# INTEGER PROGRAMMING APPROACHES FOR AUTOMATED PLANNING

by

Menkes Hector Louis van den Briel

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

ARIZONA STATE UNIVERSITY

August 2008

INTEGER PROGRAMMING APPROACHES FOR AUTOMATED PLANNING

by

Menkes Hector Louis van den Briel

has been approved

August 2008

Graduate Supervisory Committee:

Subbarao Kambhampati, Co-Chair
John Fowler, Co-Chair
Ahmet Keha
Goran Konjevod
Thomas Vossen

ACCEPTED BY THE GRADUATE COLLEGE

ABSTRACT

Automated planning is concerned with developing automated methods for generating and reasoning about sequences of actions to perform certain tasks or achieve certain goals. Since reasoning about a course of actions to achieve desired outcomes is an important hallmark of intelligent behavior, automated planning has been one of the longstanding sub-areas of artificial intelligence.

The use of integer programming approaches for automated planning has received only little attention, and is generally not thought to be competitive with the more traditional approaches for automated planning. This research, however, shows that the use of integer programming can provide effective approaches for solving automated planning problems, and has the potential to address quality and optimization criteria that arise in real world planning problems.

In particular, this research makes the following contributions. (1) It introduces novel integer programming approaches where causal considerations are separated from sequencing considerations, and where more general interpretations of action parallelism are introduced. The combination of these ideas leads to very effective integer programming formulations that are solved using a branch-and-cut algorithm. (2) It shows how to exploit these novel integer programming formulations in solving partial satisfaction problems, which are planning problems that typically require an increased emphasis on the modeling and handling of plan quality. (3) It develops a novel framework based on linear programming that gives rise to finding provably optimal plans. Moreover, this framework can also be used in heuristic search approaches for automated planning.

ACKNOWLEDGMENTS

In June 1999, about one year before I completed my Master's degree in econometrics from the University of Maastricht, I attended the International Symposium on Artificial Intelligence, Robotics and Automation for Space at the European Space Research and Technology Center (ESTEC) in Noordwijk, Netherlands. It was at this symposium where I first got interested in pursuing a PhD.

I wanted to combine my educational background in operations research with my interest in space and so I looked at many different universities. One university that stood out was Arizona State University (ASU). ASU is home to one of 17 data centers worldwide, which archives planetary images for scientific and educational use. ASU is in charge of the thermal emission spectrometer (TES) onboard the 2001 Mars Odyssey spacecraft and the mini-TES, one of the instruments attached to the Mars rovers of Spirit and Opportunity. Moreover, at the time, ASU had its own satellite program that provided students the opportunity to participate in designing, fabricating, and testing a small satellite.

Once I started my PhD at ASU I realized that getting involved in a space related project was not going to be easy. At the end of my first year I still had not been able to find a research topic of my interest. So, without much to hold on to, I was debating whether or not to leave ASU for another school. I looked at several other universities and one that caught my attention was the Surrey Space Center at the University of Surrey in the south east part of England. I applied and went to visit the university over the summer and gotten very excited about their program. So much so, that I was pretty much certain to start there in the upcoming fall semester.

When I got back from my trip, however, I heard about an interesting project on airplane boarding. After carefully considering my options I decided to work on this project and stay for one more year at ASU. This way I would at least complete the requirements for an MS and leave with a degree.

While working on the airplane boarding project I decided to make one more serious attempt to find a research topic in order to continue my PhD at ASU. I emailed several professors in the Geological Sciences Department at ASU that were involved in space related projects. Among the several replies that I received one was from Jack Farmer, back then an investigator of the Mars 2003 Landing Site Steering Committee. He asked me to drop by his office and while he may not have realized, the chat that I had with him is the main reason why this thesis is written (and likewise why I decided to stay at ASU to complete my PhD). Hence, I simply cannot thank him enough for his time that day.

When Jack pointed out that automated planning and scheduling are important and relevant problems for the Mars rovers, I knew almost instantly that this would provide a perfect middle ground in which I could blend my educational background with my personal interests. I had already worked with John Fowler on scheduling problems and in order to learn more about automated planning problems I started meeting with Subbarao (Rao) Kambhampati. After some time I began working with Rao and I familiarized myself with some of the previous research by Thomas Vossen. A couple of years after that, this thesis on *Integer Programming Approaches for Automated Planning* is the result. While it is no way related to any particular space mission, rovers like the ones on Mars increasingly depend on automated planning and scheduling technology.

John, Rao, and Thomas played an important role during my years as a PhD student. I owe a lot to them for their support and collaboration. John sparked my interest in scheduling, funded me through my difficult first year, and has always been a source of encouragement to me. Rao provided a stimulating research environment in which I had the opportunity and freedom to explore various topics of my interest even those not related to automated planning. Thomas gave me a better understanding of the art of modeling and pushed me to test the limits of my creativity. I am indebted to all three of them because without their help this thesis would never have been completed.

There are many other people who I would like to acknowledge, but a special word of thanks goes to Rene Villalobos. With him and Gary Hogg I worked on the airplane boarding project, which quickly turned into my most fun research project that I have ever been involved with. I have to thank Rene for giving me the opportunity to work with him and making me aware about the importance of validating analytic models. Without his support and collaboration this project would never have received all the media attention that is has gotten to date.

Another special word of thanks goes to Omar Ahumada and his wife Christy. We share many memorable experiences and I especially like to thank them for the travels that we have done together, for the warm and generous hospitality of their family and friends in Mexico, for the movie nights that we used to have, but above all I like to thank them for their friendship.

Furthermore, I like to thank all current and past co-researchers from the EAL, LOC, MASM, and Yochan labs. I developed many contacts and friendships within these labs and with one of them, Jennifer (Jen) Bekki, I made the following bet a couple of years ago. Whoever graduates second may borrow the graduation outfit of the one who graduates first. While at some point we were heading for a tie, by now, I can confirm that I will be wearing Jen's graduation cap to the commencement and convocation in December. Additionally, I also like to thank the people outside of ASU from Donkey Fire, FLAME, FOI, ICAPS, ISC, PARC, and RxRunning for making my stay here so much more enjoyable.

Tot slot gaat mijn dank uit naar mijn familie en vrienden in Nederland en Spanje. In het bijzonder wil ik mijn ouders Daan en Liesbeth en mijn broer Wouter bedanken voor hun onuitputtende steun en belangstelling voor mijn onderzoek. Vooral vele malen dank voor al jullie telefoontjes. Het is altijd erg leuk om van het thuisfront te horen, maar verder verbaasd het mij iedere keer weer dat het zo fijn is om gewoon even Nederlands te spreken.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

# 1 INTRODUCTION

*A goal without a plan is just a wish.*

*Antoine de Saint-Exupery (1900-1944)*

The main objective of this research is to develop and analyze integer programming approaches for automated planning problems.

The problem of automated planning is about creating a computer program that can produce a plan, where a plan is a sequence of actions that will transform the world from some given initial state to a desired goal state. The key here is that planning is *domain independent*. The world in which you want to plan could be almost anything. Hence, many computational problems can be cast as planning problems. In fact, any problem that is in the class PSPACE can be cast as a planning problem, and thus any problem that is in the class NP can be cast as a planning problem as well.

Because of its generality, there are many applications and potential application areas for automated planning technology. An ideal application area for planning is in robotics that have to operate in harsh environments, such as underwater robots and deep-space operating satellites. Other applications of planning technology include but are not limited to general game playing, manufacturing, composition of web services, and cyber security.

A closely related problem to planning is the problem of scheduling. Scheduling is an ordering problem. Given a number of tasks or actions, scheduling is the problem of deciding when to execute each action. Planning can be seen as a generalization of scheduling. Planning is not only an ordering problem, it is also a selection problem. Specifically, planning is deciding both what actions need to be done, and when to execute each action.

By analyzing the computational complexity of planning and scheduling one can observe that most of the interesting scheduling problems, like certain job shop and flow shop problems, and certain single and parallel machine problems, are NP-complete. Planning, on the other hand, is

PSPACE-complete. One way of getting an intuitive understanding for this is to be aware of that solution plans may require exponentially many actions.

By comparing planning with scheduling one can observe that scheduling problems have traditionally been studied by the operations research community and they typically involve solving hard optimization problems. Planning problems, on the other hand, have traditionally been studied by the artificial intelligence community and typically involve solving hard feasibility problems.

Integer programming approaches have been, and still are, very popular in solving scheduling problems. Since planning can be seen as a generalization of scheduling, one would expect that integer programming approaches are also very popular in solving planning problems. Surprisingly the opposite is true. Planning is an area that is dominated by constraint satisfaction, satisfiability, A* and heuristic search methods.

There has been very little focus by the artificial intelligence community on integer programming approaches for automated planning. There are several reasons for this lack of interest. For a long time, it was assumed that integer programming-based approaches for planning simply don't work. In particular, there was a general consensus that satisfiability-based approaches are much faster than integer programming-based approaches. Moreover, traditionally there has been very little focus on plan quality in general. This is due to the complexity of planning. Planning is PSPACE-complete, so why would one want to find the optimal plan when finding any feasible plan is already hard enough?

This research tries to counter these reasons and in short shows that integer programming-based approaches do work for solving planning problems and can compete with satisfiability-based approaches. Moreover, this research is part of a more general shift in focus in recent years in the artificial intelligence community towards handling plan quality. In particular, this research has resulted in the following contributions.

- Van den Briel and Kambhampati [80] describe Optiplan, an integer programming-based planner. Optiplan was judged the second best performer in four out of seven domains of the International Planning Competition. While Optiplan was not able to compete with satisfiability encodings it did open up ideas for improved integer programming approaches for automated planning.

- Van den Briel, Vossen and Kambhampati [85, 86] describe a series of novel integer programming formulations where causal considerations are separated from sequencing considerations, and where more general interpretations of action parallelism are introduced. Experimental results show that the performance of these formulations scale up with the satisfiability encodings for automated planning.

- Van den Briel et al. [84], and Do et al. [24] leverage the integer programming formulations described in this research in partial satisfaction planning. Partial satisfaction problems have recently gained a lot of attention as plan quality has become an important issue in automated planning.

- In addition, Van den Briel et al. [78] describe the development of a novel framework based on linear programming that gives rise to finding provably optimal plans. This framework can also be used in heuristic search approaches for automated planning as described by Benton, van den Briel and Kambhampati [5].

### 1.1 Outline of this thesis

Chapter 2 starts with the basics of planning and introduces some of the planning terminology. The planning problem is formally defined and different planning formalisms for planning are presented. Discussed are STRIPS and SAS+, two popular planning formalisms that provide a language to represent states, goals and actions. This chapter also provides a brief description

of the early history of planning and highlights a few successful applications in which automated planning plays a crucial role. This chapter ends with a brief description of some benchmark problems that are often used for the empirical comparison of different planning systems.

In Chapter 3 planning by integer programming is discussed. This chapter starts out with a background and an overview of the previous work and discusses the transformation of a planning problem into an integer programming problem. In planning, transformation approaches are based on either transforming the state space or plan space and presented are two naive integer programming formulations for these transformations. This chapter also provides a description of the Optiplan system, the first integer programming based planner to successfully participate in the international planning competition. The integer programming formulation of Optiplan is presented and a summary of its computational performance is given.

Chapter 4 presents three novel integer programming formulations for automated planning that rely on two key ideas. First, changes in individual state variables are modeled as flows in an appropriately defined network. As a consequence, the resulting formulations can be interpreted as network flow problems with additional side constraints. Second, causal considerations are separated from ordering considerations to generalize the notion of action parallelism. The three formulations are solved using a branch-and-cut algorithm and yield impressive results.

Chapter 5 talks about the types of planning problems where there are not enough resources to satisfy all the goals, and thus a planner must select a subset of the goals. These oversubscribed, or partial satisfaction planning problems give rise to additional challenges. This chapter provides an overview of partial satisfaction problems and discusses their computational complexity. One specific partial satisfaction planning problem in which the objective is to find a plan with the highest net benefit (i.e. total goal utility minus total action cost) is discussed in more detail.

An outline of directions for further research is given in Chapter 6. This chapter proposes to extend the current formulations to explore more expressive planning domains, involving time

and resource constraints. Moreover, a novel idea based on representing the planning problem as a set of concurrent automata and modeling them as network flow models is described. By generating the parallel composition of two or more automata one can create additional network flow constraints that strengthen the formulation. When minimizing the number of actions in the plan, the linear programming relaxation of this formulation yields very strong lower bounds and seem to suggest that this approach holds considerable promise.

Finally, a brief summary and concluding remarks are given in Chapter 7.

# 2 AUTOMATED PLANNING

*If you fail to plan, then plan to fail.*

*Unknown author*

Humans seem to plan effortlessly, but creating a computer program that can do the planning for them is a difficult challenge. Plans are created by searching through a space of possible actions until a sequence of actions is discovered that can accomplish the given task. This chapter introduces the planning problem and is organized as follows. Section 2.1 discusses some of the planning terminology through a simple example. Section 2.2 provides a brief overview of the history of planning and Section 2.3 describes a few applications in which automated planning has been successfully applied. Section 2.4 defines the planning problem more formally by describing the planning formalism STRIPS and SAS+. This chapter concludes with a few short descriptions of benchmark instances that are often used for testing and comparing different planning systems.

## 2.1 The Planning Problem

Automated planning is concerned with the process of developing methods for generating and reasoning about sequences of actions to achieve certain goals. A planner is an agent that uses some form of search to generate a sequence of actions that when executed from a given initial state will achieve some desired state. The state of the system is represented by a set of state variables that often take the value true or false. Actions are specified in the form of operators, where an operator is defined in terms of its pre-conditions, which define the applicability conditions of the action, and its post-conditions which define the effects of the action.

Planning problems are similar but in some sense more general than scheduling problems. What separates planning from scheduling is that solving a planning problem typically involves determining both *what* actions to do and *when* to do them. Scheduling, on other hand, typically involve "only" the order in which a pre-specified set of actions should occur. In other words, given a set of actions, scheduling typically involves the ordering of actions so as to meet some criteria.


Fig. 1. A simple logistics problem instance.

The simplest form of planning is known as classical planning, which is described by a system that is:

- *Static*, there are no exogenous events that could change the state of the system.
- *Fully observable*, at any point in time the planner has complete knowledge about the state of the system.
- *Deterministic*, the outcome of applying an action can be determined beforehand.
- *Finite*, there are finitely many states and actions.

A typical example of a classical planning problem is a logistics problem. This problem involves a number of packages, a number of trucks, and a number of locations. Each package and each truck starts out at one of the locations. The goal is to transport each package to its final destination. The goal may also define a final destination for some or all of the trucks. Consider, for example, an instance of a simple logistics problem with only one package, one truck, and two locations 1 and 2 as given in Figure 1. In this particular example both the package and the truck are at location 1 in the initial state and the goal is to have the package at location 2. The actions in the logistics domain describe loading a package from a location into a truck at that location, unloading a package from a truck to a location at that location, and driving a truck from one location to another. These actions are given in their standard planning domain description language (PDDL) [32] in Figure 2.

To solve a classical planning problem, a planner could simply enumerate all possible sequences of actions, evaluate their outcomes, and select one sequence of actions that satisfies the goal.

```
Action:          load(? p, ? t, ?l)
Preconditions:   at(? p, ? l)
                 at(? t, ? l)
Effects:         in(? p, ? t)
                 not at(? p, ? l)

Action:          unload(? p, ? t, ?l)
Preconditions:   in(? p, ? t)
                 at(? t, ? l)
Effects:         at(? p, ? l)
                 not in(? p, ? t)

Action:          drive(? t, ? l1, ?l2)
Preconditions:   at(? t, ? l1)
Effects:         at(? t, ? l2)
                 not at(? t, ? l1)
```

Fig. 2. A description of the load, unload, and drive actions in the logistics domain. The indicator "?" is used to refer to an object of some type.

Classical planning, however, is a hard combinatorial problem that is known to be PSPACE-complete. Currently, the best planners either use local search methods or heuristics.

### 2.2 A Brief History of Planning

The field of automated planning arose during the late 1950s from the motivation of creating a general purpose problem solver that would exhibit human problem solving characteristics. One of the first attempts was the General Problem Solver (GPS) by Newell and Simon [58]. GPS introduced the seminal idea of means-ends analysis, a technique that compares the current state with the goal, detects the difference between them, and applies an action that is likely to reduce this difference.

In the late 1960s, Fikes and Nilsson [30] introduced the Stanford Research Institute Problem Solver (STRIPS). STRIPS introduced an effective way to represent the states and operators in the planning domain. STRIPS operators are represented in terms of their pre- and post-conditions using lists containing the action's applicability conditions and lists containing the set of state variables that the action makes true or false. The STRIPS planner is one of the most famous

early planning systems and its operator representation formed a basis for many later approaches in planning.

The ideas of hierarchical planning and partial order planning emerged in the mid 1970s. Hierarchical planning was introduced by Sacerdoti [68] in a planner called ABSTRIPS. ABSTRIPS is similar to STRIPS, but instead of planning in the original problem space it plans in a hierarchy of abstraction spaces. Different levels of abstraction are created by treating some preconditions as more critical than other preconditions. The problem is solved in a stepwise procedure by progressively inserting more operators that were ignored at higher levels of abstraction.

Partial order planning was introduced by two systems: NOAH by Sacerdoti [69] and NONLIN by Tate [75]. The concept of partial order planning is based on the principle of least commitment planning, which delays decision making regarding the order of the actions until the last possible moment. Partial order planning was popular until the Graphplan planner was introduced in the mid 1990s.

Graphplan was developed by Blum and Fürst [7] and outperformed all other planners that were around at the time. Its efficient planning graph structure estimates the information that could possibly be reached from the initial state. The planning graph is a leveled graph with each level containing more information as more becomes reachable. The actual Graphplan planner alternates between two phases: a planning graph construction phase, and a planning graph search phase. The search phase searches backwards in the planning graph, starting from the goals it looks for helpful actions that have compatible pre- and post-conditions. If no solution plan is found the planning graph is extended by an extra level and search starts again.

Since the introduction of Graphplan many other planners have been developed. In particular, the last decade has seen several incredible improvements increasing both the scale and the complexity of problem instances that can now be solved with in automated planning technology. Some of which have led to successful implementations in real world applications.

## 2.3 Planning Applications

Automated planning is an active area of research that has proceeded along two directions: (1) domain independent planning and (2) domain dependent planning.

Domain independent planning, which is really the heart of automated planning, is concerned with the general form of plan synthesis. The techniques that have been proposed in this line of work are applicable to a wide variety of planning domains, but in practice, have only been effective in a few industrial and commercial planning applications. Poor performance and not being able to solve large scale problem instances contribute to their limited use to date. On the other hand, domain dependent planning technology, which is concerned with the application of planning to specific domains, has been highly successful in several real world planning applications. Some of these applications are described next.

### 2.3.1 Satellite and Spacecraft Operations

One of the most famous applications of automated planning techniques is the autonomous remote agent (RA) software system [56, 55]. The RA software system is based on automated planning techniques and was successfully tested during an experiment onboard the Deep Space One spacecraft between May 17 and May 21, 1999. It was the first time that an automated planner ran on the flight processor of a spacecraft. Automated planning in the RA software system is performed by the planner and scheduler (PS) component. The PS component tightly integrates planning and scheduling techniques. It selects and schedules appropriate actions to achieve mission goals, and synchronizes actions to allocate global resources (like power and data storage capacity) over time. Three flight experiments were run using the RA software system, two of which required onboard planning by the PS component. The generated plans included actions as camera imaging for optical navigation and engine thrusting to assure that the Deep Space One spacecraft would remain on track. The success of the PS component can be attributed in part by the use of an

expressive framework to represent concurrent actions, action durations, and time deadlines and the use of domain-dependent heuristics to guide the search for a solution plan.

Apart from Deep Space One, several other examples have successfully been implemented in satellite and spacecraft operations, including: OPTIMUM-AIV by Aarup et al. [1], CASPER by Chien et al. [19], and CONSAT by Rodríguez-Moreno et al. [67].

In general, spacecraft and satellite operations form a great application area for automated planning technology. Currently, there are several hundreds of operational satellites orbiting Earth. With many more satellites scheduled to be launched in the future, the automatic control of them becomes a very important problem. Also, as future space missions keep expanding our boundaries further into space, it is of key importance to rely increasingly more on automated planning technologies so as to lower the cost and risk that these missions may bring.

### 2.3.2 Mobile Robotics

Mobile robotics is an area that shows great potential for integrating planning techniques. Currently, most robots lack planning capabilities, but when planning is integrated it often takes on several forms, such as: path and motion planning, navigation planning, manipulation planning, and mission planning. Probably the most well studied planning problems in mobile robotics are that of path and motion planning. Path planning is the problem of finding a collision free path from an initial position to a goal position. Motion planning, on the other hand, is the problem of determining a collision free trajectory from one configuration to another. One example is TEMPEST by Tompkins, Stentz, and Wettergreen [76], a path planner for long-range planetary navigation. TEMPEST finds feasible routes that respect resource and other operational constraints for navigating a rover through a rocky landscape. Although, TEMPEST is still under development, it has already successfully been tested on a rover traversing over a simulated Mars terrain.

Other applications of path and motion planning that integrate planning techniques include: Hilare, a car like robot with a trailer and an arm by Lamiraux, Sekhavat and Laumond [52], and the humanoid robot HRP by Kanehiro et al. [45].

Overall, robotics is a perfect application area for automated planning technology. Already some examples have entered the mainstream culture, with successes like the robotic vacuum cleaners and the robotic lawn mowers. Even though these examples work only in very restricted environments and do not perform planning as such, robotics will continue to spread in our everyday live, replacing more and more appliances in increasingly more challenging domains. Automated planning technology will eventually become a basic need for advanced robotics operating in diverse and complex environments.

### 2.3.3 Manufacturing Operations

One example in manufacturing is the Interactive Manufacturability Analysis and Critiquing System (IMACS) by Nau et al. [57]. IMACS analyzes the manufacturability of designs for machined parts by generating and evaluating manufacturing operation plans. An operation plan is a sequence of machining operations capable of creating the required part from a given object, and is feasible if it satisfies the design tolerances. IMACS incorporates algorithms to recognize portions of a Computer Aided Design (CAD) model that can be created by machining operations. The machining operations that are considered by IMACS include end milling, side milling, face milling, and drilling. When creating an operation plan, there are potentially infinitely many different machining operations that are capable of creating the various portions of the machined part. In IMACS, this is addressed by using some sort of preprocessing on the machining operations. Even though IMACS is a prototype written for research purposes, it provides a designer useful feedback about the manufacturability of a design.

Currently, only very few successful commercial planning systems are being used by manufacturing industries. Geometric aspects and tolerance information of the manufactured goods often make the planning problem too demanding, but with the recent advances in automated planning technology this may change in the near future. Manufacturing is a great potential application area for automated planning technology. In the manufacturing of goods, industries always strive to maximize their productivity, maintain a high level of quality, and minimize operating cost all at the same time. The way these industries can accomplish this is by introducing automating in their manufacturing operations. Automation in manufacturing reduces labor costs and improves the throughput of the system. Moreover, automation can improve the quality of the manufactured good as it reduces variation in the manufacturing process.

## 2.4 Planning Formalisms

Planning systems face two important issues, first modeling of actions and their effects, and second organizing the search for plans. It is important to describe how the actions change the state in which they are executed and describe the parts of the state that remain unaffected after their execution. Planning formalisms provide a general framework for encoding planning problems. Several formalisms have been introduced over the years including: STRIPS, ADL, TWEAK, SAS, and SAS+.

### 2.4.1 STRIPS

STRIPS is probably the most famous planning formalism. In STRIPS states are represented by a set of propositional atoms, and a set of operators that map states into states. A STRIPS planner takes as input an initial state, a goal, and a set of operators. The task of a STRIPS planner is to find a plan that leads from the initial state to the goal.

Formally, STRIPS is defined as follows:

**Definition 2.4.1.** An instance of a STRIPS planning problem is described by a tuple $\Pi = \langle P, O, I, G \rangle$ where:

- $P$ is a finite set of propositional atoms also called the conditions. For each propositional atom $p \in P$, $s[p]$ denotes the value of $p$ in a state $s$.
- $O$ is a finite set of operators of the form $\langle pre, add, del \rangle$, where $pre$ denotes the pre-conditions (precondition list), $add$ denotes the positive post-conditions (add list), and $del$ denotes the negative post-conditions (delete list). For each operator $o \in O$, $pre(o), add(o)$ and $del(o)$ denotes $pre, add$ and $del$ respectively.
- $I \subseteq P$ denotes the initial state.
- $G \subseteq P$ denotes the goal, a conjunction of positive conditions.

### 2.4.2 SAS+

SAS+ is a planning formalism that is at the basis of some of the integer programming formulations that will described in later sections of this document. There are two main differences between the SAS+ formalism and the STRIPS formalism. First, SAS+ uses multi-valued state variables instead of propositional atoms. In STRIPS an atom can have one of two possible values, namely true or false. In SAS+, a state variable can have an arbitrary number of defined values in addition to the undefined value $u$. Second, in SAS+ operators can have one or more *prevail* conditions on top of the normal pre- and post-conditions. A prevail condition specifies a certain value that must hold before and during the execution of that action.

Formally, SAS+ is defined as follows:

**Definition 2.4.2.** An instance of a SAS+ planning problem is described by a tuple $\Pi = \langle C, O, s_I, s_G \rangle$ where:

- $C$ is a finite set of state variables. Each state variable $c \in C$ has an associated finite domain of values $D_c$. For each state variable $c \in C$, $s[c]$ denotes the value of $c$ in a state $s$.

- $O$ is a finite set of operators of the form $\langle pre, post, prev \rangle$, where $pre$ denotes the pre-conditions, $post$ denotes the post-conditions, and $prev$ denotes the prevail-conditions. For each operator $o \in O$, $pre(o), add(o)$ and $prev(o)$ denotes $pre, add$ and $prev$ respectively. The following two restrictions are imposed on all operators $o \in O$: (1) Once a state variable is made defined, it can never be made undefined. Hence, for all $c \in C$, if $pre[c] \neq u$ then $pre[c] \neq post[c] \neq u$; (2) A prevail- and post-condition of an operator can never define the same state variable. Hence, for all $c \in C$, either $post[c] = u$ or $prev[c] = u$ or neither.

- $s_I \subseteq S$ denotes the initial state.

- $s_G \subseteq S$ denotes the goal.

### 2.5 Benchmark Instances

Several benchmark instances are available through the International Planning Competition (IPC). The IPC is a bi-annual event that was first organized in 1998. Since then, the competition has introduced a diverse collection of benchmark domains that is often used for testing and comparing different planning systems. In order to give an idea of the sorts of problems automated planning is concerned with, four of these domains are described next.

**Blocksworld** The blocksworld problem is a standard planning benchmark domain. In this domain, there is a table and a number of blocks that can be placed either on the table or stacked on top of another block. The goal is to rearrange the blocks by moving them from the table to another block, from a block another block to the table, or from a block to another block. A restricted version of this domain uses a robot arm for stacking, unstacking, putting down, and picking up a block.

**Logistics** The logistics problem is another standard planning benchmark domain. In this domain, there are a number of cities and in each city a number of locations. Trucks can move between locations in the same city, and airplanes can move between locations that are designated as airports. The goal is to transport packages from their initial location to their respective destinations. There are many variations of this domain that introduce complications such as fuel restrictions and carrying capacity.

**Gripper** The gripper problem involves two rooms and a number of balls. The balls are located in either of the two rooms and must be transported from one room to the other using a gripper. While the gripper problem is computationally easy, several planning systems have a hard time in solving large and even medium sized problem instances.

**TPP** The travelling purchase problem (TPP) is a relatively recent planning domain and is a generalization of the travelling salesman problem. In this domain, there is a set of products and a set of markets. Each market is provided with a limited amount of each product at a known price. The TPP problem consists in selecting a subset of markets such that a given demand of each product can be purchased, minimizing the routing cost and the purchasing cost. This problem arises in several routing and scheduling applications.

**Freecell** Freecell is a popular solitaire card game and one of the many puzzle/game benchmark domains in planning. In this domain, cards are randomly dispersed over some playing piles. The object of the game is to build up foundation piles that must start with an Ace and follow suit in ascending order. Cards can be moved by following a number of rules and by using a number of free cells.

**Airport** The airport problem is a real application benchmark domain that deals with the problem of controlling the ground traffic on an airport. The goal is to guide the inbound airplanes to their designated gates and the outbound airplanes to the runways respecting separation

standards. The optimization version of this domain requires that the total airplane taxiing time is minimized. The largest available instances in this domain are modeled after Munich International Airport.

# 3 PLANNING BY INTEGER PROGRAMMING

*Adventure is just bad planning.*

*Roald Amundsen (1872 - 1928)*

One way to solve a planning problem is by transforming it into a form that is amenable to special purpose algorithms. Examples like this include: converting the planning problem into a satisfiability (SAT) problem and solve the corresponding SAT instance using a SAT solver [48]; converting the planning problem into a constraint satisfaction problem (CSP) and solve the corresponding CSP instance using a CSP solver [25, 77]; converting the planning problem into a large logical formula that can be expressed as a binary decision diagram and then use model checking to determine the status of the formula [29, 39]; and through converting the planning problem into an integer programming (IP) problem and solve the corresponding instance of the IP problem using an IP solver [87].

This chapter is about solving planning problems by converting them into integer programming problems and is organized as follows. Section 3.1 provides a brief background on IP and describes the previous work using IP as a means for planning. In Section 3.2 the transformation of a STRIPS planning problem to a corresponding integer programming problem is discussed. Two naive transformations are presented, one based on state space planning and the other on partial order planning. Section 3.3 describes Optiplan, the first IP based to successfully participate in the international planning competition. Part of this chapter is based on Van den Briel and Kambhampati [80] of which a preliminary version appeared in Van den Briel and Kambhampati [79].

## 3.1 Background

A *mixed integer program* is represented by a linear objective function and a set of linear inequalities:

$$\min\{cx : Ax \geq b, x_1, ..., x_p \geq 0 \text{ and integer}, x_{p+1}, ..., x_n \geq 0\},$$

where $A$ is an $(m \times n)$ matrix, $c$ is an $n$-dimensional row vector, $b$ is an $m$-dimensional column vector, and $x$ an $n$-dimensional column vector of variables. If all variables are continuous ($p = 0$) we have a *linear program*, if all variables are integer ($p = n$) we have an *integer program*, and if $x_1, ..., x_p \in \{0, 1\}$ we have a *mixed 0-1 program*. The set $S = \{x_1, ..., x_p \geq 0 \text{ and integer}, x_{p+1}, ..., x_n \geq 0 : Ax \geq b\}$ is called the *feasible region*, and an $n$-dimensional column vector $x$ is called a *feasible solution* if $x \in S$. Moreover, the function $cx$ is called the *objective function*, and the feasible solution $x^*$ is called an *optimal solution* if the objective function is as small as possible, that is, $cx^* = \min\{cx : x \in S\}$

Mixed integer programming provides a rich modeling formalism that is more general than propositional logic. Any propositional clause can be represented by one linear inequality in 0-1 variables, but a single linear inequality in 0-1 variables may require exponentially many clauses [40].

The most widely used method for solving (mixed) integer programs is by applying a branch-and-bound algorithm to the *linear programming relaxation*, which is much easier to solve. The linear programming (LP) relaxation is a linear program obtained from the original (mixed) integer program by relaxing the integrality constraints:

$$\min\{cx : Ax \geq b, x_1, ..., x_n \geq 0\}$$

Generally, the LP relaxation is solved at every node in the branch-and-bound tree, until (1) the LP relaxation gives an integer solution, (2) the LP relaxation value is inferior to the current best feasible solution, or (3) the LP relaxation is infeasible, which implies that the corresponding (mixed) integer program is infeasible.

An *ideal* formulation of an integer program is one for which the solution of the linear programming relaxation is integral. Even though every integer program has an ideal formulation [89], in practice it is very hard to characterize the ideal formulation as it may require an exponential number of inequalities. In problems where the ideal formulation cannot be determined, it is often

desirable to find a *strong* formulation of the integer program. Suppose that the feasible regions $P_1 = \{x \in R^n : A_1 x \geq b_1\}$ and $P_2 = \{x \in R^n : A_2 x \geq b_2\}$ describe the linear programming relaxations of two IP formulations of a problem. Then we say that formulation for $P_1$ is stronger than formulation for $P_2$ if $P_1 \subset P_2$. That is, the feasible region $P_1$ is subsumed by the feasible region $P_2$. In other words $P_1$ improves the quality of the linear relaxation of $P_2$ by removing fractional extreme points.

There exist numerous powerful software packages that solve mixed integer programs. In our experiments we make use of the commercial solver CPLEX 10.0 [41], which is currently one of the best LP/IP solvers.

The use of integer programming techniques to solve artificial intelligence planning problems has an intuitive appeal, especially given the success IP has had in solving similar types of problems. For example, IP has been used extensively for solving problems in transportation, logistics, and manufacturing. Examples include crew scheduling, vehicle routing, and production planning problems [42]. One potential advantage is that IP techniques can provide a natural way to incorporate several important aspects of real-world planning problems, such as numeric constraints and objective functions involving costs and utilities.

Planning as integer programming has, nevertheless, received only limited attention. Only a handful of works have explored the use of IP techniques in automated planning. The first appears to have been by Bylander [15], who used a linear programming formulation as a heuristic in partial order planning. His results, however, did not scale well against graph based and satisfiability based planners.

Vossen et al. [87] discuss the importance of developing strong IP formulations, by comparing two formulations for classical planning. First, they consider an IP formulation based on a straightforward translation of the propositional representation by Satplan [47]. In this formulation, the variables simply take on the value 1 if a certain proposition is true, and 0 otherwise,

and the assertions expressed by IP constraints directly correspond to the logical conditions of Satplan's propositional representation. Second, they consider an IP formulation in which the original propositional variables are replaced by *state change* variables. State change variables take the value 1 if a certain proposition is added, deleted, or persisted, and 0 otherwise. While the straightforward translation of SAT-based encodings yields only mediocre results, the less intuitive formulation based on the representation of state changes results in considerable performance improvements.

Dimopoulos [21] discusses a number of ideas that further improve the state change formulation by Vossen and his colleagues. In particular, by exploiting certain mutexes, valid inequalities can be obtained that tighten the formulation. Some of these ideas are implemented in the IP-based planner Optiplan [80].

A somewhat different approach that relies on domain-specific knowledge is proposed by Bockmayr and Dimopoulos [8, 9]. These formulations provide very strong relaxations, but are limited to solving domain specific problems only.

The use of LP and IP has also been explored for non-classical planning. Dimopoulos and Gerevini [22] describe an IP formulation for temporal planning and Wolfman and Weld [88] use LP formulations in combination with a SAT-based planner to solve resource planning problems. Kautz and Walser [50] use IP formulations for resource planning problems that incorporate action costs and complex objectives.

### 3.2 Transformation of the Planning Problem

All the previous work in planning by integer programming have been based on transforming the state space to an IP formulation. There are, however, at least two substantially different transformations of a planning problem depending on the how the search space is defined. The search space can be defined as the set of all possible states, or as the set of all possible plans.

The following notation is used. $F$ is the set of propositions and $A$ is the set of actions. The propositions that are true in the initial state and the propositions that must be true in the goal are given by the sets $I$ and $G$ respectively. Furthermore, the following sets are defined:

- $pre(a) \subseteq F, \forall a \in A$, set of preconditions of action $a$;
- $add(a) \subseteq F, \forall a \in A$, set of add effects of action $a$;
- $del(a) \subseteq F, \forall a \in A$, set of delete effects of action $a$;
- $pre_f \subseteq A, \forall f \in F$, set of actions that have proposition $f$ as precondition;
- $add_f \subseteq A, \forall f \in F$, set of actions that have proposition $f$ as add effect;
- $del_f \subseteq A, \forall f \in F$, set of actions that have proposition $f$ as delete effect;

### 3.2.1 Transformation for State Space Planning

In a state space formulation the search space is defined by the set of all possible states, and the objective is to find a sequence of actions that transform the initial state into a state that satisfies the goal. Transformation of planning problems based on state space planning will often show better performance if they are combined with graph based planning that help to reduce the size of the search space. A straightforward integer programming formulation of a state space formulation of a planning problem is given as follows.

For every action $a \in A$ at step $t$ a binary variable is introduced:

$$x_{a,t} = \begin{cases} 1 & \text{if action } a \text{ is executed in plan step } t, \\ 0 & \text{otherwise.} \end{cases} \quad \forall a \in A, t \in \{1, ..., T\} \tag{1}$$

For every proposition $f$ at step $t$ a binary variable is introduced:

$$y_{f,t} = \begin{cases} 1 & \text{if proposition } f \text{ is true in plan step } t, \\ 0 & \text{otherwise.} \end{cases} \quad \forall f \in F, t \in \{0, ..., T\} \tag{2}$$

The IP formulation is given by:

$$\text{MIN} \sum_{a \in A, t \in \{1,...,T\}} x_{a,t} \tag{3}$$

subject to

$$y_{f,0} = 1 \qquad \forall f \in I \tag{4}$$

$$y_{f,0} = 0 \qquad \forall f \notin I \tag{5}$$

$$y_{f,T} = 1 \qquad \forall f \in G \tag{6}$$

$$x_{a,t} \leq y_{f,t-1} \qquad \forall f \in F, a \in pre_f, t \in \{1,...,T\} \tag{7}$$

$$y_{f,t} \leq \sum_{a \in add_f} x_{a,t} \qquad \forall f \in F, t \in \{1,...,T\} \tag{8}$$

$$x_{a_1,t} + x_{a_2,t} \leq 1 \qquad \forall a_1 \in del_f, a_2 \in pre_f \cap add_f, t \in \{1,...,T\} \tag{9}$$

$$y_{f,t} \in \{0,1\} \qquad \forall f \in F, t \in \{0,...,T\} \tag{10}$$

$$x_{a,t} \in \{0,1\} \qquad \forall a \in A, t \in \{1,...,T\} \tag{11}$$

Constraints (4) and (5) specify initial state conditions, and constraints (6) specify the goal state conditions. Precondition constraints are modeled by (7) and imply that actions can be executed only if their preconditions are satisfied. Constraints (8) model the chaining conditions between actions and propositions, they imply that add effects of actions executed in plan step $t$ must be true at the end of plan step $t$. Conflict exclusion constraints are given by (9). They restrict two actions to be executed in the same step if one actions deletes the precondition or add effect of the other action. The binary constraints are given by (10) and (11).

### 3.2.2 Transformation for Partial Ordering Planning

In a plan space formulation of planning, the search space is defined by the set of all possible plans. In this case, a plan is represented by a partially ordered set of actions. The advantage of this representation is that the order between any two actions is defined only if there is some reason

for it. In order to talk about causal links and orderings between a pair of actions the following additional notation is used. The initial state and goal state are represented by the actions $a_I$ and $a_G$ respectively. For each $a_1, a_2 \in \{A \cup a_I \cup a_G\}$, $f \in F$ such that $f \in add(a_1)$ and $f \in pre(a_2)$ we define $L$ to be the set of causal links, and for each $a_1, a_2 \in \{A \cup a_I \cup a_G\}$ we define $P$ to be the set of action pairs.

A straightforward integer programming formulation of a plan space formulation of a planning problem is given as follows.

For every action $a$ a binary variable is introduced:

$$x_a = \begin{cases} 1 & \text{if action } a \text{ is executed,} \\ 0 & \text{otherwise.} \end{cases} \qquad \forall a \in A \cup a_I \cup a_G \tag{12}$$

For every causal link $(a_1, a_2, f) \in L$ a binary variable is introduced:

$$y_{a_1,a_2,f} = \begin{cases} 1 & \text{if proposition } f \text{ is produced by action } a_1 \\ & \text{and consumed by action } a_2 \\ 0 & \text{otherwise.} \end{cases} \qquad \forall f \in F, a_1, a_2 \in \{A \cup a_I \cup a_G\} \tag{13}$$

For every action pair $(a_1, a_2) \in P$ a binary variable is introduced:

$$z_{a_1,a_2} = \begin{cases} 1 & \text{if action } a_1 \text{ is ordered before action } a_2 \\ 0 & \text{otherwise.} \end{cases} \qquad \forall a_1, a_2 \in \{A \cup a_I \cup a_G\} \tag{14}$$

The IP formulation is given by:

$$\text{MIN} \sum_{a \in A} x_a \tag{15}$$

subject to

$$x_{a_I} = 1 \tag{16}$$

$$x_{a_G} = 1 \tag{17}$$

$$x_{a_2} = \sum_{(a_1,a_2,f) \in L} y_{a_1,a_2,f} \qquad \forall a_2 \in \{A \cup a_G\} \tag{18}$$

$$y_{a_1,a_2,f} \leq x_{a_2} \qquad \forall (a_1, a_2, f) \in L \tag{19}$$

$$y_{a_1,a_2,f} \leq z_{a_1,a_2} \qquad \forall (a_1, a_2, f) \in L \tag{20}$$

$$z_{a_1,a_2} + z_{a_2,a_1} \geq x_{a_1} + x_{a_2} - 1 \qquad \forall a_1, a_2 \in \{A \cup a_I \cup a_G\} \tag{21}$$

$$z_{a_1,a_2} + z_{a_2,a_1} \leq 1 \qquad \forall (a_1, a_2) \in P \tag{22}$$

$$z_{a_1,a_2} + z_{a_2,a_3} + z_{a_3,a_1} \leq 2 \qquad \forall (a_1, a_2) \in P \tag{23}$$

$$y_{a_1,a_2,f} + z_{a_1,a_3} + z_{a_3,a_2} \leq 2 \qquad \forall (a_1, a_2, f) \in L, (a_1, a_3), (a3, a_2) \in P \tag{24}$$

$$x_a \in \{0, 1\} \qquad \forall a \in \{A \cup a_I \cup a_G\} \tag{25}$$

$$y_{a_1,a_2,f} \in \{0, 1\} \qquad \forall (a_1, a_2, f) \in L \tag{26}$$

$$z_{a_1,a_2} \in \{0, 1\} \qquad \forall (a_1, a_2) \in P \tag{27}$$

Constraints (16) and (17) specify the initial and goal state conditions by forcing the start and end action to be executed. Preconditions constraints are given by (18) and imply that if an action is executed then all its preconditions must be linked to effects of other actions using the causal link variables. Constraints (19) imply that for each causal link the corresponding providing action is executed, and constraints (20) merely fixes the order of the actions defined by the causal link. Ordering constraints are given by (21)-(23). These constraints eliminate cycles and ensure transitivity. Constraints (24) make sure that the plan is free of threats. The binary constraints are given by (25)-(27).

### 3.3 The Optiplan planner

Optiplan was the first integer programming (IP) based planner to take part in the international planning competition (IPC). In the fourth IPC, which was held in 2004, it was judged second best performer in four out of seven competition domains in the optimal track for propositional domains.

The Optiplan planner unifies IP-based and graph-based planning. Its architecture is very similar to Blackbox by Kautz and Selman [48], a planner that unifies satisfiability-based and graph-based planning; and to GP-CSP by Do and Kambhampati [25], a planner that unifies constraint satisfaction-based and graph-based planning. Like Blackbox and GP-CSP, Optiplan works in two phases. In the first phase a planning graph is built and transformed into an IP formulation, and in the second phase the IP formulation is solved by the commercial solver ILOG CPLEX [41].

Optiplan's IP formulation is an extension to the state change formulation by Vossen et al. [87]. One of the practical differences is that the state change formulation by Vossen and his colleagues considers all ground actions and propositions, whereas Optiplan only considers the actions and propositions that are instantiated by Graphplan [7]. It is well known that the use of planning graph has a significant effect on the size of the encoding as it reduces the search space. For instance, Kautz and Selman [48] as well as Kambhampati [43] point out that Blackbox's success over Satplan [47] is mainly explained by Graphplan's ability to produce better, more refined, propositional structures than Satplan.

Another practical difference is that Optiplan allows propositions to be deleted without them being required as preconditions. Such state changes are not modeled in the state change model, and therefore Optiplan can be considered to be a more general formulation. One more, although minor, implementation detail between Optiplan and the state change model is that Optiplan reads in PDDL files.

### 3.3.1 Mathematical model

Optiplan is a planning graph based planner and works as follows. First the planning graph is built to the plan step where all the goal propositions appear non-mutex. Second the planning graph is compiled into an integer program, which is subsequently solved. If no plan is found, the planning graph is extended by one step and the new graph is compiled into an integer program and solved again. This process is repeated until a plan is found or a maximum step has been reached.

The IP formulation that is used by Optiplan uses the notation that has been introduced previously. Variables are defined for each plan step $t \in \{1, ..., T\}$ in the planning graph. There are variables for the actions and there are variables for the possible state changes, but only those variables that are reachable and relevant by planning graph analysis are instantiated. For all $a \in A$, $t \in \{1, ..., T\}$ we have the action variables

$$x_{a,t} = \begin{cases} 1 & \text{if action } a \text{ is executed in plan step } t, \\ 0 & \text{otherwise.} \end{cases}$$

The "no-op" actions are not included in the $x_{a,t}$ variables but are represented separately by the state change variable $y_{f,t}^{maintain}$.

Optiplan uses the state change variables as defined by Vossen et al. [87]. State change variables are used to model transitions in the world state. That is, a proposition is true if and only if it is added to the state by either $y_{f,t}^{add}$ or $y_{f,t}^{preadd}$, or if it is persisted from the previous state by $y_{f,t}^{maintain}$. Optiplan extends the state change formulation by introducing extra state change variables, $y_{f,t}^{del}$, that allow actions to delete propositions without requiring them as preconditions. The IPC4 domains of Airport and PSR include many actions with such delete effects, making the original state change formulation ineffective. For all $f \in F$, $t \in \{1, ..., T\}$ we have the following state change variables:

$$y_{f,t}^{maintain} = \begin{cases} 1 & \text{if proposition } f \text{ is propagated in plan step } t, \\ 0 & \text{otherwise.} \end{cases}$$

$$y_{f,t}^{preadd} = \begin{cases} 1 & \text{if action } a \text{ is executed in plan step } t \text{ such that } a \in pre_f \wedge a \notin del_f, \\ 0 & \text{otherwise.} \end{cases}$$

$$y_{f,t}^{predel} = \begin{cases} 1 & \text{if action } a \text{ is executed in plan step } t \text{ such that } a \in pre_f \wedge a \in del_f, \\ 0 & \text{otherwise.} \end{cases}$$

$$y_{f,t}^{add} = \begin{cases} 1 & \text{if action } a \text{ is executed in plan step } t \text{ such that } a \notin pre_f \wedge a \in add_f, \\ 0 & \text{otherwise.} \end{cases}$$

$$y_{f,t}^{del} = \begin{cases} 1 & \text{if action } a \text{ is executed in plan step } t \text{ such that } a \notin pre_f \wedge a \in del_f, \\ 0 & \text{otherwise.} \end{cases}$$

In summary: $y_{f,t}^{maintain} = 1$ if the truth value of a proposition is propagated; $y_{f,t}^{preadd} = 1$ if an action is executed that requires a proposition and does not delete it; $y_{f,t}^{predel} = 1$ if an action is executed that requires a proposition and deletes it; $y_{f,t}^{add} = 1$ if an action is executed that does not require a proposition but adds it; and $y_{f,t}^{del} = 1$ if an action is executed that does not require a proposition but deletes it. The complete IP formulation of Optiplan is given by the following objective function and constraints.

**Objective**

For STRIPS planning problems, no optimization is required as any feasible solution will compose a feasible plan. The search for a solution, however, may be guided by an objective function such as the minimization of the number of actions, which is currently implemented in Optiplan. The objective function is given by:

$$\min \sum_{a \in A} \sum_{t \in \{1,\ldots,T\}} x_{a,t} \tag{28}$$

Since the constraints guarantee feasibility we could have used any linear objective function. For example, we could easily set up an objective to deal with cost-sensitive plans (in the context of non-uniform action cost), utility-sensitive plans (in the context of over-subscription and goal utilities), or any other metric that can be transformed to a linear expression. Indeed this flexibility to handle any linear objective function is one of the advantages of IP formulations.

**Constraints**

The requirements on the initial and goal transition are given by:

$$y_{f,0}^{add} = 1 \qquad \forall f \in I \tag{29}$$

$$y_{f,0}^{add} + y_{f,0}^{maintain} + y_{f,0}^{preadd} = 0 \qquad \forall f \notin I \tag{30}$$

$$y_{f,T}^{add} + y_{f,T}^{maintain} + y_{f,T}^{preadd} \geq 1 \qquad \forall f \in G \tag{31}$$

Where constraints (29), and (30) add the initial propositions in step 0 so that they can be used by the actions that appear in the first layer (step 1) of the planning graph. Constraints (31) represent the goal state requirements, that is, propositions that appear in the goal must be added or propagated in step $T$.

The state change variables are linked to the actions by the following effect implication constraints. For each $f \in F$ and $t \in \{1, ..., T\}$ we have:

$$\sum_{a \in add_f \setminus pre_f} x_{a,t} \geq y^{add}_{f,t} \tag{32}$$

$$x_{a,t} \leq y^{add}_{f,t} \qquad \forall a \in add_f \setminus pre_f \tag{33}$$

$$\sum_{a \in del_f \setminus pre_f} x_{a,t} \geq y^{del}_{f,t} \tag{34}$$

$$x_{a,t} \leq y^{del}_{f,t} \qquad \forall a \in del_f \setminus pre_f \tag{35}$$

$$\sum_{a \in pre_f \setminus del_f} x_{a,t} \geq y^{preadd}_{f,t} \tag{36}$$

$$x_{a,t} \leq y^{preadd}_{f,t} \qquad \forall a \in pre_f \setminus del_f \tag{37}$$

$$\sum_{a \in pre_f \wedge del_f} x_{a,t} = y^{predel}_{f,t} \tag{38}$$

Where constraints (32) to (38) represent the logical relations between the action and state change variables. The equality sign in (38) is because all actions that have $f$ as a precondition and as a delete effect are mutually exclusive. This also means that we can substitute out the $y^{predel}_{f,t}$ variables, which is what we have done in the implementation of Optiplan. We will, however, use the variables here for clarity. Mutexes also appear between different state change variables and these are expressed by constraints as follows:

$$y^{add}_{f,t} + y^{maintain}_{f,t} + y^{del}_{f,t} + y^{predel}_{f,t} \leq 1 \tag{39}$$

$$y^{preadd}_{f,t} + y^{maintain}_{f,t} + y^{del}_{f,t} + y^{predel}_{f,t} \leq 1 \tag{40}$$

Where constraints (39) and (40) restrict certain state changes from occurring in parallel. For example, $y^{maintain}_{f,t}$ (propagating proposition $f$ at step $t$) is mutually exclusive with $y^{add}_{f,t}$, $y^{del}_{f,t}$, and $y^{predel}_{f,t}$ (adding or deleting $f$ at step $t$).

Finally, the backward chaining requirements and binary constraints are represented by:

$$y_{f,t}^{preadd} + y_{f,t}^{maintain} + y_{f,t}^{predel} \leq y_{f,t-1}^{preadd} + y_{f,t-1}^{add} + y_{f,t-1}^{maintain} \qquad \forall f \in F, t \in \{1, ..., T\} \quad (41)$$

$$y_{f,t}^{preadd}, y_{f,t}^{predel}, y_{f,t}^{add}, y_{f,t}^{del}, y_{f,t}^{maintain} \in \{0, 1\} \quad (42)$$

$$x_{a,t} \in \{0, 1\} \quad (43)$$

Where constraints (41) describe the backward chaining requirements, that is, if a proposition $f$ is added or maintained in step $t-1$ then the state of $f$ can be changed by an action in step $t$ through $y_{f,t}^{preadd}$, or $y_{f,t}^{predel}$, or it can be propagated through $y_{f,t}^{maintain}$. Constraints (42) and (43) are the binary constraints for the state change and action variables respectively.

### 3.3.2 Example

In this example, we show how some of the constraints are initialized and we comment on the interaction between the state change variables and the action variables.

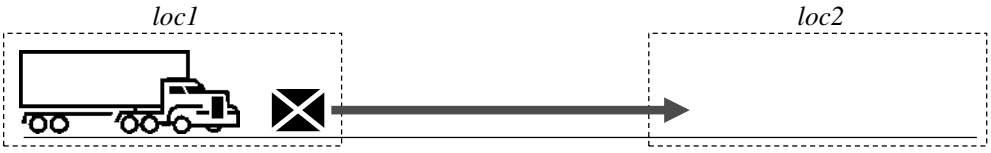


Fig. 3. A simple logistics example

Consider a simple logistics example in which there are two locations, two trucks, and one package. The package can only be transported from one location to another by one of the trucks. We built a formulation for three plan steps. The initial state is that the package and the trucks are all at location 1 as given in Figure 3. The initial state constraints are:

$$x_{pack\text{-}at\text{-}loc1,0}^{add} = 1$$

$$x_{truck1\text{-}at\text{-}loc1,0}^{add} = 1$$

$$x_{f,0}^{add}, x_{f,0}^{maintain}, x_{f,0}^{preadd} = 0 \qquad f \neq I$$

The goal is to get the package at location 2 in three plan steps, which is expressed as follows:

$$x^{add}_{pack\text{-}at\text{-}loc2,3} + x^{maintain}_{pack\text{-}at\text{-}loc2,3} + x^{preadd}_{pack\text{-}at\text{-}loc2,3} \geq 1$$

We will not write out all effect implication constraints, but we will comment on a few of them. If $x^{add}_{f,t} = 1$ for a certain fluent $f$, then we have to execute at least one action $a$ that has $f$ as an add effect and not as a precondition. For example:

$$y_{unload\text{-}truck1\text{-}at\text{-}loc2,t} + \geq x^{add}_{pack\text{-}at\text{-}loc2,t}$$

The state changes for $x^{del}_{f,t}$ and $x^{preadd}_{f,t}$ have a similar requirement, that is if we change the state through *del* or *preadd* then we must execute at least one action $a$ with the corresponding effects. On the other hand, if we execute an action $a$ then we must change all fluent states according to the effects of $a$. For example:

$$y_{unload\text{-}truck1\text{-}at\text{-}loc2,t} \leq x^{add}_{pack\text{-}at\text{-}loc2,t}$$

$$y_{unload\text{-}truck1\text{-}at\text{-}loc2,t} \leq x^{preadd}_{truck1\text{-}at\text{-}loc2,t}$$

$$y_{unload\text{-}truck1\text{-}at\text{-}loc2,t} = x^{predel}_{pack1\text{-}in\text{-}truck1,t}$$

There is a one-to-one correspondence (note the equality sign) between the execution of actions and the $x^{predel}_{f,t}$ state change variables. This is because, actions that have the same *predel* effect must be mutex. Mutexes are also present between state changes. For example, a fluent $f$ that is maintained (propagated) cannot be added or deleted. The only two state changes that are not mutex are the *add* and the *preadd*. This is because the *add* state change behaves like the *preadd* state change if the corresponding fluent is already present in the state of the world. This is why we introduce two separate mutex constraints, one that includes the *add* state change and one that includes the *preadd*. An example for the constraints on the mutex state changes are as

follows:

$$x^{add}_{pack1\text{-}in\text{-}truck1,t} + x^{maintain}_{pack1\text{-}in\text{-}truck1,t} + x^{del}_{pack1\text{-}in\text{-}truck1,t} + x^{predel}_{pack1\text{-}in\text{-}truck1,t} \leq 1$$

$$x^{preadd}_{pack1\text{-}in\text{-}truck1,t} + x^{maintain}_{pack1\text{-}in\text{-}truck1,t} + x^{del}_{pack1\text{-}in\text{-}truck1,t} + x^{predel}_{pack1\text{-}in\text{-}truck1,t} \leq 1$$

The state of a fluent can change into another state only if correct state changes have occurred previously. Hence, a fluent can be deleted, propagated, or used as preconditions in step $t$ if and only if it was added or propagated in step $t-1$. For example:

$$x^{preadd}_{pack1\text{-}in\text{-}truck1,t} + x^{maintain}_{pack1\text{-}in\text{-}truck1,t} + x^{predel}_{pack1\text{-}in\text{-}truck1,t} \leq$$

$$x^{preadd}_{pack1\text{-}in\text{-}truck1,t-1} + x^{add}_{pack1\text{-}in\text{-}truck1,t-1} + x^{maintain}_{pack1\text{-}in\text{-}truck1,t-1}$$

This simple problem has a total of 107 variables (41 action and 66 state change) and 91 constraints. However, planning graph analysis fixes 53 variables (28 action and 25 state change) to zero. After substituting these values and applying presolve techniques that are built in the ILOG CPLEX solver, this problem has only 13 variables and 17 constraints. The solution for this example is given in Table I. Note that, when there are no actions that actively delete $f$, there is nothing that ensures $x^{maintain}_{f,t}$ to be true whenever $f$ was true in the preceding state. Since negative preconditions are not allowed, having the option of letting $x^{maintain}_{f,t}$ be false when it should have been true cannot cause actions to become executable when they should not be. We will not miss any solutions because constraints (31) ensure that the goal fluents are satisfied, therefore forcing $x^{maintain}_{f,t}$ to be true whenever this helps us generate a plan.

### 3.3.3 Computational Results

First we compare Optiplan with the original state change model, and then we check how Optiplan performed in the IPC of 2004.

Optiplan and the original state change formulation are implemented in two different languages. Optiplan is implemented in C++ using Concert Technology, which is a set of libraries that allow

TABLE I

SOLUTION TO THE SIMPLE LOGISTICS EXAMPLE. ALL DISPLAYED VARIABLES HAVE VALUE 1 AND ALL OTHER VARIABLES HAVE VALUE 0.

| $t = 0$ | $t = 1$ | $t = 2$ | $t = 3$ |
|---|---|---|---|
| $x^{add}_{pack\text{-}at\text{-}loc1,0}$ | $y_{load\text{-}truck1\text{-}at-loc1,1}$ | $y_{drive\text{-}truck1\text{-}loc1\text{-}loc2,2}$ | $y_{unload\text{-}truck1\text{-}at-loc2,3}$ |
| | $x^{add}_{pack1\text{-}in\text{-}truck1,1}$ | $x^{maintain}_{pack1\text{-}in\text{-}truck1,2}$ | $x^{add}_{pack\text{-}at\text{-}loc2,3}$ |
| | $x^{predel}_{pack\text{-}at\text{-}loc1,1}$ | | $x^{predel}_{pack1\text{-}in\text{-}truck1,3}$ |
| $x^{add}_{truck1\text{-}at\text{-}loc1,0}$ | $x^{preadd}_{truck1\text{-}at\text{-}loc1,1}$ | $x^{add}_{truck1\text{-}at\text{-}loc2,2}$ | $x^{preadd}_{truck1\text{-}at\text{-}loc2,3}$ |
| | | $x^{predel}_{truck1\text{-}at\text{-}loc1,2}$ | |

you to embed ILOG CPLEX optimizers [41], and the original state change model is implemented in AMPL [31], which is a modeling language for mathematical programming. In order to compare the formulations that are produced by these two implementations, they are written to an output file using the MPS format. MPS is a standard data format that is often used for transferring linear and integer linear programming problems between different applications. Once the MPS file, which contains the IP formulation for the planning problem, is written, it is read and solved by ILOG CPLEX 8.1 on a Pentium 2.67 GHz with 1.00 GB of RAM.

Table III shows the encoding size of the two implementations, where the encoding size is characterized by the number of variables and the number of constraints in the formulation. Both the encoding size before and after applying ILOG CPLEX presolve is given. Presolve is a problem reduction technique [13] that helps most linear programming problems by simplifying, reducing and eliminating redundancies. In short, presolve tries to remove redundant constraints and fixed variables from the formulation, and aggregate (substitute out) variables if possible.

From the encoding size before presolve, which is the actual encoding size of the problem, we can see how significant the use of planning graphs is. Optiplan, which instantiates only those propositions and actions that are reachable and relevant through planning graph analysis, produces encodings that in some cases are over one order of magnitude smaller than the encodings

TABLE II

ENCODING SIZE OF THE ORIGINAL STATE CHANGE FORMULATION AND OPTIPLAN BEFORE AND AFTER ILOG CPLEX PRESOLVE. #VAR. AND #CONS. GIVE THE NUMBER OF VARIABLES AND CONSTRAINTS RESPECTIVELY.

| | State change model | | | | Optiplan | | | |
|---|---|---|---|---|---|---|---|---|
| | Before presolve | | After presolve | | Before presolve | | After presolve | |
| Problem | #Var. | #Cons. | #Var. | #Cons. | #Var. | #Cons. | #Var. | #Cons. |
| bw-sussman | 486 | 878 | 196 | 347 | 407 | 593 | 105 | 143 |
| bw-12step | 3900 | 7372 | 1663 | 3105 | 3534 | 4998 | 868 | 1025 |
| bw-large-a | 6084 | 11628 | 2645 | 5022 | 5639 | 8690 | 1800 | 2096 |
| att-log0 | 1932 | 3175 | 25 | 35 | 117 | 149 | 0 | 0 |
| log-easy | 24921 | 41457 | 1348 | 2168 | 2534 | 3029 | 437 | 592 |
| log-a | 50259 | 85324 | 3654 | 6168 | 5746 | 7480 | 1479 | 2313 |

TABLE III

PERFORMANCE AND ENCODING SIZE OF THE ORIGINAL STATE CHANGE FORMULATION AND OPTIPLAN. #VAR. AND #CONS. GIVE THE NUMBER OF VARIABLES AND CONSTRAINTS AFTER CPLEX PRESOLVE, AND #NODES GIVE THE NUMBER OF NODES EXPLORED DURING BRANCH-AND-BOUND BEFORE FINDING THE FIRST FEASIBLE SOLUTION.

| | State change model | | | | Optiplan | | | |
|---|---|---|---|---|---|---|---|---|
| Problem | #Var. | #Cons. | #Nodes | Time | #Var. | #Cons. | #Nodes | Time |
| bw-sussman | 196 | 347 | 0 | 0.01 | 105 | 143 | 0 | 0.01 |
| bw-12step | 1663 | 3105 | 19 | 4.28 | 868 | 1025 | 37 | 1.65 |
| bw-large-a | 2645 | 5022 | 2 | 8.45 | 1800 | 2096 | 0 | 0.72 |
| bw-large-b | 6331 | 12053 | 14 | 581.92 | 4780 | 5454 | 10 | 72.58 |
| att-log0 | 25 | 35 | 0 | 0.01 | 0 | 0 | 0 | 0.01 |
| att-log1 | 114 | 164 | 0 | 0.03 | 29 | 35 | 0 | 0.01 |
| att-log2 | 249 | 371 | 10 | 0.07 | 81 | 99 | 0 | 0.01 |
| att-log3 | 2151 | 3686 | 15 | 0.64 | 181 | 228 | 0 | 0.03 |
| att-log4 | 2147 | 3676 | 12 | 0.71 | 360 | 507 | 0 | 0.04 |
| att-loga | 2915 | 4968 | 975 | 173.56 | 1479 | 2312 | 19 | 2.71 |
| rocket-a | 1532 | 2653 | 517 | 32.44 | 991 | 1644 | 78 | 5.48 |
| rocket-b | 1610 | 2787 | 191 | 9.90 | 1071 | 1788 | 24 | 3.12 |
| log-easy | 1348 | 2168 | 43 | 0.96 | 437 | 592 | 0 | 0.04 |
| log-a | 3654 | 6168 | 600 | 145.31 | 1479 | 2313 | 19 | 2.66 |
| log-b | 4255 | 6989 | 325 | 96.47 | 1718 | 2620 | 187 | 14.06 |
| log-c | 5457 | 9111 | 970 | 771.36 | 2413 | 3784 | 37 | 16.07 |

produced by the original state change model, which instantiates all ground propositions and actions.

Although the difference in the encoding size reduces substantially after applying presolve, planning graph analysis still finds redundancies that presolve fails to detect. Consequently, the encodings produced by Optiplan are smaller than the encodings that are produced by the original state change model.

The performance (and the encoding size after presolve) of Optiplan and the original state change model are given in Table III. Performance is measured by the time to find the first feasible solution. The results show the overall effectiveness of using planning graph analysis. For all problems Optiplan not only generates smaller encodings it also performs better than the encodings generated by the state change model.

**IPC Results**

Optiplan participated in the propositional domains of the optimal track in the IPC 2004. In this track, planners could either minimize the number of actions, like BFHSP and Semsyn; minimize makespan, like CPT, HSP*a, Optiplan, Satplan04, and TP-4; or minimize some other metric.

The IPC results of the makespan optimal planners are given in Figure 4. All results were evaluated by the competition organizers by looking at the runtime and plan quality graphs. Also, all planners were compared to each other by estimating their asymptotic runtime and by analyzing their solution quality performance. Out of the seven competition domains, Optiplan was judged second best in four of them. This is quite remarkable because integer programming has hitherto not been considered competitive in planning.

Optiplan reached second place in the Optical Telegraph and the Philosopher domains. In these domains Optiplan is about one order of magnitude slower than Satplan04, but it clearly outperforms all other participating planners. In the Pipesworld Tankage domain, Optiplan was

awarded second place together with Satplan04, and in the Satellite domain Optiplan, CPT, and Semsyn all tied for second place. In the other domains Optiplan did not perform too well. In the Airport domain, Optiplan solves the first 17 problems and problem 19, but it takes the most time to do so. For the Pipesworld Notankage and the PSR domains, Optiplan not only is the slowest it also solves the fewest number of problems among the participating planners.

In looking at the domains and problems where Optiplan has difficulty scaling, it can be observed that these are problems lead to very large IP encodings. Since the size of the encoding is a function of plan length, Optiplan often fails to solve problems that have long solution plans. One way to resolve this issue is to de-link the encoding size from solution length, which is done in Chapter 4 and in the work by Rintanen, Heljanko and Niemelä [65]. In fact, the formulations described in Chapter 4 present novel IP encodings that (1) model transitions in the individual propositions as separate but loosely coupled network flow problems, and that (2) control the encoding length by generalizing the notion of parallelism.

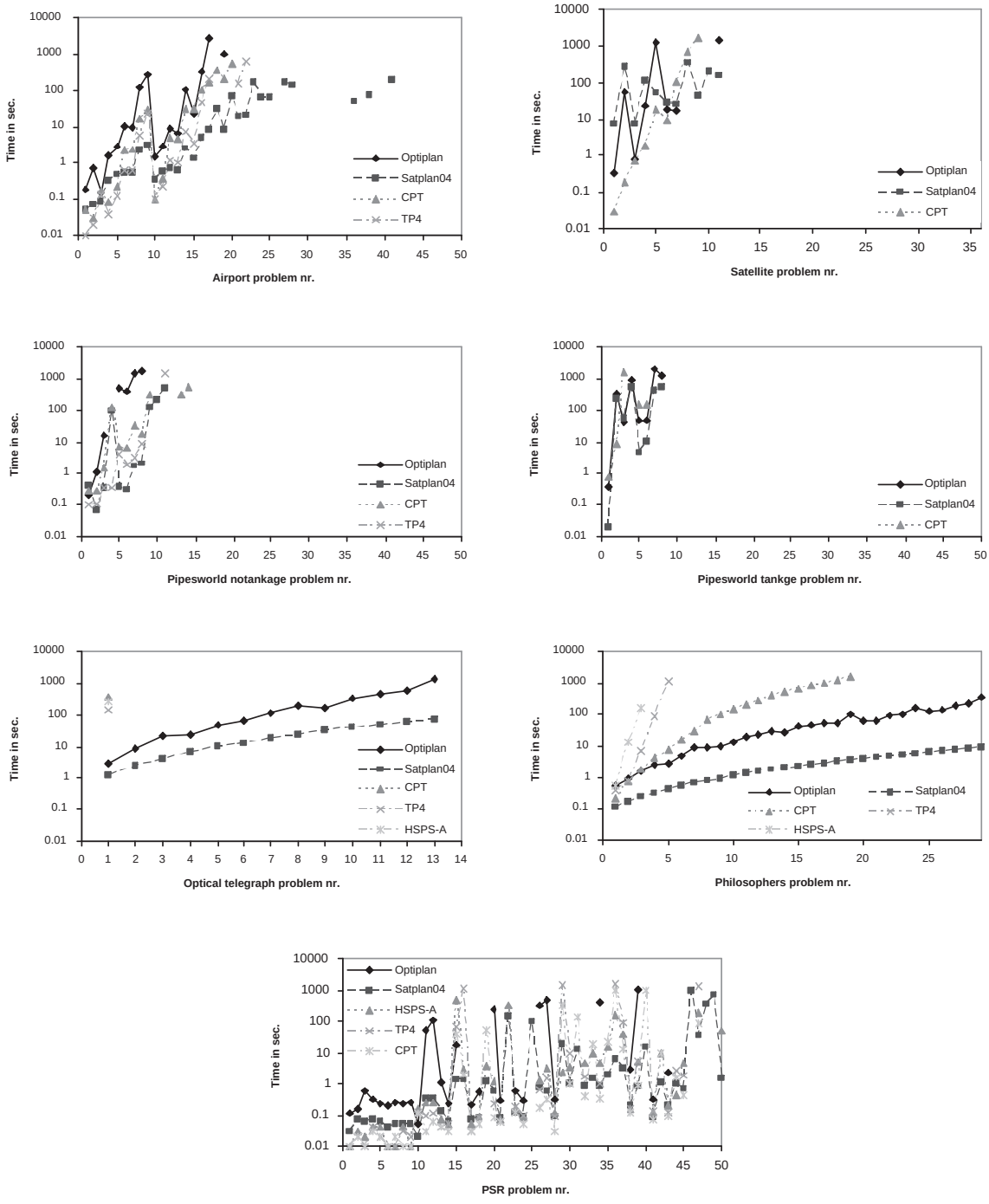


Fig. 4. IPC 2004 results for the makespan optimal planners.

# 4 PLANNING BY INTEGER PROGRAMMING REVISED: A BRANCH-AND-CUT ALGORITHM

*Just because something doesn't do what you planned it to do doesn't mean it's useless.*

*Thomas A. Edison (1847-1931)*

Despite the potential flexibility offered by integer programming approaches for automated planning, in practice planners based on integer programming have not been competitive with those based on constraint satisfaction and satisfiability encodings. This state of affairs is more a reflection on the type of integer programming formulations that have been tried until now, rather than any inherent shortcomings of integer programming as a combinatorial substrate for planning.

This chapter is based on Van den Briel, Vossen, and Kambhampati [85, 86] and presents a number of integer programming formulations that model planning as a set of loosely coupled network flow problems. Section 4.1 gives an overview of this approach and Section 4.2 provides a series of integer programming formulations that each adopt a different network representation. In Section 4.3 a branch-and-cut algorithm is described that is used for solving these formulations. It provides a general background on the branch-and-cut concept and shows how it is applied to our formulations by means of an example. Section 4.4 provides experimental results to determine which characteristics in our approach have the greatest impact on performance. Related work is discussed in Section 4.5 and some conclusions are given in Section 4.6.

## 4.1 Introduction

While integer programming approaches for automated planning have not been able to scale well against other compilation approaches (i.e. satisfiability and constraint satisfaction), they have been extremely successful in the solution of many real-world large scale optimization problems. Given that the integer programming framework has the potential to incorporate several important aspects of real-world automated planning problems (for example, numeric quantities and objective functions involving costs and utilities), there is significant motivation to investigate more effective

integer programming formulations for classical planning as they could lay the groundwork for large scale optimization (in terms of cost and resources) in automated planning. In this paper, we study a novel decomposition based approach for automated planning that yields very effective integer programming formulations.

Decomposition is a general approach to solving problems more efficiently. It involves breaking a problem up into several smaller subproblems and solving each of the subproblems separately. In this paper we use decomposition to break up a planning problem into several interacting (i.e. loosely coupled) components. In such a decomposition, the planning problem involves both finding solutions to the individual components and trying to merge them into a feasible plan. This general approach, however, prompts the following questions: (1) what are the components, (2) what are the component solutions, and (3) how hard is it to merge the individual component solutions into a feasible plan?

### 4.1.1 The Components

We let the components represent the state variables of the planning problem. Figure 5 illustrates this idea using a small logistics example, with one truck and a package that needs to be moved from location 1 to location 2. There are a total of five components in this example, one for each state variable. We represent the components by an appropriately defined network, where the network nodes correspond to the values of the state variable (for atoms this is $T$ = true and $F$ = false), and the network arcs correspond to the value transitions. The source node in each network, represented by a small in-arc, corresponds to the initial value of the state variable. The sink node(s), represented by double circles, correspond to the goal value(s) of the state variable. Note that the effects of an action can trigger value transitions in the state variables. For example, loading the package at location 1 makes the atom *pack-in-truck* true and *pack-at-loc*1 false. In addition, loading the package at location 1 requires that the atom *truck-at-loc*1 is true.

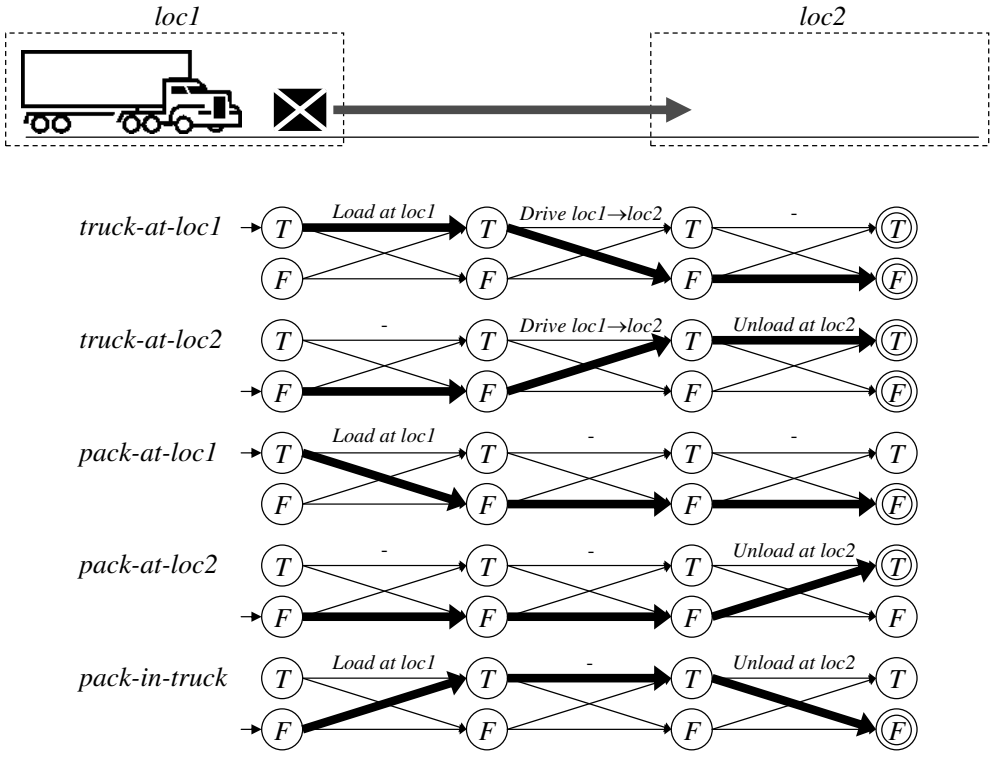


Fig. 5. Logistics example broken up into five components (binary-valued state variables) that are represented by network flow problems.

While the idea of components representing the state variables of the planning problem can be used with any state variable representation, it is particularly synergistic with multi-valued state variables. Multi-valued state variables provide a more compact representation of the planning problem than their binary-valued counterparts. Therefore, by making the conversion to multi-valued state variables we can reduce the number of components and create a better partitioning of the constraints. Figure 6 illustrates the use of multi-valued state variables on our small logistics example. There are two multi-valued state variables in this problem, one to characterize the location of the truck and one to characterize the location of the package. In our network representation, the nodes correspond to the state variable values (1 = *at-loc*1, 2 = *at-loc*2, and $t$ = *in-truck*), and the arcs correspond to the value transitions.

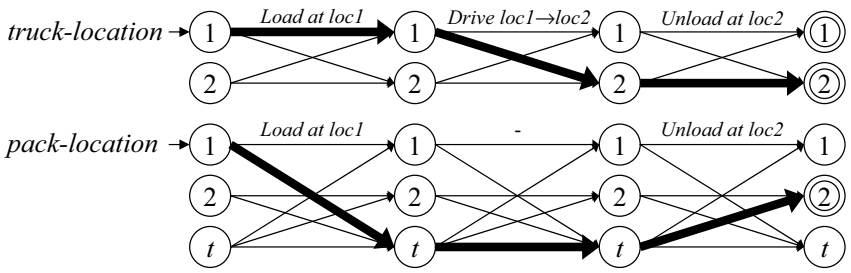


Fig. 6. Logistics example broken up into two components (multi-valued state variables) that are represented by network flow problems.

### 4.1.2 The Component Solutions

We let the component solutions represent a path of value transitions in the state variables. In the networks, nodes and arcs appear in layers. Each layer represents a plan period in which, depending on the structure of the network, one or more value transitions can occur. The networks in Figures 5 and 6 each have three layers (i.e. plan periods) and their structure allows values to persist or change exactly once per period. The layers are used to solve the planning problem incrementally. That is, we start with one layer in each network and try to solve the planning problem. If no plan is found, all networks are extended by one extra layer and a new attempt is made to solve the planning problem. This process is repeated until a plan is found or a time limit is reached. In Figures 5 and 6, a path (i.e. a solution) from the source node to one of the sink nodes is highlighted in each network. Since the execution of an action triggers value transitions in the state variables, each path in a network corresponds to a sequence of actions. Consequently, *the planning problem can be thought of as a collection of network flow problems where the problem is to find a path (i.e. a sequence of actions) in each of the networks.* However, interactions between the networks impose side constraints on the network flow problems, which complicate the solution process.

### 4.1.3 The Merging Process

We solve these loosely coupled networks using integer programming formulations. One design choice we make is that we expand all networks (i.e. components) together, so the cost of finding solutions for the individual networks as well as merging them depends on the difficulty of solving the integer programming formulation. This, in turn, typically depends on the size of the integer programming formulation, which is partly determined by the number of layers in each of the networks. The simplest idea is to have the number of layers of the networks equal the length of the plan, just as in sequential planning where the plan length equals the number of actions in the plan. In this case, there will be as many transitions in the networks as there are actions in the plan, with the only difference that a sequence of actions corresponding to a path in a network could contain no-op actions.

An idea to reduce the required number of layers is by allowing multiple actions to be executed in the same plan period. This is exactly what is done in Graphplan [7] and in other planners that have adopted the Graphplan-style definition of parallelism. That is, two actions can be executed in parallel (i.e. in the same plan period) as long as they are non-interfering. In our formulations we adopt more general notions of parallelism. In particular, we relax the strict relation between the number of layers in the networks and the length of the plan by changing the network representation of the state variables. For example, by allowing multiple transitions in each network per plan period we permit interfering actions to be executed in the same plan period. This, however, raises issues about how solutions to the individual networks are searched and how they are combined. When the network representations for the state variables allow multiple transitions in each network per plan period, and thus become more flexible, it becomes harder to merge the solutions into a feasible plan. Therefore, to evaluate the tradeoffs in allowing such flexible representations, we look at a variety of integer programming formulations.

We refer to the integer programming formulation that uses the network representation shown in Figures 5 and 6 as the *one state change* model, because it allows at most one transition (i.e. state change) per plan period in each state variable. Note that in this network representation a plan period mimics the Graphplan-style parallelism. That is, two actions can be executed in the same plan period if one action does not delete the precondition or add-effect of the other action. A more flexible representation in which values can change at most once and persist before and after each change we refer to as the *generalized one state change* model. Clearly, we can increase the number of changes that we allow in each plan period. The representations in which values can change at most twice or $k$ times, we refer to as the *generalized two state change* and the *generalized k state change* model respectively. One disadvantage with the generalized $k$ state change model is that it creates one variable for each way to do $k$ value changes, and thus introduces exponentially many variables per plan period. Therefore, another network representation that we consider allows a path of value transitions in which each value can be visited at most once per plan period. This way, we can limit the number of variables, but may introduce cycles in our networks. The integer programming formulation that uses this representation is referred to as the *state change path* model.

In general, by allowing multiple transitions in each network per plan period (i.e. layer), the more complex the merging process becomes. In particular, the merging process checks whether the actions in the solutions of the individual networks can be linearized into a feasible plan. In our integer programming formulations, ordering constraints ensure feasible linearizations. There may, however, be exponentially many ordering constraints when we generalize the Graphplan-style parallelism. Rather than inserting all these constraints in the integer programming formulation up front, we add them as needed using a branch-and-cut algorithm. A branch-and-cut algorithm is a branch-and-bound algorithm in which certain constraints are generated dynamically throughout the branch-and-bound tree.

We show that the performance of our integer programming (IP) formulations show new potential and are competitive with SATPLAN04 [46]. This is a significant result because it forms a basis for other more sophisticated IP-based planning systems capable of handling numeric constraints and non-uniform action costs. In particular, the new potential of our IP formulations has led to their successful use in solving partial satisfaction planning problems [24]. Moreover, it has initiated a new line of work in which integer and linear programming are used in heuristic state-space search for automated planning [5, 78].

### 4.2 Formulations

This section describes four IP formulations that model the planning problem as a collection of loosely coupled network flow problems. Each network represents a state variable, in which the nodes correspond to the state variable values, and the arcs correspond to the value transitions. The state variables are based on the SAS+ planning formalism [4], which is a planning formalism that uses multi-valued state variables instead of binary-valued atoms. An action in SAS+ is modeled by its pre-, post- and prevail-conditions. The pre- and post-conditions express which state variables are changed and what values they must have before and after the execution of the action, and the prevail-conditions specify which of the unchanged variables must have some specific value before and during the execution of an action. A SAS+ planning problem is described by a tuple $\Pi = \langle C, A, s_0, s_* \rangle$ where:

- $C = \{c_1, ..., c_n\}$ is a finite set of state variables, where each state variable $c \in C$ has an associated domain $V_c$ and an implicitly defined extended domain $V_c^+ = V_c \cup \{u\}$, where $u$ denotes the *undefined value*. For each state variable $c \in C$, $s[c]$ denotes the value of $c$ in state $s$. The value of $c$ is said to be *defined* in state $s$ if and only if $s[c] \neq u$. The total state space $S = V_{c_1} \times ... \times V_{c_n}$ and the partial state space $S^+ = V_{c_1}^+ \times ... \times V_{c_n}^+$ are implicitly defined.

- $A$ is a finite set of actions of the form $\langle pre, post, prev \rangle$, where $pre$ denotes the pre-conditions, $post$ denotes the post-conditions, and $prev$ denotes the prevail-conditions. For each action $a \in A$, $pre[c], post[c]$ and $prev[c]$ denotes the respective conditions on state variable $c$. The following two restrictions are imposed on all actions: (1) Once the value of a state variable is defined, it can never become undefined. Hence, for all $c \in C$, if $pre[c] \neq u$ then $pre[c] \neq post[c] \neq u$; (2) A prevail- and post-condition of an action can never define a value on the same state variable. Hence, for all $c \in C$, either $post[c] = u$ or $prev[c] = u$ or both.

- $s_0 \in S$ denotes the initial state and $s_* \in S^+$ denotes the goal state. While SAS+ planning allows the initial state and goal state to be both partial states, we assume that $s_0$ is a total state and $s_*$ is a partial state. We say that state $s$ is *satisfied* by state $t$ if and only if for all $c \in C$ we have $s[c] = u$ or $s[c] = t[c]$. This implies that if $s_*[c] = u$ for state variable $c$, then any defined value $f \in V_c$ satisfies the goal for $c$.

To obtain a SAS+ description of the planning problem we use the translator component of the Fast Downward planner [36]. The translator is a stand-alone component that contains a general purpose algorithm which transforms a propositional description of the planning problem into a SAS+ description. The algorithm provides an efficient grounding that minimizes the state description length and is based on the preprocessing algorithm of the MIPS planner [28].

In the remainder of this section we introduce some notation and describe our IP formulations. The formulations are presented in such a way that they progressively generalize the Graphplan-style parallelism through the incorporation of more flexible network representations. For each formulation we will describe the underlying network, and define the variables and constraints. We will not concentrate on the objective function as much because the constraints will tolerate only feasible plans.

### 4.2.1 Notation

For the formulations that are described in this paper we assume that the following information is given:

- $C$: a set of state variables;
- $V_c$: a set of possible values (i.e. domain) for each state variable $c \in C$;
- $E_c$: a set of possible value transitions for each state variable $c \in C$;
- $G_c = (V_c, E_c)$ : a directed domain transition graph for every $c \in C$;

State variables can be represented by a domain transition graph, where the nodes correspond to the possible values, and the arcs correspond to the possible value transitions. An example of the domain transition graph of a variable is given in Figure 7. While the example depicts a complete graph, a domain transition graph does not need to be a complete graph.

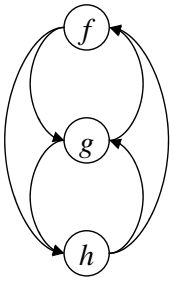


Fig. 7. An example of a domain transition graph, where $V_c = \{f, g, h\}$ are the possible values (states) of $c$ and $E_c = \{(f, g), (f, h), (g, f), (g, h), (h, f), (h, g)\}$ are the possible value transitions in $c$.

Furthermore, we assume as given:

- $E_c^a \subseteq E_c$ represents the effect of action $a$ in $c$;
- $V_c^a \subseteq V_c$ represents the prevail condition of action $a$ in $c$;
- $A_c^E := \{a \in A : |E_c^a| > 0\}$ represents the actions that have an effect in $c$, and $A_c^E(e)$ represents the actions that have the effect $e$ in $c$;

- $A_c^V := \{a \in A : |V_c^a| > 0\}$ represents the actions that have a prevail condition in $c$, and $A_c^V(f)$ represents the actions that have the prevail condition $f$ in $c$;

- $C^a := \{c \in C : a \in A_c^E \cup A_c^V\}$ represents the state variables on which action $a$ has an effect or a prevail condition.

Hence, each action is defined by its effects (i.e. pre- and post-conditions) and its prevail conditions. In SAS+ planning, actions can have at most one effect or prevail condition in each state variable. In other words, for each $a \in A$ and $c \in C$, we have that $E_c^a$ and $V_c^a$ are empty or $|E_c^a| + |V_c^a| \leq 1$. An example of how the effects and prevail conditions affect one or more domain transition graphs is given in Figure 8.

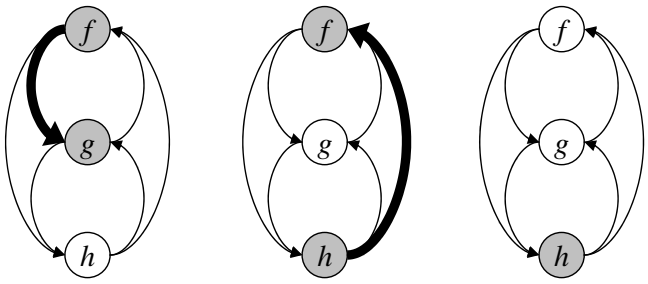


Fig. 8. An example of how action effects and prevail conditions are represented in a domain transition graph. Action $a$ has implications on three state variables $C^a = \{c_1, c_2, c_3\}$. The effects of $a$ are represented by $E_{c_1}^a = \{(f, g)\}$ and $E_{c_2}^a = \{(h, f)\}$, and the prevail condition of $a$ is represented by $V_{c_3}^a = \{h\}$.

In addition, we use the following notation:

- $V_c^+(f)$: to denote the in-arcs of node $f$ in the domain transition graph $G_c$;

- $V_c^-(f)$: to denote the out-arcs of node $f$ in the domain transition graph $G_c$;

- $P_{c,k}^+(f)$: to denote paths of length $k$ in the domain transition graph $G_c$ that end at node $f$. Note that $P_{c,1}^+(f) = V_c^+(f)$.

- $P_{c,k}^-(f)$: to denote paths of length $k$ in the domain transition graph $G_c$ that start at node $f$. Note that $P_{c,1}^-(f) = V_c^-(f)$.

- $P^{\sim}_{c,k}(f)$: to denote paths of length $k$ in the domain transition graph $G_c$ that visit node $f$, but that do not start or end at $f$.

### 4.2.2 One State Change (1SC) Formulation

Our first IP formulation incorporates the network representation that we have seen in Figures 5 and 6. The name *one state change* relates to the number of transitions that we allow in each state variable per plan period. The restriction of allowing only one value transition in each network also restricts which actions we can execute in the same plan period. It happens to be the case that the network representation of the 1SC formulation incorporates the standard notion of action parallelism which is used in Graphplan [7]. The idea is that actions can be executed in the same plan period as long as they do not delete the precondition or add-effect of another action. In terms of value transitions in state variables, this is saying that actions can be executed in the same plan period as long as they do not change the same state variable (i.e. there is only one value change or value persistence in each state variable).

**4.2.2.1 State Change Network:** Figure 9 shows a single layer (i.e. period) of the network which underlies the 1SC formulation. If we set up the IP formulation with $T$ plan periods, then there will be $T + 1$ layers of nodes and $T$ layers of arcs in the network (the zeroth layer of nodes is for the initial state and the remaining $T$ layers of nodes and arcs are for the successive plan periods). For each possible state transition there is an arc in the state change network. The horizontal arcs correspond to the persistence of a value, and the diagonal arcs correspond to the value changes. A solution path to an individual network follows the arcs whose transitions are supported by the action effect and prevail conditions that appear in the solution plan.

**4.2.2.2 Variables:** We have two types of variables in this formulation: action variables to represent the execution of an action, and arc flow variables to represent the state transitions in each network. We use separate variables for changes in a state variable (the diagonal arcs in the

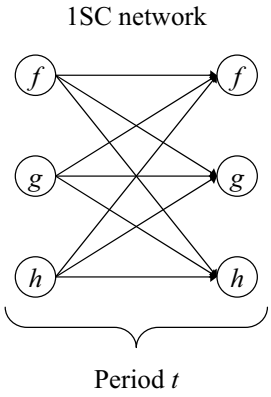


Fig. 9. One state change (1SC) network.

1SC network) and for the persistence of a value in a state variable (the horizontal arcs in the 1SC network). The variables are defined as follows:

- $x_t^a \in \{0,1\}$, for $a \in A, 1 \leq t \leq T$; $x_t^a$ is equal to 1 if action $a$ is executed at plan period $t$, and 0 otherwise.
- $\bar{y}_{c,f,t} \in \{0,1\}$, for $c \in C$, $f \in V_c$, $1 \leq t \leq T$; $\bar{y}_{c,f,t}$ is equal to 1 if the value $f$ of state variable $c$ persists at period $t$, and 0 otherwise.
- $y_{c,e,t} \in \{0,1\}$, for $c \in C, e \in E_c, 1 \leq t \leq T$; $y_{c,e,t}$ is equal to 1 if the transition $e \in E_c$ in state variable $c$ is executed at period $t$, and 0 otherwise.

**4.2.2.3 Constraints:** There are two classes of constraints. We have constraints for the network flows in each state variable network and constraints for the action effects that determine the interactions between these networks. The 1SC integer programming formulation is:

- State change flows for all $c \in C,\ f \in V_c$

$$\sum_{e \in V_c^-(f)} y_{c,e,1} + \bar{y}_{c,f,1} = \begin{cases} 1 & \text{if } f = s_0[c] \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

$$\sum_{e \in V_c^-(f)} y_{c,e,t+1} + \bar{y}_{c,f,t+1} = \sum_{e \in V_c^+(f)} y_{c,e,t} + \bar{y}_{c,f,t} \quad \text{for } 1 \le t \le T-1 \quad (2)$$

$$\sum_{e \in V_c^+(f)} y_{c,e,T} + \bar{y}_{c,f,T} = 1 \quad \text{if } f = s_*[c] \quad (3)$$

- Action implications for all $c \in C,\ 1 \le t \le T$

$$\sum_{a \in A : e \in E_c^a} x_t^a = y_{c,e,t} \quad \text{for } e \in E_c \quad (4)$$

$$x_t^a \le \bar{y}_{c,f,t} \quad \text{for } a \in A, f \in V_c^a \quad (5)$$

Constraints (1), (2), and (3) are the network flow constraints for state variable $c \in C$. Constraint (1) ensures that the path of state transitions begins in the initial state of the state variable and constraint (3) ensures that, if a goal exists, the path ends in the goal state of the state variable. Note that, if the goal value for state variable $c$ is undefined (i.e. $s_*[c] = u$) then the path of state transitions may end in any of the values $f \in V_c$. Hence, we do not need a goal constraint for the state variables whose goal states $s_*[c]$ are undefined. Constraint (2) is the flow conservation equation and enforces the continuity of the constructed path.

Actions may introduce interactions between the state variables. For instance, the effects of the *load* action in our logistics example affect two different state variables. Actions link state variables to each other and these interactions are represented by the action implication constraints. For each transition $e \in E_c$, constraints (4) link the action execution variables that have $e$ as an effect (i.e. $e \in E_c^a$) to the arc flow variables. For example, if an action $x_t^a$ with effect $e \in E_c^a$ is executed, then the path in state variable $c$ must follow the arc represented by $y_{c,e,t}$. Likewise, if we choose to follow the arc represented by $y_{c,e,t}$, then exactly one action $x_t^a$ with $e \in E_c^a$ must be executed. The summation on the left hand side prevents two or more actions from interfering with each other, hence only one action may cause the state change $e$ in state variable $c$ at period $t$.

Prevail conditions of an action link state variables in a similar way as the action effects do. Specifically, constraint (5) states that *if* action $a$ is executed at period $t$ ($x_t^a = 1$), *then* the prevail condition $f \in V_c^a$ is required in state variable $c$ at period $t$ ($\bar{y}_{c,f,t} = 1$).

### 4.2.3 Generalized One State Change (G1SC) Formulation

In our second formulation we incorporate the same network representation as in the 1SC formulation, but adopt a more general interpretation of the value transitions, which leads to an unconventional notion of action parallelism. For the G1SC formulation we relax the condition that parallel actions can be arranged in *any order* by requiring a weaker condition. We allow actions to be executed in the same plan period as long as *there exists* some ordering that is feasible. More specifically, within a plan period a set of actions is feasible if (1) there exists an ordering of the actions such that all preconditions are satisfied, and (2) there is at most one state change in each of the state variables. This generalization of conditions is similar to what Rintanen, Heljanko and Niemelä [66] refer to as the $\exists$-step semantics semantics.

To illustrate the basic concept, let us again examine our small logistics example introduced in Figure 5. The solution to this problem is to load the package at location 1, drive the truck from location 1 to location 2, and unload the package at location 2. Clearly, this plan would require three plan periods under Graphplan-style parallelism as these three actions interfere with each other. If, however, we allow the *load at loc*1 and the *drive loc*1 $\rightarrow$ *loc*2 action to be executed in the same plan period, then there exists some ordering between these two actions that is feasible, namely load the package at the location 1 before driving the truck to location 2. The key idea behind this example should be clear: while it may not be possible to find a set of actions that can be linearized in any order, there may nevertheless exist *some* ordering of the actions that is viable. The question is, of course, how to incorporate this idea into an IP formulation.

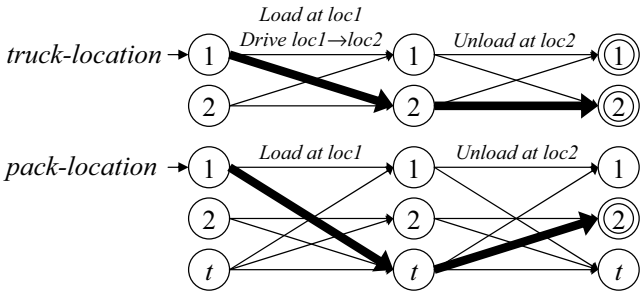


Fig. 10. Logistics example represented by network flow problems with generalized arcs.

This example illustrates that we are looking for a set of constraints that allow sets of actions for which: (1) all action preconditions are met, (2) there exists an ordering of the actions at each plan period that is feasible, and (3) within each state variable, the value is changed at most once. The incorporation of these ideas only requires minor modifications to the 1SC formulation. Specifically, we need to change the action implication constraints for the prevail conditions and add a new set of constraints which we call the ordering implication constraints.

**4.2.3.1 State Change Network:** The minor modifications are revealed in the G1SC network. While the network itself is identical to the 1SC network, the interpretation of the transition arcs is somewhat different. To incorporate the new set of conditions, we implicitly allow values to persist (the dashed horizontal arcs in the G1SC network) at the tail and head of each transition arc. The interpretation of these implicit arcs is that in each plan period a value may be required as a prevail condition, then the value may change, and the new value may also be required as a prevail condition as shown in Figure 11.

**4.2.3.2 Variables:** Since the G1SC network is similar to the 1SC network the same variables are used, thus, action variables to represent the execution of an action, and arc flow variables to represent the flow through each network. The difference in the interpretation of the state change arcs is dealt with in the constraints of the G1SC formulation, and therefore does not introduce any new variables. For the variable definitions, we refer to Section 4.2.2.2.

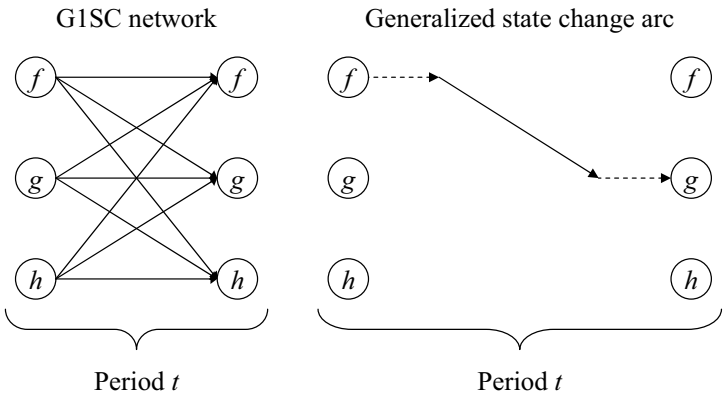


Fig. 11. Generalized one state change (G1SC) network.

**4.2.3.3 Constraints:** We now have three classes of constraints, that is, constraints for the network flows in each state variable network, constraints for linking the flows with the action effects and prevail conditions, and ordering constraints to ensure that the actions in the plan can be linearized into a feasible ordering.

The network flow constraints for the G1SC formulation are identical to those in the 1SC formulation given by (1)-(3). Moreover, the constraints that link the flows with the action effects are equal to the action effect constraints in the 1SC formulation given by (4). The G1SC formulation differs from the 1SC formulation in that it relaxes the condition that parallel actions can be arranged in any order by requiring a weaker condition. This weaker condition affects the constraints that link the flows with the action prevail conditions, and introduces a new set of ordering constraints. These constraints of the G1SC formulation are given as follows:

- Action implications for all $c \in C$, $1 \leq t \leq T$

$$x_t^a \quad \leq \quad \bar{y}_{c,f,t} + \sum_{e \in V_c^+(f)} y_{c,e,t} + \sum_{e \in V_c^-(f)} y_{c,e,t} \quad \text{for } a \in A, f \in V_c^a \tag{6}$$

- Ordering implications

$$\sum_{a \in V(\Delta)} x_t^a \quad \leq \quad |V(\Delta)| - 1 \text{ for all cycles } \Delta \in G^{prec} \tag{7}$$

Constraint (6) incorporates this new set of conditions for which actions can be executed in the same plan period. In particular, we need to ensure that for each state variable $c$, the value $f \in V_c$ holds if it is required by the prevail condition of action $a$ at plan period $t$. There are three possibilities: (1) The value $f$ holds for $c$ throughout the period. (2) The value $f$ holds initially for $c$, but the value is changed to a value other than $f$ by another action. (3) The value $f$ does not hold initially for $c$, but the value is changed to $f$ by another action. In either of the three cases the value $f$ holds at some point in period $t$ so that the prevail condition for action $a$ can be satisfied. In words, the value $f$ may prevail implicitly as long as there is a state change that includes $f$. As before, the prevail implication constraints link the action prevail conditions to the corresponding network arcs.

The action implication constraints ensure that the preconditions of the actions in the plan are satisfied. This, however, does not guarantee that the actions can be linearized into a feasible order. Figure 11 indicates that there are implied orderings between actions. Actions that require the value $f$ as a prevail condition must be executed before the action that changes $f$ into $g$. Likewise, an action that changes $f$ into $g$ must be executed before actions that require the value $g$ as a prevail condition. The state change flow and action implication constraints outlined above indicate that there is an ordering between the actions, but this ordering could be cyclic and therefore infeasible. To make sure that an ordering is acyclic we start by creating a directed *implied precedence graph* $G^{prec} = (V^{prec}, E^{prec})$. In this graph the nodes $a \in V^{prec}$ correspond to the actions, that is, $V^{prec} = A$, and we create a directed arc (i.e. an ordering) between two nodes $(a, b) \in E^{prec}$ if action $a$ has to be executed before action $b$ in time period $t$, or if $b$ has to be executed after $a$. In particular, we have

$$E^{prec} = \bigcup_{\substack{(a,b)\in A\times A, c\in C, f\in V_c^a, e\in E_c^b: \\ e\in V_{c,f}^-}} (a,b) \cup \bigcup_{\substack{(a,b)\in A\times A, c\in C, g\in V_c^b, e\in E_c^a: \\ e\in V_{c,g}^+}} (a,b)$$

The implied orderings become immediately clear from Figure 12. The figure on the left depicts the first set of orderings in the expression of $E^{prec}$. It says that the ordering between two actions $a$ and $b$ that are executed in the same plan period is implied if action $a$ requires a value to prevail that action $b$ deletes. Similarly, the figure on the right depicts second set of orderings in the expression of $E^{prec}$. That is, an ordering is implied if action $a$ adds the prevail condition of $b$.

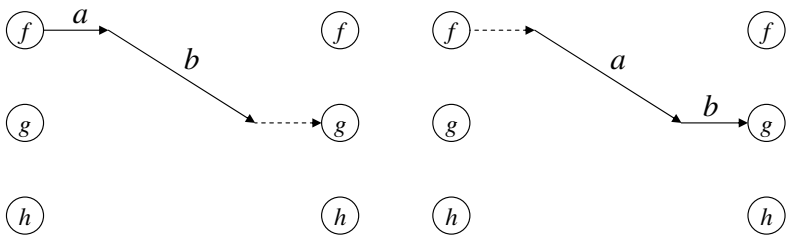


Fig. 12. Implied orderings for the G1SC formulation.

The ordering implication constraints ensure that the actions in the final solution can be linearized. They basically involve putting an $n$-ary mutex relation between the actions that are involved in each cycle. Unfortunately, the number of ordering implication constraints grows exponentially in the number of actions. As a result, it will be impossible to solve the resulting formulation using standard approaches. We address this complication by implementing a branch-and-cut approach in which the ordering implication constraints are added dynamically to the formulation. This approach is discussed in Section 4.3.

### 4.2.4 Generalized $k$ State Change (G$k$SC) Formulation

In the G1SC formulation actions can be executed in the same plan period if (1) there exists an ordering of the actions such that all preconditions are satisfied, and (2) there occurs at most one value change in each of the state variables. One obvious generalization of this would be to relax the second condition and allow at most $k_c$ value changes in each state variable $c$, where $k_c \leq |V_c| - 1$. By allowing multiple value changes in a state variable per plan period we, in fact, permit a series of value changes. Specifically, the G$k$SC model allows series of value changes.

Obviously, there is a tradeoff between loosening the networks versus the amount of work it takes to merge the individual plans. While we have not implemented the G$k$SC formulation, we provide some insight in this tradeoff by describing and evaluating the G$k$SC formulation with $k_c = 2$ for all $c \in C$ We will refer to this special case as the generalized two state change (G2SC) formulation. One reason we restrict ourselves to this special case is that the general case of $k$ state changes would introduce exponentially many variables in the formulation. There are IP techniques, however, that deal with exponentially many variables [20], but we will not discuss them here.

**4.2.4.1 State Change Network:** The network that underlies the G2SC formulation is equivalent to G1SC, but spans an extra layer of nodes and arcs. This extra layer allows us to have a series of two transitions per plan period. All transitions are generalized and implicitly allow values to persist just as in the G1SC network. Figure 13 displays the network corresponding to the G2SC formulation. In the G2SC network there are generalized one and two state change arcs. For example, there is a generalized one state change arc for the transition $(f, g)$, and there is a generalized two state changes arc for the transitions $\{(f, g), (g, h)\}$. Since all arcs are generalized, each value that is visited can also be persisted. We also allow cyclic transitions, such as, $\{(f, g), (g, f)\}$ if $f$ is not the prevail condition of some action. If we were to allow cyclic transitions in which $f$ is a prevail condition of an action, then the action ordering in a plan period can not be implied anymore (i.e. the prevail condition on $f$ would either have to occur before the value transitions to $g$, or after it transitions back to $f$). Thus if there is no prevail condition on $f$ then we can safely allow the cyclic transition $\{(f, g), (g, f)\}$.

**4.2.4.2 Variables:** As before we have variables representing the execution of an action, and variables representing the flows over one state change (diagonal arcs) or persistence (horizontal arcs). In addition, we have variables representing paths over two consecutive state changes. Hence, we have variables for each pair of state changes $(f, g, h)$ such that $(f, g) \in E_c$ and $(g, h) \in$

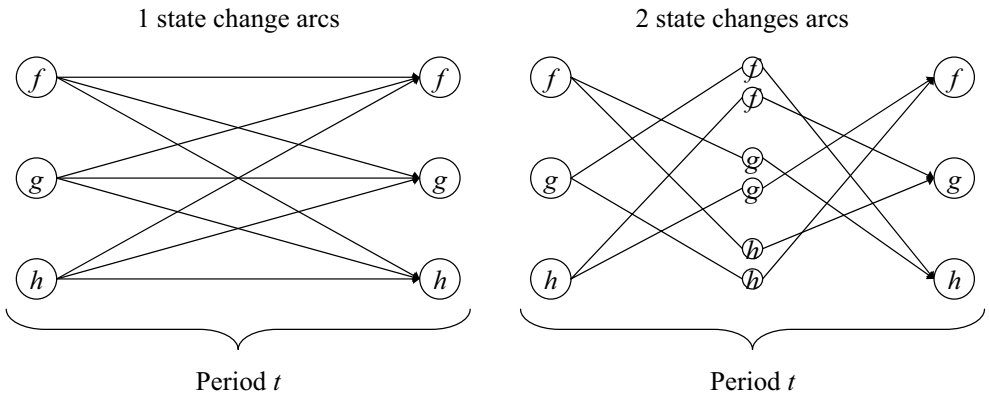


Fig. 13. Generalized two state change (G2SC) network. On the left the subnetwork that consists of generalized one state change arcs and no-op arcs, on the right the subnetwork that consists of the generalized two state change arcs. The subnetwork for the two state change arcs may include cyclic transitions, such as, $\{(f, g), (g, f)\}$ as long as $f$ is not the prevail condition of some action.

$E_c$. We will restrict these paths to visit unique values only, that is, $f \neq g$, $g \neq h$, and $h \neq f$, or if $f$ is not a prevail condition of any action then we also allow paths where $f = h$. The variables from the G1SC formulation are also used in G2SC formulation. There is, however, an additional variable to represent the arcs that allow for two state changes:

- $y_{c,e_1,e_2,t} \in \{0, 1\}$, for $c \in C$, $(e_1, e_2) \in P_{c,2}$, $1 \leq t \leq T$; $y_{c,e_1,e_2,t}$ is equal to 1 if there exists a value $f \in V_c$ and transitions $e_1, e_2 \in E_c$, such that $e_1 \in V_c^+(f)$ and $e_2 \in V_c^-(f)$, in state variable $c$ are executed at period $t$, and 0 otherwise.

**4.2.4.3 Constraints:** We again have our three classes of constraints, which are given as follows:

- State change flows for all $c \in C,\ f \in V_c$

$$\sum_{(e_1,e_2)\in P^-_{c,2}(f)} y_{c,e_1,e_2,1} + \sum_{e\in V^-_c(f)} y_{c,e,1} + \bar{y}_{c,f,1} = \begin{cases} 1 & \text{if } f = s_0[c] \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

$$\sum_{(e_1,e_2)\in P^-_{c,2}(f)} y_{c,e_1,e_2,t+1} + \sum_{e\in V^-_c(f)} y_{c,e,t+1} + \bar{y}_{c,f,t+1} = \sum_{(e_1,e_2)\in P^+_{c,2}(f)} y_{c,e_1,e_2,t} + \sum_{e\in V^+_c(f)} y_{c,e,t} + \bar{y}_{c,f,t} \quad \text{for } 1 \le t \le T-1 \tag{9}$$

$$\sum_{(e_1,e_2)\in P^+_{c,2}(f)} y_{c,e_1,e_2,T} + \sum_{e\in V^+_c(f)} y_{c,e,T} + \bar{y}_{c,f,T} = 1 \quad \text{if } \{f \in s_*[c]\} \tag{10}$$

- Action implications for all $c \in C,\ 1 \le t \le T$

$$\sum_{a\in A: e\in E^a_c} x^a_t = y_{c,e,t} + \sum_{(e_1,e_2)\in P_{c,2}: e_1=e \vee e_2=e} y_{c,e_1,e_2,t} \quad \text{for } e \in E_c \tag{11}$$

$$x^a_t \le \bar{y}_{c,f,t} + \sum_{e\in V^+_c(f)} y_{c,e,t} + \sum_{e\in V^-_c(f)} y_{c,e,t} + \sum_{(e_1,e_2)\in P^{\sim}_{c,2}(f)} y_{c,e_1,e_2,t} + \sum_{(e_1,e_2)\in P^+_{c,2}(f)} y_{c,e_1,e_2,t} + \sum_{(e_1,e_2)\in P^-_{c,2}(f)} y_{c,e_1,e_2,t} \quad \text{for } a \in A, f \in V^a_c \tag{12}$$

- Ordering implications

$$\sum_{a\in V(\Delta)} x^a_t \le |V(\Delta)| - 1 \text{ for all cycles } \Delta \in G^{prec} \tag{13}$$

Constraints (8), (9), and (10) represent the flow constraints for the G2SC network. Constraints (11) and (12) link the action effects and prevail conditions with the corresponding flows, and constraint (13) ensures that the actions can be linearized into some feasible ordering.

### 4.2.5 State Change Path (PathSC) Formulation

There are several ways to generalize the network representation of the G1SC formulation and loosen the interaction between the networks. The G$k$SC formulation presented one generalization that allows up to $k$ transitions in each state variable per plan period. Since it uses exponentially many variables another way to generalize the network representation of the G1SC formulation

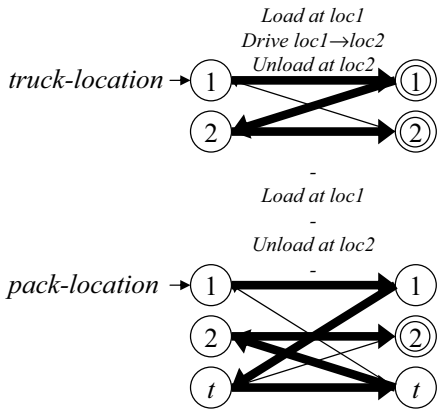


Fig. 14. Logistics example represented by network flow problems that allow a path of value transitions per plan period such that each value can be true at most once.

is by requiring that each value can be true at most once per plan period. To illustrate this idea we consider our logistics example again, but we now use a network representation that allows a path of transitions per plan period as depicted in Figure 14.

Recall that the solution to the logistics example consists of three actions: first load the package at location 1, then drive the truck from location 1 to location 2, and last unload the package at location 2. Clearly, this solution would not be allowed within a single plan period under Graphplan-style parallelism. Moreover, it would also not be allowed within a single period in the G1SC formulation. The reason for this is that the number of value changes in the *package-location* state variable is two. First, it changes from *pack-at-loc*1 to *pack-in-truck*, and then it changes from *pack-in-truck* to *pack-at-loc*2. As before, however, there does exists an ordering of the three actions that is feasible. The key idea behind this example is to show that we can allow multiple value changes in a single period. If we limit the value changes in a state variable to simple paths, that is, in one period each value is visited at most once, then we can still use implied precedences to determine the ordering restrictions.

**4.2.5.1 State Change Network:** In this formulation each value can be true at most once in each plan period, hence the number of value transitions for each plan period is limited to $k_c$ where $k_c = |V_c| - 1$ for each $c \in C$. In the PathSC network, nodes appear in layers and correspond

to the values of the state variable. However, each layer now consists of twice as many nodes. If we set up an IP encoding with a maximum number of plan periods $T$ then there will be $T$ layers. Arcs within a layer correspond to transitions or to value persistence, and arcs between layers ensure that all plan periods are connected to each other.

Figure 15 displays a network corresponding to the state variable $c$ with domain $V_c = \{f, g, h\}$ that allows multiple transitions per plan period. The arcs pointing rightwards correspond to the persistence of a value, while the arcs pointing leftwards correspond to the value changes. If more than one plan period is needed the curved arcs pointing rightwards link the layers between two consecutive plan periods. Note that with unit capacity on the arcs, any path in the network can visit each node at most once.

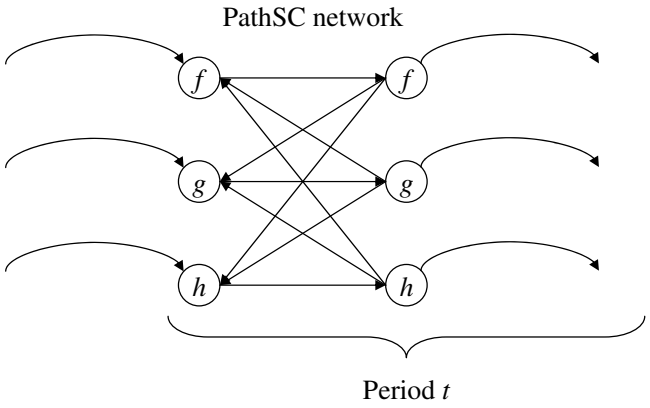


Fig. 15. Path state change (PathSC) network.

**4.2.5.2 Variables:** We now have action execution variables and arc flow variables (as defined in the previous formulations), and linking variables that connect the networks between two consecutive time periods. These variables are defined as follows:

- $z_{c,f,t} \in \{0, 1\}$, for $c \in C, f \in V_c, 0 \leq t \leq T$; $z_{c,f,t}$ is equal to 1 if the value $f$ of state variable $c$ is the end value at period $t$, and 0 otherwise.

**4.2.5.3 Constraints:** As in the previous formulations, we have state change flow constraints, action implication constraints, and ordering implication constraints. The main difference is the underlying network. The PathSC integer programming formulation is given as follows:

- State change flows for all $c \in C,\ f \in V_c$

$$z_{c,f,0} = \begin{cases} 1 & \text{if } f = s_0[c] \\ 0 & \text{otherwise.} \end{cases} \tag{14}$$

$$\sum_{e \in V_c^+(f)} y_{c,e,t} + z_{c,f,t-1} = \bar{y}_{c,f,t} \tag{15}$$

$$\bar{y}_{c,f,t} = \sum_{e \in V_c^-(f)} y_{c,e,t} + z_{c,f,t} \quad \text{for } 1 \le t \le T-1 \tag{16}$$

$$z_{c,f,T} = 1 \quad \text{if } f \in s_*[c] \tag{17}$$

- Action implications for all $c \in C,\ 1 \le t \le T$

$$\sum_{a \in A : e \in E_c^a} x_t^a = y_{c,e,t} \quad \text{for } e \in E_c \tag{18}$$

$$x_t^a \le \bar{y}_{c,f,t} \quad \text{for } f \in V_c^a \tag{19}$$

- Ordering implications

$$\sum_{a \in V(\Delta)} x_t^a \le |V(\Delta)| - 1 \text{ for all cycles } \Delta \in G^{prec'} \tag{20}$$

Constraints (14)-(17) are the network flow constraints. For each node, except for the initial and goal state nodes, they ensure a balance of flow (i.e. flow-in must equal flow-out). The initial state node has a supply of one unit of flow and the goal state node has a demand of one unit of flow, which are given by constraints (14) and (17) respectively. The interactions that actions impose upon different state variables are represented by the action implication constraints (18) and (19), which have been discussed earlier.

The implied precedence graph for this formulation is given by $G^{prec'} = (V^{prec'}, E^{prec'})$. It has an extra set of arcs to incorporate the implied precedences that are introduced when two actions

imply a state change in the same class $c \in C$. The nodes $a \in V^{prec'}$ again correspond to actions, and there is an arc $(a, b) \in E^{prec'}$ if action $a$ has to be executed before action $b$ in the same time period, or if $b$ has to be executed after $a$. More specifically, we have

$$E^{prec'} = E^{prec} \cup \bigcup_{\substack{(a,b)\in A\times A, c\in C, f\in V_c, e\in E_c^a, e'\in E_c^b: \\ e\in V_c^+(f)\wedge e'\in V_c^-(f)}} (a, b)$$

As before, the ordering implication constraints (20) ensure that the actions in the solution plan can be linearized into a feasible ordering.

### 4.3 Branch-and-Cut Algorithm

IP problems are usually solved with an LP-based branch-and-bound algorithm. The basic structure of this technique involves a binary enumeration tree in which branches are pruned according to bounds provided by the LP relaxation. The root node in the enumeration tree represents the LP relaxation of the original IP problem and each other node represents a subproblem that has the same objective function and constraints as the root node except for some additional bound constraints. Most IP solvers use an LP-based branch-and-bound algorithm in combination with various preprocessing and probing techniques. In the last few years there has been significant improvement in the performance of these solvers [6].

In an LP-based branch-and-bound algorithm, the LP relaxation of the original IP problem (the solution to the root node) will rarely be integer. When some integer variable $x$ has a fractional solution $v$ we branch to create two new subproblems, such that the bound constraint $x \leq \lfloor v \rfloor$ is added to the left-child node, and $x \geq \lceil v \rceil$ is added to the right-child node. This branching process is carried out recursively to expand those subproblems whose solution remains fractional. Eventually, after enough bounds are placed on the variables, an integer solution is found. The

value of the best integer solution found so far, $Z^*$, is referred to as the incumbent and is used for pruning.

In a minimization problem, branches emanating from nodes whose solution value $Z_{\text{LP}}$ is greater than the current incumbent, $Z^*$, can never give rise to a better integer solution as each child node has a smaller feasible region than its parent. Hence, we can safely eliminate such nodes from further consideration and prune them. Nodes whose feasible region have been reduced to the empty set, because too many bounds are placed on the variables, can be pruned as well.

When solving an IP problem with an LP-based branch-and-bound algorithm we must consider the following two decisions. If several integer variables have a fractional solution, which variable should we branch on next, and if the branch we are currently working on is pruned, which subproblem should we solve next? Basic rules include use the "most fractional variable" rule for branching variable selection and the "best objective value" rule for node selection.

For our formulations a standard LP-based branch-and-bound algorithm approach is very ineffective due to the large number (potentially exponentially many) ordering implication constraints in the G1SC, G2SC, and PathSC formulations. While it is possible to reduce the number of constraints by introducing additional variables [54], the resulting formulations would still be intractable for all but the smallest problem instances. Therefore, we solve the IP formulations with a so-called *branch-and-cut* algorithm, which considers the ordering implication constraints implicitly. A branch-and-cut algorithm is a branch-and-bound algorithm in which certain constraints are generated dynamically throughout the branch-and-bound tree. A flowchart of our branch-and-cut algorithm is given in Figure 16.

If, after solving the LP relaxation, we are unable to prune the node on the basis of the LP solution, the branch-and-cut algorithm tries to find a violated cut, that is, a constraint that is valid but not satisfied by the current solution. This is also known as the *separation problem*. If one or more violated cuts are found, the constraints are added to the formulation and the LP is

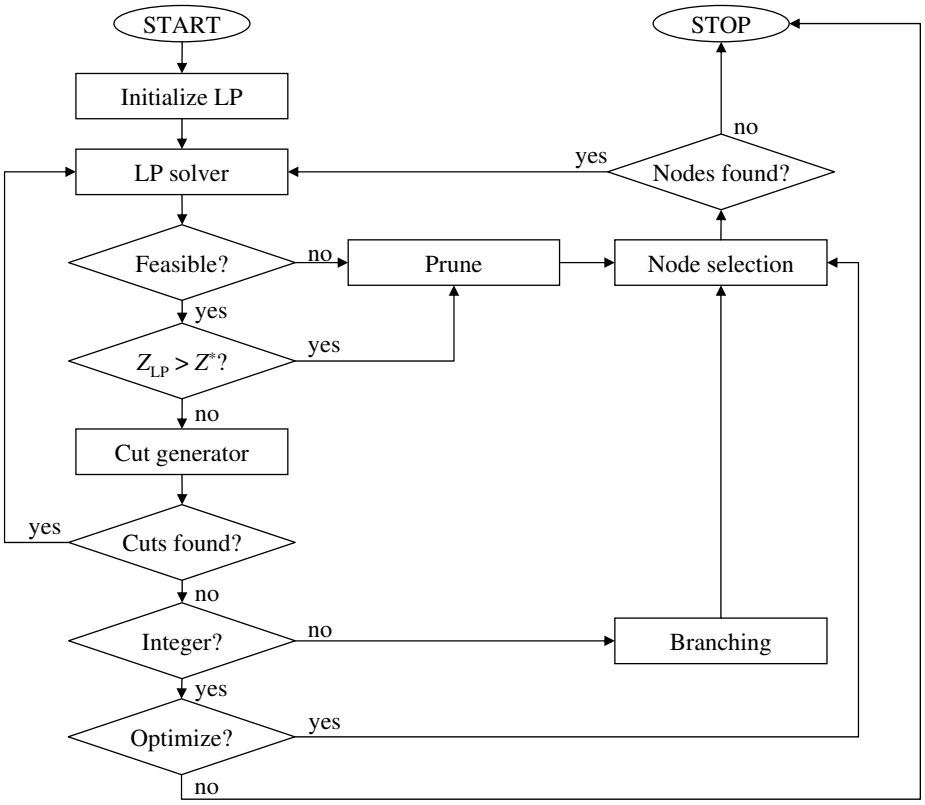


Fig. 16. Flowchart of our branch-and-cut algorithm. For finding any feasible solution (i.e. optimize = no) the algorithm stops as soon as the first feasible integer solution is found. When searching for the optimal solution (i.e. optimize = yes) for the given formulation we continue until no open nodes are left.

solved again. If none are found, the algorithm creates a branch in the enumeration tree (if the solution to the current subproblem is fractional) or generates a feasible solution (if the solution to the current subproblem is integral).

The basic idea of branch-and-cut is to leave out constraints from the LP relaxation of which there are too many to handle efficiently, and add them to the formulation only when they become binding at the solution to the current LP. Branch-and-cut algorithms have successfully been applied in solving hard large-scale optimization problems in a wide variety of applications including scheduling, routing, graph partitioning, network design, and facility location problems [16].

In our branch-and-cut algorithm we can stop as soon as we find the first feasible solution, or we can implicitly enumerate all nodes (through pruning) and find the optimal solution for a given objective function. Note that our formulations can only be used to find bounded length optimal plans. That is, find the optimal plan given a plan period (i.e. a bounded length). In our experimental results, however, we focus on finding feasible solutions.

### 4.3.1 Constraint Generation

At any point during runtime that the cut generator is called we have a solution to the current LP problem, which consists of the LP relaxation of the original IP problem plus any added bound constraints and added cuts. In our implementation of the branch-and-cut algorithm, we start with an LP relaxation in which the ordering implication constraints are omitted. So given a solution to the current LP relaxation, which could be fractional, the separation problem is to determine whether the solution violates one of the omitted ordering implication constraints. If so, we identify the violated ordering implication constraints, add them to the formulation, and resolve the new problem.

**4.3.1.1 Cycle Identification:** In the G1SC, G2SC, and PathSC formulations an ordering implication constraint is violated if there is a cycle in the implied precedence graph. Separation

problems involving cycles occur in numerous applications. Probably the best known of its kind is the traveling salesman problem in which subtours (i.e. cycles) are identified and subtour elimination constraints are added to the current LP. Our algorithm for separating cycles is based on the one described by Padberg and Rinaldi [60]. We are interested in finding the shortest cycle in the implied precedence graph, as the shortest cycle cuts off more fractional extreme points. The general idea behind this approach is as follows:

1. Given a solution to the LP relaxation, determine the subgraph $G_t$ for plan period $t$ consisting of all the nodes $a$ for which $x_t^a > 0$.

2. For all the arcs $(a, b) \in G_t$, define the weights $w_{a,b} := x_t^a + x_t^b - 1$.

3. Determine the shortest path distance $d_{a,b}$ for all pairs $((a, b) \in G_t)$ based on arc weights $\bar{w}_{a,b} := 1 - w_{a,b}$ (for example, using the Floyd-Warshall all-pairs shortest path algorithm).

4. If $d_{a,b} - w_{b,a} < 0$ for some arc $(a, b) \in G_t$, there exists a violated cycle constraint.

While the general principles behind branch-and-cut algorithms are rather straightforward, there are a number of algorithmic and implementation issues that may have a significant impact on overall performance. At the heart of these issues is the trade-off between computation time spent at each node in the enumeration tree and the number of nodes that are explored. One issue, for example, is to decide when to generate violated cuts. Another issue is which of the generated cuts (if any) should be added to the LP relaxation, and whether and when to delete constraints that were added to the LP before. In our implementation, we have only addressed these issues in a straightforward manner: cuts are generated at every node in the enumeration tree, the first cut found by the algorithm is added, and constraints are never deleted from the LP relaxation. However, given the potential of more advanced strategies that has been observed in other applications, we believe there still may be considerable room for improvement.

**4.3.1.2 Example:** In this section we will show the workings of our branch-and-cut algorithm on the G1SC formulation using a small hypothetical example involving two state variables $c_1$ and $c_2$, five actions $A1$, $A2$, $A3$, $A4$, and $A5$, and one plan period. In particular we will show how the cycle detection procedure works and how an ordering implication constraint is generated.

Figure 17 depicts a solution to the current LP of the planning problem. For state variable $c_1$ we have that actions $A1$ and $A2$ have a prevail condition on $g$, $A4$ has a prevail condition on $h$, and action $A3$ has an effect that changes $g$ into $h$. Likewise, for state variable $c_2$ we have that action $A4$ has an effect that changes $g$ into $f$, action $A5$ changes $g$ into $h$, and action $A1$ has a prevail condition on $f$. Note that the given solution is fractional. Therefore some of the action variables have fractional values. In particular, we have $x^{A1} = x^{A4} = 0.8$, $x^{A5} = 0.2$, and $x^{A2} = x^{A3} = 1$. In other words, actions $A2$ and $A3$ are fully executed while actions $A1$, $A4$ and $A5$ are only fractionally executed. Clearly, in automated planning the fractional execution of an action has no meaning whatsoever, but it is very common that the LP relaxation of an IP formulation gives a fractional solution. We simply try to show that we can find a violated cut even when we have a fractional solution. Also, note that the actions $A4$ and $A5$ have interfering effects in $c_2$. While this would generally be infeasible, the actions are executed only fractionally, so this is actually a feasible solution to the LP relaxation of the IP formulation.

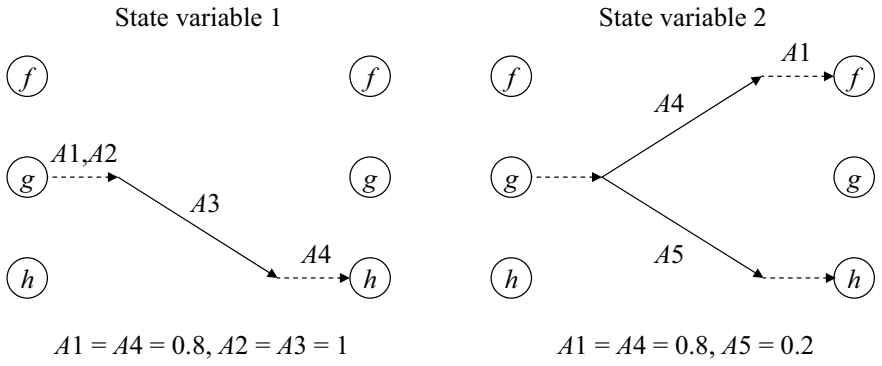


Fig. 17. Solution to a small hypothetical planning example. The solution to the current LP has flows over the indicated paths and executes actions $A1$, $A2$, $A3$, $A4$, and $A5$.

In order to determine whether the actions can be linearized into a feasible ordering we first create the implied precedence graph $G^{prec} = (V^{prec}, E^{prec})$, where we have $V^{prec} = \{A1, A2, A3, A4, A5\}$ and $E^{prec} = \{(A1, A3),(A2, A3),(A3, A4),(A4, A1)\}$. The ordering $(A1, A3)$, for example, is established by the effects of these actions in state variable $c_1$. $A1$ has a prevail condition $g$ in $c_1$ while $A3$ changes $g$ to $h$ in $c_1$, which implies that $A1$ must be executed before $A3$. The other orderings are established in a similar way. The complete implied precedence graph for this example is given in Figure 18.

The cycle detection algorithm gets the implied precedence graph and the solution to the current LP as input. Weights for each arc $(a, b) \in E^{prec}$ are determined by the values of the action variables in the current solution. We have the LP solution that is given in Figure 17, so in this example we have $w_{A1,A3} = w_{A3,A4} = 0.8$, $w_{A2,A3} = 1$, and $w_{A4,A1} = 0.6$. The length of the shortest path from $A1$ to $A4$ using weights $\bar{w}_{a,b}$ is equal to 0.4 $(0.2 + 0.2)$. Hence, we have $d_{A1,A4} = 0.4$ and $w_{A4,A1} = 0.6$. Since $d_{A1,A4} - w_{A4,A1} < 0$, we have a violated cycle (i.e. violated ordering implication) that includes all actions that are on the shortest path from $A1$ to $A4$ (i.e. $A1$, $A3$, and $A4$, which can be retrieved by the shortest path algorithm). This generates the following ordering implication constraint $x_1^{A1} + x_1^{A3} + x_1^{A4} \leq 2$, which will be added to the current LP. Note that this ordering constraint is violated by the current LP solution, as $x_1^{A1} + x_1^{A3} + x_1^{A4} = 0.8 + 1 + 0.8 = 2.6$. Once the constraint is added to the LP, the next solution will select a set of actions that does not violate the newly added cut. This procedure continues until no cuts are violated and the solution is integer.

### 4.4 Experimental Results

The described formulations are based on two key ideas. The first idea is to decompose the planning problem into several loosely coupled components and represent these components by an appropriately defined network. The second idea is to reduce the number of plan periods

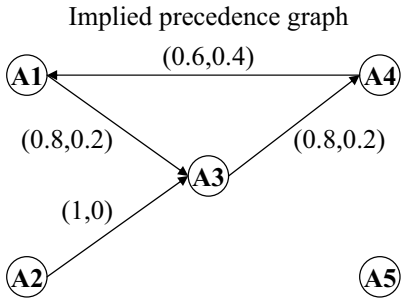


Fig. 18. Implied precedence graph for this example, where the labels show $(w_{a,b}, \bar{w}_{a,b})$.

by adopting different notions of parallelism and use a branch-and-cut algorithm to dynamically add constraints to the formulation in order to deal with the exponentially many action ordering constraints in an efficient manner.

To evaluate the tradeoffs of allowing more flexible network representations we compare the performance of the one state change (1SC) formulation, the generalized one state change formulation (G1SC), the generalized two state change (G2SC) formulation, and the state change path (PathSC) formulation. For easy reference, an overview of these formulations is given in Figure 19.

In our experiments we focus on finding feasible solutions. Note, however, that our formulations can be used to do bounded length optimal planning. That is, given a plan period (i.e. a bounded length), find the optimal solution.

### 4.4.1 Experimental Setup

To compare and analyze our formulations we use the STRIPS domains from the second and third international planning competitions (IPC2 and IPC3 respectively). That is, Blocksworld, Logistics, Miconic, Freecell from IPC2 and Depots, Driverlog, Zenotravel, Rovers, Satellite, and Freecell from IPC3. We do not compare our formulations on the STRIPS domains from IPC4 and IPC5 mainly because of a peripheral limitation of the current implementation of the G2SC and PathSC formulations. In particular, the G2SC formulation cannot handle actions that change a

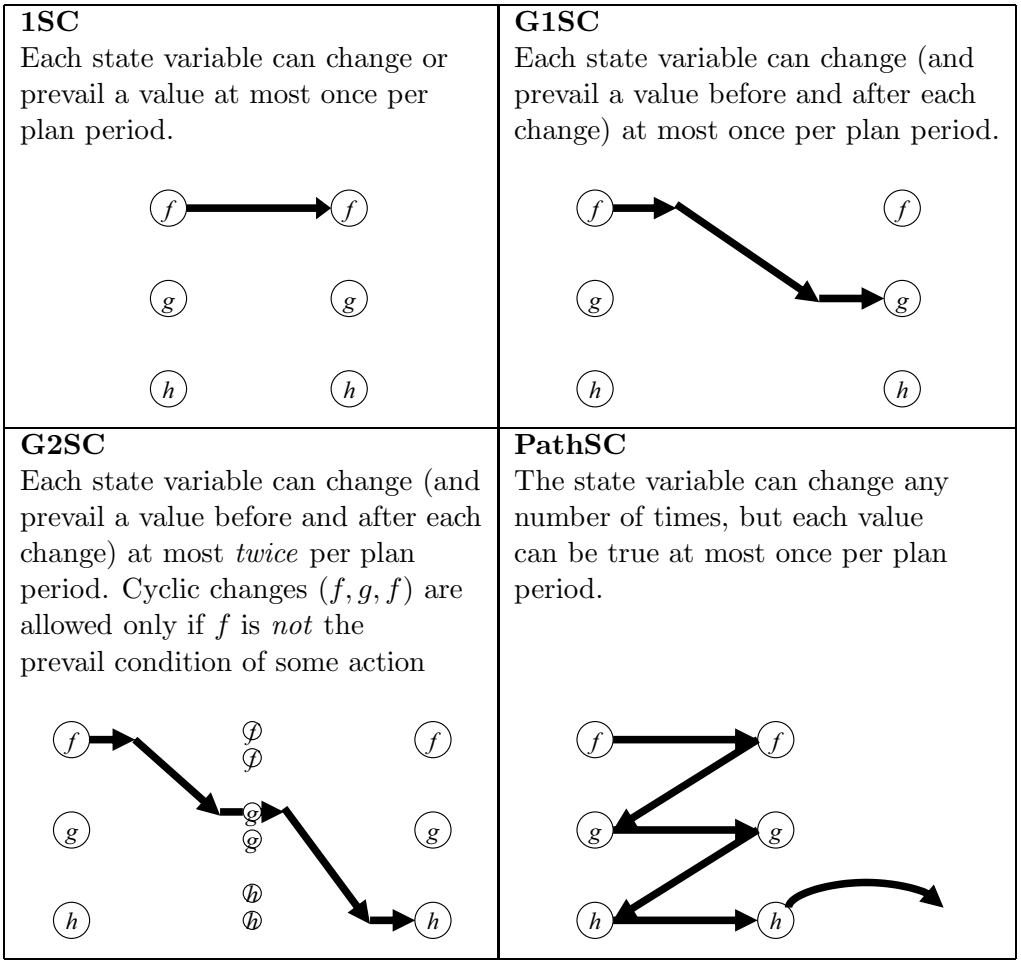


Fig. 19. Overview of the 1SC, G1SC, G2SC, and PathSC formulations.

state variable from an undefined value to a defined value, and the PathSC formulation cannot handle such actions if the domain size of the state variable is larger than two. Because of these limitations we could not test the G2SC formulation on the Miconic, Satellite and Rovers domains, and we could not test the PathSC formulation on the Satellite domain.

In order to setup our formulations we translate a STRIPS planning problem into a multi-valued state description using the translator of the Fast Downward planner [36]. Each formulation uses its own network representation and starts by setting the number of plan periods $T$ equal to one. We try to solve this initial formulation and if no plan is found, $T$ is increased by one, and then try to solve this new formulation. Hence, the IP formulation is solved repeatedly until the first feasible plan is found or a 30 minute time limit (the same time limit that is used in the

international planning competitions) is reached. We use CPLEX 10.0 [41], a commercial LP/IP solver, for solving the IP formulations on a 2.67GHz Linux machine with 1GB of memory.

We set up our experiments as follows. First, in Section 4.4.2 we provide a brief overview of our main results by looking at aggregated results from IPC2 and IPC3. Second, in Section 4.4.3, we give a more detailed analysis on our loosely coupled encodings for planning and focus on the tradeoffs of reducing the number of plan periods to solve a planning problem versus the increased difficulty in merging the solutions to the different components. Third, in Section 4.4.4 we briefly touch upon how different state variable representations of the same planning problem can influence performance.

### 4.4.2 Results Overview

In this general overview we compare our formulations to the following planning systems: Optiplan [80], SATPLAN04 [46], SATPLAN06 [49], and Satplanner [66][1].

Optiplan is an integer programming based planner that participated in the optimal track of the fourth international planning competition[2]. Like our formulations, Optiplan models state transitions but it does not use a factored representation of the planning domain. In particular, Optiplan represents state transitions in the atoms of the planning domain, whereas our formu-

[1]We note that that SATPLAN04, SATPLAN06, Optiplan, and the 1SC formulation are "step-optimal" while the G1SC, G2SC, and PathSC formulations are not. There is, however, considerable controversy in the planning community as to whether the step-optimality guaranteed by Graphplan-style planners has any connection to plan quality metrics that users would be interested in. We refer the reader to Kambhampati [44] for a longer discussion of this issue both by us and several prominent researchers in the planning community. Given this background, we believe it is quite reasonable to compare our formulations to step-optimal approaches, especially since our main aim here is to show that IP formulations have come a long way and that they can be made competitive with respect to SAT-based encodings. This in turn makes it worthwhile to consider exploiting other features of IP formulations, such as their amenability to a variety of optimization objectives as we have done in our recent work [78].

[2]A list of participating planners and their results is available at http://ipc04.icaps-conference.org/

lations use multi-valued state variables. Apart from this, Optiplan is very similar to the 1SC formulation as they both adopt the Graphplan-style parallelism.

SATPLAN04, SATPLAN06, and Satplanner are satisfiability based planners. SATPLAN04 and SATPLAN06 are versions of the well known system SATPLAN [47], which has a long track record in the international planning competitions. Satplanner has not received that much attention, but is among the state-of-the-art in planning as satisfiability. Like our formulations Satplanner generalizes the Graphplan-style parallelism to improve planning efficiency.

The main results are summarized by Figure 20. It displays aggregate results from IPC2 and IPC3, where the number of instances solved (y-axis) is drawn as a function of log time (x-axis). We must note that the graph with the IPC2 results favors the PathSC formulation over all other planners. However, as we will see in Section 4.4.3, this is mainly a reflection of its exceptional performance in the Miconic domain rather than its overall performance in IPC2. Morever, the graph with the IPC3 results does not include the Satellite domain. We decided to remove this domain, because we could not run it on the public versions of SATPLAN04 and SATPLAN06 nor the G2SC and PathSC formulations. While the results in Figure 20 provide a rather coarse overview, they sum up the following main findings.

- *Factored planning using loosely coupled formulations helps improve performance.* Note that all integer programming formulations that use factored representations, that is 1SC, G1SC, G2SC, and PathSC (except the G2SC formulation which could not be tested on all domains), are able to solve more problem instances in a given amount of time than Optiplan, which does not use a factored representation. Especially, the difference between 1SC and Optiplan is remarkable as they both adopt the Graphplan-style parallelism. In Section 4.4.3, however, we will see that Optiplan does perform well in domains that are either serial by design or have a significant serial component.

- *Decreasing the encoding size by relaxing the Graphplan-style parallelism helps improve performance.* This is not too surprising, Dimopoulos et al. [23] already note that a reduction in the number of plan periods helps improve planning performance. However, this does not always hold because of the tradeoff between reducing the number plan periods versus the increased difficulty in merging the solutions to the different components. In Section 4.4.3 we will see that different relaxations of Graphplan-style parallelism lead to different results. For example, the PathSC formulation shows superior performance in Miconic and Driverlog, but does poorly in Blocksworld, Freecell, and Zenotravel. Likewise, the G2SC formulation does well in Freecell, but it does not seem to excel in any other domain.

- *Planning as integer programming shows new potential.* The conventional wisdom in the planning community has been that planning as integer programming cannot compete with planning as satisfiability or constraint satisfaction. In Figure 20, however, we see that the 1SC, G1SC and PathSC formulation can compete quite well with SATPLAN04. While SATPLAN04 is not state-of-the-art in planning as satisfiability anymore, it does show that planning as integer programming has come a long way. The fact that IP is competitive allows us to exploit its other virtues such as optimization [24, 5, 78].

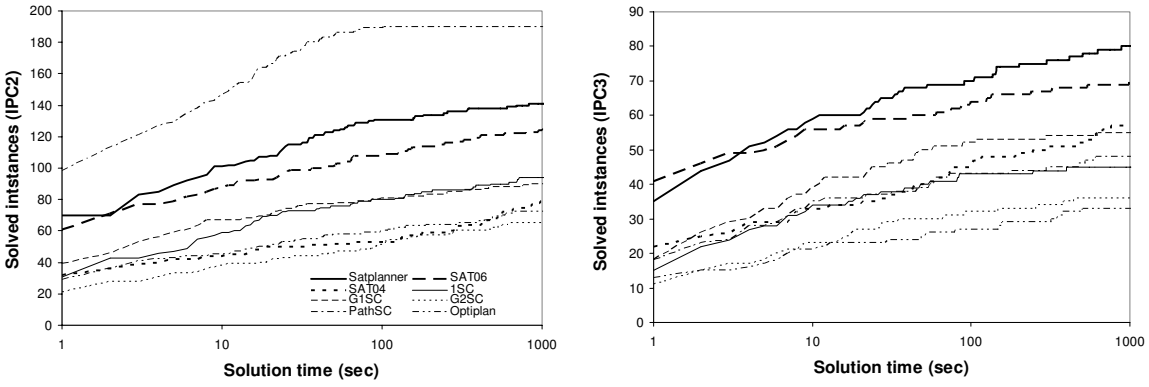


Fig. 20. Aggregate results of the second and third international planning competitions.

### 4.4.3 Comparing Loosely Coupled Formulations for Planning

In this section we compare our IP formulations and try to evaluate the benefits of allowing more flexible network representations. Specifically, we are interested in the effects of reducing the number of plan periods required to solve the planning problem versus dealing with merging solutions to the different components. Reducing the number of plan periods can lead to smaller encodings, which can lead to improved performance. However, it also makes the merging of the loosely coupled components harder, which could worsen performance.

In order to compare our formulations we will analyze the following two things. First, we examine the performance of our formulations by comparing their solution times on problem instances from IPC2 and IPC3. In this comparison we will include results from Optiplan as it gives us an idea of the differences between a formulation based on Graphplan and formulations based on loosely coupled components. Moreover, it will also show us the improvements in IP based approaches for planning. Second, we examine the number of plan periods that each formulation needs to solve each problem instance. Also, we will look at the tradeoffs between reducing the number of plan periods and the increased difficulty in merging the solutions of the loosely coupled components. In this comparison we will include results from Satplanner because, just like our formulations, it adopts a generalized notion of the Graphplan-style parallelism.

We use the following figures and table. Figure 21 shows the total solution time (y-axis) needed to solve the problem instances (x-axis), Figure 22 shows the number of plan periods (y-axis) to solve the problem instances (x-axis), and Table IV shows the number of ordering constraints that were added during the solution process, which can be seen as an indicator of the merging effort. The selected problem instances in Table IV represent the five largest instances that could be solved by all of our formulations (in some domains, however, not all formulations could solve at least five problem instances).

The label *GPsteps* in Figure 22 represents the number of plan steps that SATPLAN06, a state-of-the-art Graphplan-based planner, would use. In the Satellite domain, however, we use the results from the 1SC formulation as we were unable to run the public version of SATPLAN06 in this domain. We like to point out that Figure 22 is not intended to favor one formulation over the other, it simply shows that it is possible to generate encodings for automated planning that use drastically fewer plan periods than Graphplan-based encodings.

**4.4.3.1 Results: Planning Performance:** Blocksworld is the only domain in which Optiplan solves more problems than our formulations. In Zenotravel and Satellite, Optiplan is generally outperformed with respect to solution time, and in Rovers and Freecell, Optiplan is generally outperformed with respect to the number of problems solved. As for the other IP formulations, the G1SC provides the overall best performance and the performance of the PathSC formulation is somewhat irregular. For example, in Miconic, Driverlog and Rovers the PathSC formulation does very well, but in Depots and Freecell it does rather poorly.

In the Logistics domain all formulations that generalize the Graphplan-style parallelism (i.e. G1SC, G2SC, and PathSC) scale better than the 1SC formulation and Optiplan, which adopt the Graphplan-style parallelism. Among G1SC, G2SC, and PathSC formulations there is no clear best performer, but in the larger Logistics problems the G1SC formulation seems to do slightly better. The Logistics domain provides a great example of the tradeoff between flexibility and merging. By allowing more actions to be executed in each plan period, generally shorter plans (in terms of number of plan periods) are needed to solve the planning problem (see Figure 22), but at the same time merging the solutions to the individual components will be harder as one has to respect more ordering constraints (see Table IV).

**Optiplan versus 1SC.** If we compare the 1SC formulation with Optiplan, we note that Optiplan fares well in domains that are either serial by design (Blocksworld) or in domains that have a significant serial aspect (Depots). We think that Optiplan's advantage over the 1SC

formulation in these domains is due to the following two possibilities. First, our intuition is that in serial domains the reachability and relevance analysis in Graphplan is stronger in detecting infeasible action choices (due to mutex propagation) than the network flow restrictions in the 1SC formulation. Second, it appears that the state variables in these domains are more tightly coupled (i.e. the actions have more effects, thus transitions in one state variable are coupled with several transitions in other state variables) than in most other domains, which may negatively affect the performance of the 1SC formulation.

**1SC versus G1SC.** When comparing the 1SC formulation with the G1SC formulation we can see that in all domains, except in Blocksworld and Miconic, the G1SC formulation solves at least as many problems as the 1SC formulation. The results in Blocksworld are not too surprising and can be attributed to semantics of this domain. Each action in Blocksworld requires one state change in the state variable of the arm (*stack* and *putdown* change the status of the arm to $arm - empty$, and *unstack* and *pickup* change the status of the arm to $holding - x$ where $x$ is the block being lifted). Since, the 1SC and the G1SC formulations both allow at most one state change in each state variable, there is no possibility for the G1SC formulation to allow more than one action to be executed in the same plan period. Given this, one may think that the 1SC and G1SC formulations should solve at least the same number of problems, but in this case the prevail constraints (5) of the 1SC formulation are stronger than the prevail constraints (6) of the G1SC formulation. That is, the right-hand side of (6) subsumes (i.e. allows for a larger feasible region in the LP relaxation) than the right-hand side of (5). In Figure 21 we can see this slight advantage of 1SC over G1SC in the Blocksworld domain.

The results in the Miconic domain are, on the other hand, not very intuitive. We would have expected the G1SC formulation to solve at least as many problems as the 1SC formulation, but this did not turn out to be the case. One thing we noticed is that in this domain the G1SC

formulation required a lot more time to determine that there is no plan for a given number of plan periods.

**G1SC versus G2SC and PathSC.** Table IV only shows the five largest problems in each domain that were solved by the formulations, yet it is representative for the whole set of problems. The table indicates that when Graphplan-style parallelism is generalized, more ordering constraints are needed to ensure a feasible plan. On average, the G2SC formulation includes more ordering constraints than the G1SC formulation, and the PathSC formulation in its turn includes more ordering constraints than the G2SC formulation. The performance of these formulations as shown by Figure 21 varies per planning domain. The PathSC formulation does well in Miconic and Driverlog, the G2SC formulation does well in Freecell, and the G1SC does well in Zenotravel. Because of these performance differences, we believe that the ideal amount of flexibility in the generalization of Graphplan-style parallelism is different for each planning domain.

**4.4.3.2 Results: Number of Plan Periods:** In Figure 22, we see that in all domains the flexible network representation of the G1SC formulation is slightly more general than the 1-linearization semantics that is used by Satplanner. That is, the number of plan periods required by the G1SC formulation is always less than or equal to the number of plan periods used by Satplanner. Moreover, the flexible network representation of the G2SC and PathSC formulations are both more general than the one used by the G1SC formulation. One may think that the network representation of the PathSC formulation should provide the most general interpretation of action parallelism, but since the G2SC network representations allows some values to change back to their original value in the same plan period this is not always the case.

In the domains of Logistics, Freecell, Miconic, and Driverlog, the PathSC never required more than two plan periods to solve the problem instances. For the Miconic domain this is very easy to understand. In Miconic there is an elevator that needs to bring travelers from one floor to another. The state variables representation of this domain has one state variable for the elevator

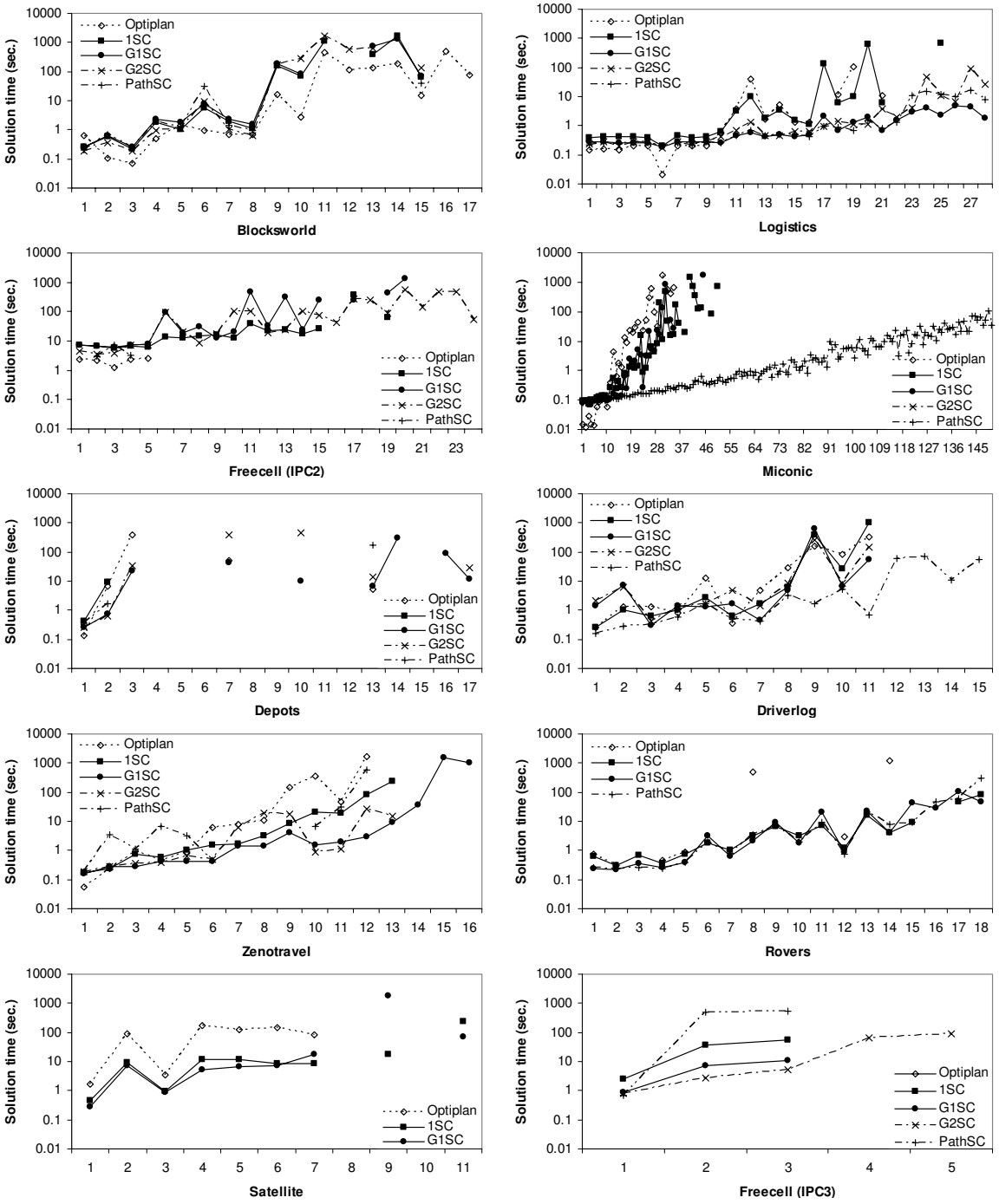


Fig. 21. Solution times in the planning domains of the second and third international planning competition.

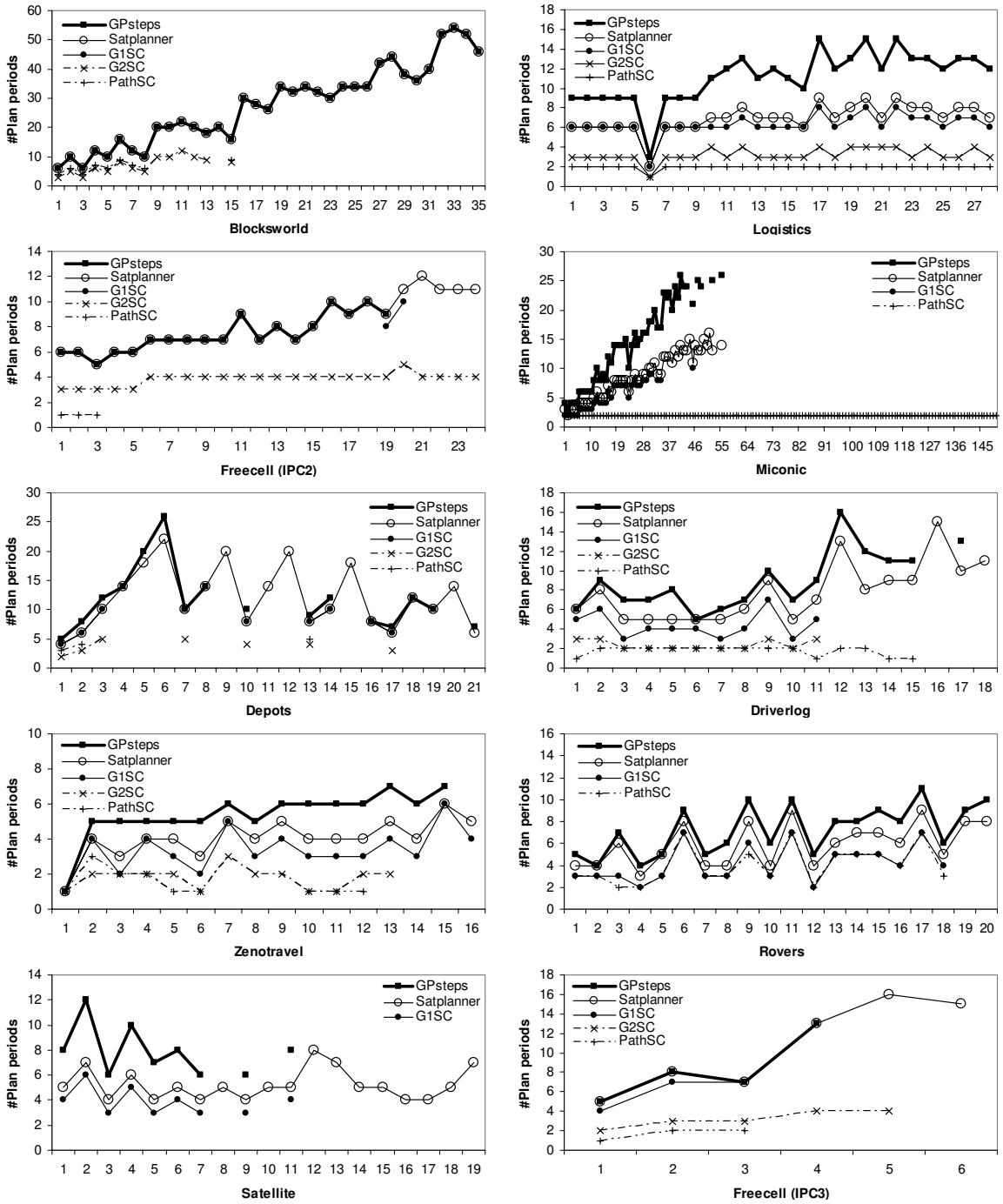


Fig. 22. Number of plan periods required for each formulation to solve the planning problems in the second and third international planning competition.

TABLE IV

NUMBER OF ORDERING CONSTRAINTS, OR CUTS, THAT WERE ADDED DYNAMICALLY THROUGH THE SOLUTION PROCESS TO PROBLEMS IN IPC2 (LEFT) AND IPC3 (RIGHT). A DASH '-' INDICATES THAT THE IP FORMULATION COULD NOT BE TESTED ON THE DOMAIN AND A STAR '*' INDICATES THAT THE FORMULATION COULD NOT SOLVE THE PROBLEM INSTANCE WITHIN 30 MINUTES.

| Problem | G1SC | G2SC | PathSC |
|---|---|---|---|
| Blocksworld | | | |
| 5-1 | 0 | 0 | 16 |
| 5-2 | 0 | 0 | 62 |
| 6-0 | 0 | 0 | 5 |
| 6-1 | 0 | 0 | 5 |
| 8-2 | 0 | 0 | 62 |
| Logistics | | | |
| 10-1 | 0 | 7 | 11 |
| 11-0 | 0 | 0 | 10 |
| 11-1 | 0 | 0 | 49 |
| 12-0 | 0 | 16 | 9 |
| 14-0 | 0 | 7 | 110 |
| Freecell | | | |
| 2-1 | 0 | 0 | 0 |
| 2-2 | 0 | 0 | 0 |
| 2-3 | 0 | 0 | 84 |
| 2-4 | 0 | 2 | 0 |
| 5-5 | 0 | 3 | * |
| Miconic | | | |
| 6-4 | 0 | - | 0 |
| 7-0 | 0 | - | 0 |
| 7-2 | 0 | - | 2 |
| 7-3 | 0 | - | 4 |
| 9-4 | 0 | - | 5 |

| Problem | G1SC | G2SC | PathSC |
|---|---|---|---|
| Depots | | | |
| 1 | 0 | 0 | 7 |
| 2 | 0 | 0 | 2 |
| 10 | 0 | 2 | * |
| 13 | 0 | 0 | 30 |
| 17 | 0 | 0 | * |
| Driverlog | | | |
| 7 | 1 | 16 | 2 |
| 8 | 32 | 62 | 108 |
| 9 | 58 | 80 | 32 |
| 10 | 5 | 5 | 69 |
| 11 | 6 | 30 | 11 |
| Zenotravel | | | |
| 5 | 0 | 4 | 485 |
| 6 | 0 | 5 | 6 |
| 10 | 0 | 0 | 214 |
| 11 | 0 | 0 | 586 |
| 12 | 0 | 60 | 6259 |
| Rovers | | | |
| 14 | 0 | - | 13 |
| 15 | 12 | - | 11 |
| 16 | 1 | - | 92 |
| 17 | 1 | - | 4 |
| 18 | 1 | - | 192 |
| Satellite | | | |
| 5 | 2 | - | - |
| 6 | 6 | - | - |
| 7 | 8 | - | - |
| 9 | 14 | - | - |
| 11 | 0 | - | - |
| Freecell | | | |
| 1 | 0 | 0 | 3 |
| 2 | 0 | 0 | 2480 |
| 3 | 0 | 1 | 1989 |
| 4 | * | 1 | * |
| 5 | * | 0 | * |

and two for each traveler (one to represent whether the traveler has boarded the elevator and one to represent whether the traveler has been serviced). Clearly, one can devise a plan such that each value of the state variable is visited at most twice. The elevator simply could visit all floors and pickup all the travelers, and then visit all floors again to bring them to their destination floor.

### 4.4.4 Comparing Different State Variable Representations

An interesting question is to find out whether different state variable representations lead to different performance benefits. In our loosely coupled formulations we have components that represent multi-valued state variables. However, the idea of modeling value transitions as flows in an appropriately defined network can be applied to any binary or multi-valued state variable representation. In this section we concentrate on the efficiency tradeoffs between binary and multi-valued state descriptions. As there are generally fewer multi-valued state variables than binary atoms needed to describe a planning problem, we can expect our formulations to be more compact when they use a multi-valued state description. For this comparison we only concentrate on the G1SC formulation as it showed the overall best performance among our formulations. In our recent work [83] we analyze different state variable representations in more detail.

Table V compares the encoding size for the G1SC formulation on a set of problems using either a binary or multi-valued state description. The table clearly shows that the encoding size becomes significantly smaller (both before and after CPLEX presolve) when a multi-valued state description is used. The encoding size before presolve gives an idea of the impact of using a more compact multi-valued state description, whereas the encoding size after presolve shows how much preprocessing can be done by removing redundancies and substituting out variables.

Figure 23 shows the total solution time (y-axis) needed to solve the problem instances (x-axis). Since we did not make any changes to the G1SC formulation, the performance differences are the

result of using different state descriptions. In several domains the multi-valued state description shows a clear advantage over the binary state description when using the G1SC formulation, but there are also domains in which the multi-valued state description does not provide too much of an advantage. In general, however, the G1SC formulation using a multi-valued state description leads to the same or better performance than using a binary state description. In all our tests, we encountered only one problem instance (Rovers pfile10) in which the binary state description notably outperformed the multi-valued state description.

### 4.5 Related Work

There are only few integer programming-based planning systems. Bylander [15] considers an IP formulation based on converting the propositional representation given by Satplan [47] to an IP formulation with variables that take the value 1 if a certain proposition is true, and 0 otherwise. The LP relaxation of this formulation is used as a heuristic for partial order planning, but tends to be rather time-consuming. A different IP formulation is given by Vossen et al. [87]. They consider an IP formulation in which the original propositional variables are replaced by state change variables. State change variables take the value 1 if a certain proposition is added, deleted, or persisted, and 0 otherwise. Vossen et al. show that the formulation based on state change variables outperforms a formulation based on converting the propositional representation. Van den Briel and Kambhampati [80] extend the work by Vossen et al. by incorporating some of the improvements described by Dimopoulos [21]. Other integer programming approaches for planning rely on domain-specific knowledge [8, 9] or explore non-classical planning problems [22, 50].

In our formulations we model the transitions of each state variable as a separate flow problem, with the individual problems being connected through action constraints. The Graphplan planner introduced the idea of viewing planning as a network flow problem, but it did not decompose the

TABLE V

FORMULATION SIZE FOR BINARY AND MULTI-VALUED STATE DESCRIPTION OF PROBLEM INSTANCES FROM THE IPC2 AND IPC3 IN NUMBER OF VARIABLES (#VA), NUMBER OF CONSTRAINTS (#CO), AND NUMBER OF ORDERING CONSTRAINTS, OR CUTS, (#CU) THAT WERE ADDED DYNAMICALLY THROUGH THE SOLUTION PROCESS.

| | Binary | | | | Multi | | | |
|---|---|---|---|---|---|---|---|---|
| | Before presolve | | After presolve | | Before presolve | | After presolve | |
| Problem | #va | #co | #va | #co | #va | #co | #va | #co |
| Blocksworld | | | | | | | | |
| 6-2 | 7645 | 12561 | 5784 | 9564 | 5125 | 7281 | 3716 | 5409 |
| 7-0 | 10166 | 16881 | 7384 | 12318 | 6806 | 9741 | 4761 | 6967 |
| 8-0 | 11743 | 19657 | 9947 | 16773 | 7855 | 11305 | 6438 | 9479 |
| Logistics | | | | | | | | |
| 14-1 | 16464 | 16801 | 7052 | 7386 | 10843 | 11180 | 2693 | 3007 |
| 15-0 | 16465 | 16801 | 7044 | 7385 | 10844 | 11180 | 2696 | 3009 |
| 15-1 | 14115 | 14401 | 4625 | 4935 | 9297 | 9583 | 1771 | 2133 |
| Miconic | | | | | | | | |
| 6-4 | 2220 | 3403 | 1776 | 2843 | 1905 | 3088 | 428 | 1495 |
| 7-0 | 2842 | 4474 | 2295 | 3764 | 2473 | 4105 | 503 | 1972 |
| 7-2 | 2527 | 3977 | 1999 | 3287 | 2199 | 3649 | 431 | 1719 |
| Freecell(IPC2) | | | | | | | | |
| 3-3 | 128636 | 399869 | 27928 | 79369 | 25267 | 62588 | 7123 | 15588 |
| 3-4 | 129392 | 401486 | 28234 | 79577 | 23734 | 61601 | 6346 | 15101 |
| 3-5 | 128636 | 399869 | 27947 | 79444 | 23342 | 61083 | 6237 | 14931 |
| Depots | | | | | | | | |
| 7 | 21845 | 36181 | 11572 | 23233 | 17250 | 15381 | 4122 | 5592 |
| 10 | 30436 | 50785 | 13727 | 27570 | 24120 | 21713 | 4643 | 6731 |
| 13 | 36006 | 59425 | 14729 | 29712 | 27900 | 25297 | 4372 | 6806 |
| Driverlog | | | | | | | | |
| 8 | 3431 | 3673 | 2245 | 2506 | 2595 | 2513 | 1146 | 1102 |
| 10 | 4328 | 4645 | 2159 | 2333 | 3551 | 3292 | 1409 | 1171 |
| 11 | 8457 | 9101 | 5907 | 6404 | 6997 | 6471 | 3558 | 3073 |
| Zenotravel | | | | | | | | |
| 12 | 9656 | 15589 | 4294 | 7046 | 2858 | 5821 | 1051 | 2398 |
| 13 | 13738 | 21649 | 7779 | 12449 | 4466 | 8417 | 1882 | 4174 |
| 14 | 40332 | 70021 | 17815 | 32959 | 9282 | 24121 | 3367 | 10619 |
| Rovers | | | | | | | | |
| 16 | 8631 | 8093 | 5424 | 5297 | 7367 | 6637 | 4394 | 4155 |
| 17 | 25794 | 23906 | 19549 | 18384 | 22889 | 20700 | 16652 | 15257 |
| 18 | 20895 | 20241 | 12056 | 12144 | 18351 | 17377 | 10081 | 9988 |
| Satellite | | | | | | | | |
| 6 | 4471 | 4945 | 3584 | 3774 | 4087 | 4561 | 2288 | 2478 |
| 7 | 5433 | 5833 | 4294 | 4267 | 5013 | 5413 | 2974 | 2925 |
| 11 | 16758 | 21537 | 13643 | 16713 | 15578 | 20357 | 7108 | 10118 |
| Freecell(IPC3) | | | | | | | | |
| 1 | 7332 | 19185 | 2965 | 7339 | 1624 | 3265 | 624 | 1339 |
| 2 | 28214 | 76343 | 16218 | 43427 | 4873 | 11383 | 2604 | 6416 |
| 3 | 39638 | 105995 | 19603 | 50819 | 7029 | 16003 | 3394 | 8055 |

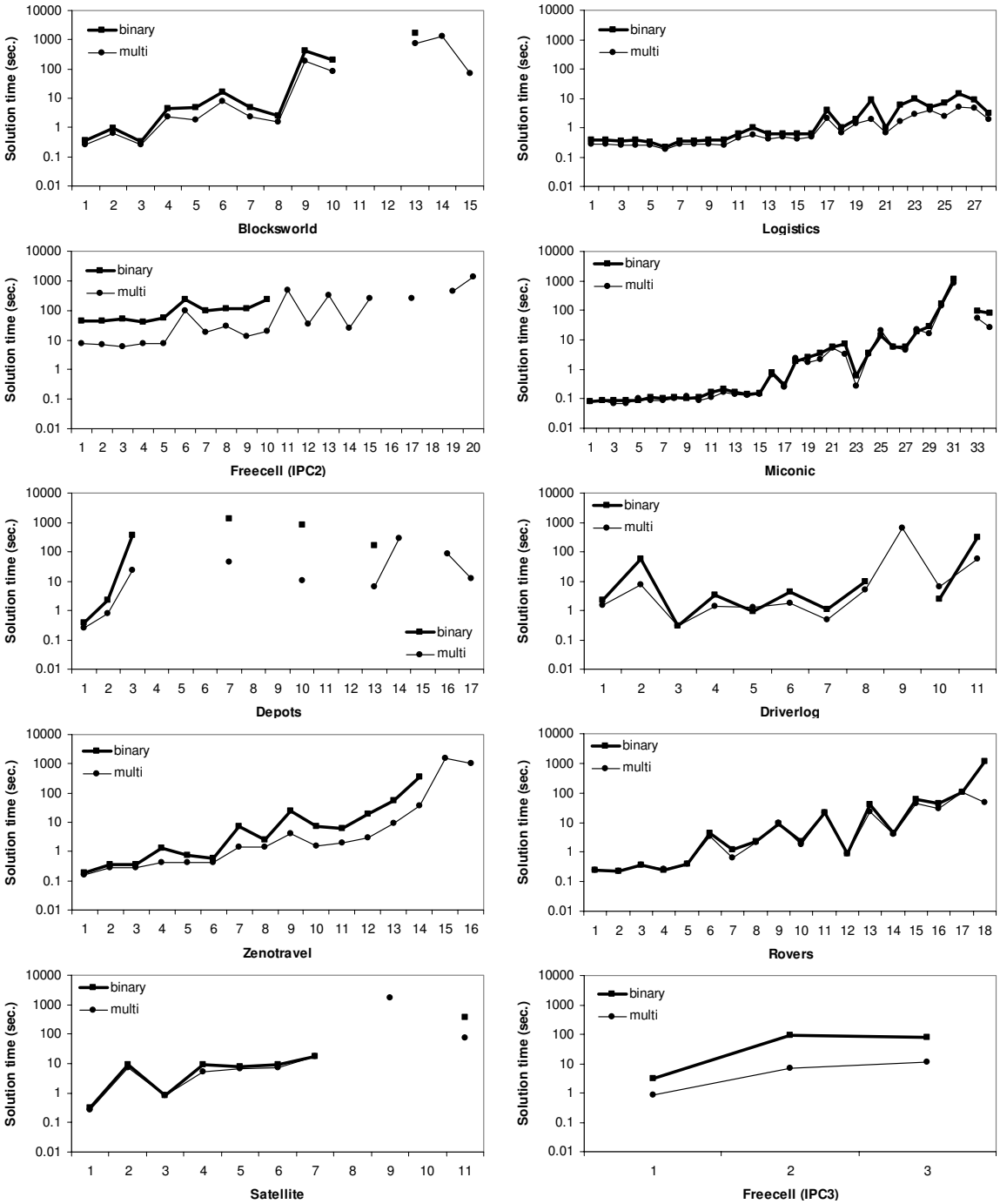


Fig. 23. Comparing binary state descriptions with multi-valued state descriptions using the G1SC formulation.

domain into several loosely coupled components. The encodings that we described are related to the loosely-coupled modular planning architecture by Srivastava, Kambhampati, and Do [74], as well as factored planning approaches by Amir and Engelhardt [2], and Brafman and Domshlak [12]. The work by Brafman and Domshlak, for example, proposes setting up a separate CSP problem for handling each factor. These individual factor plans are then combined through a global CSP. In this way, it has some similarities to our work (with our individual state variable flows corresponding to encodings for the factor plans). Although Brafman and Domshlak do not provide empirical evaluation of their factored planning framework, they do provide some analysis on when factored planning is expected to do best. It would be interesting to adapt their minimal tree-width based analysis to our scenario.

The branch-and-cut concept was introduced by Grötschel, Rünger, and Reinelt [34] and Padberg and Rinaldi [60], and has been successfully applied in the solution of many hard large-scale optimization problems [16]. In planning, approaches that use dynamic constraint generation during search include RealPlan [74] and LPSAT [88].

Relaxed definitions for Graphplan-style parallelism have been investigated by several other researchers. Dimopoulos et al. [23] were the first to point out that it is not necessary to require two actions to be independent in order to execute them in the same plan period. In their work they introduce the property of post-serializability of a set of actions. A set of actions $A' \subseteq A$ is said to be post-serializable if (1) the union of their preconditions is consistent, (2) the union of their effects is consistent, and (3) the preconditions-effects graph is acyclic. Where the preconditions-effects graph is a directed graph that contains a node for each action in $A'$, and an arc $(a, b)$ for $a, b \in A'$ if the preconditions of $a$ are inconsistent with the effects of $a$. For certain planning problems Dimopoulos et al. [23] show that their modifications reduce the number of plan periods and improve performance. Rintanen [64] provides a constraint-based implementation of their idea and shows that the improvements hold over a broad range of planning domains.

Cayrol et al. [18] introduce the notion of authorized linearizations, which implies an order for the execution of two actions. In particular, an action $a \in A$ authorizes an action $b \in A$ implies that if $a$ is executed before $b$, then the preconditions of $b$ will not be deleted by $a$ and the effects of $a$ will not be deleted by $b$. The notion of authorized linearizations is very similar to the property of post-serializability. If we were to adopt these ideas in our network-based representations it would compare to the G1SC network in which the generalized state change arcs (see Figure 11) only allows values to prevail after, but not before, each of the transition arcs.

A more recent discussion on the definitions of parallel plans is given by Rintanen, Heljanko and Niemelä [66]. Their work introduces the idea of ∃-step semantics, which says that it is not necessary that all parallel actions are non-interfering as long as they can be executed in at least one order. ∃-step semantics provide a more general interpretation of parallel plans than the notion of authorized linearizations used by LCGP [18]. The current implementation of ∃-step semantics in Satplanner is, however, somewhat restricted. While the semantics allow actions to have conflicting effects, the current implementation of Satplanner does not.

### 4.6 Conclusions

This work makes two contributions: (1) it improves state of the art in modeling planning as integer programming, and (2) it develops novel decomposition methods for solving bounded length (in terms of number of plan periods) planning problems.

We presented a series of IP formulations that represent the planning problem as a set of loosely coupled network flow problems, where each network flow problem corresponds to one of the state variables in the planning domain. We incorporated different notions of action parallelism in order to reduce the number of plan periods needed to find a plan and to improve planning efficiency. The IP formulations described in this paper have led to their successful use in solving partial satisfaction planning problems [24]. Moreover, they have initiated a new line of work in which

integer and linear programming are used in heuristic state-space search for automated planning [5, 78]. It would be interesting to see how our approach in the context of IP formulations applies to formulations based on satisfiability and constraint satisfaction.

# 5 PARTIAL SATISFACTION PLANNING

*Having a bad plan is better than having no plan at all.*

*Unknown author*

Practical planning problems often have a large number of goals, but only limited number of resources to satisfy all of them. These planning problems are called oversubscribed as there are too many goals to satisfy feasibly. Solving these oversubscribed problems poses extra challenges, in particular in the handling of plan quality in terms of action costs and goal utilities. In this chapter the problem of oversubscription, or partial satisfaction planning, is introduced in Section 5.1. Section 5.2 provides a taxonomy of partial satisfaction problems and discusses their computational complexity. Moreover, two approaches are described for the partial satisfaction problem in which the objective is to find a plan with the highest net benefit (i.e. total goal utility minus total action cost). In Section 5.3 utility dependencies are introduced and an approach is described for handling utility dependencies. Part of this chapter is based on Van den Briel et al. [84] and Do et al. [24].

## 5.1 Introduction

In classical planning the aim is to find a sequence of actions that transforms a given initial state $I$ to some goal state $G$, where $G = g_1 \wedge g_2 \wedge ... \wedge g_n$ is a conjunctive list of propositions. Plan success for these planning problems is measured in terms of whether or not all the conjuncts in $G$ are achieved. In many real world scenarios, however, the agent may only be able to partially satisfy $G$ as there may not be enough resources to satisfy all goals in $G$.

Effective handling of oversubscription or partial satisfaction planning (PSP) problems poses several challenges, including the problem of designing efficient goal selection heuristics, and an added emphasis on the need to differentiate between feasible and optimal plans. Indeed, every executable plan is feasible. For example, a trivially feasible, but likely non-optimal solution would be the "null" plan.

**Example 1** Figure 24 illustrates a small example from the rover domain [53]. In this problem, a rover needs to collect different samples. Waypoints have been identified on the surface and at each waypoint the rover may have either collect a rock sample, a soil sample, or both of them. For example, in Figure 24 $waypoint_3$ has a rock sample, while $waypoint_4$ has a soil sample. The rover needs to travel to the corresponding waypoint to collect its samples. Each travel action has a cost associated with it. For example, the cost of traveling from $waypoint_0$ to $waypoint_1$ is given by $c_{travel_{0,1}} = 10$. In addition to the $travel_{x,y}$ actions, we have two more actions *sample* and *communicate* to collect and communicate the data respectively to the lander. To simplify our problem, such actions have equal costs independently of the locations where they take place. These costs are specified by $c_{sample_{data,x}} = 5$ and $c_{communicate_{data,x,y}} = 4$. Each sample (or subgoal) has a utility attached to it. There is a utility of $u_{rock_3} = 30$ for the rock sample at $waypoint_3$, and a utility $u_{soil_4} = 20$ for a soil sample at $waypoint_4$. The goal of the rover is to find a travel plan that achieves the best cost-utility tradeoff for collecting the samples. In this example, the best plan is $P = \{travel_{0,2}$ , $sample_{rock_2,2}$, $communicate_{rock_2,2,0}$, $travel_{2,1}$, $sample_{rock_1,1}$, $communicate_{rock_1,1,0}$, $sample_{soil_1,1}$, $communicate_{soil_1,1,0}\}$ which achieves the goals $rock_1, rock_2$ and $soil_1$, and ignores the rest of the samples at $waypoint_3$ and $waypoint_4$, giving a final net benefit of 45.

Despite the ubiquity of PSP problems, surprisingly little attention has been paid to the developments of efficient approaches in solving them. Given the recent success of integer programming approaches to automated planning [85], these approaches are a good avenue to explore further for handling PSP problems. Both because of the recent improvements and the fact that PSP is a more optimization oriented problem, which integer programming is naturally equipped to handle.

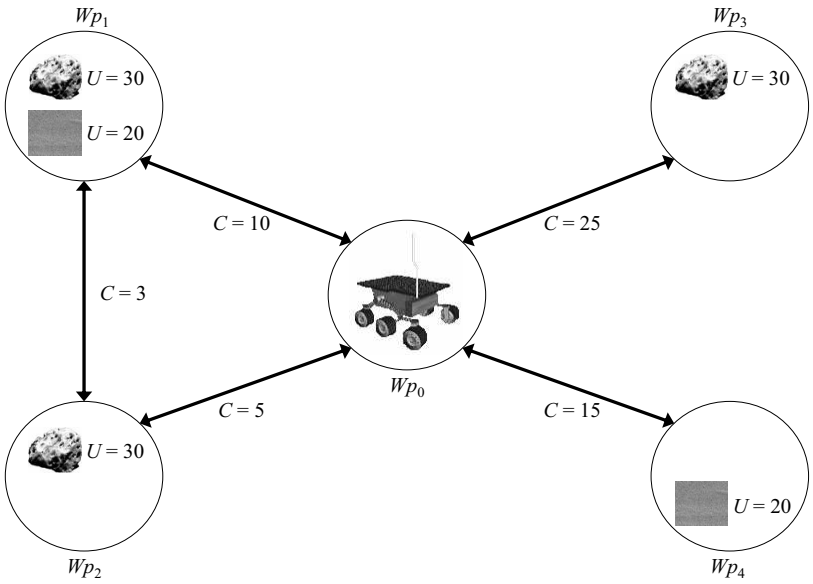


Fig. 24. Rover domain problem.

### 5.2 Problem Definition and Complexity

A classical planning $P = (F, A, I, G)$ is a tuple on the propositions ($F$), actions ($A$), initial state ($I$), and goal set ($G$). Actions are represented as three subsets of propositions; preconditions, adds, and deletes. Naturally, $I$ and $G$ are both subset of $F$. We now define the following classical planning decision problems.

**Definition 5.2.1.** PLAN EXISTENCE*: Given a planning problem $P = (F, A, I, G)$. Is there a finite executable sequence of actions $\Delta = (a_1, ..., a_n)$ that transforms $I$ into a state $S$ so that $G \subset S$?*

**Definition 5.2.2.** PLAN LENGTH*: Given a planning problem $P = (F, A, I, G)$ and a positive integer $K$. Is there a finite sequence of actions $\Delta = (a_1, ..., a_n)$ with $n \leq K$ that transforms $I$ into a state $S$ so that $G \subset S$?*

One could say that plan existence is the problem of deciding whether there exists a sequence of actions that transforms $I$ into $G$, and bounded plan existence is the decision problem that cor-

responds to the optimization problem of finding a minimum sequence of actions that transforms $I$ into $G$.

**Theorem 5.2.1.** (Bylander, 1994)

1. PLAN EXISTENCE *is PSPACE-complete.*

2. PLAN LENGTH *is PSPACE-complete.*

PLAN EXISTENCE and PLAN LENGTH both require that all goals be achieved; however, there are many practical planning problems in which it is inefficient or impossible to achieve all of the goals. We define the following decision problems to represent these partial satisfaction planning (PSP) problems.

**Definition 5.2.3.** PSP GOAL*: Given a planning problem $P = (F, A, I, G)$ and a positive integer $L$. Is there a finite sequence of actions $\Delta = (a_1, ..., a_n)$ that transforms $I$ into a state $S$ that satisfies at least $L$ of the goal conjuncts in $G$?*

**Definition 5.2.4.** PSP GOAL-LENGTH*: Given a planning problem $P = (F, A, I, G)$ and positive integers $K, L$. Is there a finite sequence of actions $\Delta = (a_1, ..., a_n)$ with $n \leq K$ that transforms $I$ into a state $S$ that satisfies at least $L$ of the goal conjuncts in $G$?*

PSP GOAL and PSP GOAL-LENGTH are generalized representations of PLAN EXISTENCE and PLAN LENGTH. Clearly, we can generalize these problems even further by introducing non-uniform *costs* for the actions and non-uniform *utilities* for the goals. We define the following decision problems.

**Definition 5.2.5.** PLAN COST*: Given a planning problem $P = (F, A, I, G)$ and, for each action, a cost, $c_a \geq 0$, and a positive integer $K$. Is there a finite sequence of actions $\Delta = (a_1, ..., a_n)$ with total cost $\sum_{a \in \Delta} c_a \leq K$ that starting from $I$ leads to a state $S$ that satisfies all the goal specifications in $G$?*

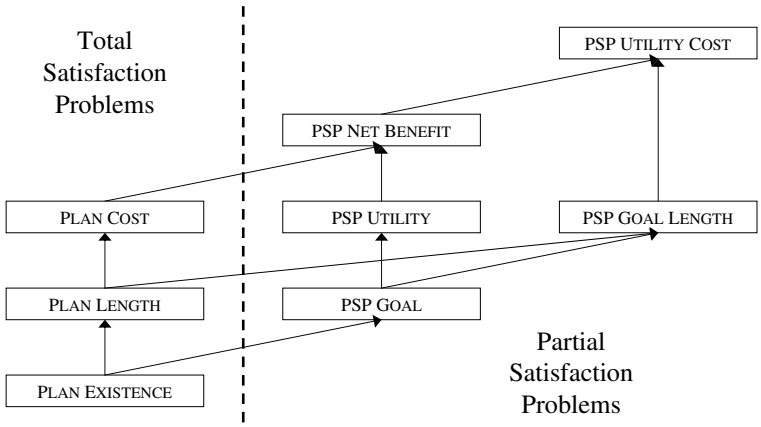


Fig. 25. Hierarchical overview of several types of complete and partial satisfaction planning problems.

**Definition 5.2.6.** PSP UTILITY*: Given a planning problem* $P = (F, A, I, G)$ *and, for each each goal specification* $f \in G$*, a utility,* $u_f \geq 0$*, and a positive integer* $L$*. Is there a finite sequence of actions* $\Delta = (a_1, ..., a_n)$ *that starting from* $I$ *leads to a state* $S$ *that satisfies* $\sum_{f \in (S \cap G)} u_f \geq L$*?*

**Definition 5.2.7.** PSP UTILITY-COST*: Given a planning problem* $P = (F, A, I, G)$ *and, for each action, a cost,* $c_a \geq 0$ *and, for each goal specification* $f \in G$*, a utility,* $u_f \geq 0$*, and positive integers* $K, L$*. Is there a finite sequence of actions* $\Delta = (a_1, ..., a_n)$ *with total cost* $\sum_{a \in \Delta} c_a \leq K$ *that starting from* $I$ *leads to a state* $S$ *that satisfies* $\sum_{f \in (S \cap G)} u_f \geq L$*?*

PLAN COST corresponds to the optimization problem of finding minimum cost plans, and PSP UTILITY corresponds to the optimization problem of finding plans that achieve maximum utility. PSP UTILITY-COST is a combination of both PLAN COST and PSP UTILITY, that is, PSP UTILITY-COST corresponds to optimizing both both utility and cost.

Figure 25 gives a taxonomic overview of both complete satisfaction planning problems (i.e. that require all goals to be satisfied) and of partial satisfaction planning problems (i.e. that do not require all goals to be satisfied). Complete satisfaction problems are identified by names starting with PLAN and partial satisfaction problems have names starting with PSP.

**Theorem 5.2.2.**

1. PLAN COST *is PSPACE-complete.*

2. PSP GOAL *is PSPACE-complete.*

3. PSP GOAL-LENGTH *is PSPACE-complete.*

4. PSP UTILITY *is PSPACE-complete.*

5. PSP UTILITY-COST *is PSPACE-complete.*

*Proof.* PSP UTILITY-COST is in PSPACE follows from Bylander [14]. PSP UTILITY-COST is PSPACE-hard because we can restrict it to PLAN EXISTENCE by allowing only instances having $\forall a \in \mathcal{A}, c_a = 1$, $\forall f \in \mathcal{F}, u_f = 1$; $K = 2^n$, and $L = |G|$. PSP UTILITY-COST subsumes every other problem, each of which subsumes PSP EXISTENCE, so every other problem is also PSPACE-complete. □

In this paper we focus on a special case of PSP UTILITY-COST, called PSP NET BENEFIT, where both costs and utilities are denominated in the same currency unit (and therefore additive). The problem of PSP NET BENEFIT corresponds to the optimization problem of finding a plan with maximum net benefit, that is, a plan that maximizes total utility minus total cost. PSP NET BENEFIT is the focus of this paper and we define it as follows.

**Definition 5.2.8.** PSP NET BENEFIT*: Given a planning problem $P = (F, A, I, G)$ and, for each action, a cost, $c_a \geq 0$ and, for each goal specification $f \in G$, a utility, $u_f \geq 0$, and a positive number $K$. Is there a finite sequence of actions $\Delta = (a_1, ..., a_n)$ that starting from $I$ leads to a state $S$ that has net benefit $\sum_{f \in (S \cap G)} u_f - \sum_{a \in \Delta} c_a \geq K$?*

### 5.2.1 An MDP Formulation for Partial Satisfaction Planning

The most obvious way to optimally solve the PSP NET BENEFIT problem is by modeling it as a fully-observable Markov Decision Process (MDP) [10]. MDPs naturally permit action cost and

goal utilities, but as we will show later, an MDP based approach for the PSP NET BENEFIT problem appears to be impractical: even the very small problems generate too many states.

State transitions in a MDP that models a PSP NET BENEFIT problem with deterministic actions have $\Pr(s_i, a, s_j) \in \{0, 1\}$. That is, the probability that state $s_j$ is reached when action $a$ is executed in state $s_i$ is equal to 1 if executing $a$ in $s_i$ is possible and results in $s_j$, otherwise 0.

The obvious way to model goals in MDP is to make goals states as sink states with reward equal to the utility of the goal. However, modeling the PSP NET BENEFIT problem as an MDP is complicated by the states that contain only a subset of the goals. For example, if the set of goals is given by $\{g_1, g_2\}$ and the only path that reaches these goals passes through a state containing only $g_1$, then we would not be able to reach $g_1$ and $g_2$ if the states containing $g_1$ are all sink states. But without sink states we could potentially collect the reward for $g_1$ more than once. To deal with this we introduce one extra state variable which we call *done*, and one extra action which we call *terminate*. The *done* variable divides the state space into two parts: a search space ($\neg$*done*) and a reward space (*done*). The terminate action causes the MDP to transition from search space to reward space with zero cost. All states in reward space are sink states and accrue some reward (possibly zero). The optimal policy in such an MDP induces a plan (from any starting state $S$) that ends with the terminate action.

We modeled a couple of smaller test problems as MDPs and solved them using SPUDD [37]. SPUDD is an MDP solver that uses value iteration on algebraic decision diagrams, which provides an efficient representation of the planning problem. Since SPUDD does not handle sink states, we introduce a variable called *dead* on top of the variable done and the action terminate. With these variables we partition the state space into: search space ($\neg$*done*, $\neg$*dead*), reward space (*done*, $\neg$*dead*), and dead space ($\neg$*done*, *dead*) or (*done*, *dead*). The transitions between these states is as follows: every action adds dead if done is true, and the terminate action adds done. This transition scheme ensures that reward space is visited only once, because applying any action

TABLE VI
SPUDD RESULTS ON THE ZENOTRAVEL DOMAIN.

| Problem | #Vars | #Actions | Time |
|---|---|---|---|
| pfile1 | 20 | 97 | 38.50 |
| pfile2 | 24 | 103 | 30.35 |
| pfile3 | 42 | 217 | >3000 |
| pfile4 | 47 | 229 | >3000 |

when done is true causes dead and no action deletes dead. Since terminate is the only action with 0 cost, the only policies with finite value induce trajectories of the form: $\Delta$, *terminate*, any action, *terminate*, etc, and optimal policies induce trajectories of the form $\Delta$, *terminate*, *terminate*, *terminate*, etc.

Table VI shows the results on the Zenotravel domain that we obtained from SPUDD. All problems were run using a discount factor of 1.0 and a tolerance factor of 0.01. We used planning graph analysis to determine the reachable fluents and actions in the problem and used them as input for Spudd. The number of variables in Table VI include the two extra variables done and dead, likewise the number of actions include the extra action terminate. As can be observed from the table, Spudd was only able to solve the two smallest problems, but solved them to optimality.

### 5.2.2 Optiplan for Partial Satisfaction Planning

Through some simple adjustments in the objective function and the constraint set Optiplan, the planner described in Chapter 3, can also be used to solve PSP NET BENEFIT problems. The adjustments involve changing the objective function to maximize the sum of goal utilities $u_f$ for each $f \in F$ minus the sum of action costs $c_a$ for each $a \in A$ as follows:

$$\sum_{f \in G, t \in \{1,\ldots,T\}} u_f(y_{f,t}^{add} + y_{f,t}^{pre-add} + y_{f,t}^{maintain}) - \sum_{a \in A, t \in \{1,\ldots,T\}} c_a x_{a,t} \quad (1)$$

The total utility at plan step $t$ is determined by the utility of the goals $u_f$ that are added, $y_{f,t}^{add}$, required but not deleted, $y_{f,t}^{pre-add}$, and propagated $y_{f,t}^{maintain}$ in the last step of the plan. The total cost is determined by the cost $c_a$ of the actions $x_{a,t}$ that are executed in the plan. Optiplan finds optimal solutions for a given plan step $t$ (i.e. bounded length plan), however, the global optimum may not be found, because Optiplan does not extend $t$ to be greater than the number of steps in the solution. There are some ways to calculate a bound on the maximum number of plan steps in the optimal plan, but these bounds are often very weak and therefore impractical to use directly in the formulation.

### 5.2.3 Experimental Results

In this section we provide a comparison between Optiplan and two heuristic approaches, AltAlt$^{ps}$ and Sapa$^{ps}$, for the PSP Net Benefit problem. AltAlt$^{ps}$ is an extension to the regression planner AltAlt [59, 70] and uses an approach to pick goals up front. Sapa$^{ps}$ is an extension to the forward search planner Sapa [**?**] and does not select goals up front, but instead uses an A$^*$ algorithm in which goals are treated as soft constraints.

Since there are no benchmark PSP Net Benefit problems, we use existing planning domains from the 2002 International Planning Competition [**?**] and modify them to support goal utilities and action costs. In particular, we use the Driverlog, Satellite and Zenotravel domains. For these domains, utilities ranging from 100 to 600 were assigned to each of the goals, and costs ranging from 10 to 800 were assigned to each of the actions by taking into account some of the characteristics of the problems. For example, in the Driverlog domain, a *drive* action is assigned a higher cost than a *load* action. Under this design, the planners achieved around 60% of the goals on average over all the domains.

All three planners were run on a 2.67Ghz CPU machine with 1.0GB RAM. The IP encodings in Optiplan were solved using ILOG CPLEX 8.1 with default settings except that the start algorithm

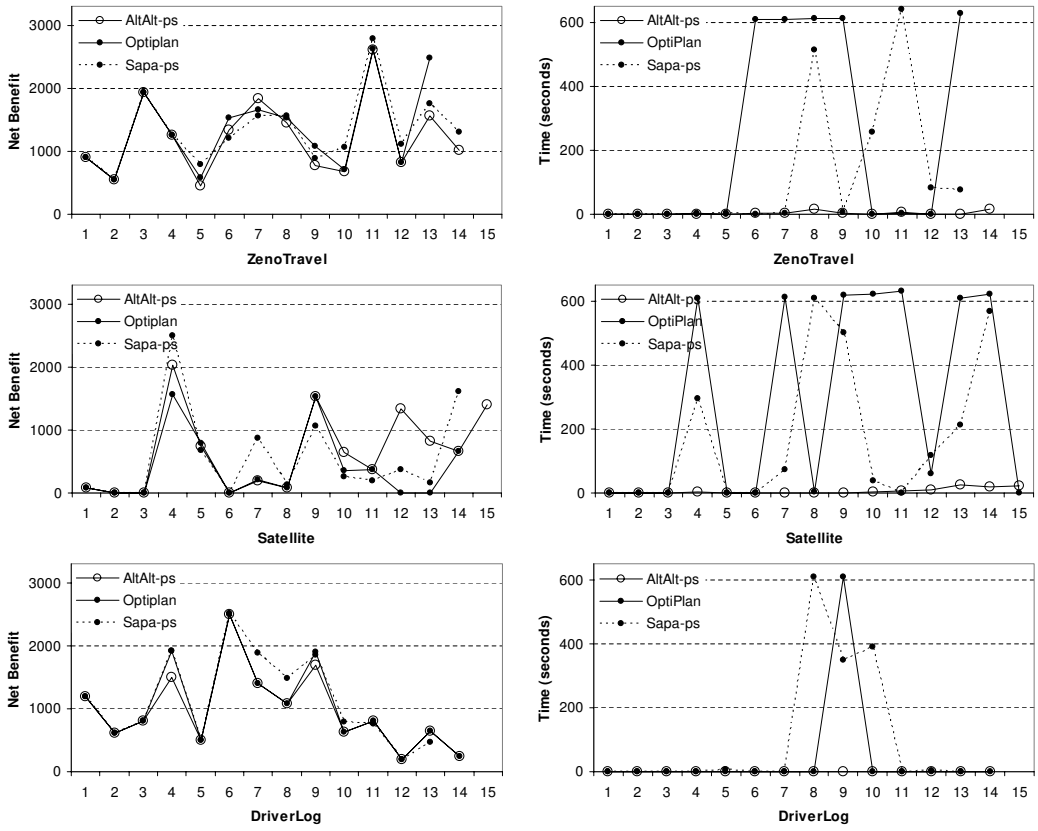


Fig. 26. Results of selected domains from the International Planning Competition.

was set to the dual problem, the variable select rule was set to pseudo reduced cost, and a time limit of 600 seconds was imposed. In case that the time limit was reached, we denoted the best feasible solution as Optiplan's solution quality. Given that Optiplan returns optimal solutions for a given number of plan steps, we provide Optiplan with a plan step by post-processing the solution of AltAlt$^{ps}$ [70].

Figure 26 shows the results of the three planners. It can be observed that in some problems instances the two heuristic planners produce better plans than Optiplan. In these problems the optimal plan requires more plan steps than the number of plan steps that is given to Optiplan. By increasing the number of plan steps in these problems, Optiplan finds plans of equal or better quality than the two heuristic planners. Also, in some problem instances Optiplan times out while the heuristic approaches return a solution in a couple of seconds. While performance is

indeed a strength of the heuristic approaches, Optiplan often does find solutions quickly, but it takes time to guarantee that there is no better plan for the given number of plan steps.

### 5.3 Partial Satisfaction with Utility Dependencies

The process of selecting goals in partial satisfaction planning is complicated by the fact that goals interact. For example, actions may help (i.e. positive interaction) or conflict (i.e. negative interaction) other actions in the process of achieving the goals. These types of interactions are referred to as cost dependencies as the cost of achieving goals individually may differ from the cost of achieving goals together. Interactions between goals can, however, also exists between their utilities. Two concrete examples of utility dependencies are mutual dependency and conditional dependency. For mutual dependency, the utility of a set of goals is different from the sum of the utility of each individual goal. For instance, the utility of having both a left and a right shoe is, in general, much higher than having either a left or a right shoe (i.e. complementing goals). On the other hand, the utility of having two cars is, in general, smaller than the sum of having one of them (i.e. substituting goals). Conditional dependency is where the utility of a goal or set of goals depend on whether or not another goals or set of goals is achieved. For instance, the utility of having a hotel reservation at a resort depends on whether or not you have already booked a trip to the resort.

In order to represent the different types of goal utility dependencies the generalized additive independence (GAI) model by Bacchus and Grove [3] can be used. The utility of the goal set $G$ can be represented by $k$ utility functions $f(G_k)$ over set of goals $G_k \subseteq G$. Now for any subset of goals $G' \subseteq G$ the utility of $G'$ is:

$$u(G') = \sum_{G_k \subseteq G'} f(G_k) \tag{2}$$

### 5.3.1 Modeling Utility Dependencies

We setup an integer programming formulation to handle utility dependencies by extending the generalized single state change (G1SC) formulation as described in Chapter 4.

To summarize, the G1SC formulation represents the planning problem as a set of loosely coupled network flow problems, where each network corresponds to one of the state variables in the planning domain. The network nodes correspond to the state variable values and the network arcs correspond to the value transitions. The planning problem is to find a path (i.e. a sequence of actions) in each network such that, when merged, the constitute a feasible plan. In the networks, nodes and arcs appear in layers, where each layer represents a plan period. The layers are used to solve the planning problem incrementally. That is, we start by performing reachability analysis to find a lower bound on the number of layers necessary to solve the problem. If no plan is found, all the networks are extended by one extra layer and the planning problem is solved again. This process is repeated until a plan is found.

In order to deal with the utility dependencies we incorporate the following extensions to the G1SC formulation:

- For each goal utility dependency function $G_k$ we add a variable $z_{G_k} \in \{0, 1\}$, where $z_{G_k} = 1$ if all goals in $G_k$ are achieved, and $z_{G_k} = 0$ otherwise.

- For each goal utility dependency function $G_k$ we add constraints to ensure that $G_k$ is satisfied if and only if all goals $g \in G_k$ are achieved, that is:

$$\sum_{f,g \in D_c : g \in G_k} y_{f,g,T} - |G_k| + 1 \leq z_{G_k} \tag{3}$$

$$z_{G_k} \leq \sum_{f \in D_c} y_{f,g,T} \quad \forall g \in D_c : g \in G_k \tag{4}$$

  Where $D_c$ is the domain of a state variable $c$, $y_{f,g,T} \in \{0, 1\}$ are variables of the IP problem that represent value changes in the state variables, and $T$ is the plan horizon. Constraint (3) represents the "if" part and ensures that if all goals in $g \in G_k$ are achieved then the

goal utility dependency function $G_k$ is satisfied (i.e. $z_{G_k} = 1$). Constraint (4) represents the "only if" and ensures that if $z_{G_k} = 1$ then all goals in $G_k$ must be achieved.

- We create an objective function to maximize the net-benefit (utility minus cost) of the plan.

$$\text{MAX} \quad \sum_{G_k} u(G_k) z_{G_k} - \sum_{a \in A, 1 \leq t \leq T} c_a x_{a,t} \tag{5}$$

Where $u(G_k)$ represents the utility of satisfying the goal utility dependency function $G_k$, and $c_a$ represents the cost of executing action $a \in A$.

The extended G1SC formulation is bounded length optimal. Global optimality cannot be guaranteed as there could still be solutions with higher net benefit that require more plan periods.

### 5.3.2 Experimental Results

We compare the extended G1SC formulation with the heuristic planners Sapa$^{ps}$ [84] and SPUDS using the $h_{relax}$ heuristic [24]. All tests use a 2.67GHz machine with 1.0GB RAM with a 600 second time limit.

The G1SC formulation finds optimal plans for a given horizon or parallel length $t$. We initialize $t$ with the horizon at which goals appear achievable by reachability analysis. If the G1SC formulation finds a solution we increment $t$ by one until the set time limit is reached. If the solution quality does not improve when $t$ increases we stop as well. Optimality cannot be guaranteed as there is always the potential that better solutions appear at longer plan horizons.

We generated a set of problems using the propositional benchmarks used in the International Planning Competition, such that utility dependencies play an important role. We varied our bounds on action cost and goal set utility values such that each problem domain focuses on a different aspect of utility dependencies. For example, in Zenotravel ending a plan with people at various location increases utility significantly. In TPP the cost of purchasing an item is similar to the utility of that item, but having more than one of the same item can significantly improve the

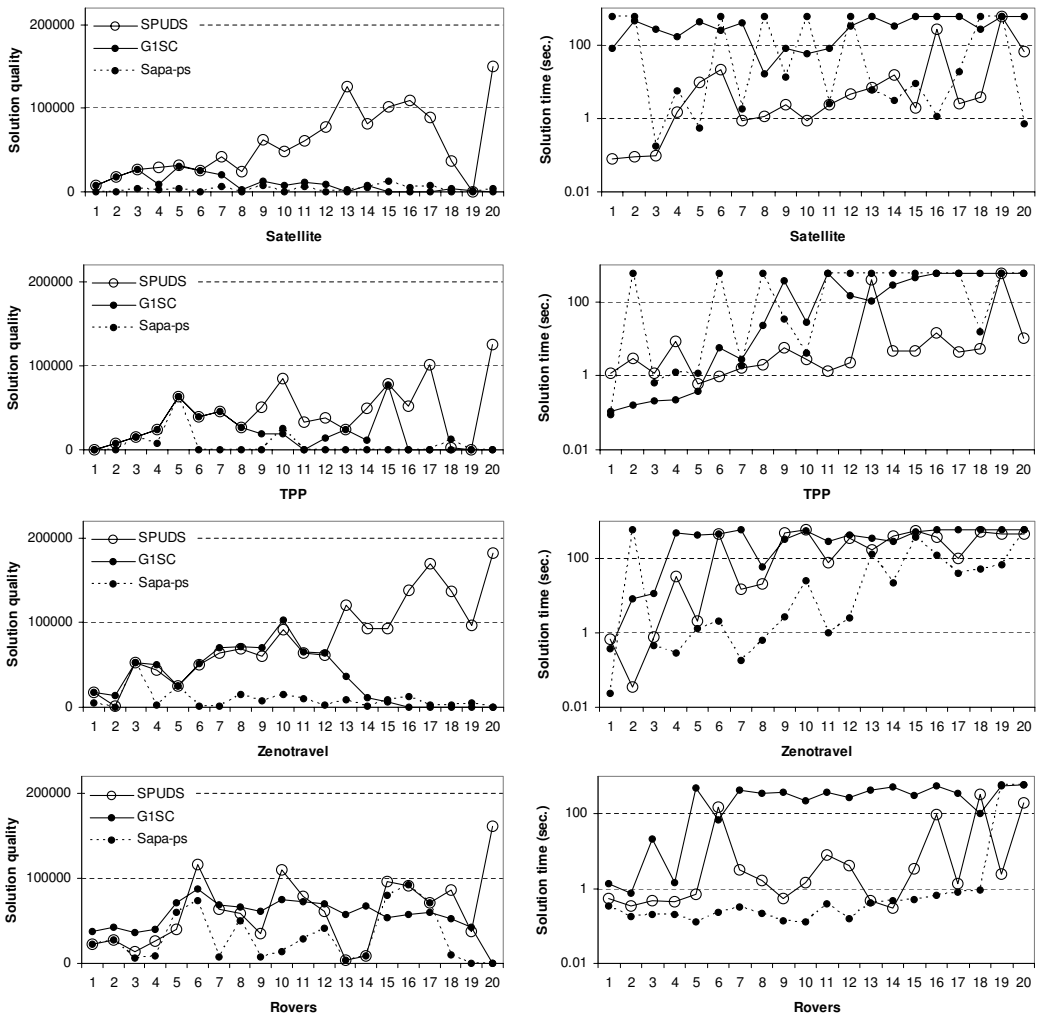


Fig. 27. Results of selected domains from the International Planning Competition.

overall utility. In Satellite, and to some extend in Rovers, certain sets of images are undesirable and lead to negative utilities. A summary of the results is given in Figure 27.

The G1SC formulation does well on several problems, especially when compared to Sapa$^{ps}$, but it turns out that as the size of the problem increases it is harder to find a good feasible solution. Also, better quality plans often require a longer planning horizons than the horizon at which the G1SC formulation stops. The horizon used by by the G1SC formulation greatly affects its performance. If the initial horizon is too short, the G1SC formulation may only achieve a small number of goals and thus loose out on some dependencies with high utilities. On the other hand,

if the initial horizon is too long, the formulation may become too large and impractical to find even a feasible, non-null plan.

# 6 DIRECTIONS FOR FURTHER RESEARCH

*People know you for what you've done, not for what you plan to do.*

*Unknown author*

Despite the obvious affinity between automated planning problems and operational research techniques, little research has been done to-date to cross-fertilize these areas. Part of the reason for this has been the focus on feasibility in automated planning. While this was understandable early on (given the computational complexity of the problem), as automated planning technology is used in real world applications, handling resources and dealing with quality considerations are increasingly at the center stage.

This chapter focusses on three directions for further research. Section 6.1 is based on Van den Briel, Kambhampati, and Vossen [81] and discusses a potential approach for planning with numeric variables. Section 6.2 is based on Van den Briel, Kambhampati, and Vossen [82] and describes a way to incorporate user preferences. Section 6.3 is based on notes and discussions with Kambhampati, Vossen, and Fowler and describes an idea for finding provably optimal plans.

## 6.1 Planning with Numeric Variables by Integer Programming

One of the most compelling reason for using integer programming (IP) and mixed integer programming (MIP) techniques in planning is when the planning problem contains numerical state variables. Numerical state variables appear in many practical planning domains and are often accompanied by linear numerical constraints and optimization criteria, which are naturally supported by the IP framework. Traditionally, IP has been used to tackle hard combinatorial optimization problems that arise in the field of operation research. However, recent work has shown that IP techniques also show great potential in their ability to solve classical AI planning problems and can compete with the most efficient SAT-based encodings [85].

### 6.1.1 Numerical State Variables

We often refer to numerical state variables as *resources*. Heuristics for planning with resources have been studied by several different works [26, 35, 38, 63]. Studies on IP formulations for resource planning, however, are not as plentiful. Wolfman and Weld [88] use LP formulations in combination with a satisfiability-based planner to solve resource planning problems, and Kautz and Walser [50] use IP formulations for resource planning problems that incorporate action costs and complex objectives.

In order to reason about resources, actions are extended to include resource preconditions and effects. Koehler [51] provides a general framework in which a resource precondition is a simple linear inequality that must hold in each state where the action is applicable, and the action effects are to produce (increase), consume (decrease), or provide (assign) the value of a resource. A resource is called *reusable* if it can only be borrowed, that is, it cannot be consumed or produced by any action. In all other cases a resource is called *consumable*. A reusable resource is *sharable* if it can be borrowed by more than one action at the same time, otherwise it is non-sharable.

The international planning competition (IPC) 2002 introduced several planning domains with resources. The language that was used in this competition, PDDL2.1 [32], incorporates the possibility to define numerical constraints and effects on numerical state variables. Table VII gives all the numeric domains of this competition and lists all the resource variables. If there exists an action in the domain that has an effect on a resource variable, then that resource variable is listed in the corresponding action effect column. In addition, a type and bounds (where $C$ is some constant) on the resource are given.

We say a resource is *monotonic* if it can only be produced (monotonic$^+$), or if it can only be consumed (monotonic$^-$). We say a resource is *nonmonotonic* if it can both be produced and consumed (nonmonotonic), and if it can be provided (nonmonotonic$^=$). These resource types are used to categorize the resource constraints in our IP formulations.

TABLE VII
THE NUMERIC DOMAINS OF THE AIPS-2002 PLANNING COMPETITION.

| **Domain** | **Increase** | **Decrease** | **Assign** | **Type** | **Bounds** |
|---|---|---|---|---|---|
| Depots | (current-load ?z) | (current-load ?z) | | nonmonotonic | $[0, C]$ |
| | (fuel-cost) | | | monotonic$^+$ | $[0, \infty)$ |
| Driverlog | (driven) | | | monotonic$^+$ | $[0, \infty)$ |
| | (walked) | | | monotonic$^+$ | $[0, \infty)$ |
| Rovers | (energy ?x) | (energy ?x) | | nonmonotonic | $[0, C)$ |
| | (recharges) | | | monotonic$^+$ | $[0, \infty)$ |
| Satellite | (fuel-used) | | | monotonic$^+$ | $[0, \infty)$ |
| | (data-stored) | | | monotonic$^+$ | $[0, \infty)$ |
| | | (fuel ?s) | | monotonic$^-$ | $[0, C]$ |
| | | (data-capacity ?s) | | monotonic$^-$ | $[0, C]$ |
| Settlers | (available ?r ?v) | (available ?r ?v) | (available ?r ?v) | nonmonotonic$^=$ | $[0, \infty)$ |
| | (available ?r ?p) | (available ?r ?p) | | nonmonotonic | $[0, \infty)$ |
| | (space-in ?v) | (space-in ?v) | | nonmonotonic | $[0, \infty)$ |
| | (resource-use) | | | monotonic$^+$ | $[0, \infty)$ |
| | (labor) | | | monotonic$^+$ | $[0, \infty)$ |
| | (pollution) | | | monotonic$^+$ | $[0, \infty)$ |
| UMT2 | (weight-load-v ?v) | (weight-load-v ?v) | | nonmonotonic | $[0, \infty)$ |
| | (volume-load-v ?v) | (volume-load-v ?v) | | nonmonotonic | $[0, \infty)$ |
| | (volume-load-l ?l) | (volume-load-l ?l) | | nonmonotonic | $[0, \infty)$ |
| Zenotravel | (onboard ?a) | (onboard ?a) | | nonmonotonic | $[0, \infty)$ |
| | (total-fuel-used) | | | monotonic$^+$ | $[0, \infty)$ |
| | | (fuel ?a) | (fuel ?a) | nonmonotonic$^=$ | $[0, C]$ |

### 6.1.2 Integer Programming Formulations

Numerical constraints such as $\sum_{j \in B} a_j x_j + \sum_{j \in C} g_j y_j \leq b$, where $a_j$ and $g_j$ are real numbers, $B$ the set of binary variables, and $C$ the set of continuous and integer variables, have received a great deal of attention in the field of mixed integer programming [71]. We integrate some of the ideas presented in this field to deal with the numerical constraints and variables that are present in resource planning domains.

Our description for resource planning can be used to extend the IP formulations given by [85]. In this presentation, we will limit our focus on dealing with the numerical state variables, the propositional variables are dealt with in the same way as in van den Briel, Vossen and Kambhampati [85]. That is, propositional variables are transformed into multi-valued state variables, and changes in the state variables are modeled as flows in an appropriately defined network. As a consequence, the resulting IP formulations can be interpreted as a network flow problem with additional side constraints.

We will use the following notation:

- $A$: the set of ground actions
- $R$: the set of resources
- $T$: the maximum number of plan steps
- $prod(a)$, $cons(a)$, $prov(a)$: the set of resources that appear respectively as produce, consume, provide effects for action $a \in A$
- $produce_{a,r}$, $consume_{a,r}$, $provide_{a,r}$: the amount of resource $r \in R$ that is respectively produced, consumed, provided by action $a \in A$

In our formulations we use actions and numerical state variables, which we define as follows:

- $x_{a,t} \in \{0,1\}$, for $a \in A, 1 \leq t \leq T$; $x_{a,t}$ is equal to 1 if action $a$ is executed at plan step $t$, and 0 otherwise.

- $z_{r,t} \geq 0$, for $r \in R, 1 \leq t \leq T$; $z_{r,t}$ represents the value of resource $r$ at plan step $t$. $z_{r,t}$ can be real or integer-valued and may be bounded from above. For now, we will assume that that each resource has a lower bound that can be normalized to 0.

Numerical state variables add constraints to the planning problem and they may appear in the optimization criteria of the planning problem. Next, we will discuss what constraints need to be added to the IP formulation in order to model the different resources.

**6.1.2.1 Monotonic resources:** Resources that behave monotonically can be modeled without introducing numerical state variables to the IP formulation. We can simply deal with these resources by adding them implicitly to the model. Let $R^{mon+}$ and $R^{mon-}$ be the set of resources of type monotonic$^+$ and monotonic$^-$ respectively. If the optimization criteria requires a monotonic resource to be minimized then we can simply setup the following objective function:

$$MIN \sum_{a \in A, 1 \leq t \leq T, r \in R^{mon+}: r \in prod(a)} produce_{a,r} x_{a,t} + \sum_{a \in A, 1 \leq t \leq T, r \in R^{mon-}: r \in cons(a)} consume_{a,r} x_{a,t}$$

Instead of representing monotonic resources by numerical state variables, we can simply deal with them by directly working on the action effects. In case a monotonic resource is bounded then we add an extra constraint to satisfy this bound. For every monotonic$^+$ resource $r$ with an upper bound $U_r$ we must add the following constraint:

$$\sum_{a \in A, 1 \leq t \leq T, r \in R^{mon+}} produce_{a,r} x_{a,t} \leq U_r$$

Similarly, for every monotonic$^-$ resource $r$ with a lower bound $L_r$ and an initial value $I_r$ we must add the following constraint:

$$\sum_{a \in A, 1 \leq t \leq T, r \in R^{mon-}} consume_{a,r} x_{a,t} \leq I_r - L_r$$

Note that in numeric planning domains where all resources are unbounded and monotonic, like the IPC 2002 numeric driverlog domain, resource planning reduces to classical planning with cost sensitive actions.

**6.1.2.2 Nonmonotonic resources:** Resources that are nonmonotonic require the use of numerical state variables in the IP formulation. Since actions may increase or decrease the value of the resource, we need to keep track of their value over time. Let $R^{non}$ be the set of nonmonotonic resources only affected by produce and consume effects of actions, and let $R^{non=}$ be the set of nonmonotonic resources that are affected by provide effects of actions. If nonmonotonic resources are to be minimized then we can add them to the objective function as follows:

$$MIN \sum_{a \in A, 1 \leq t \leq T, r \in R^{non} \cup R^{non=} : r \in prod(a)} produce_{a,r} x_{a,t} + \sum_{a \in A, 1 \leq t \leq T, r \in R^{non} \cup R^{non=} : r \in cons(a)} consume_{a,r} x_{a,t}$$

Also for each nonmonotonic resource $r \in R^{non}$ we must keep track of its value, hence we have the constraint:

$$z_{r,t-1} + \sum_{a \in A, r \in R^{non} : r \in prod(a)} produce_{a,r} x_{a,t} = \sum_{a \in A, r \in R^{non} : r \in cons(a)} consume_{a,r} x_{a,t} + z_{r,t}$$

When an action has a provide effect on a resource $r \in R^{non=}$ then the constraints for keeping track of the resource are more involved, one can use the linear inequalities as described by Kautz and Walser [50].

Integer programming provides a strong framework for dealing with numerical constraints and optimization criteria. So far, only a few researchers have looked into the application of IP techniques in planning with resources, and we believe that there is significant room for improvement. There are some potential cutting planes that we may be able to detect and add to the IP formulations. For example, the bound constraints on the monotonic resource variables can be interpreted as 0-1 knapsack constraints, and so, we may be able to find knapsack cover inequalities. The constraints on the nonmonotonic resources look very similar to constraints we see in lot-sizing problems [61], and so, we could find flow cover inequalities.

## 6.2 Planning with Preferences by Integer Programming

Preferences and trajectory constraints are two new language features in PDDL3.0 that can be used to express hard and soft constraints on plan trajectories, and that can be used to differentiate between hard and soft goals. Hard constraints and goals define a set of conditions that must be satisfied by any solution plan, while soft constraints and goals define a set of conditions that merely affect solution quality.

In particular, preferences assume a choice between alternatives and the possibility to rank or order these alternatives. In PDDL3.0, preferences can be defined on states, on action preconditions, on trajectory constraints, or on some combination of these. Since preferences may or may not be satisfied for a plan to be valid they impose soft constraints or goals on the planning problem. Trajectory constraints, on the other hand, define a set of conditions that must be met throughout the execution of the plan. They can be used to express control knowledge or simply describe restrictions of the planning domain. Since trajectory constraints define necessary con-

ditions for a plan to be valid (except in the case where the trajectory constraint is a preference) they impose hard constraints or goals on the planning problem.

Neither preferences nor trajectory constraints have yet gotten a lot of attention from the planning community, but the importance of solution quality and the efficient handling of hard and soft constraints and goals has increasingly been addressed by some recent works.

Planning with preferences is closely related to oversubscription planning. In oversubscription planning goals are treated as soft goals as there are not enough resources to satisfy all of them. This problem has been investigated by Smith [72] and further explored by several other works.

Preferences, however, are more general than soft goals as they also include soft constraints. Son and Pontelli [73] describe a language for specifying preferences in planning problems using logic programming. Their language can express a wide variety of preferences, including both soft goals and soft constraints, but it seems that it has not been used for testing yet. Empirical results for planning with preferences are provided by Rabideau, Engelhardt and Chien [62] and Brafman and Chernyavsky [11]. Rabideau, Engelhardt and Chien describe an optimization framework for the ASPEN planning system, and Brafman and Chernyavsky describe a constraint based approach for the GP-CSP planning system.

Planning with trajectory constraints is closely related to reasoning about temporal control knowledge and temporally extended goals. Edelkamp [27] handles trajectory constraints by converting a PDDL3.0 description into a PDDL2.2 description and then using a heuristic search planner.

In this paper we show numerous examples of how to express preferences and trajectory constraints by linear constraints over 0-1 variables. These constraints would need to be added to the integer programming formulation of the planning problem. Currently, we are still in the process of incorporating these constraints in the formulations described by van den Briel, Vossen and Kambhampati [85].

### 6.2.1 Modeling with 0-1 Variables

Integer programming is a powerful and natural modeling framework for solving optimization problems involving integer decision variables. The case where the integer variables are restricted to be 0 or 1 are referred to as 0-1 programming problems or binary integer programming problems. In mathematical programming, 0-1 variables are commonly used to represent binary choice, where binary choice is simply a choice between two things. For example, consider the problem of deciding whether an event should or should not occur. This decision can be modeled by a binary variable $x$, where $x = 1$, if the event occurs and, $x = 0$, if the event does not occur. Depending on the problem being considered, the event itself could be almost anything. For example, in scheduling, an event $x$ could represent whether some job should be scheduled before another job.

Here we show some standard relationships between events and how they can be modeled by 0-1 variables.

- The relation that *at most one* of a set $J$ of events is allowed to occur is represented by a packing constraint, $\sum_{j \in J} x_j \leq 1$.
- The relation that *at least one* of a set $J$ of events is allowed to occur is represented by a cover constraint, $\sum_{j \in J} x_j \geq 1$.
- The relation that *exactly one* of a set $J$ of events is allowed to occur is represented by a partitioning constraint, $\sum_{j \in J} x_j = 1$.
- The relation that neither or both events 1 and 2 must occur, that is, event 1 *equals* event 2, is represented by the linear equality $x_2 - x_1 = 0$.
- The relation that event 2 can occur only if event 1 occurs, that is, event 2 *implies* event 1, is represented by the linear inequality $x_2 - x_1 \leq 0$.

Sometimes, the occurrence of an event is limited to a set of pre-specified time periods $1 \leq t \leq T$. The decision whether an event should or should not occur at time period $t$ can can be modeled by a time-indexed binary variable $x_t$, where $x_t = 1$, if the event occurs at time period $t$ and, $x_t = 0$, if the event does not occur at time period $t$. For example, in planning, an event $x_t$ could represent the execution of an action or the truth value of a proposition at a specific plan step.

Here we show some standard relationships between time dependent events and how they can be modeled by time-indexed 0-1 variables.

- The relation that event 1 may occur *at most once* during the time horizon, is represented by the linear inequality $\sum_t x_{1,t} \leq 1$.
- The relation that event 1 must occur *sometime* during the time horizon, is represented by the linear inequality $\sum_t x_{1,t} \geq 1$.
- The relation that if event 1 occurs, event 2 must occur *sometime-before* event 1, is represented by the linear inequalities $x_{1,t} \leq \sum_{1 \leq s < t} x_{2,s}$ for all $1 \leq t \leq T$.
- The relation that if event 1 occurs, event 2 must occur *sometime-after* event 1, is represented by the linear inequalities $x_{1,t} \leq \sum_{t \leq s \leq T} x_{2,s}$ for all $1 < t \leq T$.
- The relation that event 1 must *always* occur is represented by the linear equalities $x_{1,t} = 1$ for all $1 \leq t \leq T$.
- The relation that event 1 must occur *at the end* of the time horizon is represented by the linear equality $x_{1,T} = 1$.

Note that most of these standard relationships coincide with the new modal operators: `at-most-once`, `sometime`, `sometime-before`, `sometime-after`, `always`, `at end`, in PDDL3.0. Even though there some differences in semantics, this suggests that modeling these modal oper-

ators and the preferences and trajectory constraints that are expressed by them through integer programming should be rather straightforward.

In the next two sections we give various examples of how to model preferences and trajectory constraints by linear constraints over 0-1 variables. The examples are all borrowed from the International Planning Competition resources [1].

### 6.2.2 Simple Preferences

Simple preferences are preferences that appear in the goal or that appear in the preconditions of an action. Goal preferences can be violated at most once (at the end of the plan), whereas precondition preferences can be violated multiple times (each time the corresponding action is executed).

For each goal preference in the planning problem we introduce a 0-1 variable $p$, where $p = 1$, if the goal preference is violated and, $p = 0$ if the goal preference is satisfied. Similarly, for each precondition preference for action $a$ at step $t$ ($1 \leq t \leq T$) we introduce a 0-1 variable $p_{a,t}$, where $p_{a,t} = 1$, if the precondition preference is violated for action $a$ at step $t$ and, $p_{a,t} = 0$ if the precondition preference is satisfied for action $a$ at step $t$. This way all violations can be counted for separately and given different costs in the objective function of the formulation.

Constraints for goal and precondition preferences are easily modeled by integer programming. There are only finitely many operators in PDDL3.0, including some standard operators like `or`, `and`, and `imply`, which can all be represented by one or more linear constraints.

**6.2.2.1 Examples:** In the examples we will use variables $x_{a,t}$ to denote the execution of an action $a$ at step $t$, and use variables $y_{f,t}$ to denote the truth value of a fluent $f$ at step $t$. This is slightly different from the notation and variables used in the formulations by van den

[1] http://zeus.ing.unibs.it/ipc-5/

Briel, Vossen, and Kambhampati [85], but it provides a concise representation of the resulting constraints.

In PDDL3.0, the goal preference $p_1$ "We would like that person1 is at city2" is expressed as follows.

```
(:goal (and (preference p1
  (at person1 city2))))
```

The inequality corresponding to preference $p_1$ is given by:

$$p_1 \geq 1 - y_{\text{at person1 city2},T} \tag{1}$$

Thus preference $p_1$ is violated ($p_1 = 1$) if person1 is not at city2 at the end of the plan ($y_{\text{at person1 city2},T} = 0$).

The goal preference $p_2$ "We would like that person1 or person2 is at city2" is expressed as follows.

```
(:goal (and (preference p2 (or
  (at person1 city2) (at person2 city2)))))
```

The inequality corresponding to preference $p_2$ is given by:

$$p_2 \geq 1 - y_{\text{at person1 city2},T} - y_{\text{at person2 city2},T} \tag{2}$$

Now, preference $p_2$ is violated if neither person1 nor person2 is at city2 at the end of the plan. Preference $p_2$ is satisfied when either or both person1 and person2 are at city2 at the end of the plan.

The goal preference $p_3$ "We would like that person2 is at city1 if person1 is at city1" is expressed as follows.

```
(:goal (and (preference p3 (imply
  (at person1 city1) (at person2 city1)))))
```

The inequality corresponding to preference $p_3$ is given by:

$$p_3 \geq y_{\text{at person1 city1},T} - y_{\text{at person2 city1},T} \tag{3}$$

So preference $p_3$ is violated if person2 is not at city1 while person1 is.

The goal preference $p_4$ "We would like that person3 and person4 are at city3" is expressed as follows.

```
(:goal (and (preference p4 (and
  (at person3 city3) (at person4 city3)))))
```

Note that preference $p_4$ is very similar to preference $p_1$ with the exception that $p_4$ is defined over a conjunction of fluents. For each fluent in the conjunction we will state a separate constraint, thus the inequalities corresponding to preference $p_4$ are given by:

$$p_4 \geq 1 - y_{\text{at person3 city3},T} \tag{4}$$

$$p_4 \geq 1 - y_{\text{at person4 city3},T} \tag{5}$$

Now, preference $p_4$ is violated if either or both person3 and person4 are not at city3 at the end of the plan.

Preferences over preconditions are different from goal preferences as they depend on both the execution of an action and on the state of the precondition of that action. Moreover,

a precondition preference is defined for each plan step $t$, where $1 \leq t \leq T$. In PDDL3.0, the precondition preference $p_{5,\text{fly?}a\text{?}c1\text{?}c2,t}$ "We would like that some person is in the aircraft" whenever we fly aircraft ?$a$ from city ?$c$1 to city ?$c$2 is expressed as follows:

```
(:action fly
 :parameters (?a - aircraft ?c1 ?c2 - city)
 :precondition (and (at ?a ?c1)
   (preference p5
     (exists (?p - person) (in ?p ?a))))
 :effect (and (not (at ?a ?c1))
   (at ?a ?c2)))
```

The inequalities corresponding to each ground fly ?a ?c1 ?c2 action is given by:

$$p_{5,\text{fly?}a\text{?}c1\text{?}c2,t} \geq x_{\text{fly ?a ?c1 ?c2,t}} - \sum_{?p} y_{\text{in ?p ?a,}t} \quad \forall 1 \leq t \leq T \tag{6}$$

Thus, preference $p_{5,\text{fly?}a\text{?}c1\text{?}c2,t}$ is violated at step $t$ if we fly aircraft ?$a$ from city ?$c$1 to city ?$c$2 at step $t$ ($x_{\text{fly ?a ?c1 ?c2,t}} = 1$) without having any passenger ?$p$ onboard at step $t$ ($y_{\text{in ?p ?a,}t} = 0$, for each ?$p$).

### 6.2.3 Qualitative Preferences

In propositional planning, qualitative preferences include trajectory constraints and preferences over trajectory constraints none of which involve numbers. Given the space limitations we will mainly concentrate on the trajectory constraints here that use the new modal operators of PDDL3.0 in this section.

There is a general rule of thumb for the operators `forall` and `always`. `forall` indicates that the trajectory constraint must hold for each object to which it is referring to. For example, `forall (?b - block)` means that the trajectory must hold for each instantiation of $?b$, thus we generate the trajectory constraint for all blocks $?b$. `always` in propositional planning is equivalent to saying for all $t$, thus we generate the trajectory constraint for all $t$ where $1 \leq t \leq T$.

Constraints for trajectories are easily modeled by integer programming through observing the different operators carefully. It is often the case, that the trajectory constraint simply represent one of the standard relationships described earlier in this paper.

**6.2.3.1 Examples:** In PDDL3.0 the trajectory constraint "A fragile block can never have something above it" is expressed as follows.

```
(:constraints (and (always (forall (?b - block)
  (implies (fragile ?b) (clear ?b))))))
```

The inequality corresponding to this trajectory constraint corresponds to the relation that fragile *implies* clear for all blocks $?b$, for all steps $t$, where $1 \leq t \leq T$. It is given by:

$$y_{\text{fragile ?b},t} - y_{\text{clear ?b},t} \leq 0 \quad \forall ?b, 1 \leq t \leq T \tag{7}$$

The trajectory constraint "Each block should be picked up at most once" which is expressed as follows.

```
(:constraints (and (forall (?b - block)
  (at-most-once (holding ?b)))))
```

It translates to an *at most once* relation for all blocks ?$b$ and is given by:

$$y_{\text{holding ?b,0}} + \sum_{a \in A, 1 \leq t \leq T : \text{holding ?b} \in ADD(a)} x_t^a \leq 1 \quad \forall ?b \tag{8}$$

Likewise the trajectory constraint "Each block should be picked up at least once" is expressed as follows.

```
(:constraints (and (forall (?b - block)
  (sometime (holding ?b)))))
```

This translates to a *sometime* relation for all blocks ?$b$ and is given by:

$$\sum_t y_{\text{holding ?b},t} \geq 1 \quad \forall ?b \tag{9}$$

Continuing in the same way, the trajectory constraint "A truck can visit city1 only if it has visited city2 sometime before" is expressed in PDDL3.0 as follows.

```
(:constraints (and (forall (?t - truck)
  (sometime-before
    (at ?t city1) (at ?t city2)))))
```

The corresponding inequality describes a *sometime-before* relationship for all trucks ?$t$ and is given by:

$$\sum_{1 \leq s < t} y_{\text{at ?t city2},s} \geq y_{\text{at ?t city1},t} \quad \forall ?t, 1 \leq t \leq T \tag{10}$$

Similarly the trajectory constraint "If a taxi has been used and it is at the depot, then it has to be cleaned."

```
(:constraints (and (forall (?t - taxi)
  (sometime-after (and (at ?t depot) (used ?t))
    (clean ?t)))))
```

This translates to a *sometime-after* relationship for all taxis $?t$. Note, however, that this trajectory constraint has two conditions, which if satisfied, require that taxi $?t$ is to be cleaned. The inequality corresponding to this trajectory constraint is given by:

$$y_{\text{at ?t depot},t} + y_{\text{used ?t},t} - 1 \leq \sum_{t \leq s \leq T} y_{\text{clean ?t},s} \quad \forall ?t, 1 \leq t \leq T \tag{11}$$

Now, if taxi $?t$ is at the depot at step $t$ ($y_{\text{at ?t depot},t} = 1$) and if it has been used ($y_{\text{used ?t},t} = 1$), then it must be cleaned sometime after step $t$ ($\sum_{t \leq s \leq T} y_{\text{clean ?t},s} \geq 1$).

More examples can be presented, but these seem sufficient to point out that integer programming provides a natural framework for modeling propositional planning with preferences and trajectory constraints.

### 6.3 Optimal Planning

The main objective is to develop effective IP methods that yield provably optimal solutions. Even though several heuristic approaches that address specific optimality criteria have been proposed [84, 50], exact methods for optimal planning are practically non-existent.

At first sight, it may appear rather straightforward to incorporate general optimality criteria within the IP models we proposed in our earlier work. However, simply using a more general objective function with these formulations will not guarantee that the resulting solution represents an optimal plan. Because we limit the number of plan steps, the resulting solution will only be optimal with respect to a given upper bound on the number of plan steps. This notion of optimality, though, has little practical relevance, since plan steps are somewhat artificial

constructs that may vary according to the underlying parallelism. A key issue, therefore, is to determine whether a bound is sufficient, that is, a guarantee that no better solutions can be found by increasing the number of plan steps. The derivation of sufficient bounds that are tight is difficult; while possible (i.e., the number of possible states in a planning problem is a trivial bound), the large number of plan steps is likely to render the problem intractable.

As such, the development of exact methods for optimal planning poses a number of fundamental challenges. The issues mentioned above illustrate a limitation for *any* approach that relies on the notion of plan steps and planning graphs, which suggests a need for different problem representations and solution methods.

While there are different ways to represent classical planning, here we use *deterministic automata* [17] to describe planning problems. This representation is analogous to what is know as the state variable representation for classical planning [33], and relies on a finite set of state variables $V$.

### 6.3.1 Deterministic Automaton Representation

As a first step we define classical planning problems as deterministic automata, following the definition in [17].

**Definition 6.3.1.** Given a set of state variables $V = \{v_1, ..., v_n\}$, a planning problem on $V$ is a deterministic automaton

$$G \quad = \quad (S, A, \gamma, \Gamma, s_0, g)$$

where:

- $S$ is a finite set of states. $S \subseteq \prod_{v \in V} D_v$ where $D_v$ is the domain of state variable $v$. A state $s \in S$ is denoted $s = \{(v = c) | v \in V\}$, where $c \in D_v$.

- $A$ is a finite set of actions associated with the transitions in $G$. Each action $a \in A$ is a 2-tuple $(precond(a), effect(a))$, where $precond(a)$ is a set of expressions of the form $(v = c)$ on the state variables that define the preconditions of $a$, and $effect(a)$ is a set of assignments of values to state variables of the form $(v \leftarrow c)$ that define the effects of $a$.

- $\gamma : S \times A \rightarrow S$ is the *transition function*. $\gamma(s, a) = \{(v = c)|v \in V\}$, where $c$ is specified by an assignment $(v \leftarrow c)$ in *effects*$(a)$ if there is such an assignment, otherwise $(v = c) \in s$.

- $\Gamma : S \rightarrow 2^A$ is the *active action function*, where $\Gamma^-$ represents the active applicable action function, and $\Gamma^+$ represents the active relevant action function. An action $a$ is applicable to a state $s \in S$ if and only if $a \in \Gamma^-(s)$, where $\Gamma^-(s) = \{a \in A|(v = c) \in precond(a) \wedge (v = c) \in s\}$. An action $a$ is relevant to a state $s \in S$ if and only if $a \in \Gamma^+(s)$, where $\Gamma^+(s) = \{a \in A|(v \leftarrow c) \in effects(a) \wedge (v = c) \in s\}$.

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

- $g \subseteq S$ is a set of *goal states*.

An automaton is deterministic if $\gamma(s, a) = s'$ and $\gamma(s, a) = s''$ imply that $s' = s''$ for all $a \in A$ and $s, s', s'' \in S$. Intuitively, the automaton $G$ operates as follows. It starts in the initial state $s_0$ and upon the execution of an action $a \in \Gamma^-(s_0)$ it transitions to the state $\gamma(s_0, a) \in S$ . This process then continues based on the transitions for which $\gamma$ is defined. A string (i.e. sequence of actions) $\pi$ is processed in the following way. First, let $A^*$ denote the set of all finite-length strings in $A$, including the string consisting of no actions $\varepsilon$. Second, we can extend $\gamma$ from domain $S \times A$ to domain $S \times A^*$ as follows

$$\gamma(s,\varepsilon) = s, \tag{12}$$

$$\gamma(s,\pi a) = \begin{cases} \gamma(\gamma(s,\pi),a) & \text{if } a \in \Gamma^-(s) \\ undefined & \text{otherwise} \end{cases} \tag{13}$$

where (12) states that on the empty string $\varepsilon$ the automaton cannot change. Now, (13) shows how to compute the state on the string $\pi a$. That is, first compute the state $\gamma(s,\pi)$ on the string $\pi$, and then compute the state $\gamma(s,a)$.

Plans may now be interpreted in terms of the languages generated by the corresponding automaton, which leads to the following definition.

**Definition 6.3.2.** The language *generated* by $G$ is

$$\mathcal{L}(G) := \{\pi \in A^* : \gamma(s_0,\pi) \text{ is defined}\}$$

The language *accepted* by $G$, which correspond to solutions of the planning problem, is

$$\mathcal{L}^*(G) := \{\pi \in \mathcal{L}(G) : \gamma(s_0,\pi) \in g\}$$

The simplest way to represent an automaton is to consider its directed graph representation, or *state transition diagram*. In a state transition diagram nodes represent states, labeled arcs represent transitions between these states, and $\Gamma^-(s)$ and $\Gamma^+(s)$ correspond to the set of in- and out-arcs for a state $s \in S$ respectively. In this representation, the notion of plans being generated and accepted is easily understood by examining the state transition diagram. The plans $\mathcal{L}(G)$ corresponds to the set of all directed simple paths in the state transition diagram that start at the initial state $s_0$. Moreover, the plans $\mathcal{L}^*(G)$ is the subset of $\mathcal{L}(G)$ consisting only of the directed simple paths that end in a goal state $s \in g$.

A typical example of a classical planning problem is a logistics problem. In a logistics problem there are a number of locations, a number of packages, and a number of trucks that can carry

and move packages from one location to another. The goal is to move the packages from their origin to their destination.

**Example 6.3.1. (Logistics)**

Figure 28 shows the state transition diagram corresponding to the automaton $G_{logistics}$ of a simple logistics problem involving two state variables $V = \{package(p1), truck(t1)\}$, with domains $D_{package(p1)} = \{loc1, loc2, t1\}$ and $D_{truck(t1)} = \{loc1, loc2\}$ denoting the location of package $p1$ and truck $t1$ respectively. The set of nodes is the state set of the automaton, $S = \{(loc1, loc2), (loc1, loc1), (t1, loc1), (t1, loc2), (loc2, loc2), (loc2, loc1)\}$, where a state $s$ is denoted by the pair $(c_1, c_2)$ such that $c_1 \in D_{package(p1)}$ and $c_2 \in D_{truck(t1)}$. The set of labels for the transitions is the action set of the automaton, $A = \{load(p1, t1, loc1),\ load(p1, t1, loc2),\ unload(p1, t1, loc1),\ unload(p1, t1, loc2),\ drive(loc1, loc2),\ drive(loc1, loc2)\}$. The arcs in the graph provide a graphical representation of the transition function of the automaton, which is denoted as $\gamma((loc1, loc2), drive(loc2, loc1)) = (loc1, loc1),\ \gamma((loc1, loc1), load(p1, t1, loc1)) = (t1, loc1)$ and so forth. The initial state is marked by an arrow, and the goal states are marked by double circles. Hence, in the initial state we have that package $p1$ is at location $loc1$ and the truck $t1$ is at location $loc2$, and the goal is to have package $p1$ at location $loc2$.

### 6.3.2 Concurrent Automata Representation

The number of states in a classical planning problem is generally exponential, which makes the deterministic automata representation impractical. We therefore describe a more compact representation in which classical planning is defined using a set of *concurrent automata* [17].

**Definition 6.3.3.** Let $V = \{v_1, ..., v_n\}$ be a given set of state variables, and $G = (S, A, \gamma, \Gamma, s_0, g)$ a planning problem on $V$. For each $v \in V$ we define a deterministic automaton

$$G_v \quad = \quad (D_v, A_v, \gamma_v, \Gamma_v, c_{0,v}, g_v)$$

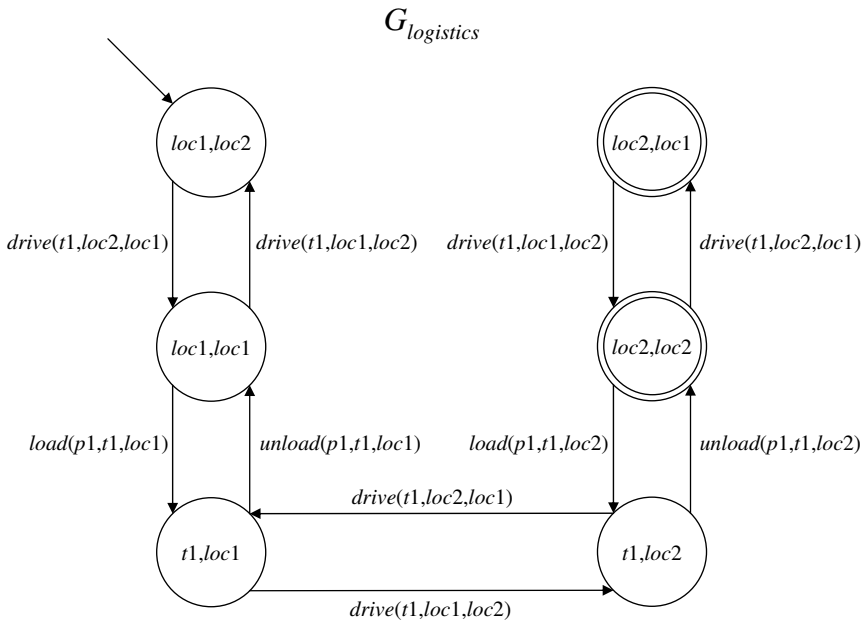


Fig. 28. Deterministic automaton representation of the simple logistics example.

where:

- $D_v$ is a finite set of states corresponding to the domain of state variable $v$.
- $A_v := \{a \in A | (v \leftarrow c) \in \mathit{effects}(a)\}$ is a finite set of actions associated with the transitions in $G_v$. Note that, $A = \bigcup_{v \in V} A_v$.
- $\gamma_v : D_v \times A_v \rightarrow D_v$ is the transition function. $\gamma_v(c_1, a) = c_2$, where $c_2$ is specified by an assignment $(v \leftarrow c_2)$ in $\mathit{effects}(a)$ if there is such an assignment.
- $\Gamma_v : D_v \rightarrow 2^{A_v}$ is the active action function. An action $a$ is applicable to a state $c \in D_v$ if and only if $a \in \Gamma_v^-(c)$, where $\Gamma_v^-(c) = \{a \in A_v | (v = c) \in \mathit{precond}(a) \wedge c \in D_v\}$. An action $a$ is relevant to a state $c \in D_v$ if and only if $a \in \Gamma_v^+(c)$, where $\Gamma_v^+(c) = \{a \in A_v | (v \leftarrow c) \in \mathit{effects}(a) \wedge c \in D_v\}$.
- $c_{0,v} \in D_v$ is the initial state of state variable $v$.
- $g_v \subseteq D_v$ is a set of goal states of state variable $v$.

A set of concurrent automata operates very much like a deterministic automaton. That is, each automaton $G_v$, starts in the initial state $c_{0,v}$ and upon the execution of an action $a \in \Gamma_v^-(c_{0,v})$ it transitions to the state $\gamma(c_{0,v}, a) \in D_v$. This process then continues for each automaton $G_v$ based on the transitions for which $\gamma_v$ is defined. Some automata may communicate over common actions. A *common action* is an action that appears in the action set $A_v$ of two or more automata $G_v$. For example, the common actions of $G_1$ and $G_2$ are the actions in $A_1 \cap A_2$. A common action can only be executed if all automata involved execute it simultaneously, whereas a *private action*, those actions that appears in the action set $A_v$ of only one automaton $G_v$, can be executed whenever possible.

In the concurrent automata representation, a plan corresponds to a directed walk (i.e. a path with repeated vertices and arcs allowed) in the state transition diagram of each automata that starts at the initial state $s_0$ and ends in a goal state $s \in g$.

**Example 6.3.2. (Logistics continued)**

The state transition diagrams in Figure 29 give a concurrent automata representation for our simple logistics problem introduced in Example 6.3.1. The automaton on the left $G_{package(p1)}$ represents the state variable for package $p1$ with $D_{package(p1)} = \{loc1, loc2, t1\}$, and the automaton on the right $G_{truck(t1)}$ represents the state variable for truck $t1$ with $D_{truck(t1)} = \{loc1, loc2\}$. The sets of nodes correspond to the state sets of the automata, and the sets of labels for the transitions correspond to the action sets of the automata. Note that, $load(p1, t1, loc1)$, $load(p1, t1, loc2)$, $unload(p1, t1, loc1)$, $unload(p1, t1, loc2)$ are common actions as they appear in the action set of both automata. Hence, the action $load(p1, t1, loc1)$ can only be executed if both automata execute it simultaneously. That is, if package $p1$ is at location $loc1$ and truck $t1$ is also at location $loc1$. The actions $drive(loc1, loc2)$, $drive(loc2, loc1)$ are private actions and thus can be executed whenever they are applicable.

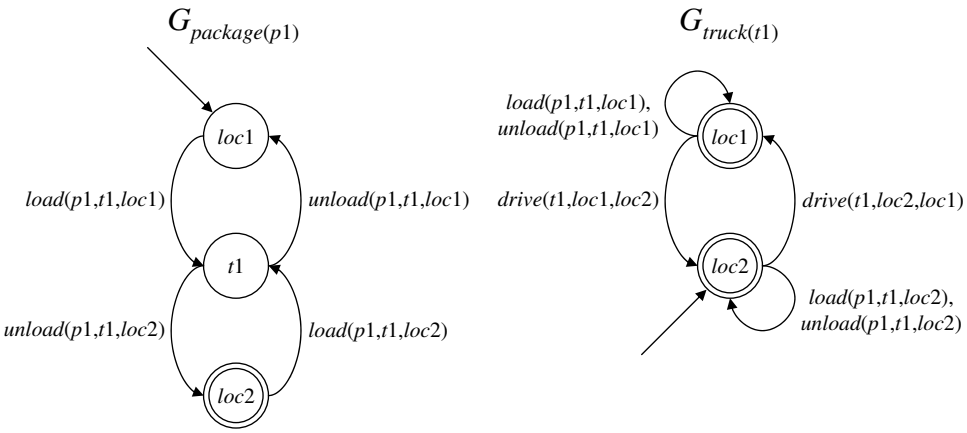


Fig. 29. Concurrent automata representation of the simple logistics example.

One way to model the joint behavior of a set of automata that operate concurrently is by applying the parallel composition operation. Parallel composition indicates that two or more transitions in different automata should be carried out in parallel. Thus, in a parallel composition automata are synchronized on the common actions.

**Definition 6.3.4.** The *parallel composition* of the two automata $G_1$ and $G_2$ is the automaton

$$G_1 \| G_2 := (S_1 \times S_2, A_1 \cup A_2, \gamma_{1\|2}, \Gamma_{1\|2}, (c_{0,1}, c_{0,2}), g_1 \times g_2)$$

where

$$\gamma_{1\|2}((c_1, c_2), a) := \begin{cases} (\gamma_1(c_1, a), \gamma_1(c_2, a)) & \text{if } a \in \Gamma_1^-(c_1) \cap \Gamma_2^-(c_2) \\ (\gamma_1(c_1, a), c_2) & \text{if } a \in \Gamma_1^-(c_1) \backslash A_2 \\ (c_1, \gamma_1(c_2, a)) & \text{if } a \in \Gamma_2^-(c_2) \backslash A_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

and for both $\Gamma_{1\|2}^-(c_1, c_2)$ and $\Gamma_{1\|2}^+(c_1, c_2)$ we have

$$\Gamma_{1\|2}(c_1, c_2) := [\Gamma_1(c_1) \cap \Gamma_2(c_2)] \cup [\Gamma_1(c_1) \backslash A_2] \cup [\Gamma_2(c_2) \backslash A_1]$$

In order to characterize the languages (i.e. plans) generated by $G_1 \| G_2$ in terms of those by $G_1$ and $G_2$, we define projections and their corresponding inverse maps. A projection $P_v$ is a type

of operation performed in automata theory on strings (i.e. sequences of actions) and languages from a set of actions, say $A_1 \cup A_2$, to a smaller set of actions $A_v$ where $v = 1, 2$.

**Definition 6.3.5.** The *projections* $P_1$ and $P_2$ for strings are defined as follows

$$P_v : (A_1 \cup A_2)^* \to A_v^* \quad \text{for } v = 1, 2$$

where:

$$\begin{aligned} P_v(\varepsilon) &:= \varepsilon \\ P_v(a) &:= \begin{cases} e & \text{if } e \in A_v \\ \varepsilon & \text{if } e \notin A_v \end{cases} \\ P_v(\pi a) &:= P_v(\pi)P_v(a) \quad \text{for } \pi \in (A_1 \cup A_2)^*, a \in (A_1 \cup A_2) \end{aligned}$$

and their corresponding inverse maps $P_1^{-1}$ and $P_2^{-1}$ are defined by

$$P_v^{-1} | A_v^* \to 2^{(A_1 \cup A_2)^*}$$

where:

$$P_v^{-1}(\pi) := \{\sigma \in (A_1 \cup A_2)^* | P_v(\sigma) = \pi\}$$

In words, the projection operation $P_1(P_2)$ takes a string formed from the larger set of actions $A_1 \cup A_2$ and removing actions in it that do not belong to the smaller set of actions $A_1(A_2)$. Accordingly, the inverse projection operation $P_1^{-1}(P_2^{-1})$ takes a string $\pi$ in the smaller set of actions $A_1(A_2)$ and returns the set of all strings from the larger set of actions $A_1 \cup A_2$ that project to $\pi$. The projections $P_v$ and their inverses $P_v^{-1}$ can be extended to languages by simply applying them to all the strings in the language.

**Definition 6.3.6.** The projections $P_1$ and $P_2$ and their corresponding inverse maps $P_1^{-1}$ and $P_2^{-1}$ for languages $L$ are defined as follows

$$P_v(L) := \{\pi \in A_v^* | \exists \sigma \in L(P_v(\sigma) = \pi)\} \qquad \text{for } L \subseteq (A_1 \cup A_2)^*, v = 1, 2$$

$$P_v^{-1}(L_v) := \{\sigma \in (A_1 \cup A_2)^* | \exists \pi \in L_v(P_v(\sigma) = \pi)\} \qquad \text{for } L_v \subseteq A_v^*, v = 1, 2$$

Using the above defined projections, we can now characterize the languages resulting from a parallel composition

**Definition 6.3.7.** The language generated by $G_1 \| G_2$ is

$$\mathcal{L}(G_1 \| G_2) := P_1^{-1}[\mathcal{L}(G_1)] \cap P_2^{-1}[\mathcal{L}(G_2)]$$

The language accepted by $G_1 \| G_2$ is

$$\mathcal{L}^*(G_1 \| G_2) := P_1^{-1}[\mathcal{L}^*(G_1)] \cap P_2^{-1}[\mathcal{L}^*(G_2)]$$

**Example 6.3.3. (Parallel composition)**

The parallel composition of the automata $G_{package(p1)}$ and $G_{truck(t1)}$ in Example 6.3.2 is the automaton $G_{logistics}$ in Example 6.3.1. Projections from the larger automaton $G_{logistics}$ to either of the smaller automata $G_{package(p1)}$ or $G_{truck(t1)}$ are easily understood by examining the corresponding state transition diagrams. For example, when we project the string (i.e. sequence of actions) $\{drive(loc2, loc1), load(p1, t1, loc1), drive(loc2, loc1), unload(p1, t1, loc2)\}$ onto $G_{package(p1)}$ we obtain the string $\{load(p1, t1, loc1), unload(p1, t1, loc2)\}$. Note that when we project languages from the larger automaton to the either of the smaller automata that we obtain subset inclusion. That is, all strings that are generated (accepted) by $G_{logistics}$ and projected onto $G_{package(p1)}$ or $G_{truck(t1)}$ are also generated (accepted) by $G_{package(p1)}$ or $G_{truck(t1)}$ respectively, but not necessarily vice versa. For example, the string $\pi = \{unload(p1, t1, loc2), drive(t1, loc2, loc1), load(p1, t1, loc1)\}$ is both generated and accepted by $G_{truck(t1)}$, but is not generated nor accepted by $G_{logistics}$.

Example 6.3.3 shows that we have

$$P_v[\mathcal{L}(G_1 \| G_2)] \subseteq \mathcal{L}(G_v) \quad \text{for } v = 1, 2$$

The same result also holds for the accepted language $\mathcal{L}^*$. In particular, Example 6.3.3 shows that the parallel composition of automata may prevent some strings to occur in the generated

(accepted) languages of the individual automata. This is due to the constraints imposed by the synchronization of the common actions in the parallel composition.

The parallel composition operation on automata is commutative and associative [17]. Thus, given the automata $G_1$, $G_2$, and $G_3$, we have that $G_1 \| G_2 = G_2 \| G_1$, and $G_1 \| (G_2 \| G_3) = (G_1 \| G_2) \| G_3$. As a result, we can extend the definition of parallel composition and their projections to sets of automata.

**Definition 6.3.8.** The parallel composition of the set of automata $V$ is the automaton

$$G_V := \|_{\{v \in V\}} G_v$$

Moreover, the parallel composition of the sets of automata $X$ and $Y$, such that $X \cap Y = \emptyset$ is the automaton

$$G_X \| G_Y := \|_{\{v \in (X \cup Y)\}} G_v$$

**Definition 6.3.9.** Given the sets of automata $X$ and $Y$, such that $X \cap Y = \emptyset$. The language generated by $G_X \| G_Y$ is

$$\mathcal{L}(G_X \| G_Y) := P_X^{-1}[\mathcal{L}(g_X)] \cap P_Y^{-1}[\mathcal{L}(g_Y)]$$

The language accepted by $G_X \| G_Y$ is

$$\mathcal{L}^*(G_X \| G_Y) := P_X^{-1}[\mathcal{L}^*(g_X)] \cap P_Y^{-1}[\mathcal{L}^*(g_Y)]$$

Now, for any set of automata $V$ we have the following result

$$P_W[\mathcal{L}(G_V)] \subseteq \mathcal{L}(G_W) \quad \text{for } W \subseteq V$$

Likewise, the same holds for the accepted language $\mathcal{L}^*$. This is an important result as it indicates that the generated (accepted) language of $G_V$ projected onto $G_W$ is contained in the generated (accepted) language of $G_W$. In terms of planning, which relies on a given set of state variables

$V$, this result implies that we can use any subset of state variables $W \subseteq V$ as a relaxation of the original planning problem. Note that, a problem is a relaxation of the original problem if any feasible solution to the original problem (i.e. any string $\pi$ that is accepted by the automaton $G_V$) is also a feasible solution to the relaxation problem (i.e. the projection of $\pi$ onto $G_W$ is a string $\sigma$ that is accepted by the automaton $G_W$).

**Definition 6.3.10.** Given the sets of automata $V$ and $W$, such that $W \subseteq V$. The projection for languages $L(G_V)$ onto $G_W$ is called a *perfect projection* if

$$P_W[L(G_V)] = L(G_W)$$

**Theorem 6.3.1.** Let $V$, $W$, $W'$ be sets of automata, such that $W \subseteq V$ and $W' = V \backslash W$. If $A_W \cap A_{W'} = \emptyset$ then the projection for languages $L(G_V)$ onto $G_W$ is a perfect projection.

*Proof.* We must show that if $A_W \cap A_{W'} = \emptyset$ then $P_W[L(G_V)] = L(G_W)$ (and similarly $P_{W'}[L(G_V)] = L(G_{W'})$). Hence, we must show that the inverse projection for any string $\pi \in L(G_W)$ returns a nonempty set of strings $\sigma \in A_V^*$ that project to $\pi$. In case the inverse projection of a string $\pi \in L(G_W)$ returns the empty set, then we have $P_W[L(G_V)] \subseteq L(G_W)$ (i.e. no perfect projection). Since $\pi \in L(G_W)$ we have that $\pi \in A_W^*$. Moreover, since $A_W \cap A_{W'} = \emptyset$ and $A_V = A_W \cup A_{W'}$ we have that $\pi \in A_V^*$. In other words, because there are no common actions between $A_W$ and $A_{W'}$ any string that is accepted by $G_W$ must have a nonempty inverse projection on $G_V$, just take the string itself. □

The theorem is rather trivial, but useful nonetheless as it characterizes perfect projections.

### 6.3.3 Deterministic Automaton Formulation

Our ultimate goal is to use integer programming for optimal planning. In the previous section we have seen that a planning problem can be represented as a deterministic automaton and as

a system of concurrent automata. This section and the next describe an integer programming formulation for deterministic automata.

When generating an integer programming formulation for a deterministic automaton it is useful to consider its state transition diagram. We introduce integer variables $x_{\gamma(s,a)} \in \mathbb{Z}^+$, for $s \in S, a \in \Gamma^-(s)$ for each transition $\gamma(s,a)$, where $x_{\gamma(s,a)}$ is equal to 1 if the transition $\gamma(s,a)$ is executed, and 0 otherwise. Moreover, for each state $s \in S$ we generate flow constraints stating that the flow into state $s$ (i.e. the number of times the automaton transitions to state $s$) must equal the flow out of state $s$ (i.e. the number of times the automaton transitions from state $s$). There is an implicit flow into the initial state $s_0$ and an implicit flow out of the set of goal states $g$. A nonnegative cost $c_a$ is associated with each action $a \in A$. The objective function is to minimize the cost of the actions that are executed in the deterministic automaton, which in terms of planning, corresponds to minimizing the cost of the plan. Mathematically, a deterministic automaton is formulated as the following integer programming problem:

$$\min \sum_{s \in S, a \in \Gamma^-(s)} c_a x_{\gamma(s,a)}$$

subject to:

$$\sum_{a \in \Gamma^+(s)} x_{\gamma(s,a)} - \sum_{a \in \Gamma^-(s)} x_{\gamma(s,a)} \geq \begin{cases} 1 & \text{if } s \in g, s \neq s_0 \\ -1 & \text{if } s = s_0, s \notin g \\ 0 & \text{otherwise} \end{cases} \qquad \forall s \in S \qquad \text{(DFA)}$$

$$x_{\gamma(s,a)} \in \mathbb{Z}^+ \qquad \forall s \in S, a \in \Gamma^-(s)$$

It is important to note that the minimum cost plan never visits a state $s \in S$ more than once. If a plan visits a state more than once then it contains a loop, and so, cannot not be optimal. Hence, the optimal solution to this integer programming formulation corresponds to a directed simple path from the initial state to one of the goal states in the deterministic automaton. As

a result, the optimal solution can be found by solving the corresponding linear programming relaxation of the constraint set (DFA), or by using any shortest path algorithm. Unfortunately, the deterministic automaton representation is impractical as the number of states in a classical planning problem is generally exponential. Hence, even the most efficient shortest path algorithm would not be useful.

### 6.3.4 Concurrent Automata Formulation

When generating an integer programming formulation for a set of concurrent automata we can simply take the deterministic automaton formulation and apply it to each automaton $G_v$, for $v \in V$. The resulting formulation would then, however, only model the individual automata and not their interactions. Since some automata may communicate over common actions we introduce extra variables and constraints to link the automata. We introduce integer variables $x_a \in \mathbb{Z}^+$ for each action $a \in A$, where $x_a$ is equal to the number of times action $a$ is executed. Moreover, we generate linking constraints that equate the number of times an action $a$ is executed in each automaton for which $a \in A_v$. The objective function, which can now be expressed using the $x_a$ variables, minimizes the cost of the actions that are executed in the set of concurrent automata. Mathematically, this results in the following integer programming problem:

$$\min \sum_{v \in V, c \in D_v, a \in \Gamma^-(c)} c_a x_a$$

subject to:

$$\begin{array}{lll} (\text{DFA})_v & \forall v \in V & (\text{CA}) \\ \sum_{c \in D_v, a \in \Gamma_v^-(c)} x_{\gamma_v(c,a)} = x_a & \forall v \in V, a \in A_v & \\ x_a \in \mathbb{Z}^+ & \forall a \in A_V & \end{array}$$

The constraints in this integer programming formulation are necessary, but not sufficient to model all the semantics of the concurrent automata representation. Two characteristics are ignored. First, due the presence of directed cycles and multiple goal states in the state transition graphs corresponding to the automata, the solution may end up with disjoint circular flows. Second, the flow constraints ignore information about the relative ordering of the actions.

**Example 6.3.4. (Logistics solution)**

The formulation for our simple logistics problem using the concurrent automata representation as shown in Example 6.3.2 generates two sets of flow constraints and a set of linking constraints. By setting $c_a = 1$ for each $a \in A_{logistics}$ we obtain the solution where $x_{load(p1,t1,loc1)} = x_{unload(p1,t1,loc2)} = 1$, indicated by the bold arcs in Figure 30, with and all other variables equal to zero. The transition graph of $G_{truck(t1)}$ shows that the constraints fail to capture that solutions to the individual automata must convert to a string of actions that is accepted by the automaton. Moreover, even if the solution converts to a string of actions that is accepted by the individual automata, the constraints fail to capture the relative ordering of the actions. For example, $x_{load(p1,t1,loc1)} = x_{unload(p1,t1,loc2)} = x_{drive(t1,loc2,loc1)} = 1$, is a solution to the integer programming formulation for this example, but it is not accepted by the set of concurrent automata. In $G_{package(p1)}$ we must execute $load(p1, t1, loc1)$ before $unload(p1, t1, loc2)$, while in $G_{truck(t1)}$ we must execute $unload(p1, t1, loc2)$ before $drive(t1, loc2, loc1)$ before $load(p1, t1, loc1)$.

**Example 6.3.5. (Logistics solution continued)**

We can strengthen the formulation for our simple logistics problem by creating the parallel composition of the automata $G_{package(p1)}$ and $G_{truck(t1)}$. The parallel composition generates the new automaton shown in Figure 6.3.5. The bold arcs indicate the solution as found by the formulation. From the figure, it should be clear that by generating the parallel composition we can formulate flow constraints over the new automaton, some of which may be violated by the

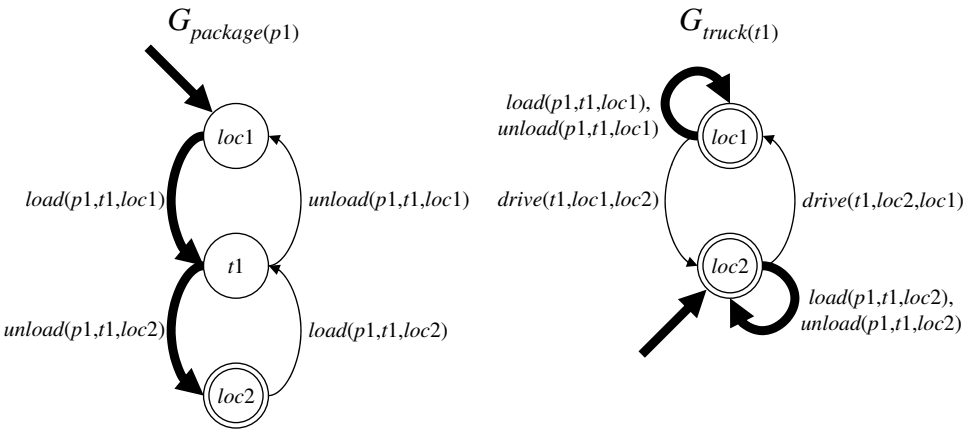


Fig. 30. Solution to the simple logistics example.

current solution. In this particular example, we can observe that the flow constraint is violated at the following nodes: $(loc1, loc2)$, $(loc1, loc1)$, $(t1, loc1)$, $(t1, loc2)$, and $(loc2, loc2)$. By adding these constraints to our existing formulation we obtain the solution where $x_{drive(t1,loc2,loc1)} = x_{load(p1,t1,loc1)} = x_{drive(t1,loc1,loc2)} = x_{unload(p1,t1,loc2)} = 1$, which happens to translate into a feasible plan.

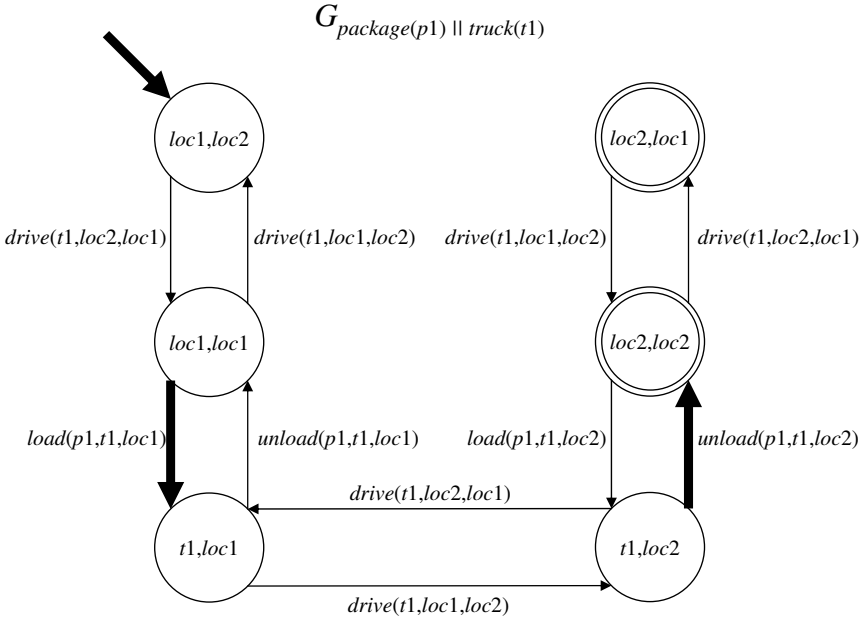


Fig. 31. Solution to the simple logistics example.

# 7 CONCLUSIONS

*I plan on living forever. So far, so good.*

*Unknown author*

Over the last few years, substantial progress has been made in developing effective integer programming formulation for automated planning. By participating in the international planning competition, initial work showed that integer programming-based methods were competitive with more common approaches. The integer programming-based planning system, Optiplan [79], was judged second best performer in four out of seven competition domains of the optimal track for propositional domains. In subsequent work [85, 86], novel integer programming formulations were introduced that incorporated a *branch-and-cut* algorithm, where certain constraints are dynamically generated and added to the formulation. Extensive experimental results showed that the resulting approach significantly outperformed the best available satisfiability based planner. More recently [78], integer programming formulations and strengthening techniques have been developed that can yield heuristic estimates in traditional artificial intelligence-based search techniques. Initial results indicate that this approach holds considerable promise.

Overall, this work has focused on the development of strong integer programming formulations to solve (or support methods that solve) classical planning problems. While the formulations that have been described have substantial differences, they all rely on two key innovations:

- First, planning is represented as a set of interdependent network flow problems, where each network corresponds to one of the state variables in the planning domain. The network nodes correspond to the values of the state variable and the network arcs correspond to the value transitions. The planning problem is to find a path (a sequence of actions) in each network such that, when merged, they constitute a feasible plan. While this idea can be used with any state variable representation, it particularly applies to multi-valued state variables.

- Second, causal considerations are separated from the action sequencing considerations to generalize the common notion of action parallelism based on planning graphs. This concept suggests that it should be possible to arrange parallel actions in any order with exactly the same outcome. By relaxing this condition, new concepts of parallelism can be adopted that significantly improve planning performance.

The combination of these ideas yields a flexible framework that allows for a variety of progressively more general IP formulations, and likewise a more general approach toward reasoning about plan interdependencies.

In summary, this research has developed and analyzed various integer programming approaches for automated planning problems. It has shown that integer programming-based approaches can be effective in solving planning problems and that they can compete with satisfiability-based approaches. Moreover, this research is part of a more general shift in focus in the artificial intelligence community towards handling plan quality.

## REFERENCES


[1] M. Aarup, M.M. Arentoft, Y. Parrod, I. Stokes, H. Vadon, and J. Stader. *Intelligent Scheduling*, chapter Optimum-AIV: A Knowledge-Based Planning and Scheduling System for Spacecraft AIV, pages 451–469. Morgan Kaufmann, 1994.

[2] E. Amir and B Engelhardt. Factored planning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-2003)*, pages 929–935, 2003.

[3] F. Bacchus and A. Grove. Graphical models for preference and utility. In *Proceedings of the 11th Annual Conference on Uncertainty in Artificial Intelligence (UAI-1995)*, pages 3–10, 1995.

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

[5] J. Benton, M.H.L. van den Briel, and S. Kambhampati. A hybrid linear programming and relaxed plan heuristic for partial satisfaction planning problems. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS-2007)*, pages 34–41, 2007.

[6] R.E. Bixby. Solving real-world linear programs: A decade and more of progress. *Operations Research*, 50(1):3–15, 2002.

[7] A. Blum and M. Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-1995)*, pages 1636–1642, 1995.

[8] A. Bockmayr and Y. Dimopoulos. Mixed integer programming models for planning problems. In *Proceedings of the Workshop Constraint Problem Reformulation (CP-1998)*, 1998.

[9] A. Bockmayr and Y. Dimopoulos. Integer programs and valid inequalities for planning problems. In *Proceedings of the 5th European Conference on Planning (ECP-1999)*, pages 239–251. Springer-Verlag, 1999.

[10] C. Boutilier, T. Dean, and S. Hanks. Decision-theoretic planning: Structural assumptions and computations leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[11] R.I. Brafman and Y. Chernyavsky. Planning with goal preferences and constraints. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 182–191, 2005.

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

[13] A.L. Brearley, G. Mitra, and H.P. Williams. Analysis of mathematical programming problems prior to applying the simplex algorithm. *Mathematical Programming*, 8:54–83, 1975.

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

[15] T. Bylander. A linear programming heuristic for optimal planning. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-1997)*, pages 694–699, 1997.

[16] A. Caprara and M. Fischetti. *Annotated Bibliographies in Combinatorial Optimization*, chapter Branch and Cut Algorithms, pages 45–63. John Wiley and Sons, 1997.

[17] C.G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 1999.

[18] M. Cayrol, P. Régnier, and V. Vidal. Least commitment in graphplan. *Artificial Intelligence*, 130(1):85–118, 2001.

[19] S. Chien, R. Knight, A. Stechert, R. Sherwood, and G. Rabideau. Integrated planning and execution for autonomous spacecraft. In *Proceedings of the IEEE Aerospace Conference (IAC-1999)*, pages 263–271, 1999.

[20] G. Desaulniers, J. Desrosiers, and M.M. Solomon, editors. *Column Generation*. Springer, 2005.

[21] Y. Dimopoulos. Improved integer programming models and heuristic search for AI planning. In *Proceedings of the 6th European Conference on Planning (ECP-2001)*, pages 301–313. Springer-Verlag, 2001.

[22] Y. Dimopoulos and A. Gerevini. Temporal planning through mixed integer programming. In *Proceeding of the Workshop on Planning for Temporal Domains (AIPS-2002)*, pages 2–8, 2002.

[23] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in nonmonotic logic programs. In *Proceedings of the 4th European Conference on Planning (ECP-1997)*, pages 167–181, 1997.

[24] M.B. Do, J. Benton, M.H.L. van den Briel, and S. Kambhampati. Planning with goal utility dependencies. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-2007)*, pages 1872–1878, 2007.

[25] M.B. Do and S. Kambhampati. Solving planning graph by compiling it into a CSP. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000)*, pages 82–91, 2000.

[26] M.B. Do and S. Kambhampati. SAPA: A domain-independent heuristic metric temporal planner. In *Proceedings of the 6th European Conference on Planning (ECP-2001)*, pages 109–120, 2001.

[27] S. Edelkamp. Efficient planning with state trajectory constraints. In J. Sauer, editor, *Proceedings Workshop Planen, Scheduling und Konfigurieren / Entwerfen*, pages 89–99, 2005.

[28] S. Edelkamp and M. Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. In *Proceedings of the 5th European Conference on Planning (ECP-1999)*, pages 135–147. Springer-Verlag, 1999.

[29] S. Edelkamp and M. Helmert. On the implementation of MIPS. In *Proceedings of Workshop on Model-Theoritic Approaches to Planning, Artificial Intelligence Planning and Scheduling (AIPS-2000)*, pages 18–25, 2000.

[30] R.E. Fikes and N.J. Nilsson. Strips: A new appraoch to the application of theorem-proving to problem solving. *Artificial Intelligence*, 2(3):189–208, 1971.

[31] R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, Belmont, CA, 1993.

[32] M. Fox and D. Long. PDDL2.1: An extension of PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20, 2003.

[33] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[34] M Grötschel, M. Jünger, and G. Reinelt. A cutting plane algorithm for the linear ordering problem. *Operations Research*, 32:1195–1220, 1984.

[35] P. Halsum and H. Geffner. Heuristic planning with time and resources. In *Proceedings of the 6th European Conference on Planning (ECP-2001)*, pages 121–132, 2001.

[36] M. Helmert. The Fast Downward planning system. *Journal of Artifical Intelligence Research*, 26:191–246, 2006.

[37] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier. Spudd: Stochastic planning using decision diagrams. In *Proceedings of the 15th Annual Conference on Uncertainty in Artificial Inteligence (UAI-1999)*, pages 279–288, 1999.

[38] J. Hoffmann. Extending FF to numerical state variables. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-2002)*, pages 571–575, 2002.

[39] S. Hölldobler and H-P. Störr. Solving the entailment problem in the fluent calculus using binary decision diagrams. In *Proceedings of Workshop on Model-Theoritic Approaches to Planning, Artificial Intelligence Planning and Scheduling (AIPS-2000)*, 2000.

[40] J.N. Hooker. A quantitative approach to logical inference. *Decision Support Systems*, 4:45–69, 1988.

[41] ILOG Inc., Mountain View, CA. *ILOG CPLEX 8.0 user's manual*, 2002.

[42] E.L. Johnson, G.L. Nemhauser, and M.W.P. Savelsbergh. Progress in linear programming based branch-and-bound algorithms: An exposition. *INFORMS Journal on Computing*, 12:2–23, 2000.

[43] S. Kambhampati. Challenges in bridging plan synthesis paradigms. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-1997)*, pages 44–49, 1997.

[44] S. Kambhampati and commentary from other planning researchers. On the suboptimality of optimal planning track at ipc 2006. http://raos-ruminations.blogspot.com/2006/07/on-suboptimality-of-optimal-planning.html, July 2006.

[45] F. Kanehiro, M. Inaba, H. Inoue, H. Hirukawa, and S. Hirai. Developmental software environment that is applicable to small-size humanoids and life-size humanoids. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2001)*, pages 4084–4089, 2001.

[46] H. Kautz. SATPLAN04: Planning as satisfiability. In *Working Notes on the International Planning Competition*, pages 44–45, 2004.

[47] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI-1992)*, 1992.

[48] H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-1995)*, pages 318–325, 1995.

[49] H. Kautz and B. Selman. SATPLAN04: Planning as satisfiability. In *Working Notes on the 5th International Planning Competition*, pages 45–46, 2006.

[50] H. Kautz and J.P. Walser. State-space planning by integer optimization. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-1999)*, pages 526–533, 1999.

[51] J. Koehler. Planning under resource constraints. In *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-1998)*, pages 489–493, 1998.

[52] F. Lamiraux, S. Sekhavat, and J. Laumond. Motion planning and control for Hilare pulling a trailer. *IEEE Transactions on Robotics and Automation*, 15(4):640–652, 1999.

[53] D. Long and M. Fox. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*, 20:1–59, 2003.

[54] K. Martin. Using separation algorithms to generate mixed integer model reformulations. *Operations Research Letters*, 10:119–128, 1991.

[55] N. Muscettola, P. Nayak, B. Pell, , and B. Williams. Remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence*, 103(1-2):5–47, 1998.

[56] N. Muscettola, B. Smith, C. Fry, S. Chien, K. Rajan, G. Rabideau, and D. Yan. On-board planning for new millennium deep space one autonomy. In *Proceedings of the IEEE Aerospace Conference (IAC-1997)*, volume 1, pages 303–318, Snowmass at Aspen CO, 1997.

[57] D.S. Nau, W.C. Regli, and S.K. Gupta. AI planning versus manufacturing-operation planning: A case study. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-1995)*, pages 1670–1676, 1995.

[58] A. Newell and H. Simon. *Computers and Thought*, chapter GPS, a Program that simulates human thought, pages 279–293. McGraw Hill, NY, 1963.

[59] X. Nguyen, S. Kambhampati, and R. Nigenda. Planning graph as the basis for deriving heuristics for plan synthesis by state space and (csp) search. *Artificial Intelligence*, 135:73–123, 2001.

[60] M.W. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33:60–100, 1991.

[61] Y. Pochet and L.A. Wolsey. *Combinatorial Optimization: Papers from the DIMACS Special Year*, volume 20 of *DIMACS Series in Discrete Mathematics and Computer Science*, chapter Algorithms and reformulations for lot-sizing problems, pages 245–294. American Mathematical Society, 1995.

[62] G. Rabideau, B. Engelhardt, and S. Chien. Using generic preferences to incrementally improve plan quality. In *Proceedings of the 2nd NASA International Workshop on Planning and Scheduling for Space*, pages 11–16, 2000.

[63] I. Refandis and Ioannis Vlahavas. Heuristic planning with resources. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-2000)*, pages 521–525, 2000.

[64] J. Rintanen. A planning algorithm not based on directional search. In *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR-1998)*, pages 617–624, 1998.

[65] J. Rintanen, K. Heljanko, and I. Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. Technical Report 216, Albert-Ludwigs-Universität Freiburg, , Institut für Informatik, 2005.

[66] J. Rintanen, K. Heljanko, and I. Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12):1031–1080, 2006.

[67] M.D. Rodríguez-Moreno, D. Borrajo, and D. Meziat. An AI planning-based tool for scheduling satellite nominal operations. *AI Magazine*, 25(4):9–27, 2004.

[68] E.D. Sacerdoti. Planning in a hierachy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[69] E.D. Sacerdoti. The nonlinear nature of plans. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence (IJCAI-1975)*, pages 206–214, 1975.

[70] R. Sanchez and S. Kambhampati. Altalt-p: Online paralelization of plans with heuristic state search. *Journal of Artificial Intelligence Research*, 19:631–657, 2003.

[71] M.W.P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming. *ORSA Journal on Computing*, 6(4):445–454, 1994.

[72] D. Smith. Choosing objectives in over-subscription planning. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 393–401, 2004.

[73] T.C. Son and E. Pontelli. Planning with preferences using logic programming. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 247–260, 2004.

[74] B. Srivastava, S. Kambhampati, and M.B. Do. Planning the project management way: Efficient planning by effective integration of causal and resource reasoning in realplan. *Artificial Intelligence*, 131(1-2):73–134, 2001.

[75] A. Tate. Generating project networks. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI-1997)*, 1977.

[76] P. Tompkins, A. Stentz, and D. Wettergreen. Global path planning for mars rover exploration. In *Proceedings of the IEEE Aerospace Conference (IAC-2004)*, pages 801–815, 2004.

[77] P. van Beek and X. Chen. CPlan: A constraint programming approach to planning. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-1999)*, pages 585–590, 1999.

[78] M.H.L. van den Briel, J. Benton, S. Kambhampati, and T. Vossen. An LP-based heuristic for optimal planning. In *Proceedings of the 13th International Conference of Principles and Practice of Constraint Programming (CP-2007)*, pages 651–665, 2007.

[79] M.H.L. van den Briel and S. Kambhampati. Optiplan: Unifying IP-based and graph-based planning. In *Working Notes on the International Planning Competition*, pages 18–20, 2004.

[80] M.H.L. van den Briel and S. Kambhampati. Optiplan: Unifying IP-based and graph-based planning. *Journal of Artificial Intelligence Research*, 24:623–635, 2005.

[81] M.H.L. van den Briel, S. Kambhampati, and T. Vossen. Planning with numerical state variables through mixed integer programming. In *Proceedings of the Poster Session (ICAPS-2005)*, pages 5–8, 2005.

[82] M.H.L. van den Briel, S. Kambhampati, and T. Vossen. Planning with preferences and trajectory constraints by integer programming. In *Proceedings of the Workshop on Preferences and Soft Constraints in Planning*, pages 19–22, 2006.

[83] M.H.L. van den Briel, S. Kambhampati, and T. Vossen. Fluent merging: A general technique to improve reachability heuristics and factored planning. In *Proceedings of the Workshop Heuristics for Domain-Independent Planning: Progress, Ideas, Limitations, Challenges (ICAPS-2007)*, 2007.

[84] M.H.L. van den Briel, R. Sanchez, M.B. Do, and S. Kambhampati. Effective approaches for partial satisfaction (oversubscription) planning. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-2004)*, pages 562–569, 2004.

[85] M.H.L. van den Briel, T. Vossen, and S. Kambhampati. Reviving integer programming approaches for AI planning: A branch-and-cut framework. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS-2005)*, pages 161–170, 2005.

[86] M.H.L. van den Briel, T. Vossen, and S. Kambhampati. Loosely coupled formulations for automated planning: An integer programming perspective. *Journal of Artificial Intelligence Research*, 31:217–257, 2008.

[87] T. Vossen, M. Ball, A. Lotem, and D.S. Nau. On the use of integer programming models in AI planning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-1999)*, pages 304–309, 1999.

[88] S.A. Wolfman and D.S. Weld. The LPSAT engine and its applicationto resource planning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-1999)*, pages 310–317, 1999.

[89] L.A. Wolsey. *Integer Programming.* John Wiley and Sons, 1998.