Domain-Independent Cost-Optimal Planning in ASP

by

David Spies

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

We examine various techniques in SAT-based (Satisfiability) planning and explore how they can be applied and further improved in the context of ASP (Answer Set Programming). First, we look at the 2006 planner SATPlan and show that their encoding, when translated directly into ASP, enjoys a significantly reduced search space over the very same encoding in SAT. Next we tackle the problem of reducing the large encoding size which prohibits the best SAT-based planners from handling the larger planning problems that appear in planning competitions. In particular we give a simple encoding for reducing the number of expressions required to handle action-mutex constraints. After that we tackle the problem of how to reduce the size of fluent-mutex constraints and show that this can be done far more effectively than in previous work by covering the mutex-graph with multicliques (complete multi-partite subgraphs). Finally we address the problem of cost-optimal planning in ASP. While most attempts at planning in SAT have been either cost-agnostic or at best constrained to planning at a fixed-makespan, those that have attempted to handle cost-optimal planning at any makespan introduce significant space-overhead, to the point of prohibiting them from tackling larger planning problems. We show that by exploiting stable model semantics, we can produce a far more space-efficient ASP-planner which guarantees makespan-agnostic cost-optimality. Using lessons learned from this, we develop an entirely new and different approach to planning, stepless planning which reduces the grounded

problem space even more and relies heavily on stable model semantics for its correctness.

Preface

This thesis was written with significant help from my supervisor Jia You, who added references and short descriptions of background material including significant revisions to the introduction. He continually helped me find places that needed further explanation and clarification, pointed me to lots of great resources, and helped me work through many of the problems I encountered along the way.

Acknowledgements

Thank you to Professor Jia You, my supervisor who helped me through writing this and who taught me so much over the time I spent at the University of Alberta. Then three years later after I'd left the university and all but given up on completing my MSc, he reached out to me and offered to really help me finish this. I cannot overstate my appreciation and respect for all his help, support, kindness, and encouragement.

Also, thank you to my co-supervisor Professor Ryan Hayward for his help and support.

I also want to thank Professors Martin Mueller, Michael Buro, Zach Friggstad and all my friends in the UofA Programming Club.

Thank you to the Department of Computing Science and the University of Alberta for providing me this opportunity to learn and research in a stimulating and supportive environment.

Contents

Chapter 1

Introduction

Planning is one of the most celebrated research problems in artificial intelligence for building goal-oriented intelligent artifacts. Simply put, planning is the problem of finding a sequence of actions that lead from some initial state to a goal state. A planning problem is described by a collection of available actions, each with some preconditions and effects, an initial state, and final state. The STRIPS planning framework [13] provides the basis for expressing domain-independent planning problems in an appropriate action language. Planning in this form is often called offline planning as compared to more complex planning problems in dynamic environments. The book by Ghallab et al. [24] gives an excellent account of the theory and practice of solving planning problems. For empirically evaluating state-of-the-art planning systems, the International Planning Competition (IPC) has been carried out as an event in the context of the International Conference on Automated Planning and Scheduling (ICAPS).

Two of the successful approaches to planning are GraphPlan [5] and SATPlan [28], [29] (also see [33], [37], [46]). The former is based on searching a state-graph where nodes are collection of fluents and edges are actions. A number of systems based on GraphPlan have been implemented as open source (check, e.g., GraphPlan in Wikipedia). SATPlan is based on the idea of planning as satisfiability - formulating a planning problem by a collection of propositional clauses and using an efficient SAT solver to search for a plan. Once a satisfying truth value assignment is found, a plan can be extracted

from it.

Along with boolean satisfiability and the advancement of its solver techniques, Answer Set Programming (ASP) [21] (also see, [3], [11], [12], [20], [21], [30], [42]) has emerged as a prominent approach to declarative problem solving and knowledge representation. ASP provides a paradigm for uniformly encoding NP-complete problems as well as problems in the second level of the polynomial hierarchy, namely Σ_2^P-hard problems, by supporting disjunctive rules. Compared to other approaches, ASP has some unique advantages, e.g., it allows cardinality constraints and constraints requiring that solutions be acyclic to be encoded compactly and handled efficiently. Highly competitive ASP solvers already (e.g., [14], [16], [19]), are used in applications in molecular biology, planning and scheduling, and solving puzzles and games.

This report investigates a new approach to domain-independent planning in ASP. Current ASP planners such as PLASP [15] do not use nearly as sophisticated techniques as modern SAT planners. In this report, we start by bringing ASP planning up to speed, translating the rules presented in [28] into ASP. We further find ways to reduce the large instance size by analyzing the mutex graph, improving on techniques presented in [36]. Finally, we explore the domain of cost-optimal planning, basing our encoding on techniques presented in [40], but exploiting stable model semantics to create a much more efficient, simple, and successful cost-optimal planner.

The next few chapters present the background material necessary to understand this report.

Chapter 2 introduces SAT (Boolean Satisfiability) as a paradigm for encoding and solving problems. We briefly overview modern techniques for SAT-solving and explore the question of how and when to reduce the size of an encoding in a way that would be beneficial to the solver.

Chapter 3 introduces ASP (Answer Set Programming) as a paradigm for encoding and solving problems. We first look at grounded ASP in the SModels format [42] and discuss the role *stable models* play in an encoding. We then present enough of the higher-level ASP-Core-2 language [7] to understand the

programs presented in this paper. This chapter finishes with an example and a note about iterative solving in ASP.

Chapter 4 provides a brief overview of STRIPS planning. STRIPS is a paradigm for encoding a planning problem so that it may be solved by any domain-independent planner. We write a short naive ASP planner to illustrate the input that is assumed in this paper. This encoding is then revised to allow for parallel actions.

Chapter 5 creates the planner ASPPlan by translating the rules from SATPlan directly into ASP. We then make some observations about the resulting encoding and some optimizations.

Chapter 6 introduces a class of graphs called multicliques and shows how to reduce the number of clauses used to express mutex relations in a planning problem by finding a multiclique-covering of the mutex graph.

Chapter 7 brings the entire paper together to produce our first cost-optimal planner. We first examine how to tell when an instance is unsolvable by restricting the set of candidate solution plans to those that "make progress", and then use this to create a cost-optimal planner, AASPPlan. We then look at the problem of delete-free planning and use stable models to generate a linear-size encoding. This is then appended to AASPPlan to serve as a heuristic "suffix" layer.

Chapter 8 applies the same idea as described in Chapter 7, which enables us to create our second cost-optimal planner, SteplessPlan. As the name suggests, in this planner there is no notion of layers or makespan. To the best of our knowledge, this is the first time that cost-optimal planning is solved by a stepless planner in a logic-based language.

Chapter 9 reports experimental results and compares with those of Robinson et al. [40], the only work that we can find that addresses *globally*[1] cost-optimal planning in a logic-based approach. Our early expectation of stepless planning was that the difficulty of the problems involved and the use of disjunctive ASP would make it underperformant, but our experiments showed

[1] As opposed to for a fixed makespan

that this expectation was wrong and that in fact stepless planning may have significant practical implications.

Chapter 10 provides a summary and discusses future directions. Indeed, this thesis is a first step in using ASP to encode cost-optimal planning problems. To make it practical many more issues need to be investigated.

Appendix A presents a proof that the various definitions of multicliques given in Chapter 6 are equivalent and gives an extra definition.

Appendix B gives a more compact representation of mutual-exclusion cliques in SAT than exists in the literature.

Appendix C lists encodings of key ASP planners used in this paper.

Chapter 2

Boolean Satisfiability (SAT)

In Boolean Satisfiability, a given problem is encoded by a collection of clauses, say Γ (such a collection is called a *SAT instance*, or just a *program*), and the problem has a solution if and only if there is a truth value assignment that satisfies Γ. A satisfying truth value assignment can be interpreted to provide a solution to the given problem. In this context, a clause is a disjunction of literals, where a literal is either a proposition or the negation of a proposition. In the literature, such a proposition is often called a *variable*. SAT is known to be NP-complete.

2.1 Hamiltonian Path as SAT

Let us start by examining a simple problem, the Hamiltonian-Path problem. Given a graph $G = (V, E)$, imagine we wish to represent the problem "find a hamiltonian path on G" as a Boolean Satisfiability instance. Recall that a hamiltonian path on a graph (directed or undirected) is a path in the graph that visits each vertex exactly once.

The simplest and most common format for encoding problems in SAT is DIMACS CNF format. In DIMACS format, every variable is a positive integer and a rule is a single 0-terminated line of negative (for negated variables) and positive integers denoting a disjunction of literals. So, for instance, the expression $((a \vee b) \wedge (\neg b \vee c))$ might be represented as:

```
c A simple DIMACS SAT instance
```

```
p cnf 3 2
1 2 0
-2 3 0
```

In this example, the first line is a comment and the second line indicates the number of variables and clauses (two and three respectively). The last two lines are the clauses themselves. The variables are numbered 1, 2, and 3 (corresponding to a, b, and c respectively). Every (disjunctive) clause sits on one line and is 0-terminated. Where a number is negative, this indicates a negated variable.

This format is simple and easy for a program to parse. However, one would not write a serious SAT instance in this format by hand. For that, you must write code to generate a CNF file.

Here is a SAT description of the problem "find a hamiltonian path on graph $G = (V, E)$ with $|V| = n$". We include n^2 boolean variables labeled $x_{(v,k)}|\,(v \in V, 1 \leq k \leq n)$ (where each vertex is represented by a number) to indicate the solution includes vertex v at position k. The constraints are as follows:

1. Each vertex has at most one position:

   $\bigwedge_{v \in V, j \in 1 \ldots n, k \in 1 \ldots n, j<k} (\neg x_{v,j} \vee \neg x_{v,k})$

2. Every vertex has at least one position:

   $\bigwedge_{v \in V} (\bigvee_{(k \in 1 \ldots n)} x_{v,k})$

3. If vertex v takes position $k > 1$, then some neighbor of v takes position $k - 1$:

   $\bigwedge_{v \in V, k \in 2 \ldots n} (\neg x_{v,k} \vee \bigvee_{(w \in V)} x_{w,k-1} | (w, v) \in E)$

4. Some vertex has position n

   $\bigvee_{v \in V} x_{v,n}$

This is a fairly straightforward program which certainly encodes the Hamiltonian Path program as a SAT instance. It is probably the simplest solution, but (as we shall see) not the most efficient.

2.2 Solving Techniques

This thesis focuses primarily on how to optimize problems before they reach the solver. Ideally, we want to express each problem in a way that simply encodes everything we know about the problem, while at the same time uses as few variables/clauses as possible. Thus, we are not particularly concerned with what happens once the program actually reaches the solver. In general, larger encodings may not necessarily be harder or easier. Sometimes a larger encoding includes redundant code that serves as extra constraints thus guiding the search more effectively. Furthermore, the heuristic used to select decision literals can matter greatly as well. But for the purposes of this thesis, we will assume it is always beneficial to reduce the big-$\mathcal{O}$ complexity of the encoding size.

This is not too strong an assumption. For random SAT instances, there is a well-known phenomenon called *phase transition* [22], [23] (similarly for random programs in ASP [47]). But for structured problems, the sheer size of an encoding can significantly influence the reasoning performance. One of the reasons is that it is known that a modern SAT solver typically spends 80-90% of its running time on computing unit propagation [46], which in theory takes linear time in the size of the underlying SAT instance but in practice can significantly impact the performance of a SAT solver. While it is indeed possible that a SAT or ASP solver might more quickly solve instances generated by an encoding that requires $2n$ clauses than one that requires n clauses, it is unlikely that the solver would perform even faster on an encoding that requires n^2 clauses, as linear time on unit propagation actually becomes quadratic time. One piece of evidence to support this argument is that modern SAT solvers have removed the process of *lookahead*, which though may assign some vari-

ables without search, its quadratic running time for each invocation causes too much overhead.

Modern complete SAT solvers use a technique known as Conflict-Directed Clause Learning (CDCL) [46]. This is an extension of the earlier algorithm known as Davis Putnam Logemann Loveland (DPLL) [9]. We will describe DPLL in detail.

2.2.1 DPLL

Here we give a brief overview of DPLL and then follow that up with a description of CDCL.

DPLL maintains a partial assignment while performing a backtracking search. Briefly, DPLL can be summarized as:

1. Initialize the search with an empty partial assignment (no variables are assigned).

2. Select a boolean variable according to some *heuristic*. The efficiency of the algorithm depends heavily on the choice of variable which is why better heuristics can make a big difference in how well the solver performs. One popular heuristic is VSIDS (Variable State Independent, Decaying Sum) [34].

3. Assign this variable either True or False (again according to the heuristic).

4. Perform *Unit Propogation* and *Pure Literal Elimination* assigning all variables which are fixed by the current partial assignment. Eliminate all satisfied clauses.

5. If a conflict is encountered, backtrack until we reach a variable for which we have not tried both values (True and False). If no such variable is found, return "No Solution". Otherwise flip the value of the variable found and return to step 4.

6. If there are more unsatisfied clauses go back to step 2 selecting another variable.

7. Otherwise, assign the remaining variables arbitrarily and return the resulting assignment as a solution.

Step 4 mentions *Unit Propogation* and *Pure Literal Elimination*. These two operations take a partial assignment together with a collection of clauses, and augment that assignment with more literals that can be inferred from the current partial assignment. Additionally, Unit Propogation may result in a "conflict", which indicates that there is no solution using the current partial assignment (resulting in backtracking in step 5).

- *Pure Literal Elimination*: For any boolean variable x, if all the clauses containing x use it only as a positive literal, then assign x to True. Similarly, if x is used in the program only as a negative literal $\neg x$ then assign it to false. Clearly, this will never cause any conflicts. Then eliminate any clauses containing the variable x since they are now satisfied.

- *Unit Propogation*: Take all singleton clauses and assign the corresponding variables accordingly. That is, if we have some clause x, then we assign the variable x to be true. Similarly, if we have a clause $\neg x$, we assign x to be false. Now wherever the variable x occurs in another clause, if it occurs with the opposite sign we eliminate x from the clause, and if it occurs with the same sign we eliminate the clause entirely since it is now satisfied. If we ever create an empty clause in this way or assign the same variable to be both True and False, this indicates we have reached a conflict and must backtrack (as per step 5).

2.2.2 CDCL

CDCL (Conflict Directed Clause Learning) [46] behaves similarly to DPLL except in the way it handles conflicts. In CDCL, whenever a conflict is encountered, rather than simply backtracking, we use information about the

conflict to construct a new clause and add it to the global list of clauses. After doing this the solver *backjumps* to the last point at which our new clause would have taken us in a different direction. In this way, it is possible to skip over deciding variables whose choice would not affect the current conflict.

2.3 Encoding Efficiency

There are plenty of metrics by which we can determine the efficiency of a particular encoding. The most obvious ones are of course, *number of variables* and *number of clauses*. Besides this, two other valuable metrics we might use are *total number of literals among all the clauses* and *number of* ≥ 3*-literal clauses*.

With respect to these four metrics, let us consider the SAT encoding of Hamiltonian path given in Section 2.1. Given a graph with n vertices and m edges, our three metrics over the last problem give:

- Variables: $n^2 \in \theta(n^2)$
- Clauses: $n\binom{n}{2} + 2n + 1 \in \theta(n^3)$
- Literals: $2n\binom{n}{2} + n^2 + 2n + m \in \theta(n^3)$
- Non-binary clauses: $2n + 1 \in \theta(n)$

This can be broken down as follows:

1. First we have $n\binom{n}{2}$ binary clauses asserting a vertex has no more than one position.
2. Next we have n clauses, one for each vertex, each of length n which assert that the vertex must take some position. (Literals: n^2)
3. Then, $n\,(n-1)$ clauses, $n-1$ for each vertex v, all of length $\deg(v)+1$ asserting that v is connected to its predecessor. (Literals: $(n-1)\,(2m+n)$)
4. Finally, a single length-n clause ensuring that the path contains a last element.

Looking at rule 1, we can recognize each cluster of $\binom{n}{2}$ binary clauses as a collection of mutual exclusion constraints on a clique of size n. As illustrated in [36], given such a clique, we can exploit a trade-off between auxilliary variables and clauses to get a more compact representation. In particular [36] points to two other representations; one which uses $\lceil \log_2 n \rceil$ extra variables and $n \lceil \log_2 n \rceil$ binary clauses, and another which uses $n-1$ extra variables and only $3n-4$ binary clauses. In Appendix B, we present an alternative encoding which is a slight improvement on this linear representation using only $3n-6$ binary clauses and $\left\lceil \frac{n}{2} \right\rceil - 2$ auxiliary variables.

Substituting this last encoding for ours in rule 1 above, we now use a total of $n\lfloor \frac{3n}{2} - 2 \rfloor$ variables over only $n(4n-6)$ clauses, totaling $n(8n-13)+2m(n-1)$ literals (of course the number of non-binary clauses is unchanged).

Chapter 3

Answer Set Programming

Although it is ostensibly the premise of this whole paper, comparing ASP encodings with SAT encodings may be like comparing apples and oranges. They are, ultimately, two different paradigms employed to solve NP-hard problems. There are two distinctions we must recognize, one which may be seen as purely semantic and the other which is rooted deeply in the underlying theory of ASP.

Let us start with the semantic distinction. An ASP program (or more specifically ASP-Core-2 as we use in this paper [43]) is not a collection of rules and variables as is SAT, but it is rather a higher-level language for generating such a collection. This is why a program written in ASP can be more properly compared with a SAT generator than with the SAT instance it generates.

If we want something we can compare with SAT, we should instead be comparing the SModels format which is the language of a program generated by a standard grounder (most popularly LPARSE or GRINGO) when we run it on an ASP program. As with DIMACS CNF format, the SModels language is intended primarily to be both generated and parsed by a program, and not coded by hand except perhaps in dealing with toy instances. This is, of course, what we really mean when we make claims about the comparitive efficiency of a SAT encoding and an ASP encoding. We are referring to the class of SModels programs the ASP code can generate, not the actual ASP code itself.

The deeper reason is that an ASP solver must accept only well-supported models. It is solving an inherently more difficult problem than a SAT solver. It is not clear whether this translates to asymptotically slower solving times. In

practice, ASP solvers tend to be a few years behind SAT solvers performance-wise, but this is generally thought to be primarily because there is a much larger community of researchers and developers working on SAT than on ASP; not because of any inherent differences between the two paradigms.

3.1 ASP

3.1.1 Overview

An ASP program passes through multiple stages on the way to finding a solution. Initially, we write code in the full ASP language using variables and numeric operations and constructs. Then a grounder (such as LPARSE or GRINGO) is run to produce the *grounded* program. This program is expressed in a simpler language with no arithmetic or variables and only atomic propositions. A solver such as SMODELS or CLASP is then run on this to find solutions. We'll start by describing what a grounded program looks like and then back up and explain about variables.

3.1.2 Basic Rules

This section briefly explores the formal SModels language. For further details, the reader is referred to [43] and [7]. Note, we will not actually look at SModels code, but rather at the restricted subset of ASP-Core 2 rules which bear a one-to-one mapping to rules in SModels input format. In particular, this means we will avoid variables. Alternatively, this can be seen as the language of programs which are output by GRINGO when run with the –text option.

A grounded ASP/SModels program is in many respects similar to a SAT instance. Both types of programs can be seen as a collection of atoms, A, and constraints, R, for which we wish to answer the question, "Does there exist an assignment of truth values, $\mu \in A \to \{T, F\}$ which satisfies R?" All constraints available in SAT are also available in ASP. However, ASP additionally allows the user the ability to encode some constraints which cannot be concisely expressed by a SAT instance.

An atom in ASP Core-2 is written as a well-parenthesized expression in which each open-parentheses is preceded by a function name and the leaves are terms (the outermost function is called a predicate). Function/predicates names must all begin with a lowercase letter; terms may be either words beginning with a lower-case letter or integers. The following are all examples of atoms in ASP

```
a
bluebird(cavern)
cartiledge(17)
my(hovercraft(is,full),of(eels))
```

An ASP program consists of a set of rules which express relations between atoms. Every ASP rule has a body and a head. A *basic rule* is expressed as:

$$h \leftarrow b_1^+, b_2^+, \ldots, b_m^+, \neg b_1^-, \ldots, \neg b_n^-$$

Here h is an atom which is the head of the rule. $b_1^+, \ldots, b_m^+$ are the atoms which make up the positive part of the body of r, and $b_1^-, \ldots, b_n^-$ are atoms which make up the negative part of the body of r.

The actual programs that run in an ASP system are slightly different in syntax from the above definitions. The main difference is that we write the symbols `:-` for $\leftarrow$, and every rule ends with a period. Lines beginning with % are comments. In the following discussions, we may use both notations without confusion.

The following are examples of basic rules in ASP (Although ASP allows separating basic rule body atoms with both ';' and ',', we will use ';' throughout this paper, since ',' is overloaded and has a different meaning in more complex rules).

```
sidewalk(wet) :- raining.
day(nice) :- not cloudy; day_of_week(saturday).
```

```
impossible :- raining; not cloudy.
pigs(flying) :- impossible.
```

The meaning of $\leftarrow$ in this context is not quite as straightforward as in the language of boolean formulas. In SAT, $a \leftarrow b$ is syntactic sugar for $a \vee \neg b$. In ASP, $a \leftarrow b$ additionally asserts that a is *supported* if b is true. Any solution M (which is a set of ground atoms called a *stable model*) to an ASP instance must satisfy two criteria:

- M satisfies all clauses in P.
- Every true atom in M is *well-supported*.

There are a couple of ways we can define well-supportedness. We will first define the *positive reduct* of a program together with a candidate model [7].

Definition 1. Given an ASP instance P and a boolean assignment M to atoms appearing in P, the *positive reduct* of M with respect to P, denoted by P_M^+, is obtained by:

For each rule $r \in P$ where $r = h \leftarrow b_1^+, \ldots, b_m^+, \neg b_1^-, \ldots, \neg b_n^-$, if $M\left(b_i^-\right) =$ False for all b_i^- then $r' \in P_M^+$ where $r' = h \leftarrow b_1^+, \ldots, b_m^+$.

In other words r' is contained in the positive reduct if and only if the negative part of its body is satisfied by M.

Definition 2. Given a program P, a *stable model* M of P, also called an *answer set* of P, is a $\subset$-minimal model of P_M^+. A stable model is also said to be *well-supported*.

When a model M is not a stable model of program P, it must be the case that it is not a minimal model of P's positive reduct. This means that there exists some subset $M^- \subsetneq M$ such that M^- satisfies P_M^+. Any atom $a \in M \setminus M^-$ is called *unsupported* in M.

To illustrate the basic idea of this definition, let us consider a propositional basic program P that consists of two rules

$$a \leftarrow not\ b$$
$$b \leftarrow b$$

When *not* is treated as classic negation and $\leftarrow$ as classic local implication, P has three models, $M_1 = \{a\}$, $M_2 = \{a, b\}$, and $M_3 = \{b\}$. M_1 is a stable model, as it is clearly a (subset) minimal model of $P^+_{M_1} = \{a \leftarrow,\ b \leftarrow b\}$. The next two models are not stable. For example, M_2 is not a minimal model of $P^+_{M_2} = \{b \leftarrow b\}$, since $\emptyset$ is a model of $P^+_{M_2}$.

Intuitively, well-supportedness requires that every atom in a stable model be supported by a *non-circular* derivation. In the example above, atom b can never be supported in this way and it thus is not in any stable model of the program. There is also a view of *dependency*: the head atom b depends on itself, and thus a *positive loop* is formed. As another example, consider the program $\{a \leftarrow b,\ b \leftarrow a\}$, where a and b together form a positive loop. *Loop formulas* have been studied extensively in ASP (e.g., see [31], [32], [45]).

The orientation of $\leftarrow$ is important in stable model semantics. For example, $a \leftarrow not\ b$ has no relation with $not\ a \rightarrow b$. The mechanism of positive reduct re-enforces that this is not a possible interpretation.

Another feature in ASP is to deny a possible solution. In general, we can write a rule in a program, say

$$f \leftarrow p, not\ f$$

to express that there is no solution (stable model) that contains p. One can verify that, if p is in a solution, we will literally have the rule, $f \leftarrow not\ f$, for which we are running into something like Barber's Paradox: f is in a stable model (thus it should be derivable) iff f is not in it. In ASP, for convenience, a constraint like the above is written as

$$\leftarrow p$$

It's worth noting that using just basic rules, we can already model NP-complete problems. This follows from the fact that SAT problems are expressible in ASP. Given a collection of atoms, $S = \{a_1, \ldots, a_n\}$, the following basic propositional program can be used to compute any subset of S: for all $1 \leq i \leq n$,

$$a_i \leftarrow not\ a'_i$$
$$a'_i \leftarrow not\ a_i$$

where a_i is new proposition intuitiveluy representing the opposite of a_i. For example, if $n = 3$, then $\{a_1, a'_2, a'_3\}$ is a stable model representing that a_1 is in the subset while a_2 and a_3 are not.

Note: In practice with other rule-types available, this encoding is unnecessary and generally inefficient. Instead we can make use of choice rules (see rule 3 in Section 3.1.3).

Now, let us consider a program that actually runs in any implemented ASP system. This program represents a system of wall-sockets, surge protectors, and lightbulbs. Each surge protector can be switched into safe-mode which means it does not provide power. Otherwise it provides power to anything plugged into it so long as it is itself powered.

```
% There is one wall-socket sock(s), which is always powered
%rule: r1
powered(sock(s)).

% We have 4 surge-protectors prot(1), prot(2), prot(3), and prot(4).
% prot(1) is plugged into the wall socket which means it is powered
% if sock(s) is powered
% rule: r2
powered(prot(1)) :- powered(sock(s)).

% prot(2) and prot(3) are plugged into prot(1). prot(2) is in safemode
% rule: r3
powered(prot(2)) :- powered(prot(1)); not safemode(prot(1)).
% rule: r4
powered(prot(3)) :- powered(prot(1)); not safemode(prot(1)).
% rule: r5
safemode(prot(2)).

% prot(4) is plugged into prot(2)
% rule: r6
```

```
powered(prot(4)) :- powered(prot(2)); not safemode(prot(2)).

% Finally we plug a lightbulb into each protector, which is on if its
% protector is powered and not in safemode
% rule: r7
on(light(1)) :- powered(prot(1)); not safemode(prot(1)).
% rule: r8
on(light(2)) :- powered(prot(2)); not safemode(prot(2)).
% rule: r9
on(light(3)) :- powered(prot(3)); not safemode(prot(3)).
% rule: r10
on(light(4)) :- powered(prot(4)); not safemode(prot(4)).
```

This instance has only one stable model:

```
powered(sock(s))
safemode(prot(2))
powered(prot(1))
powered(prot(2))
powered(prot(3))
on(light(1))
on(light(3))
```

To verify this is indeed a stable model, let us first examine which rules' bodies are satisfied by this model (we will call these rules "active"). Rule `r1` has an empty body, so it is always active. Rule `r2` is active since its single body atom (powered(sock(s))) is in our candidate solution. Rules `r3` and `r4` are active since `powered(prot(1))` is in our model, but `safemode(prot(1))` is not. Similarly, Rules `r5`, `r7`, and `r9` are active for this model. Rules `r6` and `r8` are not active because `safemode(prot(2))` is in our model (and it is part of the respective negative bodies), Rule `r10` is not active because it has `powered(prot(4))` in the body (which is not in our model)

Now let us look at the positive reduct corresponding to this model (using only the active rules 1,2,3,4,5,7,9):

```
% r1
powered(sock(s)).
% r2
powered(prot(1)) :- powered(sock(s)).
% r3
powered(prot(2)) :- powered(prot(1)).
% r4
powered(prot(3)) :- powered(prot(1)).
% r5
safemode(prot(2)).
% r7
on(light(1)) :- powered(prot(1)).
% r9
on(light(3)) :- powered(prot(3)).
```

We can see that indeed, the head of every rule appears in our model. I leave it to the reader to verify that our model is a minimal solution to this reduced program.

Now let us look at an example where the candidate solution is not minimal. We will re-arrange three of the surge protectors into a different configuration:

```
% r1
powered(sock(s)).
%prot(1) we'll leave plugged into the wall
% r2
powered(prot(1)) :- powered(sock(s)).
%We will plug prot(2) into prot(3)
% r3
powered(prot(2)) :- powered(prot(3)); not safemode(prot(3)).
```

```
%and prot(3) into prot(2)
% r4
powered(prot(3)) :- powered(prot(2)); not safemode(prot(2)).
```

Now consider the candidate solution:

```
powered(sock(s))
powered(prot(1))
powered(prot(2))
powered(prot(3))
```

This generates the positive reduct:

```
% r1
powered(sock(s)).
% r2
powered(prot(1)) :- powered(sock(s)).
% r3
powered(prot(2)) :- powered(prot(3)).
% r4
powered(prot(3)) :- powered(prot(2)).
```

in which every head appears in our candidate model. But we notice that the candidate is not minimal. There is an alternative solution to this program which is a strict subset:

```
powered(sock(s))
powered(prot(1))
```

This is sensible. Surge protectors 2 and 3 form a closed loop (a positive loop). They are not drawing power from anywhere so they cannot be powered.

We will note here (without going into detail) that the restriction on ASP programs that they be well-supported can also be expressed using something known as loop constraints (which is closely related to positive loops that we

mentioned earlier). This allows ASP programs to be expressed as SAT instances (although with potentially an exponential number of rules) [18].

The ASP solver CLASP uses a variant of CDCL on an ASP program together with a subset of the loop constraints for that program. In addition to the usual learned clauses that CDCL generates as it runs, CLASP is also able to generate learned clauses from loop constraints [17].

3.1.3 Other Rules

We will now examine the other types of rules available in grounded ASP. These rules can give a more compact grounded representation to certain types of problems.

We assume we have a program P and a model M

1. Basic rule: $h \leftarrow b_1^+ \dots b_m^+, \neg b_1^- \dots \neg b_n^-$

   Active If: $\left\{b_1^+ \dots b_m^+\right\} \subset M$ and $\left\{b_1^- \dots b_n^-\right\} \cap M = \emptyset$

   Active Constraint: $h \in M$

   Positive Reduct: $h \leftarrow b_1^+ \dots b_m^+$

   The head h may also be a special atom, $\bot$ which is restricted from being in any solution.

2. Cardinality constraint: $h \leftarrow \#\text{count}\left\{b_1^+; \dots; b_m^+, \neg b_1^-; \dots; \neg b_n^-\right\} \geq k$

   Active If: $\left|\left\{b_1^+, \dots, b_m^+\right\} \cap M\right| + \left|\left\{b_1^-, \dots, b_n^-\right\} \setminus M\right| \geq k$

   Active Constraint: $h \in M$

   Positive Reduct: $h \leftarrow \left|b_1^+, \dots, b_m^+\right| \geq d$ (where $d = k - \left|\left\{b_1^-, \dots, b_n^-\right\} \setminus M\right|$)

   A cardinality constraint enforces the restriction "of this set of atoms and negated atoms, if at least k of them are true, then h is true". In the positive reduct, the negated atoms are removed from the body and k is appropriately reduced.

3. Choice rule: $\{h_1, \dots, h_k\} \leftarrow b_1^+, \dots, b_m^+, \neg b_1^-, \dots, \neg b_n^-$

   Active If: $\left\{b_1^+, \dots, b_m^+\right\} \subset M$ and $\left\{b_1^-, \dots, b_n^-\right\} \cap M = \emptyset$

Active Constraint: None

Positive Reduct: $\bigwedge \{h_1, \ldots, h_k\} \cap M \leftarrow b_1^+, \ldots, b_m^+$

Informally, choice rules can provide support for an atom without forcing its truth. Even when the body is true, the head need not be true. If the head is true, the rule will be included in the positive reduct.

4. Weight constraint:

   $h \leftarrow \#\text{sum}\left\{b_1^+ = w_1^+; \ldots; b_m^+ = w_m^+; b_1^- = w_1^-; \ldots; b_n^- = w_n^-\right\} \geq k$

   Active If: $\sum_{i=1}^{m} w_i^+ | b_i^+ \in M + \sum_{i=1}^{n} w_i^- | b_i^- \notin M \geq k$

   Active Constraint: $h \in M$

   Positive Reduct: $h \leftarrow \sum_{i=1}^{m} w_i^+ | b_i^+ \geq d$ (where $d = k - \sum_{i=1}^{n} w_i^- | b_i^- \notin M$)

   A weight constraint behaves like a generalized cardinality constraint. Each atom (or negated atom) is assigned a weight and the head must be true if the sum of the body weights is at least k. As with a cardinality constraint, in the positive reduct the negated atoms are removed and the bound is appropriately reduced

5. Weak constraint: $\bot \approx\!\leftarrow b_1^+, \ldots, b_m^+, \neg b_1^-, \ldots, \neg b_n^-.w@l$

   When a program contains a weak constraint, this indicates to the solver that we are looking for not just any solution, but an optimal solution. A weak constraint is to ASP as a "soft" clause is to partial weighted max-SAT [40]. A weak constraint does not affect the models of a program. Instead, it is an instruction to the solver to search for models in which the body does not hold. w is the imposed cost of ignoring this constraint and l is a position. Minimizing solvers will minimize the sum of the weights of rules of the highest level and then each of the lower ones. (subject to the "strong" constraints imposed by all of the other rules)

6. Disjunctive rule: $h_1 | \ldots | h_k \leftarrow b_1^+ \ldots b_m^+, \neg b_1^- \ldots \neg b_n^-$

   Active If: $\left\{b_1^+ \ldots b_m^+\right\} \subset M$ and $\left\{b_1^- \ldots b_n^-\right\} \cap M = \emptyset$

Active Constraint: $\{h_1 \dots h_k\} \cap M \neq \emptyset$ and M is a minimal model of the reduct P_M^+ where P is the given program.

Positive Reduct: $h_1 | \dots | h_k \leftarrow b_1^+ \dots b_m^+$

A disjunctive rule enforces the restriction that if the body is true, at least one of the head atoms must be true. What makes a disjunctive rule interesting is that all of the head atoms of an active rule are retained in the positive reduct. That is, the definition of stable model in the disjunctive case is that the guessed model M is a $\subset$-minimal model of the reduct P_M^+. As a result, the presence of disjunctive rules generally raises the complexity of verifying minimality in the positive reduct from P to co-NP. As a result, programs including disjunctive rules generally belong to the class $\Sigma_2^P = NP^{NP}$ [18].

3.2 The ASP-Core-2 Language

In Section 3.1.3, we introduced a complete syntax for all the ASP-Core-2 rules which have a direct one-to-one translation to rules in grounded ASP. This paper primarily focuses on showing how we can reduce the size of grounded planning problem by using ASP, but the easiest way to generate (and discuss) grounded ASP is by using the full ASP-Core-2 language.

We will not present the entire language here (the full specification is given in [7]), but we will give just enough to understand all the ASP code used in this paper. The primary advantage ASP boasts over SModels format is variables. A variable in an ASP expression must be a term beginning with a capital leter. The following are variables:

```
V
Var
QUART
B91a
```

A variable may occur as the argument to any predicate or function (meaning inside any set of parentheses), but every variable which occurs in a rule

must occur somewhere within at least one positive body atom.

```
% These are valid rules:
t(A) :- s(A).
start :- enters(V1); exits(V2); not seesAt(V1,exitPoint(V2)).
abnormal(BRAIN) :- inside(BRAIN, frankenstein).

% These are NOT:
x :- VAR. % ERROR! Not inside any predicate
question(SELF) :- placed(OTHER). % ERROR!
% SELF does not occur in body
system(H) :- not quartered(H). % ERROR!
% H does not occur positively in body
solved :- not invented(MACHINE). % ERROR!
% MACHINE does not occur positively in body
```

When a grounder (such as GRINGO) encounters a rule with variables, it creates a rule for every possible instantiation of those variables.

For instance:

```
{a(1)}.
{a(2)}.
{a(3)}.
b(start).
b(finish).

c(X,Y) :- a(X); b(Y).
% This grounds to 6 rules:
% c(1,start) :- a(1).    c(1,finish) :- a(1).
% c(2,start) :- a(2).    c(2,finish) :- a(2).
% c(3,start) :- a(3).    c(3,finish) :- a(3).
```

The grounder will remove body atoms from a rule automatically whenever it

can guarantee the atom holds (or does not hold in the case of a negative body atom). This is why above there is no mention of b in the grounded rules.

When a variable is used to stand in for a number, it may be used as part of an arithmetic expression. The grounder will evaluate all such expressions. However, the variable must still appear (unmodified) as the argument to some predicate or (non-arithmetic) function in the positive body. The relations == and != may also be used with non-numeric variables.

```
c(3).
c(5).
f(X+1) :- c(X).
% Grounds to f(4). f(6).

g(Y) :- c(Y); Y > 3.
% Grounds to just g(5).

h(X) :- c(X+1).
% ERROR! X cannot be grounded
% unless it occurs within the positive body unmodified
```

Note that it is possible (even easy) to construct programs on which the grounder will not terminate. The problem of solving a grounded ASP program is in NP (Or Σ_2^P if the program includes disjunctive rules), but the full language ASP with functions and variables is Turing complete (and therefore can be used to express undecidable problems). It's up to us to ensure our encodings always have a finite grounded form.

A variable may appear only within a cardinality constraint and nowhere else in the positive body. Also we may omit the words #count and #sum for cardinality and weight constraints. The atoms in the constraint then include all groundings for that variable. We can further restrain the values of that variable by including comma-separated expressions after a colon (Wherever a colon occurs, it can be thought of as meaning "for all").

```
{c(1)}. {c(2)}. {c(3)}. {c(4)}.
{wildcard}
d(1). d(3). d(4).
twoOf :- {c(X); wildcard} >= 2.
% Grounds to
% twoOf :- #count {c(1); c(2); c(3); c(4); wildcard} >= 2.

selectedTwo :- {c(X) : d(X), X < 4; wildcard} >= 2.
% Grounds to
% selectedTwo :- #count {c(1); c(3); wildcard} >= 2.
```

The cardinality constraint may be written using >, <, or <=. In the case of $>n$, the grounder changes it to $>=n+1$. In the case of $<=n$, the grounder negates all the atoms inside the constraint and flips it to $>= k-n$ where k is the total number of atoms (or sum weights in case of a weight constraint) in the constraint (and < compounds both operations, first flipping, then adding 1).

Cardinality and weight constraints can also be included among a list of body atoms (even possibly containing variables themselves).

```
battery(1).
battery(2).
battery(3).
laser(l1).
{laser(l2)}.
charged(LASER) :- {hasBattery(LASER,X) : battery(X)} > 1; laser(LASER).
% Grounds to
% charged(l1) :- {hasBattery(l1,1); hasBattery(l1,2);
                  hasBattery(l1,3)} >= 2.
% charged(l2) :- __internal_atom; laser(l2).
% __internal_atom :- {hasBattery(l2,1); hasBattery(l2,2);
                      hasBattery(l2,3)} >= 2.
```

If a cardinality constraint uses an =, it is grounded to the equivalent of using two constraints, one with <= and the other with >=.

A variable may occur in only the head of a choice rule if it is grounded by a colon. In this case the choice rule encompasses all groundings of the head.

We may express both a cardinality constraint and a choice rule at the same time by including any of <, <=, >, >=, = after the braces in the head.

```
b(1). b(2). b(3).
{a(X) : b(X)}.
% Grounds to
% {a(1); a(2); a(3)}


{c(X) : b(X)} = 1.
% Grounds to
% {c(1); c(2); c(3)}.
% :- {c(1); c(2); c(3)} >= 2.
% :- {not c(1); not c(2); not c(3)} >= 3.
```

A colon can be used outside a cardinality or weight constraint directly in the body. In this case it says, for any variable which does not occur elsewhere in the body, the expression to the left ranges over all groundings of the expression to the right. If the expression to the right is itself not fixed, we must create a new variable for each instantiation.

```
b(1). b(2). b(3).
{a(1); a(2); a(3)}.
{c(1); c(2); c(3)}.
holds :- c(X) : b(X).
% Grounds to
% holds :- c(1); c(2); c(3).


holds :- c(X) : a(X).
% Grounds to
```

```
% holds :- __internal(1); __internal(2); __internal(3).
% __internal(1) :- not a(1).
% __internal(1) :- c(1).
% __internal(2) :- not a(2).
% __internal(2) :- c(2).
% __internal(3) :- not a(3).
% __internal(3) :- c(3).
```

An expression using colon in the body is a convenient way to express a (possibly lengthy) conjunction. In the first example above, grounding results in a conjunction of literals $c(.)$ generated by instances of $b(.)$. In the second example, the resulting expression is still a conjunction but we only consider those instances of $c(.)$ for which the corresponding instances of $a(.)$ hold.

A colon may occur in the head of a rule. In this case, it indicates that the rule is a disjunctive rule (ranging over instantiations of the colon).

```
c(alice). c(bob). c(carol).
a(X) : c(X).
% Grounds to
% a(alice) | a(bob) | a(carol).
```

3.3 Hamiltonian Path Revisited

An ASP program can be divided into sections. This allows us to separate the encoding from the input. For instance, a particular Hamiltonian Path instance might just be specified as a list of vertices and edges:

```
vertex(a;b;c;d).
% shorthand for
% vertex(a). vertex(b). vertex(c). vertex(d).
edge(a,b;b,a;b,c;c,b;b,d;d,b;a,d;d,a).
% edge(a,b).  edge(b,a).  ......
```

We can now write a generalized encoding for Hamiltonian Path directly in ASP (without resorting to any other languages):

```
{start(V) : vertex(V)} <= 1.
inPath(V) :- start(V).
{selected(A,B) : edge(A,B)} <= 1 :- inPath(A).
inPath(V) :- selected(_,V).
:- not inPath(V); vertex(V).
```

The first rule says we must (and may) have at most one start vertex. The second rule says: for any vertex V, if V is the start vertex then it is in the path, while the third rule says for any vertex in the path, we may select at most one outgoing edge. In the fourth rule, if an edge is selected, then its child must be in the path, and in the fifth, for every vertex V, reject all models where V is not in the path.

This encoding relies on stable model semantics to avoid loops. To see this, consider the following candidate model for our instance:

```
start(a)
selected(a,d)
selected(b,c)
selected(c,b)
inPath(a)
inPath(b)
inPath(c)
inPath(d)
```

In this case, we see that the subset:

```
inPath(b)
selected(b,c)
inPath(c)
selected(c,b)
```

forms an unsupported cycle, so this model will be rejected by the solver.

Now we can measure the ASP encoding of Hamiltonian path with respect to each of the four metrics listed in Section 2.3. When applying the same four metrics (with n as the number of vertices and m the number of edges) to the grounded version of this program, they resolve to:

- Variables: $2n + m$
- Clauses: $4n + m + 2$
- Literals: $7n + 6m$
- "Non-binary" Clauses: $n + 1$

In other words, regardless of which metric we use, the problem requires at least quadratic space to encode in SAT, but only linear space in ASP.

3.4 Iterative Grounding

There is a useful tool called CLINGO which simply combines the steps performed by GRINGO (the grounder) and CLASP (the solver). Furthermore, CLINGO allows more fine-grained control over its behavior by making it possible to solve some problem, look at the result, add more rules, and then *continue solving where the solver left off*. In other words, any learned clauses that CDCL deduced in the initial run are kept in subsequent runs.

This is an extraordinary feature which gives a high degree of control over how solving proceeds. Although we pretend in this paper (for simplicity's sake) as if we start each solve anew, anywhere you can imagine you might use this feature to speed up a succession of solves, we did.

There is an important caveat however. Clingo does not allow supportedness cycles to extend across different program sections. In other words, the following would result in undefined behavior:

```
#program(part1)
#external b.
```

```
% Indicates b will be grounded in
% a later section

a :- b.
```

Then run the solver and then later add the rule:

```
#program(part2)
b :- a.
```

The solver may report $\{a, b\}$ as a solution even though it is clearly unsupported. This is unfortunate for stepless planning (Chapter 8) since an ideal implementation would rely heavily on the ability to do exactly this and expect correct results.

Chapter 4

Strips Planning

We will not explore the theory of STRIPS quite so thoroughly as we explored ASP in Chapter 3. There are at least three reasons for this:

- For the purpose of this paper, we may assume any planning problem we encounter has already been grounded and expressed as a collection of ASP atoms which can be easily input into our planner.
- STRIPS is not as nuanced or interesting as ASP and does not require quite so much background knowledge to use effectively.
- We are interested in solving existing STRIPS planning instances, not writing our own.

4.1 Actions and Fluents

A STRIPS problem consists of actions and fluents. The term "fluent" is somewhat overloaded in the context of planning, but in this paper we use it exclusively to mean a boolean piece of state in a STRIPS planning problem.

Throughout this paper, we will use the following predicates in our program.

- $action(A)$: A is an action.
- $fluent(F)$: F is a fluent.
- $pre(A, F)$: action A has F as a precondition; when F is a precondition of action A, then action A cannot be taken unless F holds.

- $add(A, F)$: action A has F as an add-effect; when F is an add-effect of action A, then taking action A causes F to be true afterwards.
- $del(A, F)$: action A has F as a del-effect; when F is a delete-effect of action A, then taking action A causes F to be false afterwards.[1]
- $init(F)$: initial conditions
- $goal(F)$: goal conditions, i.e., these are the fluents we must make true in order to solve the problem.
- $cost(A, C)$: taking action A costs C where C is a positive integer.

We assume that $add(A, F)$ is mutually exclusive with $pre(A, F)$ or $del(A, F)$, since that would be redundant or self-contradicting respectively.

In cost-optimal planning each action has an associated cost and we wish to find a plan which minimizes the sum-cost of all actions taken.

STRIPS planning is PSPACE-complete [6]. Plan-lengths may be exponential in the worst case which is why STRIPS is not in NP (assuming NP $\neq$ PSPACE), but planning is in PSPACE because although plan-lengths may be exponential, there exists a polynomial space algorithm which determines for any planning problem whether or not it has a solution (and indeed, such an algorithm can also be used to determine the first action of the solution-plan).

4.2 Sequential Planning

We can easily write a sequential planner using ASP which is guaranteed to find a plan if one exists (much later in Section 7.1 we will discuss the problem of determining when there is no solution, but for now this will suffice).

The idea is we choose some maximum number of steps $maxsteps$, that we expect our plan might take to run, and write a program to find a plan in $maxsteps$ or less. If the ASP solver returns UNSAT, we increment $maxsteps$ by 1 and try again. In addition to the input, we are given true supported

[1] In the common case, it seems that $del(A, F)$ implies $pre(A, F)$. but this is not strictly a requirement of STRIPS.

atoms of the form: $step(0)$, $step(1)$, $step(2)$, up to $step(maxsteps - 1)$ as well as one atom $finalstep(maxsteps)$.

```
holds(F,0) :- init(F).
{happens(A,T)} :- holds(F,T) : pre(A,F); step(T).
:- {happens(A,T)} > 1; step(T).
holds(F,T+1) :- happens(A,T); add(A,F).
deleted(F,T) :- happens(A,T); del(A,F).
holds(F,T+1) :- holds(F,T), not deleted(F,T).
:- not holds(F,K); finalstep(K); goal(F).
```

The first 6 rules specify what constitutes a plan. The 7th rule instructs the solver to find a plan which achieves the goal-state.

1. F is true at time 0 iff F is an initial fluent.
2. An action A may happen at time T if all of its preconditions hold at time T.
3. At most one action may occur at any time.
4. If action A occurs at time T, then its add-effects hold at time $T + 1$.
5. If action A occurs at time T, and has F as a delete-effect, then F is deleted at time T.
6. If a fluent F holds at time T, then F still holds at time $T + 1$ unless it was deleted at time T.
7. Only accept solutions where all the goal fluents are satisfied at step K (the final step).

4.3 Taking Simultaneous Actions

A more sophistocated SAT planner can relax rule 3 from the previous section. If two actions a and b can both be taken at some time T, and if the effect of

performing action a and then action b is identical to the effect of performing action b and then action a, then we should allow a and b to happen simultaneously. In this way, we reduce the number of steps we have to search before finding a plan [4].

We may formalize this by specifying when two actions a and b are mutually exclusive (mutex) meaning they cannot occur simultaneously:

```
mutexAct(A,B) :- del(A,F); pre(B,F); A != B.
mutexAct(A,B) :- del(B,F); pre(A,F); A != B.
mutexAct(A,B) :- add(A,F); del(B,F).
mutexAct(A,B) :- add(B,F); del(A,F).
```

Two actions A and B are mutex if one deletes the other's precondition or if they have conflicting effects. Any set of actions for which no two actions are mutex under these rules could conceivably happen simultaneously (if the preconditions for all of them are satisfied at time T).

To simplify our encoding, we can now do away with rules 5 and 6 and instead add to our plan "preserving" actions for each fluent. These preserving actions can be specified as:

```
action(preserve(F)) :- fluent(F).
pre(preserve(F), F) :- fluent(F).
add(preserve(F), F) :- fluent(F).
```

where each fluent F has a corresponding *preserving action* denoted by term $preserve(F)$.

Now we have two kinds of actions: regular actions and preserving actions. It will be handy to be able to distinguish between these two kinds of actions so we will additionally add a predicate $preserving/1$.

```
preserving(preserve(F)) :- fluent(F).
```

Now, whenever such an action occurs at time T, it indicates that its argument F is held over from time T to time $T+1$. Since each preserving action

is mutex with any action which deletes its preserved fluent, this automatically enforces the restriction that a fluent can only hold at time T if it is not deleted at time T.

Under this relaxed model, the number of steps a plan uses is sometimes referred to as the plan's *makespan*. When we search for a plan with some makespan n, the steps can also be called *layers*.

4.4 The Planning Graph

Blum [4] presents a handy way to identify for each action and each fluent, what is the first layer at which this action/fluent might occur by building the *planning graph*[2]. The planning graph can be built in polynomial time with respect to makespan and can additionally give us valuable information regarding mutex relations among fluents as well as actions.

We will not detail how to build the planning graph here, but instead assume that the graph is given to us and has information encoded in the following form:

```
validAct(A,T).     % action A can occur at time T
validFluent(F,T). % fluent F can be true at time T;  when T=0, this is
                   % equivalent to init(F)
mutex(F,G).        % fluents F and G are mutually exclusive and cannot
                   % both hold simultaneously
```

[2]The planning graph of a problem is different from its state graph; the planning graph is polynomial-size and generally tractable to construct and store in memory whereas the state graph is not

Chapter 5

ASPPlan

5.1 Translating from SATPlan

In this section we examine SATPlan [28], the "standard encoding" for SAT-based planners. We show that a direct translation of their rules into ASP can account for a significant reduction in search space. This occurs because ASP only accepts stable models and in particular stable models must also be supported models. By encoding the SATPlan rules directly into ASP, we find that ASP's notion of supportedness prunes the search space in a couple useful ways.

- The *solution* space does not contain any model which includes a superfluous action. In order to be supported, an action must produce a fluent which is eventually needed. Whether restricting the solution space in this way *also* ends up restricting the search space depends on the implementation of the solver, but it strongly suggests this is a possibility
- Actions or fluents which *cannot* be useful in the sense that they do not add any fluent which eventually leads to the goal state are removed *during grounding*. This is analagous to the neededness-analysis described in [39] except that we get it for free in ASP without having to add any special preprocessing.

We can translate each of SATPlan's 5 rules into ASP rules to create a planner which I will uncreatively dub ASPPlan. `holds/2` and `happens/2`

are the only predicates which produce atomic propositions in the grounded program. The others are all elided during grounding.

Like SATPlan, we run this planner by starting at makespan k (where k is the first layer at which validFluent(F, k) holds for all goal(F)) and then incrementing the `finalStep` by 1 until we find a plan.

1. "Goals hold at level k, and the initial state at level 0".

   ```
   holds(F,K) :- goal(F); finalStep(K).
   holds(F,0) :- init(F).
   ```

   The second part of this rule (initial state at level 0) has no bearing on our encoding and can indeed be safely eliminated as no other rule makes use of it. This is true for SATPlan as well, but in any case, we have it here for the sake of consistency.

2. "If a fluent holds at level k, the disjunction of actions that have that fluent as an effect hold at level k - 1;"

   ```
   happens(A,K-1): add(A,F),validAct(A,K-1) :- holds(F,K); K > 0.
   ```

3. "Actions at each level imply their preconditions;"

   ```
   holds(F,K) :- pre(A,F); happens(A,K); validFluent(F,K).
   ```

4. "Actions with (directly) conflicting preconditions or effects are mutually exclusive, encoded as negative binary clauses;"

   ```
   :- mutexAct(A,B); happens(A,K); happens(B,K).
   ```

   If one fluent is a popular effect/precondition, this rule can blow up quadratically in space. We will find a better way to handle this in the Section 5.3.

5. "Fluents that are inferred to be mutually exclusive are encoded as negative binary clauses."

```
:- mutex(F, G); holds(F, K); holds(G, K).
```

In some domains this rule can account for most of the overhead. We will address how to better handle this problem in Chapter 6.

This is a straightforward and unsurprising encoding in every respect, but has a somewhat surprising consequence. Because ASP models are well-supported, we find that for any fluent F, "holds(F,k)" can only be true if there exists some action which requires its truth as per rule 3. Similarly, for an action A, it may only be true if there exists a fluent F which requires support from the previous time step as per rule 2. Furthermore, since rule 2 is disjunctive we find that at every layer, the set of actions which occurs is the minimal set required to support the fluents at the subsequent layer. This conforms exactly to the approach to planning in [4]: First build the planning graph, then start from the goal-state planning backwards, at each step selecting a minimal set of actions necessary to add all the preconditions for the current set of actions. That is, in the ASP translation, the neededness-analysis as carried out in [39] is accomplished automatically during search for stable models or during grounding based on the stable model semantics.

Although we have done nothing surprising or innovative in translating from one to the other, we find that where SATPlan must search the space of all conceivable plans, ASPPlan only searches those plans for which each action and each fluent is both valid and needed. Later on, when dealing with cost-optimal planning and how to encode the delete relaxation, we will actively strive to obtain this same result by forcing every fluent and every action to have support "from both ends" rather than just from the initial state.

5.2 An Observation about Mutex Actions

We would like to make a small observation here about mutex. The definition of mutex as given in [4] is overly restrictive for actions. Mutex is meant to encode two actions which cannot happen simultaneously because their order is restricted (Action A cannot take place if action B has already occured and/or

vice versa). Indeed if we allowed such actions to happen simultaneously, we risk creating a cycle of dependencies, which fails to translate into a sequential plan. However, so long as all preconditions are positive, it is not necessary to restrict actions with conflicting effects from happening simultaneously.

Indeed, if we have two actions with side effects, one which disables a crane from being used and the other which frees the crane to be used again, there is no restriction on the order of these two actions. The order determines whether or not the crane is free when the actions are completed, but if we do not intend to use the crane again, we do not care if the crane is free. We can allow these two actions to happen simultaneously and then assume the crane is disabled when they complete. There certainly may still be a plan which does not require later use of the crane.

5.3 Encoding Reduction

As mentioned in Section 5.1, some of the rules presented in ASPPlan can blow up in size when grounded. In the case of rule 4 regarding "directly conflicting actions", this happens because nearly any two actions which act on the same fluent can be considered directly conflicting.

Indeed, if we imagine a planning problem in which there is a crane which we must use to load boxes on to freighters and there are many boxes and many freighters available but only one crane, then we will have one such constraint for every two actions of the form, $load(Crate, Freighter)$, for any crate and any freighter. We already have a quadratic number of actions in the problem description size. Hence, the number of mutex constraints over *pairs* of actions is *quartic* in the initial problem description size ($crates \times crates \times freighters \times freighters$).

We would like to avoid such an explosion by introducing new predicates to keep the problem size down. We will only consider two actions to be mutex if one deletes the other's precondition. But we will take extra steps to ensure that no add-effect is later used if the same fluent is also deleted at that layer.

Here is the revised encoding of rule 4 in Section 5.1.

```
used_preserved(F,K) :- happens(A,K); pre(A,F); not del(A,F).
deleted_unused(F,K) :- happens(A,K); del(A,F); not pre(A,F).
:- {used_preserved(F,K); deleted_unused(F,K);
    happens(A,K) : pre(A,F), del(A,F)} > 1;
    valid_at(F,K).


deleted(F,K) :- happens(A,K); del(A, F).
:- holds(F,K); deleted(F,K-1).
```

Effectively, we are splitting the ways in which we care that an action A can relate to a fluent F into three different cases:

1. A has F as a precondition, but not a delete-effect.
2. A has F as a delete-effect, but not a precondition.
3. A has F as both a precondition and a delete-effect.

Now we observe that for each fluent, exactly one of four cases holds:

1. Some subset of actions from category 1 occur.
2. Some subset of actions from category 2 occur.
3. One action from category 3 occurs.
4. No action from any of them occur.

By explicitly creating two new predicates for properties 1 and 2, we have packed this restriction into one big cardinality constraint.

Further, we must account for conflicting effects, so we define one more predicate (deleted) which encapsulates the union of all actions from properties 2 and 3 (those that delete F) and assert that F cannot hold at this layer if any of those actions occured in the previous one.

Chapter 6

Mutex Graphs and Multicliques

6.1 Saving Space with Multicliques

As shown in [36], significant space-savings can be gained by considering the way in which we encode mutex constraints. The naive way given in rule 5 of Section 5.1 can certainly generate enough rules to overwhelm the underlying solver for large instances. We may view the set of mutex constraints on fluents as an undirected graph where each fluent is a vertex and each constraint is an edge. When a SAT solver selects one fluent to be true at a given layer, it can then infer by unit-propogation that each fluent joined directly by an edge with the selection must be false. Thus, the set of fluents which are true at a given layer constitute an independent set on the mutex graph (an independent set on a graph is a set of vertices where no two vertices in the set share an edge [41]; equivalently this is a clique in the complement graph).

In [36], Rintanen shows that there exist other smaller encodings besides the naive approach of listing out every individual binary constraint and implies that since these encodings are smaller, they must be superior. In their experiments, they use instances of the AIRPORTS domain from the 2004 IPC planning competition. This domain is notable because of the vast number of mutex constraints it generates. The larger instances of this problem emit complex mutex graphs which can overwhelm the underlying SAT solver if encoded naively (in a one-constraint-per-edge fashion).

Rintanen shows that the mutex graphs in these planning problems (even

in benchmark AIRPORTS) tend to be highly structured and that in SAT it is possible to cover the mutex graph (somewhat more compactly) with cliques (complete subgraphs) or with bicliques (complete bipartite subgraphs). A biclique can be expressed in SAT using only one auxiliary variable and one binary clause per assignment. Rintanen demonstrates that cliques can be expressed using only a logarithmic set of bicliques. He concludes that the best way to express a mutex graph in SAT is with a biclique edge-covering.

In this paper we intend to show that for ASP, cardinality constraints give us more power than is available in SAT and indeed we can directly encode a mutex graph by its clique covering (without the extra cost of a logarithmic factor), but further we can eliminate the choice of whether to use cliques or bicliques entirely and instead cover the graph with *multicliques* (complete multi-partite subgraphs) which is a generalization of both. Indeed, we find that with multicliques, the number of clauses (namely ASP rules) required to encode mutex constraints can be further reduced over Rintanen's results.

Definition 3. A multiclique has a few equivalent definitions which we will list here:

- A partitioned graph such that for any two vertices v and w there exists an edge between v and w if and only if v and w belong to separate partitions.
- A graph whose complement is a cluster graph (a set of disjoint cliques) (See Appendix A for third equivalent definition.)

Given a multiclique covering, we can encode a constraint graph in ASP as:

```
% Covering is given by inPartition(F,P) if fluent F belongs
% to partition P, and inMulticlique(P,M) if partition P
% belongs to multiclique M.

partitionHolds(P,K) :- holds(F,K); inPartition(F,P).
:- {p(P,K): partitionHolds(P,K),inMulticlique(P,M)} > 1;
```

```
multiclique(M); layer(K).
```

Here we have a cardinality constraint expressing the rule that among all partitions P of multiclique M, at most one holds at time-step K. Furthermore, if any fluent F holds at time-step K, then its corresponding partition P must also hold.

Additionally, we can avoid some unnecessary rules by handling singleton partitions specially. A singleton partition can be packed *directly* into the cardinality constraint rather than introduced through an auxiliary atom:

```
:- {partitionHolds(P,K):inMulticlique(P,M);
      holds(F,K):singletonPartitionOf(F,M)
    } > 1;
    multiclique(M); layer(K).
```

6.2 An Assignment-Minimum Multiclique-Covering Approximation Algorithm

As discussed above, given a planning instance, if we can construct a multiclique covering (of edges) from its mutex graph, we can use ASP to encode these constraints compactly. Now let us find an algorithm for this task.

In general, finding a minimum multiclique covering (using as few multicliques as possible) is NP-hard. To see why this is true, consider the problem of finding a minimum multiclique covering on a bipartite graph. It's easy to see that a multiclique on a bipartite graph is a biclique. Thus the minimum multiclique covering of a bipartite graph is the minimum biclique covering. The size of the minimum biclique covering of a bipartite graph is also known as its *bipartite dimension*. Finding the bipartite dimension of a graph is known to be NP-hard [2]. Thus, we can safely say in the general case that finding a minimum multiclique cover is also NP-hard (since it reduces to an NP hard problem even when restricted to bipartite graphs).

Nonetheless, we can still use approximation algorithms similar to those used in [26]. One interesting thing to notice is that under the restriction that

a multiclique must use exactly a particular set of vertices, there is always only one optimal way to partition those vertices into a multiclique to cover a maximal set of edges:

If there is a path between two vertices v and w in the complement of the induced graph, then they must belong to the same partition. If there is no path, then we might as well put them in separate partitions. Therefore the best partition is the one which makes a partition for each connected component in the complement of the induced graph.

Our algorithm is given in Algorithm 6.1. It is greedy, simple, and polynomial-time. We track the set of uncovered edges and tack multicliques on one at a time, greedily building each multiclique in such a way so as to maximize the difference $2 *$ New Edges Covered $-$ Number of Literals Used in Encoding. This algorithm is the a natural extension of the "identify-biclique" algorithm used in [36] with a couple key differences.

- We're generating multicliques rather than bicliques so there can be more than two partitions.
- Instead of *removing* edges from the graph once they've been assigned to a multiclique, we keep a separate record of "uncovered" edges which still remain to be assigned. In this way the same edge may be covered twice by different multicliques if that helps to minimize the encoding.
- Instead of optimizing for |Clauses in naive encoding|$-$|Clauses in our encoding|, we're optimizing for |*Literals* in naive encoding|$-$|*Literals* in our encoding|.[1]

We select the first vertex by finding the one incident to the most uncovered edges. We then select each subsequent vertex to greedily maximize this difference under the assumption that we will finish by adding on a "default partition" of vertices. The default partition consists of all vertices which have an edge to every vertex we have selected so far including at least two edges not yet covered.

[1]When dealing with strictly binary clauses (as in Rintanen's case), these behave identically since latter metric is just the former multiplied by two.

Algorithm 6.1 Multiclique Covering

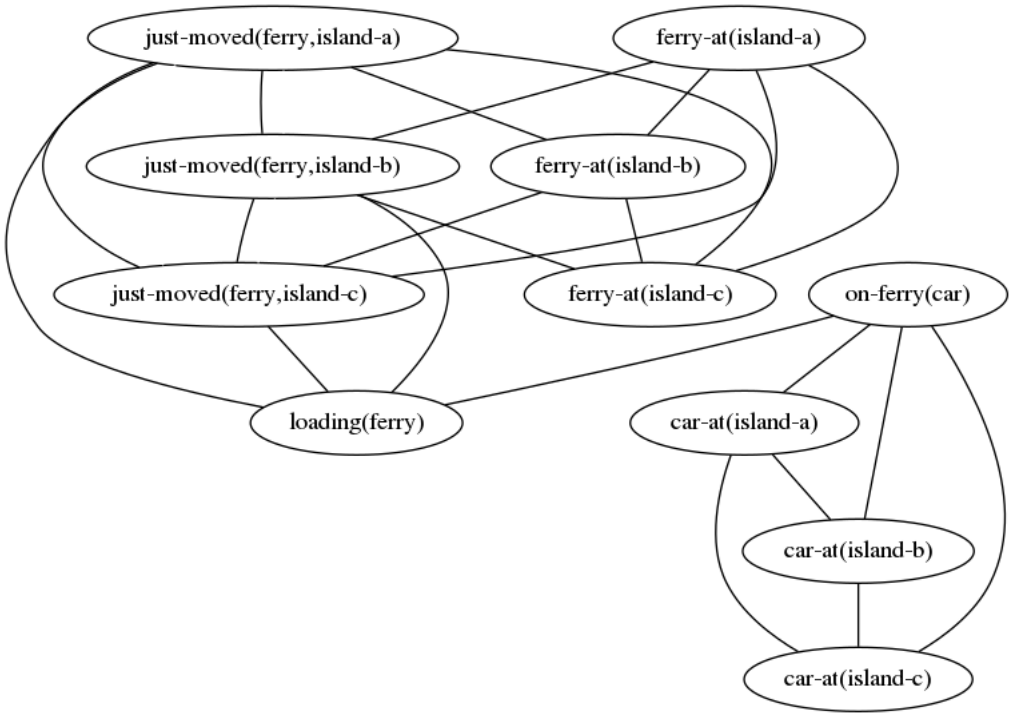
```
procedure FIND_COVER(g :: Graph) → Set MultiClique
    var uncovered ← g.edges :: Set Edge
    var multicliques ← {} :: Set MultiClique
    while uncovered.nonempty do
        new_multiclique ← next_multiclique()
        multicliques ← multicliques ∪ {new_multiclique}
        uncovered ← uncovered \ edges_covered_by(new_multiclique)
    end while
    return multicliques
end procedure
```

For more details, in Algorithm 6.1, lines 5 to 34 are helper functions. The variable *multicliques* is empty to start with. Then it iteratively adds one new multiclique at a time until all edges are covered.

Let's take a look at how this behaves on an example graph. We'll start with a mutex graph for a ferry crossing problem in which we have three islands, a ferry and a car. The ferry can be at any of the three islands and it can have just moved or be in the process of loading. The car can be on the ferry or at one of the three islands. If loading then the car is not currently on the ferry. Here's what the mutex graph for the problem looks like:

Algorithm 6.2 Multiclique Covering Helper Functions

```
type MCPartition = Set Vertex
type MultiClique = Set MCPartition
function MAKE_MULTICLIQUE(vs :: Set Vertex) → MultiClique
    return g.induced_subgraph(vs).complement().connected_components()
end function
function EDGES_COVERED_BY(mc :: MultiClique) → Edge
    return {(x, y)|p ∈ mc ∧ q ∈ mc ∧ p ≠ q ∧ x ∈ p ∧ y ∈ q}
end function
function COUNT_UNCOVERED_INCIDENT_EDGES(x :: Vertex) → ℕ
    |(g.incident_edges(x) ∩ uncovered)|
end function
procedure DEFAULTS_FOR(vs :: Set Vertex) → MCPartition
    candidates ← ⋂{g.neighbors(v)|v ∈ vs} :: Set Vertex
    return {c|c ∈ candidates, |g.incident_edges(c) ∩ uncovered| ≥ 2}
end procedure
procedure SCORE(vs :: Set Vertex) → ℤ
    multiclique :: MultiClique
    multiclique ← make_multiclique(vs) ∪ defaults_for(vs)
    newly_covered :: Set Edge
    newly_covered ← edges_covered_by(multiclique) ∩ uncovered
    complexity_cost :: ℤ
    complexity_cost ← Σ_{p∈multiclique} { 1          if |p| == 1
                                        { 2 * |p| + 1  if |p| > 1
    return 2 * |newly_covered| − complexity_cost
end procedure
procedure NEXT_MULTICLIQUE → MultiClique
    first_vertex :: Vertex
    first_vertex ← arg max_{g.vertices}(count_uncovered_incident_edges)
    var vertex_set ← {first_vertex} :: Set Vertex
    repeat
        next :: Vertex
        next ← arg max_{g.vertices}(λw. score(vertex_set ∪ {w}))
        improved ← score(vertex_set ∪ {next}) > score(vertex_set)
        if improved then
            vertex_set ← vertex_set ∪ {next}
        end if
    until ¬improved
    return make_multiclique(vertex_set ∪ defaults_for(vertex_set))
end procedure
```



Now lets run our multiclique cover algorithm on it. We get:

```
% Multiclique 0 has all singleton parts
:- {holds(just_moved(ferry,island_a),T);
    holds(just_moved(ferry,island_b),T);
    holds(just_moved(ferry, island_c),T);
    holds(loading(ferry),T)
    } > 1; step(T).
% Multiclique 1 has all singleton parts
:- {holds(car_at(island_a),T);
    holds(car_at(island_b),T);
    holds(car_at(island_c),T);
    holds(on_ferry(car),T)
    } > 1; step(T).
% Multiclique 2 has three non-singleton partitions
partitionHolds(part(2,0),T) :- holds(ferry_at(island_a),T).
partitionHolds(part(2,0),T) :- holds(just_moved(ferry,island_a),T).
partitionHolds(part(2,1),T) :- holds(ferry_at(island_b),T).
```

```
partitionHolds(part(2,1),T) :- holds(just_moved(ferry,island_b),T).
partitionHolds(part(2,2),T) :- holds(ferry_at(island_c),T).
partitionHolds(part(2,2),T) :- holds(just_moved(ferry,island_c),T).
:- {partitionHolds(part(2,0),T);
    partitionHolds(part(2,1),T);
    partitionHolds(part(2,2),T)
    } > 1; step(T).
% Multiclique 3 has two singleton parts and so is just a normal
% mutex constraint.
:- holds(loading(ferry),T); holds(on_ferry(car),T).
```

In total we have (per-layer) a grounded 10 rules with 25 literals. Had we used the naive encoding it would have been 22 rules with 44 literals so we can see this encoding is quite a bit more compact.

To give a better picture, we'll color each edge with the multiclique to which it belongs. Note that three of the edges ended up in two distinct multicliques and so are duplicated in the image:

6.3 Eventual Fluent Mutex Constraints

In Section 5.3 we found a way for the ASP solver to avoid explicitly dealing with action mutex constraints and so were able to save on grounded encoding space. But we still have a problem because the algorithm presented by Blum [4] for *generating* fluent mutex constraints in the first place requires simultaneously constructing action mutex constraints.

Indeed Rintanen [36] reports being unable to run experiments on the largest AIRPORTS instances from IPC-2004 because the action mutex constraints used so much memory they wouldn't fit in a 32-bit address space.

In this section, we find a way to circumvent this problem and were able to generate mutex constraints on the very largest (AIRPORTS-50) instance while using only about a gigabyte of memory.

Mutex constraints as defined in Blum [4] are "per-layer". You determine the set of mutex constraints *at each layer* by looking at what actions, fluents and mutex constraints were in the previous layer. Two actions are mutex if they are directly mutex or have any mutex preconditions. Two fluents are mutex if all respective pairs of causing actions are mutex. However, suppose we only care to discover and encode which fluents are *always* mutex in the sense that for *every* layer up to an arbitrarily large makespan they cannot both be true.

One way to obtain this set is to build the planning graph outward until the set of mutex constraints stabilizes. That is, we can stop once we find two consecutive layers at which the set of mutex constraints doesn't change. But this would still require tracking action mutex constraints for all pairs of actions.

The key insight is that fluents which are always mutex will be so in *sequential* planning (where exactly one action happens at each layer; see Section 4.2) as well as in parallel planning. A parallel plan is just a way of compressing a sequential plan into fewer steps so the set of pairs of things which can be true at some point will be the same regardless of how we express it.

Since a sequential plan can be expressed as a parallel plan where at most one non-preserving action happens at each layer, we can run the mutex generation algorithm under the assumption that *all* non-preserving actions are mutex with each other. Then we only need to explicitly keep track of which actions are mutex with each of the *preserving* actions. There are generally significantly fewer preserving actions than total actions. When the set of mutex fluent-pairs stabilizes, it should come out the same as if we had obtained these pairs by building the planning graph normally and waiting for the mutex fluents to stabilize.

6.4 Experiments

We implemented the multiclique generation algorithm in Haskell, representing a fluent or action as an Int and a collection of mutex constraints as an $IntMapIntSet$. Both $IntMap$ and $IntSet$ come from the *containers* package. A partition of a multiclique was represented as an $IntSet$, a multiclique as a list of partitions, and a multiclique covering as a list of multicliques.

We ran this algorithm on the same instances as Rintanen (as well as on the AIRPORTS-50 instance, the largest problem in the set) and found a significant improvement over his results. Note that these edge-counts do *not* take into account neededness. That is, they cover many fluents and actions which irrelevant to the goal of the problem and are guaranteed not to be explored by the solver. When we accounted for neededness we found the graphs got much smaller (approximately 5-fold). But we chose not to utilize this so that our results would be better comparable to Rintanen's.

In Table 6.1, "Edges" is the number of edges in the mutex graph for each instance. "CL" is the number of grounded clauses (rules) we used to encode this graph. These clauses are a mix of binary constraints and "at most 1" cardinality constraints. Because not all the clauses are binary, we are compelled to give the sum number of literals among all the constraints. This is the "Lit" column.

instance	Edges	CL	**Lit**	Edges*	CL*	**Lit***	**R-Lit**
AP-21	181884	7531	16437	166229	2336	4783	26382
AP-22	275515	11310	25014	249173	3464	7104	42776
AP-23	371062	14969	33100	336209	4806	9929	63552
AP-24	373188	15353	33894	337385	4907	10103	60814
AP-25	467653	18834	41821	421181	6208	12816	83438
AP-26	566948	22507	50252	511401	8025	16625	100494
AP-27	571298	22777	50801	514978	8155	16890	107442
AP-28	669336	26488	59201	602737	9941	20616	132120
AP-36	324835	9870	21502	297160	3084	6306	37744
AP-37	490408	14826	32921	442256	4266	8696	61362
AP-38	487033	14678	32793	438457	4263	8682	58928
AP-39	654787	20501	45166	598421	6352	12965	89294
AP-40	656469	20486	45150	599396	6351	12956	87744
AP-41	653096	20241	44709	588884	5846	11914	84628
AP-50	2613736	76180	171944	2353222	34538	71644	-

Table 6.1: Multiclique Reduction for AIRPORTS (Abbreviated AP)

After a looking at a couple of example instances, it became immediately clear that the majority of edges belong to the first few multicliques found. After that the number of edges covered per clause drops off rapidly. Thus, if we are willing to forget a small percentage of the edges, we can reduce the number of clauses necessary to encode the graph much further. For each instance, we reran the multiclique generation algorithm terminating it as soon as it had covered 90% of the total number of edges. The resulting numbers of edges covered, clauses, and literals required are given respectively by the columns "Edges*", "CL*", and "Lit*". "R-Lit" gives the number of literals required for Rintanen's biclique encoding. It's twice the number of constraints he reports [36] since all his constraints are binary clauses (having exactly two literals).

Chapter 7

Cost-Optimal Planning in ASP

In Cost-Optimal Planning, each action is associated to a cost, and our objective is to find a plan which minimizes the sum cost of all actions. Given a particular makespan (number of steps), any ASP planner can be trivially extended to a cost-optimal planner by adding weak constraints. To do this, we need only add one weak constraint for each action at each time-step:

```
:~ happens(A,K); cost(A,C).[C,A,K]
```

the grounded rules of which instruct the solver to search for a stable model that minimizes the sum of the costs of actions.

ASPPlan, augmented with this weak constraint, we will refer to as "MinASPPlan". "MinASPPlan" finds minimum cost solutions of a given makespan to a planning problem.

Quite a lot of work has been done on cost-optimal Partial-Weighted-MaxSat planning with regard to makespan, such as in [33] and [8]. But readers should find this somewhat unsatisfactory. After all, finding a plan that is "cost-optimal with regard to makespan" is just a way to sidestep the complication the real problem (finding a globally cost-optimal plan) presents. What is makespan, but an internal artifact of the SAT approach to planning? A solution should not depend on the way in which the planner happens to order the actions. Ideally, we want an approach to planning which guarantees a globally optimal solution and makes no mention of makespan.

We are aware of only one existing paper which tackles this much more difficult problem, the paper by Robinson et al. [40] (in the following, when we say Robinson's paper, we mean this paper). We will pursue a separate investigation into globally cost-optimal planning and eventually arrive at a similar (but somewhat less cluttered) algorithm to Robinson's, gaining some novel insights into the problem along the way.

Here is a road map of this chapter. In section 7.1 we tackle the problem of determining whether a planning problem (without costs) *has* a solution at all. As we'll see, this relies on being able to make assertions about whether a given plan "makes progress" towards a solution. In section 7.2 we'll look at better ways to address whether a plan is making progress. In section 7.3 we look at the connection between deciding whether a plan has a solution and determining whether a plan is optimal with respect to action costs. Section 7.4 digresses to explore the world of delete-free planning, and finally section 7.5 makes use of that to provide a much more powerful approach to detecting cost-optimality.

7.1 Our First Complete Planner

From a theoretical standpoint, let us consider why cost-optimal planning is such a difficult problem for SAT/ASP. Any planner which guarantees cost-optimality must be a proving machine [9]. When the planner terminates with a plan of cost c, it is additionally asserting "I have proved there does not exist a plan of cost $c - 1$". But here we immediately have a problem, because all the planners we have written so far are not actually *planners* in the sense of [4]; they cannot identify when a problem has no solution. So before we can create a cost-optimal planner, we must first create a (normal) planner which can determine if a problem has no solution.

Planning is PSPACE-complete, which means in the worst case, we must expect that solving a planning problem might require us to solve an exponential number of ASP instances (or at least an exponentially-sized instance). But

exponential is not infinite, and all the approaches to planning we have outlined so far have no stopping condition at all. In other words, if there is no plan, the planners we have written will simply march on forever searching for one until somebody kills the process.

The key thing that makes planning decidable is, of course, the finite state space. Any plan which goes on too long will eventually revisit the same state twice, so we only need search for a plan among those that never visit the same state twice. We can, of course, produce a naive upper bound on the number of possible states by taking $n = 2^{|fluents|}$ and then terminate the search after n steps, but let us try to do a little better.

It is simple to add a rule to our planner stating for all time-step pairs j and k with $j < k$, "some fluent must be true at time k which was not true at time j". We will augment ASPPlan with such a rule, but first, let us replace our planner's goal condition with something a little easier to satisfy.

In place of

```
holds(F,K) :- goal(F); finalStep(K).
```

we will say

```
{holds(F,K)} :- fluent(F); finalStep(K).
```

By using a choice rule here, the planner can now choose any goal it wants and then plan towards that goal. This makes our instance always satisfiable. (Just produce any valid sequence of actions, then take the end-state and claim that was your goal).

Now here comes the "we must make progress" rule.

```
:- not holds(F,K) : not holds(F,J), fluent(F); step(J); step(K); J < K.
```

In English: "For any time-step pair J and K where $J < K$, we cannot allow that every fluent F which does not hold at J also does not hold at K"

To examine more in detail what is said in the constraint, let us consider parts of it. First,

```
:- not holds(F,K)
```

says that it is not possible that F does not hold at$K$, while the full rule is instantiated to, given a particular J and particular K, a list of elements in the form "not holds(F,K) : not holds(F,J)", one for each F. In other words, the constraint in its entirety

```
:- not holds(F,K) : not holds(F,J), fluent(F); step(J); step(K); J < K.
```

says "For every time-step J and every time-step $K >$ J, there exists some fluent F which did not hold at J but *does* hold at K."

This rule guarantees that there exists a makespan n for which our planning instance is UNSAT. This is because we are now enforcing that the state must change at every time-step to take on some value which it did not have in any previous time-step. But if there are only m reachable states in our planning instance, then for all $n > m$ this is clearly impossible.

The layer at which this program becomes UNSAT must also be the layer at which we can stop searching for a solution. After all, if we can guarantee that *all* non-repeating plans take less than n steps, and we cannot find a solution to our problem using less than n steps, then that immediately tells us no solution exists.

Hence, we can build a complete planner by running two separate computations in parallel. The first is our usual ASP planner which increases the step-length until it finds an instance for which the solver finds a model (which indicates a plan has been found). The second is our "any-goal" planner which increases the step-length until it finds an instance for which the solver returns UNSAT at which point we record the previous step-length as *maxlength*.

Once that is done, if the first instance manages to reach *maxlength* and report UNSAT, we can safely claim to have proved that no plan exists, and terminate the solve.

In a sense we have struck gold. We now have a planner which is guaranteed to eventually reach a point where it knows no plan exists. Practically, however, this approach may be of limited use. We've bounded the number of steps

required by the number of possible states, which still may be as many as $2^{|fluents|}$, but there are a couple ways this might be reduced.

First of all, this already does do a little bit better than "the number of possible states" as the step length, it bounds the maximum number of steps by the length of the longest non-repeating plan, or equivalently, the longest non-self-intersecting path in the state-transition graph. This is nothing to scoff at. Some problems have a strictly-decreasing invariant such as the game Peg Solitaire. In Peg Solitaire, there is a rectangular grid of holes, some of which initially are filled with pegs. A move consists of picking up one peg and jumping (orthogonally) over an adjacent peg into an empty space on the other side. The jumped peg is removed from the board. The game is won when only one peg remains in the middle of the board.

In this problem, every jump decreases the number of pegs on the board, so the length of the longest possible plan is bounded by the number of pegs. For such problems, using the second step length finder is an easy way to determine (in a domain-independent fashion) the maximum solution length. This is indeed one problem for which our new "complete" planner can realistically be expected to terminate quickly even when there is no solution.

7.2 Stronger Notions of Progress

Unfortunately, there also exist problems containing many independent variables which may be separately manipulated to generate a large easily-traversable state-space. For such problems, the solver can produce long plans which idly "flip bits" to avoid repeating themselves.

To make this scenario more concrete, imagine we take any unsolvable planning problem and adjoin to it a binary counter with one hundred two-state switches. In addition to the actions from the original problem, we also have two hundred actions which independently flip each of the switches in the counter (either from 0 to 1 or from 1 to 0). Even though this counter has no impact on the problem itself, it suffices to increase the length of the longest plan by a

factor of 2^{100} because for every state in the longest path, we can flip through all possible arrangements of these switches before proceeding to the next state. This easily puts the possibility of solving the problem out of reach whenever there's no solution.

One way to deal with this would be to somehow encode into our planner the knowledge that the longest possible time it can take to iterate over the possible states of two independent subproblems is the maximum of the respective longest times rather than the product. In fact, this is possible. Let us create a rule stating that any action which occurs must occur as soon as possible. Robinson presents a simpler variant of this rule (rule 26) in his suffix layer [40], but does not so carefully defend its importance and effectiveness.

We can add this rule to ASPPlan (this rule only applies to regular actions, not preserving actions[1]), but we must be careful. There are quite a few things which might prevent an action from occurring any sooner. If we leave any out, we risk rendering the problem unsolvable. For an action to be able to occur at the previous time-step, its preconditions must hold at the previous time-step, its delete-effects should not be used at the previous time-step, and its used add-effects should not be deleted at the previous time-step. There are a few other conditions which at first appear to be necessary (such as its preconditions must not be deleted at the previous time-step), but upon further consideration you may notice that all of these are redundant if our goal is specifically to prevent the action from occurring *at the current time-step*. We must borrow our definition of `deleted/2` from the modified encoding of rule 4 in Section 5.3 (which is originated from Section 5.1) and additionally add a similar definition for `used/2`.

```
deleted(F,K) :- happens(A,K); del(A F).
used(F,K) :- happens(A,K); pre(A,F); not preserving(A).
```

[1]Remember from Section 4.3, in addition to the actions defined in the problem, which we call regular actions, we also include *preserving actions* for each fluent which allows that fluent to remain true from one timestep to the next. Clearly we don't want to restrict these to happening as early as possible.

```
:- happens(A,K); K > 1;
    not preserving(A);
    holds(F,K-1) : pre(A,F);
    not used(F,K-1) : del(A,F);
    not deleted(F,K-1) : add(A,F), holds(F,K).
```

How does this defeat the 100-switch-scenario?

Well, remember that the 100 switches exponentially increased the plan length because the planner may choose to flip some switches but not others to achieve one state, but then flip those other switches later to achieve an alternative state.

For every switch this rule boils down to, "if we want to flip switch i at time t, then we must also flip switch i at time $t-1$ as well". Otherwise the solution fails this rule since the switch flip *could have occurred* one action sooner. Hence, under this rule, we have made it impossible to achieve more than 100 unnecessary states within the same plan. At each step where we do not make progress somewhere else, we must choose at least one switch to stop flipping (if we toggle the exact same set of switches as in the last step, we revert to the same overall state as two steps earlier which is forbidden by the previous rule).

More generally, one can see that wherever a planning problem has multiple independent parts, this rule forces all the parts to proceed independently and not stall needlessly. It still has some gaps though.

- Even adding a linear number of unnecessary steps is suboptimal. All the switches are independent so we really should not be adding more than one layer regardless of how many switches there are.

- The switch scenario is contrived to make our solution look better than it is. One can easily see that by using three-state switches rather than two-state switches (where each state is reachable from the other two), it is still possible to construct exponential-length plans even with this restriction in place. This is because we can still reach an exponential

number of states while continually changing every switch at every time-step.

We have a stronger definition of "make progress", which I conjecture perfectly defeats the *independent parts* problem in all its forms, but it's quite a bit more complicated and will require some preliminary definitions first.

Definition 4. A *partially-ordered plan* is a transitive directed acyclic graph G (equivalently, a partial ordering) of "action occurrences" such that all topological sorts of G are valid sequential plans.

In moving from Section 4.2 to Section 4.3 we relaxed our definition of a "plan" by imposing less ordering on a solution and allowing actions to occur simultaneously. The intention with this definition is to impose even *less* ordering and relax the definition of a plan further.

Starting with any sequential plan S, we can generalize it to its canonical partially-ordered plan as follows. If a precedes b in S, then we will say $a \prec b$ for actions a and b (adding an edge from a to b) iff any of the following holds:

1. a adds some fluent which is used as a precondition for b
2. b deletes some fluent which is used as a precondition for a (and $a \neq b$)
3. a adds a fluent which b deletes
4. a deletes some fluent and b adds the same fluent[2]
5. a and b are different instances of the same action
6. There exists an action c such that $a \prec c$ and $c \prec b$

Keep in mind, these rules only apply to a-b pairs for which a precedes b in S (otherwise, rules 3 and 4 would appear to be contradictory).

If we add a source and sink s and t respectively to any partially-ordered plan such that for all actions $s \prec a \prec t$, we can consider any s-t cut x as a

[2]This will be a *later* occurrence of that fluent

generalized "intermediate state" for this plan. To see this, take any ordering where the s-side actions all precede the t-side actions and look at what fluents hold after we have taken only the s-side actions. Let us call this state x-state.

Now here comes the strongest possible (domain-independent) definition of "make progress" I can think of.

Definition 5. A partially-ordered plan is *strongly minimal* iff, given any two s-t cuts x and y, if there exists any t-side action in x which is an s-side action in y, then there must be some fluent which is true in y-state but not in x-state. We can similarly call a sequential plan strongly minimal if its canonical partially-ordered plan is.

An action can be said to make progress if no two cuts exist on either side of the action without this property (that some new fluent occurs between them).

In a strongly minimal plan, *all* actions make progress.

This beautifully handles the one hundred 3-state switch scenario by forcing us, for each switch w, to consider the generalized intermediate state where s is flipped first, before any of the other switches, and also the state where w is flipped last. The ASP encoding of this rule is complicated and has some drawbacks which merrit further discussion. It is relegated to chapter 8 where we describe how to apply it to the stepless planner. The same technique *can* with some effort be applied to layered planners as well, but requires quite a bit of boilerplate in order to talk about *next* and *previous* occurrences of each fluent and action.

7.3 Extending No-Solution Detection to Cost-Optimality Detection

How should we determine a plan is cost-optimal. Well, there's one extremely simple dummy way. Once we have used the approach from the last section to determine the *maxlength* of a non-repeating plan for our problem, we iteratively run MinASPPlan for every makespan up to *maxlength* and then take the one who's cost is optimal among those.

Here's one somewhat quirky way to reduce the possible amount of search we must do. Instead of running the any-goal solver first, we run MinASPPlan and the any-goal solver in parallel. We can take advantage of information learned from the MinASPPlan solver even if we do not yet know the overall optimal solution, once the MinASPPlan solver produces a plan of cost C, we can treat C as an upper-bound on the cost of an optimal plan. Now, we tell the any-goal solver to only search for non-repeating plans of cost $< C$. When the any-goal solver terminates with UNSAT at some time-step n we can claim that all non-repeating plans with cost $< C$ have a makespan $< n$ which means we can stop the MinASPPlan solver after finishing time $n - 1$.

In fact, if the any-goal solver finishes first, we now have a way to use the any-goal solver even after it is reached n and returned UNSAT (if we are going to allocate two cores, one for each solver, we might as well keep them both busy). Every time the MinASPPlan solver finds a new cheaper solution (say of cost D), we can start counting back down with the any-goal solver until it can find a non-repeating plan of cost D (i.e., until the problem becomes satisfiable again). In this way, we can iteratively decrease the maximum makespan we have to search until the two solvers meet somewhere in the middle.

But why should the flow of information between the two solvers be one way. The any-goal solver uses the costs of intermediate plans produced by MinASPPlan, but the costs of plans produced by the any-goal solver itself are ignored. What if in turn we use weak constraints for the any-goal solver as well and try to *its* action costs. If the any-goal solver gives back a minimum-cost plan for layer n [3], then that cost *is a lower bound* on the optimal cost of any plan with makespan at least n

We've now established that as the MinASPPlan solver increases the makespan, it can produce successively smaller upper-bounds. Meanwhile as the any-goal solver increases the makespan it can produce successively larger lower-bounds. This gives us a much simpler way to approach this algorithm. Both of these

[3]without make-progress rules, this will always be zero, which is why we keep fussing about them.

solvers can now march forward independently and as soon as the original solver reaches a layer n where its best-known solution cost C matches the cost of the any-goal solver for that same layer, we now know that the plan with cost C is *globally* optimal (having established an equal upper and lower bound on its cost).

Let us summarize this algorithm. We have two threads iteratively solve successively larger instances for encoding I and encoding II. We will name them I and II in deference to their similarity to Robinson's [40] Variant-I and Variant-II encodings.

- I is MinASPPlan (eg. ASPPlan augmented with weak constraints weighted by action costs)
- II is the any-goal solver. It is similar to I except for two major differences
  - In place of the goal conditions, II is allowed to choose its own goal.
  - II is given some notion of progress together with the constraint that it *must make progress* at every time-step. (We may choose to include such a constraint in I as well, but it is certainly not necessary. It is not clear whether such a rule would be useful).

How we determine when to stop depends on which solver (I or II) lags behind. I's result costs will fluctuate at each layer after the first layer it can solve (unless I includes no make-progress rules in which case it is monotonically nonincreasing). We can always ignore all but the lowest cost I has obtained so far on any plan. Conversely, II's result costs should be monotonically nondecreasing. If I lags behind, then as soon as I's lowest cost is $\leq$ the II cost for the layer it is currently trying to solve, we can stop and report the solution at that layer as optimal. If II lags behind, then as soon as its cost for some layer is $\geq$ the best known I-cost so far (at *any* layer), we can stop.

The asymmetry in deciding when to stop happens because of the types of bounds I and II produce. I will never produce a cost $D < C$ if C is

the optimal cost, but II will continually increase its lower bound eventually marching straight past C and on to infinity (the point at which it returns no solution). This is why we must take into account the layer at which each lower bound was produced when determining if we are done, but we do not care what layer the upper bound was produced at.

Although we have yet to talk about delete-free planning and the suffix-layer, there's one *other* particularly subtle difference between our approach and Robinson's approach that is worth noting. In place of our notion of progress, Robinson uses a simpler but less powerful rule (rule 6 in [40]) which simply states $\bigvee_{a \in A_t} a^t$ some action must occur at every time-step. This works only if we assume the problem is solvable (which he does) and all actions have positive (nonzero) costs (which he also does, but unfortunately this tends to not hold in *actual* planning competition problems. The paper doesn't attempt to rectify this; it seems in practice while planning problems *have* zero-cost actions, most of them don't produce the exact pathological case which would cause Robinson's planner to run indefinitely. In other words they don't allow you to idly "toggle state" back and forth while avoiding incurring cost). The Robinson paper does not acknowledge the problem in which the Variant-II solver proceeds faster than the Variant-I solver, but in practice the heap of extra restrictions and accompanying difficulty introduced by the suffix layer seems to make this unlikely.

7.4 Delete-Free Planning

Before discussing the suffix-layer, we must once again digress to talk about a special subclass of planning problems which happen to be in NP. This is the class of delete-free planning problems. These are planning problems without delete-effects. Surprisingly, delete-free planning can be modeled as a graph problem. We will first present it in this way and then show how the two problems are identical.

Delete-Free Planning:

Given a directed bipartite graph $G = (X, Y, E)$ with weights on X and a goal set $Y_F \subseteq Y$, find a minimum *acyclic* subgraph $G^* = (X^*, Y^*, E^*)$ such that

1. $Y_F \subseteq Y^*$
2. If $x \in X^*$ and $(y, x) \in E$, then $y \in Y^*$ and $(y, x) \in E^*$
3. For all $y \in Y^*$, E^* contains at least one edge (x, y) (and $x \in X^*$).

What is the connection? Well X is the set of actions, and Y is the set of fluents, the (x, y) edges are add-effects and the (y, x) edges are preconditions. Y_F is the goal set and the initial set has been removed (together with all corresponding preconditions) from the graph. Rule 1 means the goal fluents must be true. Rule 2 means an action implies its preconditions. Rule 3 means every fluent must have a causing action. The graph must be acyclic to ensure the actions can occur in some order. This is possible because there's no incentive to ever take an action or cause a fluent more than once. As soon as any fluent is true, it is permanently true.

7.4.1 Delete-Free Planning: Take 1

Let us write an ASP program to solve the problem delete-free planning. Furthermore, we can trivially add the extra rule to make our plans cost-optimal. Here we do not worry about makespan or running multiple iterations of the solver. Thanks to the NP-ness of delete-free planning, we can solve this problem all in one go.

```
%Instance defined by:
%action(A), cost(A,C), fluent(F), init(F), goal(F), pre(A,F), add(A,F)


holds(F) :- init(F).
{happens(A)} :- holds(F) : pre(A,F); action(A).
holds(F) :- add(A,F); happens(A).
:- goal(F); not holds(F).
```

```
:~ happens(A); cost(A,C).[C,A]
```

We have again encountered a five-line program which, magnificently, does everything. It handily encodes the problem of delete-free planning. To be supported, an action's preconditions must hold independently of that action itself, and a fluent's causing action must not require that fluent.

However, we have lost something by encoding planning "from the ground up". Earlier, we noted how the state-space for solving a planning problem was reduced when we started from the goal, and built support up backwards. That is, an action should only happen if something needs it. Let us fix that.

7.4.2 Delete-Free Planning: Take 2

If we build up the plan backwards, we must be careful to ensure that the actions can happen in some order. As such, we need to explicitly include atoms whose only purpose is to ensure supportedness.

```
holds(F) :- goal(F).
{happens(A) : add(A,F)} >= 1 :- holds(F), not init(F).
holds(F) :- pre(A,F); happens(A).

supportFluent(F) :- init(F); holds(F).
supportAct(A) :- supportFluent(F) : pre(A,F), holds(F); happens(A).
                                                        % Rule SA
supportFluent(F) :- supportAct(A); happens(A); add(A,F); holds(F).

:- holds(F); not supportFluent(F).
:~ happens(A); cost(A,C).[C,A]
```

Now the first three rules encompass neededness. We add actions and fluents in working backwards from the goal until we encounter the initial fluents. Meanwhile the second three rules indicate whether an action or fluent is supported. Together, with the restriction that all the fluents must be supported,

these guarantee a correct plan. Essentially, for an action or fluent to occur, it now must have support both from the bottom and from the top.

Why did we express needed actions with a choice rule rather than a disjunctive rule? It turns out this is important in the domain of delete-free planning since using a disjunctive rule can sometimes render an instance UNSAT. This happens because in the positive reduct, rule SA is reduced to simply:

```
supportAct(A) :- happens(A).
```

This happens because of the implicit "not" which results from having the condition holds(F) on the right of the ":" in it.)

This unintentional reduction results in an assertion that any action which happens is automatically supported.

Consider the following simple instance:

```
action(a;b;c).
fluent(x;y;z).
pre(x,a).
add(a,y).
pre(y,b).
add(b,(x;z)).
add(c,x).
goal(x;y;z).
```

Without supportedness, a valid plan requires only actions a and b (with a causing y and b causing x. Each causes the other's precondition). With supportedness, however, we must also include c in the plan. But in the positive reduct, c becomes extraneous and so the model

$$\{happens(a), happens(b), happens(c), holds(x), holds(y), holdst(z)\}$$

is deemed not minimal since it is a superset of the smaller model

$$\{happens(a), happens(b), holds(x), holds(y), holds(z)\}.$$

Running this through a disjunctive solver will return UNSAT, but by using a choice rule, we get the proper solution

$$\{happens(a), happens(b), happens(c), holds(x), holds(y), holds(z)\}$$

This concludes our digression into delete-free planning. Now let's use it.

7.5 The Suffix Layer

Delete-Free planning gives us access to an admissible heuristic for general STRIPS planning (in case we want to plan with some form of A^*-search [25] which we don't). Given any planning problem, we can simply remove all the delete-effects to get its *delete relaxation* [40]. The minimum-cost solution to the delete-relaxation of the problem is a lower bound on the minimum-cost solution to the problem itself.

As with A^*-search, we can generate successively better lower bounds by planning normally from the starting state I to some intermediate state S chosen by the planner and then finding the minimum-cost solution to the delete relaxation for the planning problem from S to the goal state G.[4] This suggests a natural way to modify our Varaint-II encoding in Section 7.3 to find better lower bounds. We append a "suffix layer" at the end, which must generate a plan in the delete relaxation of the problem from the chosen any-goal state to the actual goal state. The costs for any actions taken in the suffix layer must be added to the total cost of our plan. Indeed, in many cases this produces a remarkable lower bound. The full code for this Variant-II planner with a suffix can be found in the appendix (We do not present it here since it is really nothing more than joining the code from the rest of the chapter together into one program).

To summarize, we have two ASP programs running simultaneously. One is the Variant-I standard ASPPlan solver with weak constraints for action costs.

[4] More precisely, we encode in ASP the problem of finding the minimum *total* cost across all possible subgoal states of $cost(normalplan) + cost(suffixplan)$ (given that the normal plan respects whichever progress rule we choose to employ)

The other is the Variant-II solver with a progress rule and appended suffix layer.

- Both solvers independently run successively on makespan 0, 1, 2 etc. until we kill them.
- Each time the Variant-I solver begins solving a new makespan, we update the current makespan being solved for.
- When the Variant-I solver finds a plan, we record the plan and its cost if this is the lowest-cost plan found so far.
- When the Variant-II solver finds an optimal plan for some makespan using the suffix layer, we record the optimal cost as a lower bound for that makespan (as well as all larger makespans).
- If the Variant-II solver ever finds an optimal plan which *doesn't* use the suffix layer, then that plan is globally optimal. We can return it as a solution and entirely ignore the Variant-I solver (This only happened twice in all of our experiments. It seems to be fairly unlikely).
- If the Variant-II solver obtains UNSAT for a layer, we can stop running it and record the cost of that and all future layers as ∞.
- Any time the best-cost plan found so far (by the Variant-I solver) is no greater than the Variant-II lower bound for the currently-solving layer or *any* earlier layer, we can stop both solvers and report that plan as an optimal solution.
- Any time the Variant-I solver is solving for a makespan whose Variant-II lower bound is ∞, we can stop the solver and return the best-cost plan found so far or "no solution" if no plan has been found.

Chapter 8

Planning without Layers: Stepless Planning

All the planners so far in this paper (and indeed, all SAT/ASP planners that I've encountered) have used layers to order the actions and fluents that occur within a planning problem. But let us consider the definition of a *partially ordered plan* from Definition 4. A partially ordered plan does not need to specify layers, it is a directed graph of action-dependencies. Any topological sort of this graph corresponds to a valid plan. Perhaps we could save space by avoiding layers entirely and embedding action-dependencies directly.

The idea here is that, just as with delete-free planning, we can create a plan by specifying only which actions and which fluents hold, and we'll rely on stable model semantics to ensure the resulting solution graph is acyclic.

There's a key difference between delete-free planning and full stepless planning though which accounts for the distinction in computation complexity. In the case of delete-free planning, no fluent holds more than once and no action occurs more than once. In stepless planning, it's possible for an action to occur multiple times. As such, we'll have to have separate atoms in our encoding representing each *occurrence* of an action. But prior to solving, we don't know how many occurrences of each action or fluent will be needed.

In this chapter we'll first present a solver that *assumes* it has enough occurrences and then in Section 8.3 we'll come back to the issue of figuring out how many of each are needed in order to produce an optimal plan. The step-

less planner is *significantly* more complicated than anything else done in this thesis so we put more care into explaining what each line of ASP code does. Additionally, to the author's knowledge no planner like this has been built before, so we'll take more care to try and bridge the gap between the standard approach to planning and the approach being presented here.

8.1 Stepless Planner Encoding

Before explaining how to figure out how many occurrences we need, let us first present the encoding under the assumption we have enough. To avoid an $O\left(|\text{actions}|^2\right)$-size encoding, we don't directly encode dependencies between actions. Instead we use the fluents as intermediate nodes in the solution graph. An *occurrence* of a fluent F will be encouded as an object

```
is(fluentOcc(F,M)).
```

where M is a sequentially-ordered index. $M = 0$ is reserved for the initial fluents. All others start at $M = 1$ (when caused by some action). Similarly

```
is(actOcc{A,N}).
```

indicates an occurrence of action A.[1] In stepless planning, there are no preserving actions since there are no layers to preserve things across. We don't utilize mutex relationships between objects. We have some interesting ideas about how to use mutex for stepless planning, but it would require further development. Whereas in our previous encodings the causes and destroyers of each fluent were implicit, here we must explicitly give which occurrence of which action A causes which fluent F to hold ($causes(A, F)$) and which occurrence of which fluent F is used as a precondition for which action A ($permits(F, A)$).

[1] We index action and fluent occurrences with numbers N and M and have symmetry-breaking rules ensuring that the occurrences happen in numerical order for a given action or fluent, but it's important to understand that these numbers are *not* layers. There's no global *step* of any kind to which they correspond. A fluent occurrence can be used as a precondition for an appropriate action occurrence regardless of what their indices are or how they relate to each other. The same goes for an action *causing* a fluent. The indices are simply to be able to distinguish between multiple occurrences of the same object; they have no global significance or relation to any other object.

Additionally we need an atom for each deleted fluent occurrence F which action occurrence A has as a precondition and deletes ($deletes(A, F)$) and also one in the rare case that an action has a fluent as a delete-effect, but not a precondition, for which occurrence of the fluent F the action A follows ($follows(A, F)$).

```
% Helper function to recognize subsequent occurences of the same
% fluent/action.
nextOcc(fluentOcc(F,0),fluentOcc(F,1)) :- fluent(F).
nextOcc(fluentOcc(F,M),fluentOcc(F,M+1)) :- is(fluentOcc(F,M)).
nextOcc(actOcc(A,N),actOcc(A,N+1)) :- is(actOcc(A,N)).

% Any fluent above level 0 which holds must have exactly one causing
% action.
{causes(actOcc(A,N),fluentOcc(F,M)) : add(A,F), is(actOcc(A,N))}=1 :-
  holds(fluentOcc(F,M)); M > 0.
% If an action causes a fluent, it happens.
happens(AO) :- causes(AO,_).
% An action cannot cause more than one occurrence of the same fluent.
:- {causes(AO,fluentOcc(F,M))} > 1; is(AO); fluent(F).
% For each precondition an action occurrence has, some occurrence of
% that fluent must permit it.
{permits(fluentOcc(F,M),actOcc(A,N)) : is(fluentOcc(F,M))}=1 :-
  happens(actOcc(A,N)); pre(A,F).
% A fluent occurrence which permits an action must hold.
holds(FO) :- permits(FO,_).
% A fluent which is used to satisfy a goal condition "permits" it.
% For each goal condition, exactly one occurrence of that fluent
% permits it.
{permits(fluentOcc(F,M),goal) : is(fluentOcc(F,M))}=1 :- goal(F).
% A fluent which permits a goal condition cannot be deleted.
:- deleted(FO); permits(FO,goal).
```

```
% An occurrence of an action deletes an occurrence of a fluent if
% it permits it and that action has the fluent as a delete effect.
deletes(actOcc(A,N),fluentOcc(F,M)) :-
  permits(fluentOcc(F,M),actOcc(A,N)); del(A,F).
% No fluent may be deleted by more than one action.
:- {deletes(_, FO)} > 1; is(FO).
% An action which deletes a fluent, but doesn't have it as a
% precondition follows some occurrence of that fluent. Can possibly
% follow occurrence index 0 even if the fluent is not an initial
% fluent (indicates this action occurs before any occurrence of
% that fluent).
{follows(actOcc(A,N),fluentOcc(F,M)) : holds(fluentOcc(F,M));
  follows(actOcc(A,N),fluentOcc(F,0))}=1 :-
  del(A,F); not pre(A,F); happens(actOcc(A,N)).
% Fluent occurrences 0 which aren't initial fluents count as "deleted".
deleted(fluentOcc(F,0)) :- fluent(F); not init(F).
% A fluent is deleted if something deletes it.
deleted(FO) :- deletes(_, FO).
% A fluent is deleted if something follows it.
deleted(FO) :- follows(_, FO).

% Weak constraint charging the cost of an action occurrence.
:~ happens(actOcc(A,N)); cost(A,V).[V,A,N]

% An occurrence of a fluent doesn't hold if its previous occurrence
% doesn't hold.
:- holds(fluentOcc(F,M+1)); not holds(fluentOcc(F,M));
    is(fluentOcc(F,M)); M > 0.
% An occurrence of an action doesn't happen if its previous occurrence
% didn't happen.
```

```
:- happens(BO); not happens(AO); nextOcc(AO,BO).
```

From this we can structure the graph and assert that it is acyclic. For each action occurrence we have an"event"; additionally there's an event for the start and end of each occurrence of each fluent. There's also an event "goal" which corresponds to the goal state being reached.

Events are grouped into *vertices* in our graph each of which contains at most one action occurrence. When an action occurrence *causes* a fluent, the action and the start of that fluent belong to the same vertex. Similarly when it *deletes* a fluent, the action and the end of the fluent belong to the same vertex. To encode this we use the predicate $inVertex/2$ which indicates that its first argument belongs to the vertex named by the second argument.

```
% Events in the graph; these will be grouped into vertices
event(start(FO)) :- holds(FO).
event(end(FO)) :- holds(FO).
event(end(fluentOcc(F,0))) :- fluent(F).
event(AO) :- happens(AO).
event(goal).

% Triggering actions
% The start of a fluent by its causing action.
actionTriggers(AO,start(FO)) :- causes(AO,FO).
% The end of a fluent by its deleting action.
actionTriggers(AO,end(FO)) :- deletes(AO,FO).

% Vertices
% If no action triggers an event, then it gets a vertex by itself.
vertex(V) :- event(V); not actionTriggers(A,V) : is(A).
% Otherwise it belongs to the vertex for its trigger action.
inVertex(E,V) :- actionTriggers(V,E).
% Every event which is the name of a vertex belongs to that vertex.
```

```
inVertex(V,V) :- vertex(V).

% Graph edges
% A fluent ends after it starts
edge(start(FO),end(FO)) :- holds(FO).
% If a fluent permits an action, then the action happens after
% the start of the fluent
edge(start(FO),AO) :- permits(FO,AO).
% If a fluent permits an action but the action doesn't delete the
% fluent, then the action happens before the end of the fluent.
edge(AO,end(FO)) :- permits(FO,AO); not deletes(AO,FO).
% An action happens after the fluent it follows
edge(end(FO),AO) :- follows(AO,FO).
% but before the next occurrence
edge(AO,start(GO)) :- follows(AO,FO); nextOcc(FO,GO); holds(GO).
% The start of the next occurrence of a fluent happens after the
% end of the previous occurrence
edge(end(FO),start(GO)) :- holds(GO); nextOcc(FO,GO).
% The next occurrence of an action happens after the previous
% occurrence
edge(AO,BO) :- happens(AO); happens(BO); nextOcc(AO,BO).

% And now we use stable models to assert that the graph is acyclic; sup(X)
% indicates that X has acyclic support going back to the root of the graph.

% The input for a given event has support if all events joined
% by any incoming edge have support.
sup(in(E)) :- sup(D) : edge(D,E); event(E).
% A vertex has support if all of its events' inputs have support.
sup(V) :- sup(in(E)) : inVertex(E,V); vertex(V).
% An event has support if its vertex has support.
```

```
sup(E) :- sup(V); inVertex(E,V).
% Every vertex must have support.
:- vertex(V); not sup(V).
```

8.2 Making Stepless Progress

Now we need a way to assert that the actions of a given stepless plan "make progress". With no layers to make assertions about, the only notion of progress we're left with is the definition 5 from Section 7.2 of a plan which is strongly minimal so we'll have to use that one somehow.

Unfortunately, this definition logically takes the form of "there does not exist a pair of st-cuts such that ...". This means that *given* a particular plan, determining whether it is strongly minimal is likely *co-NP* hard (open conjecture). Then the problem of *determining the existince of* such a plan for a given collection of atoms and fluents could possibly be Σ_2^P-hard (NP^{NP}).

Luckily, as was mentioned in rule 6 of subsection 3.1.3, ASP gives us a way to encode Σ_2^P hard problems through the use of *disjunctive* rules [3]. Briefly, to determine whether a (propositional) disjunctive program has an answer set is Σ_2^P-complete, as ASP with disjunction captures Boolean quantified formulas (QBF) with variables partitioned to two sets whereas the first set is quantified by $\exists$ followed by the second which is quantified by $\forall$. The satisfiability of such QBFs is known to coincide with the decision problems in Σ_2^P [35]. Below we will diverge briefly to a short introduction about how to encode Σ_2^P problems in ASP.

Theorem 6. *Let QBF F be*

$$\exists p_1, ..., p_k,\ \forall q_1, ..., q_l :$$
$$\theta_1(p_1, ..., p_k, q_1, ..., q_l) \vee ... \vee \theta_n(p_1, ..., p_k, q_1, ..., q_l)$$

F is satisfiable iff there exists an answer set for the following disjunctive

program P:

$$\begin{array}{l} \{p_i\} \leftarrow \\ q_j \ or \ q'_j \leftarrow \\ sat \leftarrow \theta'_m \\ q_j \leftarrow sat \\ q'_j \leftarrow sat \\ \leftarrow not \ sat \end{array}$$

where $i \in [1..k]$, $j \in [1..l]$, $m \in [1..n]$, and θ'_m means θ_m except negative q_j literals $\neg g$ are replaced by g' .

Example 7. Consider the QBF and its disjunctive program encoding below.

$$\exists p_1, p_2 \forall q : \quad (p_1 \wedge \neg q) \vee (p_2 \wedge q)$$

ASP encoding:

$$\begin{array}{l} \{p_1\} \leftarrow \\ \{p_2\} \leftarrow \\ q \ or \ q' \leftarrow \\ sat \leftarrow p_1, q' \\ sat \leftarrow p_2, q \\ q \leftarrow sat \\ q' \leftarrow sat \\ \leftarrow \ not \ sat \end{array}$$

When p_1 and p_2 are assigned to true, for either assignment to q, the QBF evaluates to true. The only answer set is: $\{p_1, p_2, q, q', sat\}$.

We now come back to encoding our stepless planner. We'll present the encoding first followed by an explanation of why it works. The key thing to notice is the total lack of negative literals. This is done intentionally as we'll explain below and leads to some somewhat awkwardly named predicates of the form *not_p*.

```
% A counterexample to strong minimality consists of two cuts, cut1
% and cut2.
cut(cut1; cut2).

% For each vertex V and each cut C, V is on either the s-side or
% the t-side of V. Note this rule is disjunctive.
```

```
onSideOf(V,s,C) | onSideOf(V,t,C) :- vertex(V); cut(C).
% An event belongs to the cut side of its vertex.
onSideOf(E,X,C) :- inVertex(E,V); onSideOf(V,X,C).
% The goal (t) is always on the t-side of cut2.
onSideOf(goal,t,cut2).
% If there's a directed edge from D to E, but D is on the t-side
% and E is on the s-side, this is not a cut (invalidating this
% counterexample to strong minimality).
not_counterexample :- edge(D,E); onSideOf(D,t,C); onSideOf(E,s,C).
% If a fluent starts on the s-side of cut2 and ends on the t-side,
% then it "holds over" cut2.
holdsOver(FO,cut2) :-
  onSideOf(start(FO),s,cut2); onSideOf(end(FO),t,cut2).
% Similarly if it starts and ends on the same side of cut1, then it
% doesn't hold over cut1.
not_holdsOver(FO,cut1) :-
  onSideOf(start(FO),X,cut1); onSideOf(end(FO),X,cut1).
% Action occurrence AO is not between cut1 and cut2 if it's on the
% s-side of cut1 or the t-side of cut2.
not_betweenCuts(AO) :- onSideOf(AO,s,cut1).
not_betweenCuts(AO) :- onSideOf(AO,t,cut2).
% If no action occurs between the two cuts, then this is not a
% counterexample.
not_counterexample :- not_betweenCuts(AO) : happens(AO).
% If there exists a fluent for which some occurrence holds over cut2,
% but no occurrence holds over cut1, then this is not a counterexample.
not_counterexample :-
  holdsOver(fluentOcc(F,_),cut2);
  not_holdsOver(fluentOcc(F,M),cut1) : holds(fluentOcc(F,M)).

% There should be no counterexample (sorry for the triple negative).
```

```
:- not not_counterexample.

% If this is not a counterexample, all atoms must hold.
onSideOf(V,s,C) :- vertex(V); cut(C); not_counterexample.
onSideOf(V,t,C) :- vertex(V); cut(C); not_counterexample.
```

To understand why this works, imagine we find a plan which satisfies these rules. Consider the candidate model which includes the atom *not_counterexample*. Because all the rules here are strictly positive, the last two rules force all the others to hold. *Any* other solution is a strict subset. Therefore if some *other* solution exists which does not include the *not_counterexample* atom, then a model including it would be rejected for not being minimal. It follows that the *only* models which include *not_counterexample* (and satisfy the triple-negative rule) are those for which no counterexample exists.

8.3 Stepless Suffix Layer

Again, if there aren't enough occurrences of a fluent or action, we can tack on a suffix layer in the same way we did with the stepped-cost-optimal planner.

First, we need to replace all uses of *goal* with a *subgoal* which is the entry-point into the suffix layer:

```
% A fluent which is used to satisfy a subgoal condition "permits" it.
% For each subgoal condition, exactly one occurrence of that fluent
% permits it.
{permits(fluentOcc(F,M),subgoal(F)) : is(fluentOcc(F,M))}=1 :-
  subgoal(F).
% A fluent which permits a subgoal condition cannot be deleted.
:- deleted(FO); permits(FO,subgoal(_)).

% subgoals are events
event(subgoal(F)) :- subgoal(F).
```

```
% Any subgoal is always on the t-side of cut2.
onSideOf(subgoal(F),t,cut2) :- subgoal(F).
```

In the ASP snippet above, I've just copy-pasted any rule with the word *goal* or $goal(F)$ in the last couple sections and replaced it with $subgoal(F)$ for the appropriate fluent F.

We'll get our subgoals from the suffix layer, which can be encoded as:

```
% All goal fluents hold in the suffix layer.
suffix(holds(F)) :- goal(F).
% If a fluent holds in the suffix layer, either some action causes it
% or it is a subgoal.
{subgoal(F); suffix(causes(A,F)) : add(A,F)} = 1 :- suffix(holds(F)).
% If an action causes a fluent in the suffix, it happens.
suffix(happens(A)) :- suffix(causes(A,_)).
% If an action occurs in the suffix layer, then all of its
% preconditions hold in ths suffix layer
suffix(holds(F)) :- suffix(happens(A)); pre(A,F).

% If any action happens in the suffix layer, then we are using the
% suffix layer.
useSuffix :- suffix(happens(_)).

% A fluent is supported in the suffix if it's a subgoal
suffix(sup(holds(F))) :- subgoal(F).
% An action is supported in the suffix if all of its preconditions are
suffix(sup(happens(A))) :-
  suffix(sup(holds(F))) : pre(A,F); suffix(happens(A)).
% A fluent is supported in the suffix if its causing action is
suffix(sup(holds(F))) :- suffix(sup(happens(A))); suffix(causes(A,F)).

% No action happens in the suffix without support
```

```
:- suffix(happens(A)); not suffix(sup(happens(A))).
% No fluent holds in the suffix without support
:- suffix(holds(F)); not suffix(sup(holds(F))).

% Actions that happen in the suffix layer impose their cost.
:~ suffix(happens(A)); cost(A,V).[V,A,suffix]
% Very weak preference to avoid using the suffix layer.
:~ useSuffix.[1@-1]
```

This is similar to the normal encoding we use for the suffix layer but there are a few key differences. First, if the suffix layer is used at all, then $useSuffix$ is true. There is a cost of 1 at level -1 for useSuffix so among plans of equal cost, the solver will prefer one which *doesn't* use the suffix to one which does. If an optimal solution doesn't use the suffix, then it must be globally optimal with respect to cost.

8.4 Counting Stepless Occurrences

We'll now add rules to enforce the use of actions and fluents from our bag so that the planner resorts to the suffix layer only when it "runs out" of something.

```
% A fluent is saturated if all occurrences of it hold (besides the 0th).
saturated(fluent(F)) :-
  holds(fluentOcc(F,M)) : is(fluentOcc(F,M)),M>0; fluent(F).
% An action is saturated if all occurrences of it happen.
saturated(action(A)) :-
  happens(actOcc(A,N)) : is(actOcc(A,N)); action(A).

% If an action happens in the suffix layer and all of its preconditions
% are subgoals, we designate it a "starting" action.
suffix(start(action(A))) :- subgoal(F) : pre(A,F); suffix(happens(A)).
% Any fluent caused by a starting action is designated a "starting"
% fluent.
```

```
suffix(start(fluent(F))) :- suffix(start(action(A))); suffix(causes(A,F)).

% Guarantees that some starting action or fluent will be saturated.
:- useSuffix; not saturated(X) : suffix(start(X)).
```

With this we know how to expand our bag of fluents and actions. Each time we get back a plan making use of the suffix layer, look at all the fluents or actions which were saturated by that plan and add another occurrence of each one.

This, coupled with our definition of making progress, is what guarantees it will eventually find a plan or determine that none exists. The suffix layer is only used because the planner ran out of something it needed and needs to request more of that item from the controlling program; not as a way to save on plan cost.

The total collection of all rules needed for the stepless planner is presented in Appendix C, Section C.6.

8.5 Example of Stepless Planning: Bridge Crossing

We will use a modified version of the bridge-crossing problem from Eiter [10].[2]

In the original problem, we have four people, Joe, Jack, William, and Averell, needing to cross a bridge in the middle of the night. The bridge is unstable and so at most two people can cross at a time. The four only have a single lantern between them and since there are planks missing it's unsafe to cross unless someone in your party is carrying the lantern. In the original

[2]Eiter et al. [10] claims to present an approach to finding globally cost-optimal plans in ASP with action costs, but make the assumption that the domain has some polynomial upper bound on plan lengths. They cite some complexity results for why this assumption is reasonable. However, they then further *implicitly* and without justification make the assumption that this upper bound is known to the solver and that the algorithm can make use of it. Their algorithm for finding cost-optimal plans amounts to constructing an ASP program which can consider plans of any makespan up to a hard-coded upper bound and returning the cheapest only among those. This technique could not be deployed in building a domain-independent planner.

problem, it takes Joe 1 minute to run across, Jack 2 minutes, William 5 minutes and Averell 10. When two people cross together they must go at the slower speed of the two. What's the fastest all four can get across considering that after each crossing somebody needs to cross back carrying the lantern?

In our version we'll add two more people Jill and Candice for a total of six people. Jill takes 3 minutes to cross and Candice takes 20 (the original problem doesn't make for a very interesting example of stepless planning).

We can now phrase the problem as follows:

```
person(joe;jack;jill;william;averell;candice)
side(side_a;side_b)
crossing_time(joe,1).
crossing_time(jack,2).
crossing_time(jill,3)
crossing_time(william,5).
crossing_time(averell,10).
crossing_time(candice,20).

fluent(lantern_at(S)) :- side(S).
fluent(at(P,S)) :- person(P); side(S).

init(at(P,side_a)) :- person(P).
init(lantern_at(side_a)).
goal(at(P,side_b)) :- person(P).

action(cross_alone(P,FROM,TO)) :-
  person(P); side(FROM); side(TO); FROM != TO.
pre(cross_alone(P,FROM,TO),at(P,FROM)) :-
  action(cross_alone(P,FROM,TO)).
pre(cross_alone(P,FROM,TO),lantern_at(FROM)) :-
  action(cross_alone(P,FROM,TO)).
add(cross_alone(P,FROM,TO),at(P,TO)) :-
```

```
  action(cross_alone(P,FROM,TO)).
add(cross_alone(P,FROM,TO),lantern_at(TO)) :-
  action(cross_alone(P,FROM,TO)).
del(cross_alone(P,FROM,TO),at(P,FROM)) :-
  action(cross_alone(P,FROM,TO)).
del(cross_alone(P,FROM,TO),lantern_at(FROM)) :-
  action(cross_alone(P,FROM,TO)).
cost(cross_alone(P,FROM,TO),C) :-
  action(cross_alone(P,FROM,TO)); crossing_time(P,C).

action(cross_together(P_SLOW,P_FAST,FROM,TO)) :-
  side(FROM); side(TO); FROM != TO;
  crossing_time(P_SLOW,T1); crossing_time(P_FAST,T2); T2 < T1.
pre(cross_together(P_SLOW,P_FAST,FROM,TO),at(P_SLOW,FROM)):-
  action(cross_together(P_SLOW,P_FAST,FROM,TO)).
pre(cross_together(P_SLOW,P_FAST,FROM,TO),at(P_FAST,FROM)):-
  action(cross_together(P_SLOW,P_FAST,FROM,TO)).
pre(cross_together(P_SLOW,P_FAST,FROM,TO),lantern_at(FROM)):-
  action(cross_together(P_SLOW,P_FAST,FROM,TO)).
add(cross_together(P_SLOW,P_FAST,FROM,TO),at(P_SLOW,TO)):-
  action(cross_together(P_SLOW,P_FAST,FROM,TO)).
add(cross_together(P_SLOW,P_FAST,FROM,TO),at(P_FAST,TO)):-
  action(cross_together(P_SLOW,P_FAST,FROM,TO)).
add(cross_together(P_SLOW,P_FAST,FROM,TO),lantern_at(TO)):-
  action(cross_together(P_SLOW,P_FAST,FROM,TO)).
del(cross_together(P_SLOW,P_FAST,FROM,TO),at(P_SLOW,FROM)):-
  action(cross_together(P_SLOW,P_FAST,FROM,TO)).
del(cross_together(P_SLOW,P_FAST,FROM,TO),at(P_FAST,FROM)):-
  action(cross_together(P_SLOW,P_FAST,FROM,TO)).
del(cross_together(P_SLOW,P_FAST,FROM,TO),lantern_at(FROM)):-
  action(cross_together(P_SLOW,P_FAST,FROM,TO)).
```

```
cost(cross_alone(P_SLOW,P_FAST,FROM,TO),C) :-
  action(cross_alone(P_SLOW,P_FAST,FROM,TO)); crossing_time(P_SLOW,C).
```

Let's run the stepless solver on this and see what happens. On the first iteration we input one occurrence of every fluent and every action as well as a bonus zeroth occurrence of each initial fluent.

```
is(fluentOcc(F,1)) :- fluent(F).
is(actOcc(A,1)) :- action(A).
is(fluentOcc(F,0)) :- init(F).
```

It gives back a directed graph of action and fluent dependencies. After topologically sorting the graph and throwing out everything that isn't an action we have the plan:

```
cross_together(jack,joe,side_a,side_b)
cross_alone(joe,side_b,side_a)
suffix cross_together(candice,averell,side_a,side_b)
suffix cross_alone(joe,side_a,side_b)
suffix cross_together(william,jill,side_a,side_b)
cost: 29
```

In the suffix layer when Candice and Averell cross from *side_a* to *side_b*, the fluent $lantern_at(side\_a)$ is not deleted (because the suffix layer encodes the delete-free relaxation of the problem), so this is still considered to be achieved when Joe, and then William and Jill cross. Nobody needs to bring the lantern back for them. The use of the suffix layer is allowed because there isn't a second occurrence of the fluent $at(joe(side_b))$, but this is a starting fluent (all 3 suffix actions are starting actions since they do not depend on each other). Since the suffix layer was used, we add a second occurrence of each of the fluents and actions which were saturated by this plan:

```
Adding:
is(fluentOcc(at(joe,side_a),2)).
```

```
is(fluentOcc(lantern_at(side_a),2)).
is(fluentOcc(lantern_at(side_b),2)).
is(fluentOcc(at(joe,side_b),2)).
is(fluentOcc(at(jack,side_b),2)).
is(actOcc(cross_together(jack,joe,side_a,side_b),2)).
is(actOcc(cross_alone(joe,side_b,side_a),2)).
```

and run it again

```
cross_together(william,joe,side_a,side_b)
cross_alone(joe,side_b,side_a)
cross_together(jill,joe,side_a,side_b)
cross_alone(joe,side_b,side_a)
suffix cross_together(candice,averell,side_a,side_b)
suffix cross_together(jack,joe,side_a,side_b)
cost: 32
```

This time we start by having William and Joe cross together and then Joe carries the lantern back, crosses with Jill and carries it back again. In the suffix layer, Candice and Averell cross together while Jack and Joe cross together (each pair making use of the same undeleted lantern). Again the suffix layer occurs because we don't have enough occurrences of $at(joe(side\_b))$.

Interestingly, a cheaper solution seems to have been skipped. Namely the plan which is identical to the cost-29 plan, but with Joe running across and running back first for a total cost of 31.

This is because such a plan fails to make progress. We can produce two cuts, namely the one at the start of the plan and the one after Joe crosses back the first time and see that no new fluents hold between the two cuts. The rules enforcing strong minimality will reject this plan.

Add another occurrence of each saturated item

```
Adding:
is(fluentOcc(at(joe,side_a),3)).
```

```
is(fluentOcc(lantern_at(side_a),3)).
is(fluentOcc(lantern_at(side_b),3)).
is(fluentOcc(at(joe,side_b),3)).
is(fluentOcc(at(jill,side_b),2)).
is(fluentOcc(at(william,side_b),2)).
is(actOcc(cross_together(jill,joe,side_a,side_b),2)).
is(actOcc(cross_together(william,joe,side_a,side_b),2)).
is(actOcc(cross_alone(joe,side_b,side_a),3)).
```

and again:

```
cross_together(william,jack,side_a,side_b)
cross_alone(jack,side_b,side_a)
cross_together(jill,jack,side_a,side_b)
suffix cross_alone(jack,side_b,side_a)
suffix cross_alone(joe,side_a,side_b)
suffix cross_together(candice,averell,side_a,side_b)
cost: 33
```

Here we have William and Jack crossing together. Then Jack crosses back alone. Jill and Jack cross together, and now Jack *would* cross back alone again taking the lantern, but there are only two occurrences of the action $cross\_alone(jack, side\_b, side\_a)$ in our bag so instead we move into the suffix layer. In the suffix layer he carries the lantern back, but because of the delete relaxation, we don't lose the fluent $at(jack, side\_b)$ so he doesn't need to cross back again. Candice and Averell use the lantern to cross as does Joe by himself.

The rest of the output from the stepless solver follows:

```
Adding:
is(fluentOcc(at(jack,side_a),2)).
is(fluentOcc(at(jack,side_b),3)).
is(actOcc(cross_together(jill,jack,side_a,side_b),2)).
```

```
is(actOcc(cross_together(william,jack,side_a,side_b),2)).
is(actOcc(cross_alone(jack,side_b,side_a),2)).

cross_together(jill,joe,side_a,side_b)
cross_alone(joe,side_b,side_a)
cross_together(william,joe,side_a,side_b)
cross_alone(joe,side_b,side_a)
cross_together(jack,joe,side_a,side_b)
cross_alone(joe,side_b,side_a)
suffix cross_alone(joe,side_a,side_b)
suffix cross_together(candice,averell,side_a,side_b)
cost: 34

Adding:
is(fluentOcc(at(joe,side_a),4)).
is(fluentOcc(lantern_at(side_a),4)).
is(fluentOcc(lantern_at(side_b),4)).
is(fluentOcc(at(joe,side_b),4)).
is(actOcc(cross_alone(joe,side_b,side_a),4)).

cross_together(jill,jack,side_a,side_b)
cross_alone(jill,side_b,side_a)
cross_together(william,jill,side_a,side_b)
suffix cross_alone(jill,side_b,side_a)
suffix cross_alone(joe,side_a,side_b)
suffix cross_together(candice,averell,side_a,side_b)
cost: 35

Adding:
is(fluentOcc(at(jill,side_a),2)).
is(fluentOcc(at(jill,side_b),3)).
```

```
is(actOcc(cross_together(william,jill,side_a,side_b),2)).
is(actOcc(cross_alone(jill,side_b,side_a),2)).

cross_together(jack,joe,side_a,side_b)
cross_alone(jack,side_b,side_a)
cross_together(jill,jack,side_a,side_b)
cross_alone(jack,side_b,side_a)
cross_together(william,jack,side_a,side_b)
suffix cross_alone(jack,side_b,side_a)
suffix cross_together(candice,averell,side_a,side_b)
cost: 36

Adding:
is(fluentOcc(at(jack,side_a),3)).
is(fluentOcc(at(jack,side_b),4)).
is(actOcc(cross_alone(jack,side_b,side_a),3)).

cross_together(jack,joe,side_a,side_b)
cross_alone(joe,side_b,side_a)
cross_together(jill,joe,side_a,side_b)
cross_alone(joe,side_b,side_a)
cross_together(candice,averell,side_a,side_b)
cross_alone(jack,side_b,side_a)
cross_together(william,joe,side_a,side_b)
cross_alone(joe,side_b,side_a)
cross_together(jack,joe,side_a,side_b)
cost: 37
```

In the last one, the suffix layer is not used so we're done. No other plans need be searched.

Chapter 9

Experiments

We ran our cost-optimal two-threaded solver and stepless solver on most of the same instances as Robinson [40] and here report results.[1]

Experiments were run on a cluster of $c3.large$ Amazon EC2 instances each with two Intel Xeon 2.8 GHz CPU cores and 3.75 GB of memory. We used GNU Parallel [44] to distribute the work of running multiple instances

For comparison, we include Robinson's reported results scaled down by a factor of $\frac{2.6}{2.8}$ to account for the difference in processor speeds.

For each domain we report the largest instance solved by each of the two-thread, stepless, and Robinson's planner where largest is measured by the amount of time it took that planner to solve the instance. Where it differs, we also report the largest-indexed instance solved by that each of the two-thread and stepless planners.

Every plan produced by either planner was validated by the Strathclyde Planning Group plan verifier VAL [27].

The column C^* is the optimal cost found for each instance. In all cases the optimal cost for the two-thread and stepless planners agree and furthermore agree with the optimal cost reported by Robinson [40] where applicable (Robinson compares his results against a *non*-SAT-based planner and our optimal costs agrees with that as well).

[1] excluding *satellites* since our planner doesn't support the : *equality* extension to PDDL and *miconic* since we couldn't find the problem files for it. Robinson was kind enough to send us the instances from his constructed ftb domain so we can report performance on that as well.

The column n is the lowest makespan at which the problem has a C^* plan (according to our Variant-I MinASPPlan solver). Our value for the makespan n agreed with all of Robinson's reported results except for in Rovers-3 where we found we only needed a makespan of 7 to produce the optimal plan while Robinson reported a required makespan of 8. We suspect this is because of the action mutex relaxation distinction from Section 5.2, but have not verified this.

n_* is the makespan at which our Variant-II suffix solver proves C^* is optimal. Interestingly, for many of instances this value was 0 which indicates that the optimal plan in the delete-free reduction of the problem has the same cost as the true optimal plan.

t_π is the time required to find the plan (by our Variant-I MinASPPlan solver). t_* is the time required to prove optimality (by our Variant-II suffix solver) t_t is the sum of these two numbers (can be thought of as "total" solve time" although the algorithm necessitates that they run in parallel, so the actual wall-clock time required to run them was the maximum, not the sum, but with two CPU cores rather than one). All reported times are measured in seconds.

n_s is the number of times the stepless solver was run for this instance (each time adding more items to its bag of fluents and actions based on what was saturated in the previous rounds). On the last of these runs it produced an optimal solution which doesn't use the suffix layer and hence is globally cost-optimal. t_s is the total time running the stepless solver across all runs.

One important distinction between the experiments run with the two-threaded solver and those run with the stepless solver is that the two-threaded solver took advantage of iterative solving as mentioned in Section 3.4. For the stepless solver this was not possible since it relies heavily on full-program-spanning loop constraints to give correct results, but as mentioned in Section 3.4, CLINGO doesn't support having loop constraints cross multiple iterative stages. So instead every time new fluents or actions are added to the stepless solver's bag, it has to restart solving from scratch.

l_s is the total time required for the last iteration of the stepless solver to run. This one run by itself is sufficient to both find the globally optimal solution *and* prove its optimality. However we know of no efficient way to find the right bag of actions and fluents in order to guarantee the optimal solution won't use the suffix layer (except by running the solver multiple times and looking at saturated actions and fluents on each run). This number is still interesting in that it provides a lower bound on the time it would take to solve the instance if CLINGO supported loop constraints crossing program section boundaries (so that we could add more occurrences and continue solving rather than having to restart). It gives us some idea of what savings such a modification to CLINGO might provide.

R_t is the total time reported by [40] to find the optimal solution scaled by a factor of $\frac{2.6}{2.8}$. A questin mark ? in this column indicates the time is unknown since it's not reported in [40]. If the solver for which this row is maximal successfully solved the largest instance reported by Robinson in this domain and found this instance to be larger, we fill with a dash mark − rather than ? in this column (our best guess as to whether Robinson's planner solved it). A − in any other column indicates the relevant planner failed to solve the instance in less than 1671.4 seconds (30 minutes scaled down by $\frac{2.6}{2.8}$). In the case of depots-2, the Variant-II suffix solver reached layer 12 before the Variant-I MinASPPlan solver and so it found an optimal no-suffix solution by itself.

All instances were solved with CLINGO version 5.2.3. The controller logic for both the two-threaded solver (handling of incremental solving, coordinating the two solvers, and figuring out when to terminate the search) and the stepless solver (figuring out which occurrences to include and topologically sorting the output) was written in Haskell using the clingo-Haskell bindings written by tsahyt (GitHub alias) to communicate with CLINGO .

We used the default configuration and options for CLINGO except that the stepless planner used the –opt-usc option which finds optimal solutions by expanding an unsatisfiable core (Definition 2 in [1]).

The ASPPlan solver and its variants as well as the two-threaded and stepless solver are available on GitHub at `https://github.com/davidspies/aspplan2`. Feel free to contact the repo owner (the author of this paper) for any help with reproducing these results.

In all domains except for *blocks*, *ftb*, and *storage*, our two-layer solver outperformed Robinson's equivalent SAT-based solver and our stepless solver outperformed both (in terms of number of instances solved). In the case of *storage*, the stepless solver and Robinson's solver each solved an instance which the other failed to solve which seems to point to the possibility that the stepless solver encounters different difficulties from a more traditional approach. One more piece of evidence favoring this conclusion is that the toy example bridge-crossing problem from Section 8.5 required a full 30 seconds to solve (whereas the two-thread solver solves it in 2 seconds) and in general we found that on small/toy problems the stepless solver's performance is abysmal compared with other approaches we tried, but when given more time scales better with larger instances.

Prior to running the full suite of experiments, the above observation gave us the mistaken impression that the stepless solver was interesting as a theoretical oddity, but fails to produce decent results in practice, since for every example we ran it on while tuning it, it seemed to run slower than the two-threaded solver. It was a pleasant surprise to discover when officially running the experiments that in fact the inverse was true.

Problem	C^*	n	n_*	t_π	t_*	t_t	n_s	t_s	l_s	t_R
blocks-12	20	20	17	0.5	1203.4	1203.9	-	-	-	?
blocks-15	16	16	12	0.4	113.4	113.8	7	89.4	33.7	?
blocks-18	26	26	16	0.9	256.8	257.7	-	-	-	3.2
blocks-23	30	-	-	-	-	-	-	-	-	29.8
blocks-25	34	-	-	-	-	-	-	-	-	27.4
depots-2	15	-	12	-	771.5	771.5	2	9.7	4.2	-
depots-13	25	-	-	-	-	-	3	475.9	137.2	-
driverlog-2	19	-	-	-	-	-	20	215.5	44.4	-
driverlog-3	12	7	3	0.1	0.9	1.0	1	0.4	0.4	450.2
driverlog-11	19	-	-	-	-	-	1	13.5	13.5	-
elevators-2	26	3	0	0.4	1.7	2.1	1	2.7	2.7	13.0
freecell-3	18	-	-	-	-	-	2	420.5	344.0	-
ftb-30	1001	25	0	1.8	0.3	2.1	1	5.5	5.5	1.8
ftb-38	601	33	0	2.7	0.2	2.9	1	3.2	3.2	1.5
ftb-39	801	33	0	3.9	0.3	4.2	1	5.6	5.6	2.2
ftb-40	1001	33	0	3.9	0.4	4.3	1	8.2	8.2	?
gripper-1	11	7	4	0.1	0.4	0.5	2	0.4	0.2	14.6
gripper-2	17	11	8	0.6	312.4	313.0	7	23.5	9.7	-
pegsol-9	5	15	11	3.9	35.9	39.8	5	131.5	46.6	386.8
pegsol-16	8	21	17	48.3	1029.0	1509.3	10	910.2	280.8	-
pegsol-18	7	-	-	-	-	-	7	1548.0	537.1	-
rovers-3	11	7	4	0.1	0.2	0.3	1	0.1	0.1	49.4
rovers-4	8	4	0	0.0	0.0	0.0	1	0.1	0.1	?
rovers-6	36	-	-	-	-	-	48	1354.3	391.0	-
rovers-9	31	-	-	-	-	-	53	1040.6	101.4	-
rovers-14	28	-	-	-	-	-	72	900.7	55.9	-
storage-7	14	14	11	0.6	42.9	43.5	10	89.2	42.4	1.1
storage-8	13	-	-	-	-	-	15	799.1	239.5	-
storage-9	11	-	-	-	-	-	9	181.0	46.0	-
storage-13	18	-	-	-	-	-	-	-	-	244.0
TPP-5	19	7	2	0.1	0.2	0.3	2	0.5	0.3	-
TPP-7	34	-	-	-	-	-	13	189.6	32.4	-
transport-1	54	5	0	0.1	0.1	0.2	2	0.5	0.3	0.2
transport-2	131	12	4	74.3	55.1	129.4	2	111.6	106.3	-
transport-11	456	9	3	0.3	1.6	1.9	2	163.4	151.4	-
transport-21	478	7	1	0.2	0.6	0.8	2	5.2	3.5	-
zenotravel-4	8	7	3	0.5	2.8	3.3	3	14.1	6.9	783.4
zenotravel-6	11	7	0	7.2	6.5	13.7	1	2.1	2.1	-
zenotravel-10	22	-	-	-	-	-	1	1387.1	1387.1	-

Chapter 10

Summary and Future Directions

We found that by using ASP instead of SAT, various inherent problems with SAT-based planning can be mitigated or even eliminated entirely in some cases *specifically* because of the added power granted by stable-model semantics. Stable model semantics allows us to encode certain complex non-local conditions without a big-O increase in the grounded size of the problem being solved. These conditions turn out to be majorly useful when applied to domain-independent planning.

In Summary:

- When taking the standard approach to planning in SAT (SATPlan) and encoding it in ASP (ASPPlan), the resulting program has both a narrower search space and narrower (less redundant) solution space thanks to stable model semantics. Amazingly we get the neededness analysis of [39] for free without any need for explicit preprocessing.

- Using cardinality constraints, we can far more compactly express mutex relationships between fluents then can be expressed using only binary clauses.

- Delete-free planning and the suffix layer can actually be expressed in *linear* size with respect to the grounded problem definition. Stable model semantics ensures that the resulting plan is acyclic and can be topologically ordered into a sequential plan.

- Disjunctive ASP together with loop constraints lets us encode an incredibly sophistocated rule guaranteeing that when searching for optimal plans, the solver will not "get stuck" dealing with iterated versions of a plan which fails to make progress.

- When we put the above two elements together, we can take an entirely new approach to planning which does away with makespans entirely and instead deals only with action and fluent dependencies. Besides being a theoretically fascinating approach to planning, when used for *cost-optimal* planning, stepless planning outperforms the slightly more traditional approach to SAT(/ASP)-based planning in practice.

There's a lot of work left to be done, and concluding this thesis where it is now was somewhat painful considering all the remaining possibilities.

Future work:

- One thing we failed to note in our experiments is that our Variant I solver exploits the fact that CLINGO outputs successively better models as it approaches the optimum solution. Thus it can in some cases find the optimum solution and prove it's optimal by way of the Variant II solver rather than waiting for CLINGO to finish running that instance and prove optimallity by exhausting the search space as it typically does.

  As has been noted in [38], determining that a given SAT or ASP problem is UNSAT seems to be *significantly* more difficult than finding a solution to one which is satisfiable (for problems of approximately the same size).

  They have an approach to planning which simultaneously searches at multiple different makespans for a solution, pausing and resuming each solver according to an exponential falloff rate. This handily avoids the timesink of trying to find solutions with too small a makespan when a simple solution might be "just around the corner" so to speak. With regard to cost-optimal planning, this can still be a useful tool for the Variant I solver since it doesn't need to prove optimality (that's the Variant II solver's job).

- In our experiments, we did not use the –opt-usc option on the Variant II (suffix) piece of the two-threaded solver since (unlike with the stepless solver) some preliminary results on toy instances seemed to indicate that this option made it less performant.

  However, there's a strong incentive to use it. When running with this option, CLINGO doesn't find multiple models until it reaches an optimal one. Instead it produces successively larger lower bounds on the cost of the optimal model until it manages to find a solution. Since the Variant II solver is meant to find lower bounds on the cost of a globally optimal solution. Any lower bound on the output of the Variant II solver then is *also* a lower bound on the global cost (just as suboptimal models for the Variant I solver are still valid upper bounds on a globally optimal model). We could use the intermediate lower bounds produced by CLINGO and not just the optimal result found at each step.

  Coupling this with the exponential falloff rate approach mentioned in the the previous bullet-point, we could construct a cost-optimal planner in which neither thread by itself ever has to waste time *proving* optimality of any solution it encounters. The optimality proof comes entirely from combining the intermediate upper and lower bounds found by each of the two threads.

  Unfortunately, it's not clear how this will integrate with iterative solving. One idea might be to fork the process before grounding the new layer so that we can come back to and continue solving the old layer as well at some later point. Ideally as we flip back and forth between layers, we should be able to share learnt clauses between them where relevant.

- Stepless planning is a brand new approach to logic-based planning and and brings with it a lot of unknowns and potentials for future work.

  – We did all this work to improve our encoding of mutex constraints using *multicliques* and then threw that out the window in order

to do stepless planning. Integrating these two things is sure to produce interesting results.

Briefly, we could add two more types of elements to our "bag of occurrencs". These would be occurrences of multicliques and of partitions of multicliques. When a fluent occurs, it has to *belong to* some partition occurence for each partition that it's part of. A partition occurrence in turn has to belong to an occurrence for its associated multiclique. Each occurence of a multiclique can have at most one occurrence of a partition belonging to it and must start after the previous occurrence. In this way, we directly encode the constraint that at most one partition of a multiclique holds at any given time.

On the flip side however, this would mean that we potentially have to "request" many more multiclique occurrences than we do action or fluent occurrences before we have enough to solve the problem. Also, this constraint may not actually be useful to the solver since it doesn't have any obvious effect when coupled with unit propogation (unlike mutex rules and mutex cardinality constraints in the layered approach which have relatively obvious and direct consequences).

– Many interesting planning domains use the : $conditional - effects$ extension to STRIPS PDDL. This allows one to specify that the effects of an action are *conditioned* on some predicate. Extending stepless planning to support : $conditional - effects$ is sure to be an interesting and extremely non-trivial challenge.

  As with mutex constraints, the lack of any notion of simultaneity in stepless planning makes it difficult to extend stepless planning in this way. An effect of an action is expected to depend on whether another fluent is true at the time the action occurs, but if the action is just a node in a directed acyclic graph, it's unclear how to know exactly *what* else is true when it occurs.

– From a performance perspective, it's painful to have to throw out all the work the stepless planner has done and restart from scratch because we didn't have enough occurrences of some fluent or action. We would love to see work on CLINGO to support loop constraints crossing program section boundaries. Using this, stepless planning could potentially become a lot more performant.

  We contacted the developers of CLINGO who told us that such a change isn't theoretically difficult, but would require keeping track of a lot more information than is currently kept by CLINGO and so would be a lot of work to implement.

References

[1] M. Alviano, C. Dodaro, J. Marques-Silva, and F. Ricca, "Optimum stable model search: Algorithms and implementation," *Journal of Logic and Computation*, 2015. 92

[2] J. Amilhastre, P. Janssen, and M.-C. Vilarem, "Computing a minimum biclique cover is polynomial for bipartite domino-free graphs," in *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1997, pp. 36–42. 44

[3] C. Baral, *Knowledge Representation, Reasoning and Declarative Problem Solving*. New York, NY: Cambridge University Press, 2003. 2, 76

[4] A. L. Blum and M. L. Furst, "Fast planning through planning graph analysis," *Artificial Intelligence*, vol. 90, no. 1, pp. 281–300, 1997. 35, 36, 39, 50, 54

[5] A. Blum and M. L. Furst, "Fast planning through planning graph analysis," *Artif. Intell.*, vol. 90, no. 1-2, pp. 281–300, 1997. 1

[6] T. Bylander, "The computational complexity of propositional strips planning," *Artificial Intelligence*, vol. 69, no. 1, pp. 165–204, 1994. 33

[7] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca, and T. Schaub, *ASP-Core-2 input language format*, `https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.01c.pdf`, ASP Standardization Working Group, 2013. 2, 13, 15, 23

[8] Y. Chen, Q. Lv, and R. Huang, "Plan-A: A cost-optimal planner based on sat-constrained optimization," *Proceedings of the 6th International Planning Competition (IPC-08)*, 2008. 53

[9] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962. 8, 54

[10] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres, "Answer set planning under action costs," *Journal of Artificial Intelligence Research*, vol. 19, pp. 25–71, 2003. 82

[11] P. Ferraris, "Answer sets for propositional theories," in *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning*, 2005, pp. 119–131. 2

[12] P. Ferraris, J. Lee, and V. Lifschitz, "Stable models and circumscription," *Artificial Intelligence*, vol. 175, no. 1, pp. 236–263, 2011. 2

[13] R. Fikes and N. J. Nilsson, "STRIPS: a new approach to the application of theorem proving to problem solving," *Artif. Intell.*, vol. 2, no. 3/4, pp. 189–208, 1971. 1

[14] M. Gebser, R. Kaminski, B. Kaufmann, P. Lühne, P. Obermeier, M. Ostrowski, J. Romero, T. Schaub, S. Schellhorn, and P. Wanko, "The potsdam answer set solving collection 5.0," *KI*, vol. 32, no. 2-3, pp. 181–182, 2018. 2

[15] M. Gebser, R. Kaminski, M. Knecht, and T. Schaub, "Plasp: A prototype for PDDL-based planning in asp," in *Proceedings of Logic Programming and Nonmonotonic Reasoning (LPNMR-11)*, Springer, 2011, pp. 358–363. 2

[16] M. Gebser, B. Kaufmann, and T. Schaub, "Conflict-driven answer set solving: From theory to practice," *Artif. Intell.*, vol. 187, pp. 52–89, 2012. 2

[17] ——, "Conflict-driven answer set solving: From theory to practice," *Artificial Intelligence*, vol. 187, pp. 52–89, 2012. 21

[18] ——, "Advanced conflict-driven disjunctive answer set solving," in *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, AAAI Press, 2013, pp. 912–918. 21, 23

[19] M. Gebser, N. Leone, M. Maratea, S. Perri, F. Ricca, and T. Schaub, "Evaluation techniques and systems for answer set programming: A survey," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, 2018, pp. 5450–5456. 2

[20] M. Gelfond, "Answer sets," in *Handbook of Knowledge Representation*, Elsevier, 2008, ch. 7, pp. 285–316. 2

[21] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming," in *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seattle, Washington: MIT Press, 1988, pp. 1070–1080. 2

[22] I. P. Gent and T. Walsh, "Easy problems are sometimes hard," *Artificial Intelligence*, vol. 70, no. 1-2, pp. 335–345, 1994. 7

[23] ——, "The SAT phase transition," in *Proceedings of European Conference on Artificial Intelligence*, 1994, pp. 105–109. 7

[24] M. Ghallab, D. S. Nau, and P. Traverso, *Automated planning - theory and practice*. Elsevier, 2004, ISBN: 978-1-55860-856-6. 1

[25] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968. 68

[26] D. S. Hochbaum, "Approximating clique and biclique problems," *Journal of Algorithms*, vol. 29, no. 1, pp. 174–200, 1998. 44

[27] R. Howey, D. Long, and M. Fox, "Val: Automatic plan validation, continuous effects and mixed initiative planning using pddl," in *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, 2004, pp. 294–301. 90

[28] H. Kautz, "Satplan04: Planning as satisfiability," *Working Notes on the Fourth International Planning Competition (IPC-04)*, pp. 44–45, 2004. 1, 2, 37, 109

[29] H. A. Kautz and B. Selman: "Pushing the envelope: Planning, propositional logic and stochastic search," in *Proceedings of AAAI/IAAI*, vol. 2, 1996, pp. 1194–1201. 1

[30] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, "The DLV system for knowledge representation and reasoning," *ACM Trans. Comput. Log.*, vol. 7, no. 3, pp. 499–562, 2006. 2

[31] V. Lifschitz and A. A. Razborov, "Why are there so many loop formulas?" *ACM Transactions on Computational Logic*, vol. 7, no. 2, pp. 261–268, 2006. 16

[32] F. Lin and Y. Zhao, "ASSAT: computing answer sets of a logic program by SAT solvers," *Artificial Intelligence*, vol. 157, no. 1-2, pp. 115–137, 2004. 16

[33] M. Maratea, "Planning as satisfiability with IPC simple preferences and action costs," *AI Communications*, vol. 25, no. 4, pp. 343–360, 2012. 1, 53

[34] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 38th Annual ACM Design Automation Conference*, 2001, pp. 530–535. 8

[35] C. H. Papadimitriou, *Computational complexity*. Addison-Wesley, 1994. 76

[36] J. Rintanen, "Compact representation of sets of binary constraints," in *Proceedings of 17th European Conference on Artificial Intelligence*, 2006, pp. 143–147. 2, 11, 42, 45, 50, 52, 106

[37] ——, "Planning and SAT," in *Handbook of Satisfiability*, 2009, pp. 483–504. 1

[38] ——, "Heuristics for planning with SAT," in *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, Springer, 2010, pp. 414–428. 96

[39] N. Robinson, C. Gretton, D. N. Pham, and A. Sattar, "A compact and efficient SAT encoding for planning.," in *Proceedings of International Conference on Automated Planning and Scheduling*, 2008, pp. 296–303. 37, 39, 95

[40] N. Robinson, C. Gretton, D.-N. Pham, and A. Sattar, "Cost-optimal planning using weighted MaxSAT," in *Proceedings of the ICAPS'10 Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems*, 2010. 2, 3, 22, 54, 58, 63, 64, 6

[41] J. M. Robson, "Algorithms for maximum independent sets," *Journal of Algorithms*, vol. 7, no. 3, pp. 425–440, 1986. 42

[42] P. Simons, I. Niemelä, and T. Soininen, "Extending and implementing the stable model semantics," *Artificial Intelligence*, vol. 138, no. 1-2, pp. 181–234, 2002. 2

[43] T. Syrjänen, *ASP-Core-2 input language format*, `http://www.tcs.hut.fi/Software/smodels/lparse.ps`, 2000. 12, 13

[44] O. Tange, *GNU Parallel 2018*. Ole Tange, Mar. 2018, ISBN: 9781387509881. DOI: `10.5281/zenodo.1146014`. [Online]. Available: `https://doi.org/10.5281/zenodo.1146014`. 90

[45] Y. Wang, J. You, L. Yuan, and Y. Shen, "Loop formulas for description logic programs," *Theoretical Computer Science*, vol. 415, pp. 60–85, 2012. 16

[46] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict-driven learning in a boolean satisfiability solver," in *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design*, IEEE Press, 2001, pp. 279–285. 1, 7–9

[47] Y. Zhao and F. Lin, "Answer set programming phase transition: A study on randomly generated programs," in *proceedings of 19th International Conference on Logic Programming*, 2003, pp. 239–253. 7

Appendix A

Equivalent Definitions of Multicliques

In Section 6.1, we gave three different (but equivalent) definitions of multicliques

1. *A partitioned graph such that for any two vertices v and w there exists an edge between v and w if and only if v and w belong to separate partitions.*
2. *A graph whose complement is a cluster graph (a set of disjoint cliques).*
3. *A graph which is induced-$(K_1 + K_2)$ free* [1]

   No induced subgraph has the form $(K_1 + K_2)$ (Figure A.0.1)

Let us demonstrate that these representations are indeed equivalent:

Proof. $1 \rightarrow 2$

Consider any graph G which is a multiclique by definition 1. In the complement graph G^c, every partition is a clique. Further, since any two vertices v and w must have an edge if they belong to separate partitions in G it follows that there are no edges between partitions in G^c, therefore, the only edges in G^c belong to cliques.

[1] *This definition was observed by user13136 on cstheory.stackexchange.com*

Figure A.0.1: $K_1 + K_2$

$2 \to 1$

Similarly if G^c is a cluster-graph, then the connected components form the partitions in G as a multiclique.

$2 \to 3$

Suppose there exists vertices x, y, and z in G whose induced graph consists of a single edge xz not joined to y. Then in G^c, we must have edges xy and yz. Then x and y belong to the same cluster, and y and z belong to the same cluster, so x and z must also belong to the same cluster. Then there is also an edge xz in G^c which contradicts the assumption that there is an edge xz in G (since G and G^c are complements).

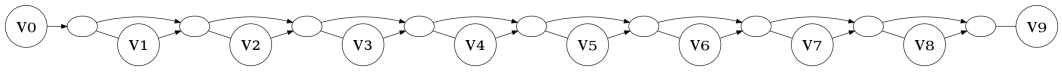
$3 \to 2$

Suppose we have a graph G that does not contain the induced subgraph $K_1 + K_2$. Equivalently, the complement graph G^c does not contain the induced subgraph P_3. Now imagine we have two vertices x and z such that there is a path from x to z in G^c but no direct edge between x and z. WLOG, let x and z be the two indirectly connected vertices for whom the distance between them is minimal in G^c. Let y be the vertex directly after x in the shortest path between them. y is closer to z than x, but since the path between x and y is shortest for any two unconnected vertices, it follows that z and y must be directly connected. Thus the triplet x-y-z has the induced subgraph P_3 which contradicts our assumption. G^c must be a cluster-graph. $\square$

Appendix B

Representing a Mutual Exclusion Clique in SAT

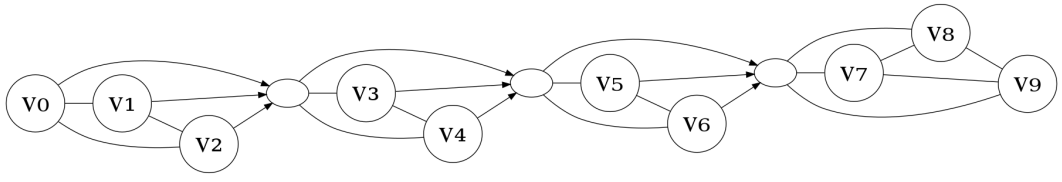
After reading [36], it is natural to wonder what is the most compact representation. Rintanen's linear representation uses $3n - 4$ binary clauses and $n - 1$ auxiliary variables. If the variables we want to represent as mutually exclusive are labeled v_1 through v_n, we can visualize Rintanen's method as:



Here, a directed edge from a to b represents a clause of the form $a \rightarrow b$ or $\neg a \vee b$

Similarly, an undirected edge represents a clause signifying a and b are mutually exclusive (i.e. $a \rightarrow \neg b$ or equivalently $\neg a \vee \neg b$). The unlabeled nodes are auxilliary variables

Our method uses only $3n-6$ binary clauses and $\lceil \frac{n}{2} \rceil - 2$ auxiliary variables. It can be expressed as:



This picture describes the encoding better than a list of boolean equations can, but nonetheless, here is a formal characterization.

First we have a triplet describing v_1

$\neg v_1 \vee \neg v_2$

$\neg v_1 \vee \neg v_3$

$\neg v_1 \vee x_1$

Then for $i \in \left[1, \left\lceil \frac{n}{2} \right\rceil - 1\right]$, we explicitly enforce mutex between each consecutive even-odd pair:

$\neg v_{2i} \vee \neg v_{2i+1}$

Then we describe the x's, for all $i \in \left[1, \left\lceil \frac{n}{2} \right\rceil - 2\right]$

$\neg v_{2i} \vee x_i$

$\neg v_{2i+1} \vee x_i$

$\neg x_i \vee \neg v_{2i+2}$

$\neg x_i \vee \neg v_{2i+3}$

For all $i \in \left[2, \left\lceil \frac{n}{2} \right\rceil - 2\right]$, we have the rule:

$\neg x_{i-1} \vee x_i$

And finally, if n is even, we have one last triplet enforcing the mutex constraints on v_n:

$\neg v_{n-1} \vee \neg v_n$

$\neg v_{n-2} \vee \neg v_n$

$\neg x_{\frac{n}{2}-2} \vee \neg v_n$

Proof. Let us totally order the v's and x's as follows: $v_i < v_j$ iff $i < j$. Furthermore, $x_i < v_j$ iff $2i + 1 < j$ (in the image, the v's and x's are arranged from left to right according to this ordering).

First we prove that for any v_i, there exists an assignment where v_i is true and v_j is false for all $j \neq i$. The assignment is: x_j is false wherever $x_j < v_i$

and x_j is true wherever $x_j > v_i$. This satisfies all the clauses. Additionally, the clauses are all satisfied whenever all the v's and x's are false.

Conversely, let usshow that given any v_i and v_j with $i < j$, we can derive $\neg v_i \vee \neg v_j$ using Robinson resolution. We have a few possible cases.

Case 1: If v_i and v_j belong to the same set (initial triplet, internal pair, or end pair/triplet), then there is a clause directly asserting $\neg v_i \vee \neg v_j$.

Case 2: Otherwise, there is a clause $\neg v_i \vee x_k$ (where x_k is the next x after v_i in our ordering) and some other clause $\neg x_m \vee \neg v_j$ (where x_m is the x immediately before v_j in our ordering) with $k \leq m$

We can resolve the sequence of clauses $\{(\neg x_k \vee x_{k+1}), (\neg x_{k+1} \vee x_{k+2}) \ldots (\neg x_{m-1} \vee x_m)\}$ to get $(\neg x_k \vee x_m)$. Resolving this with the above two clauses gives $\neg v_i \vee \neg v_j$. □

Conjecture. *For all $n \geq 5$, this representation minimizes the number of clauses required to describe a mutual exclusion clique in n variables (For $n \leq 4$, the optimal representation is the direct representation using 0 auxiliary variables $\binom{n}{2}$ clauses). Furthermore, among all such minimal representations, this one also minimizes the number of auxiliary variables required.*

Appendix C

List of Selected ASP Planners Used in this Paper

C.1 Simple Planner

At most one action per step, no plan graph, no preserving actions, no neededness. Run with an increasing finalStep until satisfiable.

```
holds(F,0) :- init(F).
{happens(A,T)} :- holds(F,T) : pre(A,F); step(T).
:- {happens(A,T)} > 1; step(T).
holds(F,T+1) :- happens(A,T); add(A,F).
deleted(F,T) :- happens(A,T); del(A,F).
holds(F,T+1) :- holds(F,T); not deleted(F,T).
:- not holds(F,K); finalStep(K); goal(F).
```

C.2 ASPPlan

Rules translated from SATPlan [28], using planning graph and preserving actions Run with an increasing finalStep until satisfiable.

```
holds(F,K) :- goal(F); finalStep(K).
holds(F,0) :- init(F).
happens(A,K-1) : add(A,F),validAct(A,K-1) :- holds(F,K); K > 0.
holds(F,K) :- pre(A,F); happens(A,K); validFluent(F,K).
```

```
:- mutexAct(A,B); happens(A,K); happens(B,K).
:- mutex(F,G); holds(F,K); holds(G,K).
```

C.3 AASPPlan

Like ASPPlan, but with a smarter encoding for mutex fluents and actions. Requires that the mutex graph be encoded as multicliques (see algorithm 6.1)

```
holds(F,K) :- goal(F); finalStep(K).
holds(F,0) :- init(F).
happens(A,K-1) : add(A,F),validAct(A,K-1) :- holds(F,K); K > 0.
holds(F,K) :- pre(A,F); happens(A,K); validFluent(F,K).

used_preserved(F,K) :- happens(A,K); pre(A,F); not del(A,F).
deleted_unused(F,K) :- happens(A,K); del(A,F); not pre(A,F).
:- {used_preserved(F,K);
    deleted_unused(F,K);
    happens(A,K) : pre(A,F),del(A,F)} > 1; valid_at(F,K).

deleted(F,K) :- happens(A, K); del(A, F).
:- holds(F,K); deleted(F,K-1).

partitionHolds(P,K) :- holds(F,K); inPartition(F,P).
:- {p(P,K): partitionHolds(P,K),inMulticlique(P,M)} > 1;
    multiclique(M); layer(K).
```

C.4 Delete-Free Planner

with action costs. No external handling needed. Run it on the problem; get an optimal solution.

```
holds(F) :- goal(F).
{happens(A) : add(A,F)} >= 1 :- holds(F); not init(F).
```

```
holdsPre(F) :- pre(A,F); happens(A).

supportFluent(F) :- init(F); holds(F).
supportAct(A) :- supportFluent(F) : pre(A,F), holds(F); happens(A).
supportFluent(F) :- supportAct(A); happens(A); add(A,F); holds(F).

:- holds(F); not supportFluent(F).
:~ happens(A); cost(A,C).[C,A]
```

C.5 Variant II encoding for Cost-Optimal ASP-Plan with Suffix Layer

(Note: Variant I is just AASPPlan + cost-optimality constraint)

```
holdsSuff(F) :- goal(F).
{happensSuff(A) : add(A,F), not preserving(A); subgoal(F)} >= 1 :-
           holdsSuff(F).
holdsSuff(F) :- pre(A,F); happensSuff(A).

supportFluent(F) :- subgoal(F), holdsSuff(F).
supportAct(A) :- supportFluent(F) : pre(A,F), holdsSuff(F); happensSuff(A).
supportFluent(F) :- supportAct(A); happensSuff(A);
                  add(A,F); holdsSuff(F).

:- holdsSuff(F); not supportFluent(F).
:~ happensSuff(A); cost(A,C).[C,A]
cheating :- happensSuff(A).

{happensFirst(A)} :- happensSuff(A); subgoal(F) : pre(A,F).
:- cheating; not happensFirst(A) : action(A).

holds(F,K) :- subgoal(F); finalStep(K).
```

```
holds(F,0) :- init(F).
happens(A,K-1) : add(A,F),validAct(A,K-1) :- holds(F, K); K > 0.
holds(F,K) :- pre(A,F); happens(A,K); validFluent(F,K).

used_preserved(F K) :- happens(A, K); pre(A,F); not del(A,F).
deleted_unused(F,K) :- happens(A, K); del(A,F); not pre(A,F).
:- {used_preserved(F,K);
    deleted_unused(F,K);
    happens(A,K) : pre(A,F),del(A,F)} > 1; valid_at(F,K).

deleted(F,K) :- happens(A,K); del(A,F).
:- holds(F,K); deleted(F,K-1).

partitionHolds(P,K) :- holds(F,K); inPartition(F,P).
:- {p(P,K): partitionHolds(P,K),inMulticlique(P,M)} > 1;
    multiclique(M); layer(K).

used(F,K) :- happens(A,K); pre(A,F); not preserving(A).

% Cost-optimality constraint
:~ happens(A,K); cost(A,C).[C,A,K]

:- happens(A,K); K > 1;
    not preserving(A);
    holds(F,K-1) : pre(A,F);
    not used(F,K-1) : del(A,F);
    not deleted(F,K-1) : add(A,F), holds(F,K).

:- happensFirst(A); K >= 1;
    finalStep(K);
    holds(F,K) : pre(A,F);
```

```
    not used(F,K) : del(A,F);
    not deleted(F,K) : add(A,F), holdsSuff(F,K).
```

C.6 Stepless Planner with Suffix Layer

Requires an external program to detect which fluents and actions are saturated each time the suffix layer is used and feed in more occurrences.

```
is(fluentOcc(F,1)) :- fluent(F).
is(actOcc(A,1)) :- action(A).
is(fluentOcc(F,0)) :- init(F).



% ============================== Problem description ============================

% Helper function to recognize subsequent occurences of the same
% fluent/action.
nextOcc(fluentOcc(F,0),fluentOcc(F,1)) :- fluent(F).
nextOcc(fluentOcc(F,M),fluentOcc(F,M+1)) :- is(fluentOcc(F,M)).
nextOcc(actOcc(A,N),actOcc(A,N+1)) :- is(actOcc(A,N)).

% Any fluent above level 0 which holds must have exactly one causing
% action.
{causes(actOcc(A,N),fluentOcc(F,M)) : add(A,F), is(actOcc(A,N))}=1 :-
  holds(fluentOcc(F,M)); M > 0.
% If an action causes a fluent, it happens.
happens(AO) :- causes(AO,_).
% An action cannot cause more than one occurrence of the same fluent.
:- {causes(AO,fluentOcc(F,M))} > 1; is(AO); fluent(F).
% For each precondition an action occurrence has, some occurrence of
% that fluent must permit it.
{permits(fluentOcc(F,M),actOcc(A,N)) : is(fluentOcc(F,M))}=1 :-
```

```
  happens(actOcc(A,N)); pre(A,F).
% A fluent occurrence which permits an action must hold.
holds(FO) :- permits(FO,_).
% A fluent which is used to satisfy a subgoal condition "permits" it.
% For each subgoal condition, exactly one occurrence of that fluent
% permits it.
{permits(fluentOcc(F,M),subgoal(F)) : is(fluentOcc(F,M))}=1 :-
  subgoal(F).
% A fluent which permits a subgoal condition cannot be deleted.
:- deleted(FO); permits(FO,subgoal(_)).

% An occurrence of an action deletes an occurrence of a fluent if
% it permits it and that action has the fluent as a delete effect.
deletes(actOcc(A,N),fluentOcc(F,M)) :-
  permits(fluentOcc(F,M),actOcc(A,N)); del(A,F).
% No fluent may be deleted by more than one action.
:- {deletes(_, FO)} > 1; is(FO).
% An action which deletes a fluent, but doesn't have it as a
% precondition follows some occurrence of that fluent. Can possibly
% follow occurrence index 0 even if the fluent is not an initial
% fluent (indicates this action occurs before any occurrence of
% that fluent).
{follows(actOcc(A,N),fluentOcc(F,M)) : holds(fluentOcc(F,M));
  follows(actOcc(A,N),fluentOcc(F,0))}=1 :-
  del(A,F); not pre(A,F); happens(actOcc(A,N)).
% Fluent occurrences 0 which aren't initial fluents count as "deleted".
deleted(fluentOcc(F,0)) :- fluent(F); not init(F).
% A fluent is deleted if something deletes it.
deleted(FO) :- deletes(_, FO).
% A fluent is deleted if something follows it.
deleted(FO) :- follows(_, FO).
```

```
% Weak constraint charging the cost of an action occurrence.
:~ happens(actOcc(A,N)); cost(A,V).[V,A,N]

% An occurrence of a fluent doesn't hold if its previous occurrence
% doesn't hold.
:- holds(fluentOcc(F,M+1)); not holds(fluentOcc(F,M));
    is(fluentOcc(F,M)); M > 0.
% An occurrence of an action doesn't happen if its previous occurrence
% didn't happen.
:- happens(BO); not happens(AO); nextOcc(AO,BO).


% ============================== Plan Event Graph ==============================

% Events in the graph; these will be grouped into vertices
event(start(FO)) :- holds(FO).
event(end(FO)) :- holds(FO).
event(end(fluentOcc(F,0))) :- fluent(F).
event(AO) :- happens(AO).
% subgoals are events
event(subgoal(F)) :- subgoal(F).

% Triggering actions
% The start of a fluent by its causing action.
actionTriggers(AO,start(FO)) :- causes(AO,FO).
% The end of a fluent by its deleting action.
actionTriggers(AO,end(FO)) :- deletes(AO,FO).

% Vertices
% If no action triggers an event, then it gets a vertex by itself.
```

```
vertex(V) :- event(V); not actionTriggers(A,V) : is(A).
% Otherwise it belongs to the vertex for its trigger action.
inVertex(E,V) :- actionTriggers(V,E).
% Every event which is the name of a vertex belongs to that vertex.
inVertex(V,V) :- vertex(V).

% Graph edges
% A fluent ends after it starts
edge(start(FO),end(FO)) :- holds(FO).
% If a fluent permits an action, then the action happens after
% the start of the fluent
edge(start(FO),AO) :- permits(FO,AO).
% If a fluent permits an action but the action doesn't delete the
% fluent, then the action happens before the end of the fluent.
edge(AO,end(FO)) :- permits(FO,AO); not deletes(AO,FO).
% An action happens after the fluent it follows
edge(end(FO),AO) :- follows(AO,FO).
% but before the next occurrence
edge(AO,start(GO)) :- follows(AO,FO); nextOcc(FO,GO); holds(GO).
% The start of the next occurrence of a fluent happens after the
% end of the previous occurrence
edge(end(FO),start(GO)) :- holds(GO); nextOcc(FO,GO).
% The next occurrence of an action happens after the previous
% occurrence
edge(AO,BO) :- happens(AO); happens(BO); nextOcc(AO,BO).

% And now we use stable models to assert that the graph is acyclic; sup(X)
% indicates that X has acyclic support going back to the root of the graph.

% The input for a given event has support if all events joined
% by any incoming edge have support.
```

```
sup(in(E)) :- sup(D) : edge(D,E); event(E).
% A vertex has support if all of its events' inputs have support.
sup(V) :- sup(in(E)) : inVertex(E,V); vertex(V).
% An event has support if its vertex has support.
sup(E) :- sup(V); inVertex(E,V).
% Every vertex must have support.
:- vertex(V); not sup(V).


% ============================== Strong minimality ==============================

% A counterexample to strong minimality consists of two cuts, cut1
% and cut2.
cut(cut1; cut2).

% For each vertex V and each cut C, V is on either the s-side or
% the t-side of V. Note this rule is disjunctive.
onSideOf(V,s,C) | onSideOf(V,t,C) :- vertex(V); cut(C).
% An event belongs to the cut side of its vertex.
onSideOf(E,X,C) :- inVertex(E,V); onSideOf(V,X,C).
% Any subgoal is always on the t-side of cut2.
onSideOf(subgoal(F),t,cut2) :- subgoal(F).
% If there's a directed edge from D to E, but D is on the t-side
% and E is on the s-side, this is not a cut (invalidating this
% counterexample to strong minimality).
not_counterexample :- edge(D,E); onSideOf(D,t,C); onSideOf(E,s,C).
% If a fluent starts on the s-side of cut2 and ends on the t-side,
% then it "holds over" cut2.
holdsOver(FO,cut2) :-
  onSideOf(start(FO),s,cut2); onSideOf(end(FO),t,cut2).
% Similarly if it starts and ends on the same side of cut1, then it
```

```
% doesn't hold over cut1.
not_holdsOver(FO,cut1) :-
  onSideOf(start(FO),X,cut1); onSideOf(end(FO),X,cut1).
% Action occurrence AO is not between cut1 and cut2 if it's on the
% s-side of cut1 or the t-side of cut2.
not_betweenCuts(AO) :- onSideOf(AO,s,cut1).
not_betweenCuts(AO) :- onSideOf(AO,t,cut2).
% If no action occurs between the two cuts, then this is not a
% counterexample.
not_counterexample :- not_betweenCuts(AO) : happens(AO).
% If there exists a fluent for which some occurrence holds over cut2,
% but no occurrence holds over cut1, then this is not a counterexample.
not_counterexample :-
  holdsOver(fluentOcc(F,_),cut2);
  not_holdsOver(fluentOcc(F,M),cut1) : holds(fluentOcc(F,M)).

% There should be no counterexample (sorry for the triple negative).
:- not not_counterexample.

% If this is not a counterexample, all atoms must hold.
onSideOf(V,s,C) :- vertex(V); cut(C); not_counterexample.
onSideOf(V,t,C) :- vertex(V); cut(C); not_counterexample.



% ============================== Suffix Layer ================================

% All goal fluents hold in the suffix layer.
suffix(holds(F)) :- goal(F).
% If a fluent holds in the suffix layer, either some action causes it
% or it is a subgoal.
{subgoal(F); suffix(causes(A,F)) : add(A,F)} = 1 :- suffix(holds(F)).
```

```
% If an action causes a fluent in the suffix, it happens.
suffix(happens(A)) :- suffix(causes(A,_)).
% If an action occurs in the suffix layer, then all of its
% preconditions hold in ths suffix layer
suffix(holds(F)) :- suffix(happens(A)); pre(A,F).

% If any action happens in the suffix layer, then we are using the
% suffix layer.
useSuffix :- suffix(happens(_)).

% A fluent is supported in the suffix if it's a subgoal
suffix(sup(holds(F))) :- subgoal(F).
% An action is supported in the suffix if all of its preconditions are
suffix(sup(happens(A))) :-
  suffix(sup(holds(F))) : pre(A,F); suffix(happens(A)).
% A fluent is supported in the suffix if its causing action is
suffix(sup(holds(F))) :- suffix(sup(happens(A))); suffix(causes(A,F)).

% No action happens in the suffix without support
:- suffix(happens(A)); not suffix(sup(happens(A))).
% No fluent holds in the suffix without support
:- suffix(holds(F)); not suffix(sup(holds(F))).

% Actions that happen in the suffix layer impose their cost.
:~ suffix(happens(A)); cost(A,V).[V,A,suffix]
% Very weak preference to avoid using the suffix layer.
:~ useSuffix.[1@-1]


% ================================ Saturated ===================================
```

```
% A fluent is saturated if all occurrences of it hold (besides the 0th).
saturated(fluent(F)) :-
  holds(fluentOcc(F,M)) : is(fluentOcc(F,M)),M>0; fluent(F).
% An action is saturated if all occurrences of it happen.
saturated(action(A)) :-
  happens(actOcc(A,N)) : is(actOcc(A,N)); action(A).

% If an action happens in the suffix layer and all of its preconditions
% are subgoals, we designate it a "starting" action.
suffix(start(action(A))) :- subgoal(F) : pre(A,F); suffix(happens(A)).
% Any fluent caused by a starting action is designated a "starting"
% fluent.
suffix(start(fluent(F))) :- suffix(start(action(A))); suffix(causes(A,F)).

% Guarantees that some starting action or fluent will be saturated.
:- useSuffix; not saturated(X) : suffix(start(X)).



% ================================================================================

#show causes/2.
#show deletes/2.
#show happens/1.
#show holds/1.
#show permits/2.
#show follows/2.

#show suffix(happens(A)) : suffix(happens(A)).
```