APPLYING CONSTRAINT SATISFACTION TECHNIQUES TO AI PLANNING PROBLEMS

by

Daniel Buettner

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Berthe Choueiry

Lincoln, Nebraska

December, 2003

APPLYING CONSTRAINT SATISFACTION TECHNIQUES TO AI PLANNING PROBLEMS

Daniel Buettner, M.S.

University of Nebraska, 2003

Advisor: Berthe Choueiry

An AI planning problem is one in which an agent capable of perceiving certain states and of performing some actions finds itself in a world, needing to achieve certain goals. A solution to a planning problem is an ordered sequence of actions that, when carried out, will achieve the desired goals.

Constraint satisfaction is a general method of problem formulation in which the goal is to find values for variables such that these values do not violate any constraints that hold between the variables. This problem formulation can be used to solve many problems in artificial intelligence, computer science, and engineering.

In this thesis, we study a particular formulation of AI planning problems as constraint satisfaction problems. Using this formulation, we study the utility of dynamic neighborhood partial interchangeability, but find that it is not applicable. We use tree decomposition to justify the choice of static variable orderings with forward checking. We also show the effectiveness of maintaining arc consistency (MAC) in solving AI planning problems so formulated. In addition, we present an improvement to the MAC algorithm that is not described in the CSP literature. The improvement to MAC uses no special properties of AI planning problems, so we expect that this improvement will prove fruitful in the context of solving general CSPs.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Berthe Y. Choueiry, for her guidance and feedback during the course of my research and the writing of this thesis.

I am grateful to Dr. Ashok Samal and Dr. Peter Revesz for being a part of my examination committee. Their suggestions have only made this thesis better.

I would also like to thank my colleagues in the Constraint Systems Laboratory, especially Anagh Lal, Eric Moss, Lin Xu, and Hui Zou, for their friendship and support while I was working on this thesis.

This research is supported by a seed grant from NASA Nebraska and the CAREER Award #0133568 from the National Science Foundation.

Contents

1 Introduction **1**
1.1 Questions addressed 2
1.2 Summary of contributions 3
1.3 Guide to thesis 4

2 Background **6**
2.1 Planning 6
2.1.1 Partial-order planning 9
2.1.2 GraphPlan 10
2.1.3 SAT solutions to planning problems 14
2.2 Constraint satisfaction 15
2.2.1 Definition 15
2.2.2 Solution extraction 17
2.3 GP-CSP 18
2.3.1 The GP-CSP planner 19
2.3.2 Details of GP-CSP 21
2.3.3 Enhancements to the CSP solver 24

3 Our changes to GP-CSP **27**
3.1 Our investigations 27
3.2 Planning problems 28
3.3 Preprocessing 29
3.4 Variable ordering 31
3.5 CSP sizes 34
3.6 Additional extensions 35

4 Exploiting partial interchangeability **36**
4.1 Background 36
4.2 Dynamic neighborhood partial interchangeability 37
4.3 Application to planning problems 39
4.4 Conclusion 41

5 Using decomposition methods **43**
5.1 Background . . . 43
5.2 Decomposing a planning problem . . . 44
5.3 Experiments . . . 46
5.4 Conclusion . . . 48

6 MAC **49**
6.1 Background . . . 49
6.2 Results . . . 50
6.2.1 Static variable ordering . . . 50
6.2.2 Dynamic variable ordering . . . 53
6.2.3 Choosing a variable ordering . . . 56
6.3 Improvement to the MAC algorithm . . . 58
6.4 Conclusion . . . 62

7 Conclusion **63**
7.1 Summarizing our contributions . . . 63
7.2 Future work . . . 64

A Implementation details **66**
A.1 Project structure . . . 66
A.2 CSP representation . . . 68
A.3 Solution extraction . . . 69
A.3.1 Further details . . . 70
A.4 Test problems . . . 72

Bibliography **75**

List of Figures

2.1 *Excerpts from a PDDL definition of a planning problem.* 8
2.2 *On the left, action A3 has been added to an existing plan. However, A3 threatens (c) which is required by A2. To resolve this threat, A3 is demoted before A1, as illustrated on the right.* 10
2.3 *Alternating layers of facts and actions. To prevent clutter, only the `noop` from F4 to F8 is shown. In reality, every fact supports a `noop`.* 11
2.4 *An example CSP: list coloring problem* 16
2.5 *The architecture of the GP-CSP system.* 19
2.6 *Excerpts from a PDDL definition of a planning problem.* 22
2.7 *Examples of a `MUTEX CONSTRAINT` and `ACTIVITY CONSTRAINT`s.* . . 23
2.8 *Example of a `FACT MUTEX CONSTRAINT`.* 24

4.1 Left: *CSP.* Right: *JDT of V2, V3.* 38
4.2 *Extended rocket problem.* 40

5.1 Top: *Original CSP.* Bottom: *CSP split into independent subproblems.* 45

A.1 *Domain definition for the rocket domain.* 73
A.2 *Problem definition for the rocket problem.* 74

List of Tables

3.1 *Results of pruning applied to planning CSPs* 30
3.2 *Constraint checks, number of nodes visited, and CPU time (in seconds) required to solve planning CSPs using various orderings. LDF is largest domain first, DDR is domain/degree ratio, DEG is degree, GPO is the default GraphPlan ordering, random is the average from 10 trials.* . . 33
3.3 *Size of various planning problems encoded as CSPs* 34

5.1 *Results of decomposition* 46

6.1 *Constraint checks, number of nodes visited, and CPU time (in seconds) required to solve planning CSPs using MAC with various static orderings. LDF is largest domain first, DDR is smallest domain/degree ratio, DEG is smallest degree, GPO is the default GraphPlan ordering.* 52
6.2 *Constraint checks, number of nodes visited, and CPU time (in seconds) required to solve planning CSPs using MAC and random variable orderings. The data presented is computed from 100 random orderings per problem. The final column tabulates the number of random orderings that exceeded the allowed 180 seconds of CPU time.* 53
6.3 *Constraint checks, number of nodes visited, and CPU time (in seconds) required to solve planning CSPs using various solution extraction methods. The first column presents results of FC using the default GraphPlan ordering, the second column presents the best performance for MAC with the associated static ordering in parentheses, the third column presents the results of MAC using dynamic least domain first variable ordering, the fourth column presents the results of MAC using dynamic domain/degree ratio variable ordering, and the final column represents the results of MAC using dynamic domain/future variables ordering.* 54
6.4 *Size of various planning problems encoded as CSPs* 56
6.5 *Average domain sizes excluding pruned variables* 57
6.6 *Density and tightness of the solution bearing level of various planning problems encoded as CSPs* 58

6.7 *Constraint checks and CPU time (in seconds) required to solve planning CSPs using the standard GraphPlan ordering. The first column presents results of the improved MAC while the second column presents results of the normal MAC. The final column shows the speedup.* . . 61

List of Algorithms

1 Improved iterative implementation of MAC 60

Chapter 1

Introduction

Much of the framework used to model and solve AI planning problems can be traced to the STRIPS solver [9]. This program was created for use with a mobile robot, in order to help it with tasks like navigation and rearranging objects in the world. Like the planner discussed in this thesis, STRIPS worked with planning problems that represented the initial conditions and the goal conditions as conjunctions of facts, while actions were represented by listing the preconditions and effects of each action. The preconditions of an action are simply the facts that must be true before an action can be applied, while the effects of an action are simply the collection of facts whose truth is changed by the application of an action. The influence of STRIPS on AI planning can be seen in the problem description syntax used by modern planners. Problems are still described by listing the initial conditions, the goals conditions and schema for available actions in terms of preconditions and effects.

There are many approaches that can be taken to solving AI planning problems. In this thesis we concentrate on a formulation of such problems as Constraint Satisfaction Problems. Constraint Satisfaction [15] is used to model and solve various problems in computer science, engineering, database, and management. Scheduling and resource

allocation problems are commonly solved using Constraint Satisfaction techniques.

After presenting a method of formulating AI planning problems as a CSP, we turn our attention to examining the applicability and effectiveness of various CSP techniques to this class of problems. In particular, we investigate variable ordering in chapter 3, Dynamic Neighborhood Partial Interchangeability in chapter 4, CSP decomposition in chapter 5, and Maintaining Arc Consistency in chapter 6.

1.1 Questions addressed

In this thesis, we address the following questions:

1. Are CSP techniques suitable for solving AI planning problems?

   Answer: As discussed in [8] and further explored here, these techniques are quite effective.

2. Is Dynamic Neighborhood Partial Interchangeability (DNPI) effective in the context of CSP formulations of AI planning problems?

   Answer: No, when formulated as a CSP as in [8], AI planning problems do not exhibit the structure required by DNPI. This is a consequence of the enumeration of objects required by planning description languages.

3. Is there an effective conjunctive decomposition for planning problems formulated as CSPs?

   Answer: No, planning graphs are too *dense*, in that a single planning graph contains solutions to many problems. Hence any subproblem that does not include the goal information will discover many, many partial solutions to problems that we are not trying to solve.

4. Is Maintaining Arc Consistency (MAC) [20] an effective algorithm for finding solutions to CSP formulations of AI planning problems?

   Answer: Yes, MAC seems to perform quite well on the tested planning problems.

5. How does MAC perform with various variable orderings?

   Answer: While Forward Checking quickly gets lost if solution extraction doesn't begin at the goal level, MAC is able to perform well with other variable orderings, including ones that do not begin at the goal level. In fact, certain dynamic ordering strategies give the best performance on tested problems.

1.2 Summary of contributions

Our contributions are as follows:

1. In the course of automatic conversion of the planning graph created by GraphPlan into a CSP, a variable pruning method was discovered. This allows us to detect certain dual-valued variables that can have no impact on the planning problem. These variables can be reduced to a single value and their constraints can safely be ignored. These variables then correspond to facts that can never be negated, and these are the sorts of facts that often express typing information. For example, the precondition for an action which loads a package into a truck should require that the thing being loaded actually *be* a package. This is generally encoded by creating a fact for each box that explicitly labels it of type package. These kinds of facts are never negated, and these are the CSP variables that we are able to prune.

2. We have shown the ineffectiveness of DNPI in the context of AI planning problems. This is a consequence of the formalisms used to express such problems

and the CSP encoding that we have used. This cannot be resolved without finding a new formalism, adding an extra layer of abstraction, or possibly by finding a different CSP encoding of planning problems.

3. The CSP literature provides no clear iterative algorithm for MAC. We provide such an algorithm, as well as an enhancement to MAC that can provide a several-fold increase in performance.

4. We identify some dynamic variable ordering strategies for MAC that seem to work well in the context of planning problems.

5. We identify areas for further research. Among these are a more thorough investigation of the improvement to MAC in the context of general CSPs and an investigation into different encodings of planning problems as CSPs.

1.3 Guide to thesis

This thesis is structured in the following way. Chapter 2 gives an introduction to the concepts behind AI planning and Constraint Satisfaction Problems (CSPs). Chapter 3 presents a method for automatically converting planning problems to CSPs and solving them in this representation. Chapter 4 discusses the concepts of Dynamic Neighborhood Partial Interchangeability (DNPI) and why it cannot be applied to our formulation of planning problems. Chapter 5 discusses a decomposition strategy for the CSPs representing planning problems. Chapter 6 features a discussion of the full lookahead solution finding method called Maintaining Arc Consistency (MAC). We show here the effectiveness of MAC in solving AI planning problems and present a simple improvement to the algorithm. Finally, chapter 7 reviews our contributions and gives some direction for future work.

Additionally, Appendix A presents an overview of the code we have written to formulate and store a planning problem as a CSP. The appendix also discusses the implementations of the solution extraction methods which we tested. The implementation is in C++ and is extended from the GP-CSP system of Do and Kambhampati [8].

Chapter 2

Background

The work described in this thesis uses Constraint Satisfaction techniques to solve classical AI Planning Problems. This chapter provides relevant background in these areas. Section 2.1 describes the ideas and techniques behind Planning, while section 2.2 introduces Constraint Satisfaction Problems. Section 2.3 introduces the planning system that our work extends.

2.1 Planning

A planning problem is one in which an agent capable of some sensing and of performing some actions finds itself in some world, needing to achieve certain goals. A simple example of such a problem is one in which two men, Jason and Alex, are initially in London with Jason wishing to travel to New York and Alex wishing to travel to Paris. There are two rockets in London, each capable of carrying one or more persons and making a single flight. A solution to a planning problem is an ordered sequence of actions that, when carried out, will achieve the desired goals.

This particular example problem is very simple and most humans can easily see that it can be solved by loading Alex into one rocket and flying it to Paris, and loading

Jason into the other rocket and flying it to New York. To solve this problem, humans will often use ad hoc methods and intuit the correct sequence of actions. However, automated planning systems must use a more rigorous methodology to ensure plan validity and, in some cases, to guarantee some notion of plan optimality.

More formally, a planning problem should specify three things [24].

1. A description of the world's initial state (as a set of facts).
2. A description of the agent's goal (as a set of facts).
3. A description of the possible actions that can be carried out to affect the state of the world. Actions are described by their preconditions and effects. The preconditions of an action are a set of facts that must be true before an action can be performed. The effects of an action are sometimes divided into an *add list* and a *delete list*, where the add list contains facts that are made true by the action and the delete list contains facts that are made false by the action.

There are several domain description languages that are used to provide a uniform syntax for encoding this information. Planning systems are written to parse these standard formats, which enables us to easily share problems and compare results. Since 1998, the Artificial Intelligence Planning Systems competition has used the PDDL [11] language to specify planning problems. Portions of a PDDL description of the example planning problem outlined above can be found in Figure 2.1. This excerpt shows two action definitions, one for `move` and one for `unload`. These actions are part of the *domain* definition. Figure 2.1 also lists the initial conditions of the world, and the goal of the planning problem; these are part of the *problem* definition.

The `action` definitions are part of the *domain* definition. The `init` and `goal` definitions form part of the *problem* definition. Domain definitions are more general than problem definitions, in that they merely express the propositions that are defined

```
(:action move
:parameters (?r ?from ?to)
:precondition (and (has-fuel ?r) (at ?r ?from) (rocket ?r) (place ?from)
                   (place ?to))
:effect (and (at ?r ?to) (not (has-fuel ?r)) (not (at ?r ?from))))

(:action unload
:parameters (?c ?r ?p)
:precondition (and (at ?r ?p) (in ?c ?r) (rocket ?r) (cargo ?c)
                   (place ?p))
:effect (and (at ?c ?p) (not (in ?c ?r))))

(:init (at r1 london) (at r2 london) (at alex london) (at jason london)
       (has-fuel r1) (has-fuel r2) (rocket r1) (rocket r2)
       (place london) (place paris) (place jfk) (cargo alex)
       (cargo jason))

(:goal (and (at alex paris)
            (at jason jfk)))
```

Figure 2.1: *Excerpts from a PDDL definition of a planning problem.*

and the actions that can be performed in a particular world. Problem definitions indicate which domain is to be used for generating actions. Additionally, they specify the initial state of the world and the goal conditions that must be achieved to solve the planning problem.

Several simplifying assumptions are made in classical AI planning:

- All actions require a single, uniform, unit of time to execute.
- Actions will be successful and produce their expected results.
- The agent knows the initial state of the world, as well as the impact of its own actions on the state of the world.
- The only change in the world is the result of the agent's own actions.

These are somewhat limiting assumptions, but some of the more recent planning systems are able to deal with problems that violate one or more of these assumptions.

Of course, such planners must be more complex to handle the relaxation of one or more of these assumptions [21]. Below, we review three main types of planning systems: POP [5], GraphPlan [3], and SAT-based [14].

2.1.1 Partial-order planning

The partial-order planner (POP) algorithm was an early planning algorithm that met with success using the idea of partial-order planning. In partial-order planning, you practice *least-commitment planning* where only essential ordering information is recorded. That is to say, if there are, for example, a pair of actions that can be executed in either order without impacting the outcome of the plan, a least commitment planner would not decide on an explicit ordering for these actions, thus leaving the decision to the execution agent.

In partial-order planning, plans are constructed incrementally, with *causal links* used to record information on why a particular action has been added to a plan. When a new action is added, all of the previous causal links must be checked to make sure that this new action does not interfere with those already chosen. An action that would interfere with another is called a *threat*. It may be possible to handle a threat by introducing an explicit ordering constraints between actions. When an action A_1 is moved before another action A_2, we say that A_1 is demoted. When A_1 is moved after A_2, we say that A_1 is promoted. This is illustrated in Figure 2.2.

The POP algorithm carries out partial-order planning in the following manner. To initialize a problem, a `*start*` action is created with the facts of the initial state as its effects. Additionally, an `*end*` action is created with the goals as its preconditions. The unsatisfied preconditions of any actions in the partial plan are added to a so-called *agenda*. Upon initialization, the only unsatisfied preconditions belong to the `*end*` action and they are the goals of the planning problem itself. The next step is

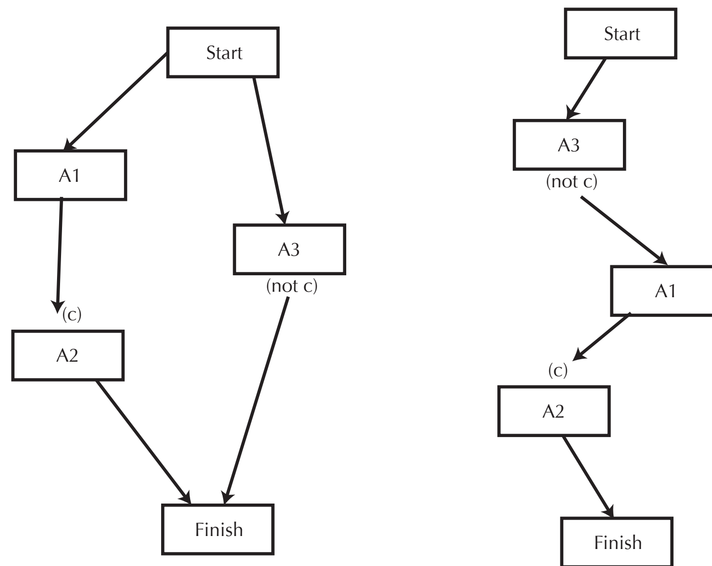

Figure 2.2: *On the left, action A3 has been added to an existing plan. However, A3 threatens (c) which is required by A2. To resolve this threat, A3 is demoted before A1, as illustrated on the right.*

to choose a fact from the agenda and find an action that can be used to produce this fact. A causal link is recorded to indicate the reason for adding this new action. The new action must be checked to ensure that it does not threaten any actions already in the plan. If there is a threat it will, if possible, be handled by introducing ordering constraints between the actions involved. Any unsatisfied preconditions of the new action are added to the agenda and the process continues until the agenda is empty.

Partial-order planning has fallen out of favor in recent years, as planners using other methods can out-perform partial-order planners in most domains [25].

2.1.2 GraphPlan

In 1995 Blum and Furst introduced GraphPlan, a method for modeling and solving planning problems that proved to be rather revolutionary [3]. It offered much better performance in terms of both CPU time and solvable problem size than the other

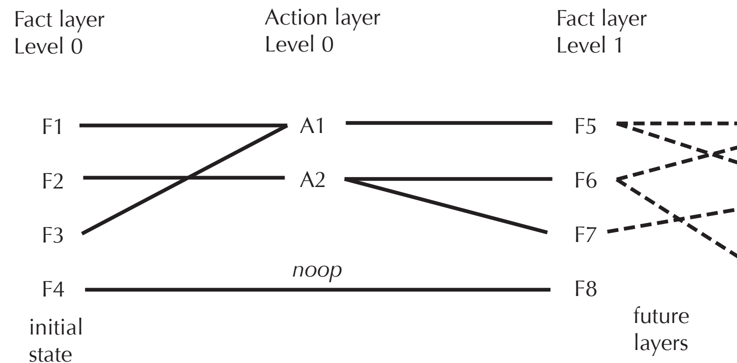

Figure 2.3: *Alternating layers of facts and actions. To prevent clutter, only the* `noop` *from F4 to F8 is shown. In reality, every fact supports a* `noop`*.*

planners of the day. While today's high performance planners are nonsystematic[1] in order to take advantage of the great speedups available to those willing to sacrifice optimality, in some cases optimality is still important, and today's planners that make that guarantee are still based on GraphPlan.

GraphPlan finds solutions to planning problems by constructing a *planning graph* level-by-level, and searching for valid plans within that graph. Planning graphs consist of alternating layers of facts and actions (see Figure 2.3). Each time a new fact layer is created, GraphPlan checks to see if the goals are found within that layer. If they are, a simple backtracking search is launched to attempt to find a valid plan. If a plan is found, it is output and GraphPlan halts. Otherwise, it extends the planning graph, creating a new action layer followed by a new fact layer.

When a planning graph is first created, it simply contains a fact layer made up of the initial conditions that specify the *initial state* of the planning problem. Planning graphs are extended by finding all actions whose prerequisites are satisfied at the most recent fact level and adding these actions in the next layer of the graph. The

[1]That is, they are based on stochastic search.

effects of these actions form the next fact layer. Of note is the fact that a so-called `noop` action exists that can take any fact as a prerequisite with that same fact as the action's only effect. In this way, once a fact becomes true, `noop`s will propagate that fact forward each time the planning graph is extended. Plan extension is carried out until all the properties of the goal state are satisfied.

During plan extension, GraphPlan discovers binary mutual exclusion (*mutex*) relationships between actions and facts. A mutex is declared between a pair of actions when:

- The effect of one action is the negation of another action's effect (termed *inconsistent effects*).
- One action deletes the precondition of another (termed *inconsistent effects*).
- The actions have preconditions that are marked as mutex (termed *competing needs*).

For example, one action may have `(door open)` as a prerequisite while the other action has `(not (door open))` as a prerequisite. Clearly, these actions cannot occur at the same time. Or one action may have `(door open)` as a prerequisite while the other action achieves the fact `(not (door open))` as an effect. It obviously matters in which order these actions are performed, so the partially ordered plans produced by Graphplan cannot allow them to be placed at the same time level.

A mutex is declared between a pair of facts when:

- One fact is the negation of the other.
- All actions supporting the facts are pairwise mutex (termed *inconsistent support*).

For example, `(door open)` and `(not (door open))` are two facts that are clearly mutex. Or if, for example, fact F_1 is an effect of only the action A_1 and fact F_2 is an effect of only the action A_2, and A_1 and A_2 are mutex actions, then we must declare F_1 and F_2 to be mutex.

Mutex relationships are specific to levels in the planning graph. A pair of facts that are mutex in one level of the planning graph may not be mutex at later levels if, for example, a new action becomes applicable that removes the problem of inconsistent support. Additionally, it should be noted that actions can only be mutex with other actions in their own level of the planning graph, and the same is true of mutex relationships between facts.

Plan extraction is first attempted when all of the goals are present in the highest fact layer and are non-mutex. Plans are extracted via a simple backtracking search, starting with the highest fact layer as the current fact layer. Each goal is put in a goal list for the current fact layer, which records which facts *must* have an action chosen to support them. The search iterates through each fact in the goal list at the current fact layer, trying to find an action to support it which is not mutex with any other actions that have been chosen. When an action is chosen, its prerequisites are added to the goal list at the next lower fact layer. When all facts in the goal list of the current fact layer have a consistent assignment of actions, the search moves to the next lower layer. The search terminates with success when it reaches the first fact layer. It backtracks when it is unable to assign an action to each fact in the goal list for a given layer. Eventually, the search will have tried all legal combinations and if it has not found a plan, it will terminate with failure. If no plan was found, the planning graph is extended and GraphPlan attempts plan extraction on this new planning graph. Since GraphPlan first searches for a solution at the earliest possible point that a plan could exist, and incrementally extends the planning graph if no

solution is found, it must find the plan that uses the fewest time steps.

The plans produced by GraphPlan are optimal in the number of time steps needed to execute the plan, but not necessarily in the total number of actions in the plan. Optimality in time steps is guaranteed by the incremental method used to build and search planning graphs. Optimality in the total number of actions cannot be guaranteed because GraphPlan produces partially ordered plans, in which actions that do not interfere with each other are allowed to occur at the same time. Therefore, a given time level can contain any number of actions. The mutex relationships that are discovered during plan extension are used during plan extraction to build partially ordered plans that do not contain inconsistencies.

Our work is an extension of the ideas used in Graphplan. Consequently, this is the most fully described of the planning methods considered.

2.1.3 SAT solutions to planning problems

Another approach that has met with success is the compilation of Planning Problems to Satisfiability Problems. Once this is done, the SAT problem can be solved by any of a number of general purpose SAT solvers. Both systematic solvers and stochastic solvers can be used, depending on what is desired from the generated plan.

There are several ways to convert a Planning Problem to a SAT problem. One encoding explicitly creates *frame* axioms that prevent unaffected facts from changing when unrelated actions occur as well as axioms that state that an action implies its preconditions and effects. Another encoding can be realised by directly converting a GraphPlan planning graph into a SAT problem [14]. For example, to encode that operators imply their preconditions, a frame axiom for the `load` operator used in the

rocket problem (see Figure 2.1) would be

$$\text{LOAD}(alex, r1, london, 2) \Rightarrow (\text{AT}(alex, london, 1) \wedge \text{AT}(r1, london, 1).$$

Additionally, each fact implies the disjunction of all the operators at the previous level that have it as an add-effect:

$$\begin{aligned} \text{IN}(alex, r1, 3) \quad \Rightarrow \quad & (\text{LOAD}(alex, r1, london, 2) \vee \\ & \text{LOAD}(alex, r1, paris, 2) \vee \text{NOOP}(\text{IN}(alex, r1), 2)). \end{aligned}$$

And finally, mutex actions are encoded as follows:

$$\neg\text{LOAD}(alex, r1, london, 2) \vee \neg\text{MOVE}(r1, london, paris, 2)$$

Having introduced planning, we now turn our attention to Constraint Satisfaction Problems.

2.2 Constraint satisfaction

Constraint satisfaction is a general method of problem formulation in which the goal is to find values for variables such that these values do not violate any constraints that hold between the variables. This problem formulation can be used to solve many problems in artificial intelligence, computer science, and engineering.

2.2.1 Definition

A Constraint Satisfaction Problem (CSP) involves a set of variables $\{V_1, V_2, \ldots, V_n\}$. Each variable V_i has an associated domain D_i which specifies the possible *values* of

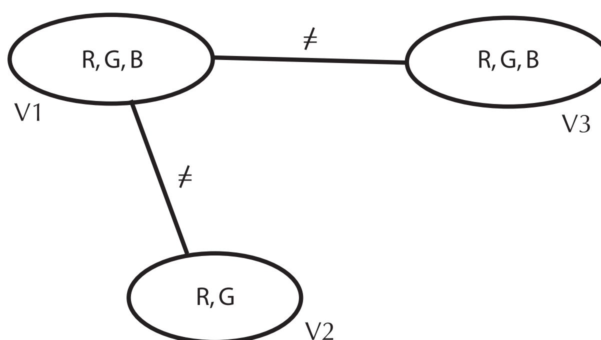

Figure 2.4: *An example CSP: list coloring problem*

the variable. Additionally, there exists a set of constraints between the variables. In the general case, constraints may be *k-ary*, but in the case of planning, only binary constraints are used, and that set looks like: $\{C_{i,j}, \ldots, C_{k,l}\}$ where the constraint $C_{i,j}$ holds between V_i and V_j. It is not necessarily the case that constraints exist between all pairs of variables. A constraint $C_{i,j}$ is a relation that restricts the values that variables V_i and V_j may simultaneously hold.

An example will make this terminology clearer. Consider a map coloring problem, in which you have a number of regions that can be colored from a restricted palette subject to the constraint that neighboring regions may not be the same color. Figure 2.4 shows a small CSP that is made up of a set of three variables $\{V_1, V_2, V_3\}$ with a set of constraints $\{C_{1,2}, C_{2,1}, C_{1,3}, C_{3,1}\}$. It is often the case that constraints are symmetric, and in such circumstances it would only really be necessary to store one of $C_{1,2}, C_{2,1}$ and $C_{1,3}, C_{3,1}$. In fact Figure 2.4 uses a single line to represent each pair of symmetric constraints. These constraints are all *mutex* constraints, which simply state that their associated variables may not assume the same value. Each variable has a domain, where $D_1 = \{R, G, B\}$, $D_2 = \{R, G\}$, and $D_3 = \{R, G, B\}$.

A CSP is solved when a value is found for each variable such that no constraints

are violated. One solution to this CSP is the set of assignments: $V_1 \Leftarrow R$, $V_2 \Leftarrow G$, $V_3 \Leftarrow B$. Of course, this CSP has more than one solution. This CSP is sufficiently small that it can be solved by inspection. However, in the case of larger CSPs, or the case that all solutions are desired, more systematic solution extraction methods are required.

2.2.2 Solution extraction

A few terms that need to be introduced to discuss algorithms that find solutions to CSPs. While looking for solutions, the variables are considered in some *order*. In the simplest case, this is a static ordering, but it is also possible to dynamically order the variables. As a partial solution is constructed, *past variables* are those that have been instantiated, while *future variables* are those that have not yet had a value assigned to them.

One of the simplest systematic methods of finding a solution to a CSP is a depth first backtracking search. This search proceeds by starting with the first variable and assigning it the first value in that variable's domain. It then moves to the next variable and checks the first value in that variable's domain against the previously assigned variable. If there is no problem it moves on to the next variable, otherwise it tries the next value in the current variable's domain against the previously assigned variable. The assignment of a value to each variable is a backtracking point. The search ends when either a satisfying solution is found, or the search space is exhaustively checked with no solution found. This algorithm is clearly described by Prosser [18].

Forward checking (FC) is a more intelligent method for finding a solution to a CSP. First described by Haralick and Elliot [12], forward checking is a *look-ahead* scheme in which consistency with future variables is ensured, removing the need to check against previously assigned variables. In forward checking, each time a variable

V_i is assigned a particular value d_j, this value is used to reduce the domains of future variables. For each future variable V_k, each value in its current domain is checked for consistency with the value d_j that has been assigned to variable V_i. Any inconsistent value is removed from the current domain of V_k. If all of the values are eliminated from a future variable (termed *domain annihilation*), we know that the current variable assignments can never lead to a solution and we must backtrack. If, after checking d_j against the future variables there has been no domain annihilation, we can consider the next variable.

Consistency is propagated forward, and the current domain of the current variable can only contain values that are consistent with all past variables. Hence, it is only necessary to check a proposed value against the future variables. This algorithm is also clearly documented by Prosser [18].

2.3 GP-CSP

Our work extends a planning system called GP-CSP, which is the work of Do and Kambhampati [8]. GP-CSP unifies the traditional GraphPlan method for planning with the Constraint Satisfaction Problem (CSP) methods for solution extraction. In this system, GraphPlan's plan extension is unchanged while GraphPlan's normal backtracking search is replaced with a CSP solver. An earlier planning system by van Beek and Chen called CPlan had already brought planning and CSPs together [23]; however, their system required hand-encoded CSPs while GP-CSP automates the process.

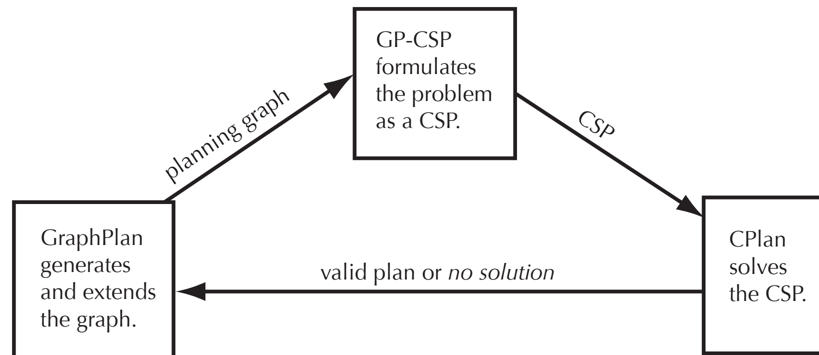

Figure 2.5: *The architecture of the GP-CSP system.*

2.3.1 The GP-CSP planner

The GP-CSP planner differs from the original GraphPlan only in the way that it looks for solutions. It generates and maintains the same planning graph as GraphPlan and carries out plan extension in the same way. But instead of launching a backtracking search for solution extraction, GP-CSP turns the *entire* planning graph into a CSP that it then attempts to solve. Just like the original GraphPlan, this attempt to find a solution either succeeds with GP-CSP halting, or it fails and the planning graph is extended. Each time the planning graph is extended and the planner attempts to find a solution, a new CSP is formulated. Figure 2.5 illustrates this process.

CSP formulation

The backtracking search used by the standard GraphPlan implementation is analogous to a dynamic constraint satisfaction problem (DCSP) as introduced by Mittal and Falkenhainer [16]. This stems from GraphPlan's use of goal sets at each fact layer. There are certain fact variables that need not be assigned, and this changes depending on which actions are chosen higher in the planning graph. In a DCSP, one has a standard CSP with the addition of an activity flag for each variable. A

satisfying assignment for the DCSP need only assign consistent values to variables that are marked as active. DCSPs also introduce the notion of *activity constraints* that are responsible for marking new variables as active. For example, one could have a constraint that says something like "if $v_1 = x$ then variables $v_i, \ldots, v_j$ must be marked as active."

To formulate the backtracking search as a DCSP, the facts are converted to variables and the actions supporting these facts become the variable domains. The constraints between the variables enforce action and fact mutexes as well as activity constraints to activate subgoals at lower levels of the planning graph. When the attempt to solve the DCSP begins, only the goal facts are marked as active and they are not allowed to be deactivated. In the backtracking search, choosing an action to achieve some goal adds that action's preconditions to the next lower goal set. Similarly, in the DCSP when an action is chosen to satisfy an active variable (achieve some goal), the activity constraints associated with that action activate new variables corresponding to that action's preconditions. There are methods for directly solving a DCSP. However, in GP-CSP this DCSP is further transformed to become a standard CSP.

This transformation is remarkably simple. All that needs to be done is to introduce an extra `null` value to the domain of each variable. A variable whose value is non-`null` is active. A variable whose value is `null` is not active and this variable can only be said to violate activation constraints. If the variables representing the goal facts each have such a `null` value, then a trivial solution is to deactivate all variables. Instead, the `null` value is not added to the domains of the goal facts. This guarantees that a satisfying assignment must achieve the goals.

2.3.2 Details of GP-CSP

Do and Kambhampati claim that GP-CSP makes use of the CSP library written by van Beek [22]. This is not entirely accurate, as GP-CSP is actually using the CSP solver code that was written for CPlan. Of course, van Beek worked on the CPlan project and the solver he wrote for it seems to have been an extension of his CSP library. An additional point of confusion in [8] is that they spend some amount of time discussing the fact that the CPlan solver is able to deal with non-binary constraints. However, the CSP formulation used by GP-CSP makes exclusive use of strictly binary constraints. The CPlan solver makes use of GAC (described in [17]) because it needs to handle non-binary constraints. This is not at all required by GP-CSP as it formulates the CSP using only binary constraints.

There are exactly three kinds of constraints that appear in the CSPs created by GP-CSP. These are `ACTIVITY CONSTRAINT`s, `MUTEX CONSTRAINT`s, and `FACT MUTEX CONSTRAINT`s. For the following discussion of constraints, it will be useful to have a concrete example of a planning problem (see Figure 2.6).

`ACTIVITY CONSTRAINTs` These are used to force activation of variables that represent prerequisites of current goal nodes (see Figure 2.7). These constraints are such that when some variable V takes on some value v_1 we must activate some set of variables $V_i, \ldots, V_j$. Recalling that each CSP variable represents a fact while its domain is the actions that can achieve that fact, consider a variable representing `(at r1 paris)`. We might try to achieve this by assigning the variable the value `(move r1 london paris)`. This would force us to mark `(has-fuel r1)`, `(at r1 london)`, `(rocket r1)`, `(place london)`, and `(place paris)` as active variables.

`FACT MUTEX CONSTRAINTs` These are directly generated by considering the mutexes

```
(:action move
:parameters (?r ?from ?to)
:precondition (and (has-fuel ?r) (at ?r ?from) (rocket ?r) (place ?from)
                   (place ?to))
:effect (and (at ?r ?to) (not (has-fuel ?r)) (not (at ?r ?from))))

(:action unload
:parameters (?c ?r ?p)
:precondition (and (at ?r ?p) (in ?c ?r) (rocket ?r) (cargo ?c)
                   (place ?p))
:effect (and (at ?c ?p) (not (in ?c ?r))))

(:init (at r1 london) (at r2 london) (at alex london) (at jason london)
       (has-fuel r1) (has-fuel r2) (rocket r1) (rocket r2)
       (place london) (place paris) (place jfk) (cargo alex)
       (cargo jason))

(:goal (and (at alex paris)
            (at jason jfk)))
```

Figure 2.6: *Excerpts from a PDDL definition of a planning problem.*

discovered by GraphPlan in its forward expansion phase. GraphPlan would, for example, note that `(in jason r1)` is mutex with `(in jason r2)`. This is represented in GP-CSP by not allowing both variables to assume non-`null` values. The CSP formulation need only note that this constraint exists (see Figure 2.8), as no other information is necessary to enforce the constraint.

`MUTEX CONSTRAINTs` Because actions are the values of the CSP variables, action mutexes are modeled by adding constraints between fact variables in the CSP. These are called `MUTEX CONSTRAINT`s. They are found as in GraphPlan, by looking for conflicting preconditions or effects. For example, this sort of constraint would be used to handle the fact that `(unload jason r2 jfk)` is mutex with `(unload alex r2 paris)`. The pairs of variables that contain these values would have a `MUTEX CONSTRAINT` placed between them to disallow this conflicting variable assignment (see Figure 2.7). This type of constraint is different

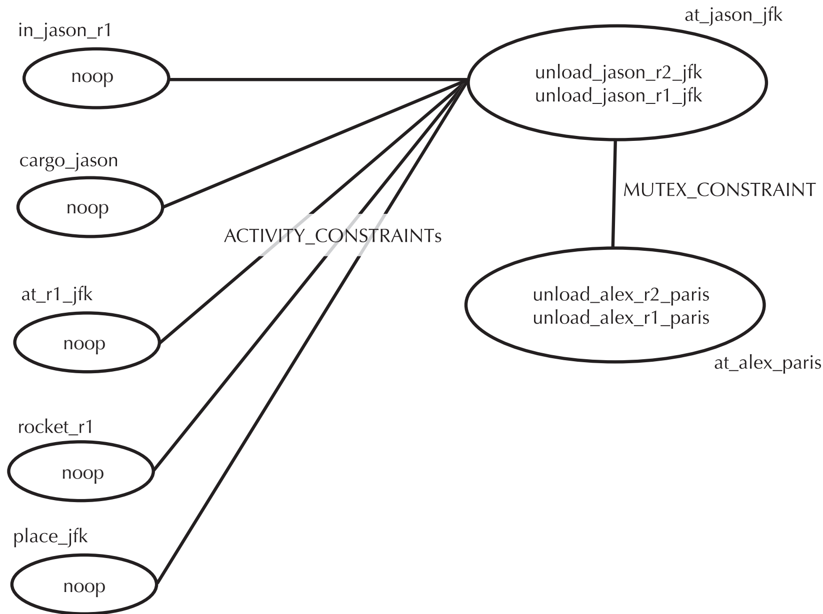

Figure 2.7: *Examples of a* `MUTEX CONSTRAINT` *and* `ACTIVITY CONSTRAINTs`*.*

from a `FACT MUTEX CONSTRAINT` in that `FACT MUTEX CONSTRAINTs` assert that two variables cannot be simultaneously active, while this type of constraint says that two variables may both be active, as long as they are not assigned particular values.

There is an additional feature of the CPlan solver that seems to be the true reason that it was used in GP-CSP. It has the ability to define constraints intensionally using functions instead of defining them in extension. Because these functions must be defined at compile time, such a solver is no longer general purpose. But by becoming a specialized solver, the memory required for constraint storage is dramatically reduced. The constraints are all represented by simply noting which kind of constraint exists between pairs of variables. In the case of `FACT MUTEX CONSTRAINTS` a constraint check simply consists of seeing if both variables have a non-`null` value. A `MUTEX CONSTRAINT` can be checked by looking in GraphPlan's mutex table to see if the

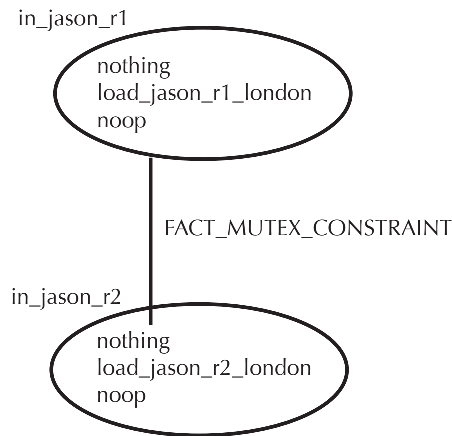

Figure 2.8: *Example of a* `FACT MUTEX CONSTRAINT`*.*

current assignment of a pair of variables is allowed or forbidden. The `ACTIVITY CONSTRAINT` is the most complex, but is still actually quite simple. For each value a_i of a variable f, we simply generate a set of binary constraints between f and each of the variables representing the preconditions of a_i.

Putting together all of the details outlined above, GP-CSP works by generating a planning graph and extending it until all of the goal facts appear at the highest fact level (exactly as in GraphPlan). At this point, the planning graph is transformed into a CSP that is given to the CPlan solver. If the solver is unable to find a valid plan, the planning graph is extended and a new CSP is created and searched.

2.3.3 Enhancements to the CSP solver

The first enhancement discussed in [8] is the addition of explanation based learning (EBL). This allows the solver to explain failures and use those explanations in subsequent searches to avoid the same failures. Their results show that in most of the cases reported, EBL decreases the CPU time required to find a solution, but in general the

speedup is less than two. The `no-goods` learned by EBL are not transfered from one CSP to the next (recall that a new CSP is generated each time the planning graph is extended). The developers of GP-CSP attempted to store this information between formulations, but found that the effort required was too great and they observed decreased performance. They suggest that this may have been due to the fact that in the CSP solver they could record inter-level (with respect to the planning graph) `no-goods` while GraphPlan only records intra-level `no-goods`. This results in GP-CSP recording a much larger number of `no-goods`, meaning that more work must be done to maintain and check the `no-good` information.

They also made use of dynamic variable and dynamic value ordering in an effort to speedup the CSP solver. By default the GAC solver uses a DCL strategy where variables are ordered first by smallest live domain (D), then by most constrained variable as measured by degree (C), then by preferring variables from higher levels of the planning graph (L). The authors also tried LDC and DLC orderings in an effort to more closely mimic the natural order of GraphPlan's backtracking search. Their experiments were not conclusive. They found that each variable ordering scheme gives better performance in certain domains, and worse performance in others. Hence they were unable to recommend any one ordering as being superior to the others.

They experimented with variable and value orderings similar to those used by the HSP planning system [4] and concluded that value ordering is not very important in comparison to variable ordering. HSP stands for heuristic search planner and is so named because it solves planning problems by using best-first hill-climbing search algorithms guided by automatically extracted heuristics.

Do and Kambhampati experimented with fairly simple level-based heuristics in which the heuristic value of a fact is the level in the planning graph where it first appears and the heuristic value of an action is based on the heuristic values of its

preconditions [13]. They tried two methods of computing the heuristic value of an action: the maximum value of any of the action's precondition, and the sum of the values of the action's preconditions. As noted above, they found that the different methods of computing the heuristic values of actions (which corresponds to CSP value ordering) had little effect on the performance of search. They also found that the variable orderings produced by the heuristic values of facts did little to improve the performance of the planner.

They next tried to automate the selection of variable ordering heuristics. As they previously noted, DCL gave better performance in some domains and LDC gave better performance in others. In a fairly ad hoc manner, they found a heuristic to select between DCL and LDC variable orderings. They calculated the value of the number of action mutexes divided by the number of actions as well as the value of the number of fact mutexes divided by the number of facts. With the appropriate thresholds for each quotient, they were able to choose the better variable ordering for each of their test problems.

> *As an aside, what they have done here is to train their heuristic on their test data. They did not test this heuristic on problems that it had never seen, and it is not clear that it would make the correct choice in such situations.*

Chapter 3

Our changes to GP-CSP

This chapter presents the changes that we have made to GP-CSP in order to investigate various CSP techniques. Section 3.1 outlines the areas that we have studied. Section 3.2 briefly describes the planning problems that we used to test our planning system. Section 3.3 introduces a preprocessing technique that can help reduce the size of planning CSPs. Section 3.4 investigates the effects of variable ordering on solution extraction. And finally, 3.5 describes the sizes of the planning problems that we are solving.

3.1 Our investigations

- We have replaced the GAC solver with a strictly binary solver that we wrote. We extended this solver to perform dynamic symmetry detection, decomposition, and the MAC algorithm.
- We have implemented our own CSP representation which maintains strictly binary constraints. We wrote code to automatically encode Graphplan planning graphs in a manner similar to that described by Do and Kambhampati. First, we

wrote a simple backtracking (BT) solver to verify that the CSP representation was correct. Then, we wrote a Forward Checking (FC) solver.

- We also have implemented DNPI, which is used to dynamically discover interchangeability in CSPs. This interchangeability is used to create *bundled solutions* in which some variables may have a set of values that interact identically with the other variables in the CSP. Hence, multiple solutions to the CSP can be found by simply iterating through the *bundled values*.

- We have also investigated a tree decomposition of planning problems when formulated as a CSP. This makes use of the structure inherent to the planning graphs created by GraphPlan.

- We have carried out some experiments to gauge the effectiveness of MAC in solving planning problems encoded as CSPs.

3.2 Planning problems

In carrying out our investigations, we tested our code using a set of benchmark planning problems, where:

- the rocket problem is as introduced in Section 2.1

- the towers of hanoi problems are what one would expect

- the gripper problem involves a robot with a pair of arms that needs to move a set of balls from one room to another

- the logistics problem involves moving several packages from location to location using various vehicles

- the bulldozer problem involves moving a vehicle around a map
- bw-large-a is a blocks world problem, wherein a set of blocks on a table need to be arranged in a particular order
- the mystery problem is a disguised problem where the actions and facts have been renamed at random, to prevent humans from giving hints to their planner.

3.3 Preprocessing

We have discovered a preprocessing step that can reduce certain variables to a single value. It was found that when searching for all solutions, the backtracking solver was outputting many plans that looked identical. When plans are printed, only “real” actions are printed; that is to say, `noop`s and variables with the value `NOTHING` are not printed. Upon further investigation we found that certain facts could be satisfied by either a `noop` or `NOTHING` with no impact on their neighbors, and so plans would look identical when output, while differing in certain variables that were not shown. For example, in the rocket problem that we keep referring to, the fact `(rocket r1)` at any fact layer higher than the initial conditions has a domain of `noop` and `NOTHING`.

The exact circumstances in which a `noop` becomes equivalent to `NOTHING` are that the variable in question has only these two values and only participates in `ACTIVITY CONSTRAINTS`. Recall that an `ACTIVITY CONSTRAINT` only says that in some cases, `NOTHING` is an unacceptable value, but otherwise any value will do. The `noop` will always satisfy such a constraint. So when a variable has only two values (which must necessarily be `NOTHING` and a `noop` by construction of the CSP) and only participates in `ACTIVITY CONSTRAINT`s, we can reduce the variable's domain to just the `noop`. Additionally, we know the variable cannot possibly violate any of its constraints, and

Table 3.1: *Results of pruning applied to planning CSPs*

CSP	# var.	# pruned	% removed
rocket problem	37	13	35.1
3 disc towers of hanoi	164	81	49.4
4 disc towers of hanoi	566	291	51.4
mystery problem 2	435	188	43.2
gripper problem 2	116	37	31.9
logistics rocket-a	280	9	3.2
bulldozer problem 1	269	164	60.9
bw-large-a	720	14	1.9

these constraints can be removed from the problem. Table 3.1 shows some results from using this pruning method.

These circumstances have only been observed to occur with the initial conditions of a problem. However, the substantial number of variables that are pruned in certain problems clearly indicates that variables other than those corresponding to initial conditions must be eligible for pruning. In fact, in many cases the pruned variables *do* correspond to initial conditions, but higher in the planning graph. In many planning problems, some of the initial conditions are facts that are never supported by any actions, often encoding typing information. Consequently, when these facts appear at higher levels in the planning graph, they are only supported by the `noop` action.

As an example, the rocket problem has facts like `(rocket r1)`, `(place paris)`, and so forth that never change. These facts appear over and over in each layer of the planning graph. In Table 3.1, we can see that the `logistics rocket-a` problem experiences relatively little pruning. This is due to the fact that the problem is encoded with the typing feature of PDDL, which eliminates the need to encode such facts as initial conditions.

Consequently, this pruning method can be seen as a way to find and ignore typing information in planning problems. The typing information has already been used by GraphPlan in creating the planning graph, as only actions with satisfied preconditions

are added to the graph. Because typing of objects is static for a given problem, no object will have its type suddenly change in the middle of the plan, and plan extraction need not be concerned with object types.

This CSP pruning method has little impact on solution extraction if the default variable ordering is used. As discussed in Section 3.4, the default variable ordering corresponds to a layer by layer traversal of the facts in the planning graph, starting at the goal layer. When using Forward Checking to solve a CSP ordered in this way, if any fact variable (F_1) has an `ACTIVITY CONSTRAINT` with a variable eligible for pruning (F_2), F_1 would necessarily be considered before F_2. Forward Checking from F_1 would ensure that F_2 has a consistent value which would never require backtracking on F_2.

If, however, a different variable ordering is used, the pruning can make a difference. Using the same notation as above, what happens if variable F_2 is considered before F_1? By construction of the CSP, the value `NOTHING` is the first in each variable's domain. F_2 would then first try `NOTHING` which could conceivably eliminate the action we require for F_1. However, this would not be noticed until a later time when we must choose a value for F_1. This would fail, and we would need to backtrack all of the way to F_2 and set its value to `noop`. Clearly, pruning variable F_2 from the CSP would prevent this backtracking from occuring.

When finding only a single solution to the CSP, there is little to prompt the discovery of this pruning method. Consequently, I suspect that Do and Kambhampati did not ever notice this effect.

3.4 Variable ordering

We tried several static orderings including:

1. Number of constraints, with the least degree first (DEG).

2. Domain size, with the largest domain first (LDF).

3. Ratio of domain size to degree, with the smallest value first (DDR).

All of these orderings produced worse runtimes than the default ordering, which is simply the order of facts as they are traversed layer-by-layer (from the goal level backwards) in the planning graph.

Table 3.2 presents some experimental data comparing the different variable orderings. Processes were killed after three minutes of computation, since GP-CSP is known to be able to solve these problems in a matter of seconds. The searches were all carried out beginning at the solution bearing level of the planning graph. The random order is an average over 10 solutions, although only the simplest problem was solvable in the allotted CPU time using a random ordering.

It seems that the default variable ordering produced by the encoding of the planning problem as a CSP may be the best. This ordering is created by traversing the facts layer by layer, starting at the highest fact level. Visiting variables in this order corresponds quite closely to the order in which GraphPlan itself examines goals and their possible supporting actions. This seems somewhat intuitive as starting with any other variables would likely force `ACTIVITY CONSTRAINT`s to become active and they may force irrelevant actions to be considered.

Indeed, starting search at any point other than the fact layer corresponding to the goals of the planning problem can cause a large number of irrelevant actions to be considered. Consider again the problem introduced in Figure 2.6. In this problem our goal is to have the facts `(at alex paris)` and `(at jason jfk)`. Of course, this planning domain could also solve a problem where we want `(at alex jfk)` and `(at jason paris)`. In fact, the same planning graph can be used to solve either problem.

Table 3.2: *Constraint checks, number of nodes visited, and CPU time (in seconds) required to solve planning CSPs using various orderings. LDF is largest domain first, DDR is domain/degree ratio, DEG is degree, GPO is the default GraphPlan ordering, random is the average from 10 trials.*

	LDF	DDR	DEG	GPO	random
rocket problem					
#CC	194	1811	1887	132	873 ($\sigma = 1051$)
NNV	42	243	244	37	
CPU time	0.02	0.04	0.02	0.01	0.01 ($\sigma = 0.003$)
3 disc towers of hanoi					
#CC	956181	-	12150671	109087	-
NNV	27390	-	173765	3300	-
CPU time	1.03	> 180	11.01	0.16	> 180
4 disc towers of hanoi					
#CC	-	-	-	-	-
NNV	-	-	-	-	-
CPU time	> 180	> 180	> 180	> 180	> 180
mystery problem 2					
#CC	-	-	-	215221	-
NNV	-	-	-	838	-
CPU time	> 180	> 180	> 180	10.48	> 180
gripper problem 2					
#CC	-	-	-	532638	-
NNV	-	-	-	15893	-
CPU time	> 180	> 180	> 180	0.51	> 180
logistics rocket-a					
#CC	-	-	-	-	-
NNV	-	-	-	-	-
CPU time	> 180	> 180	> 180	> 180	> 180
bulldozer problem 1					
#CC	-	-	-	6173	-
NNV	-	-	-	308	-
CPU time	> 180	> 180	> 180	0.13	> 180
bw-large-a					
#CC	-	-	-	57919778	-
NNV	-	-	-	169190	-
CPU time	> 180	> 180	> 180	81.32	> 180

Table 3.3: *Size of various planning problems encoded as CSPs*

CSP	1st level	soln. level	At solution bearing level		
			# var.	ave. dom. size	ave # cons. per var.
rocket problem	3	3	37	1.9	4.6
3 disc towers of hanoi	5	7	164	2.9	13.5
4 disc towers of hanoi	6	16	566	4.0	20.9
mystery problem 2	6	6	435	3.1	25.0
gripper problem 2	4	6	116	2.8	14.7
logistics rocket-a	5	8	280	3.8	44.3
bulldozer problem 1	10	10	269	2.5	10.2
bw-large-a	9	12	720	5.4	82.3

By starting solution extraction at the goal level, we immediately limit ourselves to those actions that are actually relevant to what we are attempting to accomplish. If solution extraction is started in the "middle" of the CSP, we may begin considering actions that will ultimately place `jason` at `paris`, when we really want him to be at `jfk`. Any time the goals of a planning problem can be permuted to create distinct (yet very similar) problem instances, as in the rocket example above, starting solution extraction with the goal nodes can dramatically decrease the branching factor of the search space of the CSP.

3.5 CSP sizes

The size of a CSP can have a large impact on the difficulty in finding a satisfying solution. Planning problems can generate CSPs that are quite large, which can make solution extraction slow to the point of considering the problem unsolvable.

We present some size information of this nature in Table 3.3. The first column indicates the level of the planning graph at which the goals were first non-mutex with each other. This is the heuristic that GraphPlan uses for determining when to begin search. The second column indicates the level of the planning graph at which

a solution can actually be found. The last three columns show size information from the level of the planning graph that contains the solution.

Of note is the size of the average domain. It seems that planning problems tend to have very small domains, which means that CSP techniques like value ordering can have only limited benefit. While the domain sizes are comparable amongst the different planning problems, the average number of constraints per variable varies from 4.6 to 85.7. The number of constraints is an indicator of the "complexity" of a planning domain. A constraint is introduced for each precondition of every action, and actions that undo each other's effects require further constraints.

3.6 Additional extensions

Using the ideas of the GP-CSP planning system, we have created our own CSP representation and written our own CSP solvers to be used for solution extraction. After completing the work outlined in the previous section, we turn our attention to the possible enhancements to the solver. The first technique to be considered is DNPI (discussed in chapter 4), followed by a decomposition technique (discussed in chapter 5), and finally MAC (discussed in chapter 6).

Chapter 4

Exploiting partial interchangeability

This chapter summarizes our efforts to use partial interchangeability in the context of planning problems. Section 4.1 gives some relevant information about interchangeability. Section 4.2 discusses DNPI, the particular type of interchangeability that we investigated. Section 4.3 presents our findings with regard to planning problems.

4.1 Background

Interchangeability amongst the values of a variable in a CSP is a term used to describe an equivalence that exists between these values. The idea is that if values v_1 and v_2 for variable V_i are found to be interchangeable, then any solution to the CSP with V_i taking the value v_1 will remain a solution if the value is changed to v_2.

There are different levels of interchangeability ranging from functional interchangeability to neighborhood interchangeability (NI), which allows only the variable V_i to change value. There are weaker forms of interchangeability, which allow a subset of

variables in the CSP to be affected when values are selected for variable V_i.

It is desirable to find interchangeable values for several reasons. They can help to create more *robust* solutions. If a CSP is modeling some type of physical process, say manufacturing, we can imagine that a pair of values that are marked interchangeable (or *bundled*) correspond to two different ways of building something. If one of these values is no longer available (say a machine has failed), then we can simply substitute the other value in the bundle without having to resolve the problem from scratch. Because these two values are known to be interchangeable, we know that making this substitution on the fly will not affect the correctness of our solution.

Additionally, finding bundled values can help to reduce the search space, by allowing us to replace the bundled values with a single meta-value. Hence the solver can avoid solving what end up being identical search trees. This is especially useful if all solutions for a CSP are required, which necessarily demands considering the entire search space.

4.2 Dynamic neighborhood partial interchangeability

It is important that the computational cost of finding interchangeable values not be exceedingly burdensome. For example, finding full interchangeability may require computing all solutions to the CSP. This still achieves the aim of providing robust solutions, but it does nothing to help reduce the search space. A less expensive method of finding a subset of the interchangeable values is thus desired.

Using the joint discrimination tree (JDT) introduced by Choueiry and Noubir [6] it is possible to compute dynamic neighborhood partial interchangeability (DNPI) without incurring too much additional overhead [1]. In fact, the computations re-

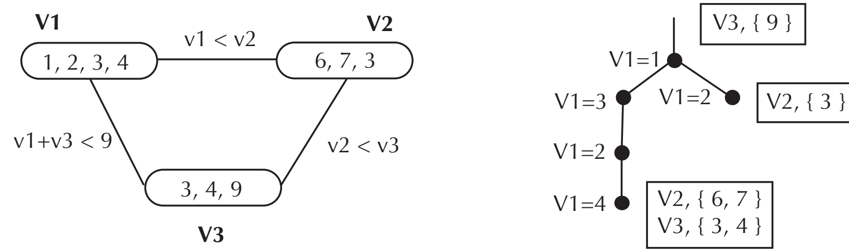

Figure 4.1: Left: *CSP.* Right: *JDT of V2, V3.*

quired by DNPI are the same as the ones used for perfomring forward checking while maintaining some extra information.

A search that makes use of DNPI is quite simple to explain. When a new variable is first considered, forward checking is performed with each of that variable's values. The JDT captures this information, as well as bundling together values that have the exact same impact on the future variables. That is, in order to be bundled together, two values must produce identical results when used in forward-checking. As an example, see figure 4.1 where the JDT shows the forward checking information, and shows us that the values of 3 and 4 for variable V1 can be bundled together.

Any sets of bundled values are replaced in the variable's domain with a single meta-value. The forward checking information is cached, and search continues as normal. When a value for the current variable is selected, the effects of forward checking are simply pulled from the cache instead of being recomputed. When meta-values are introduced, they reduce the search space. In the case that all solutions to a CSP are desired, the only additional work done by DNPI is the bookkeeping necessary to calculate interchangeable values. And when values can be bundled, DNPI will do less work, as the search space is reduced by the introduction of meta-values. In fact, even when no bundles appear in solutions, no-goods can often be bundled together, and this will again act to reduce the search space.

4.3 Application to planning problems

There are many situations in which planning problems exhibit what, to the human mind, seem to be symmetric actions. For example, there is a benchmark planning problem called the *gripper domain* in which a robot with a left arm and a right arm must move a set of balls from one room to another. Each arm can hold one ball at a time. As far as each ball is concerned, being picked up by the left arm or by the right arm makes no difference in being moved from one room to another. Similarly, each robot arm cares very little which particular ball is picked up as the order in which the balls are moved does not matter.

For the reasons outlined in section 4.1, we would like to try to automatically discover interchangeability. Recall that the CSP representation of a planning problem turns each fact in the planning graph into a variable and the actions which support some fact become the values of the variable corresponding to that fact. Using a technique like DNPI, we should be able to find any sets of actions that behave identically and use them to create robust bundled solutions.

To test the utility of this approach, a DNPI solver was implemented that could be used in conjunction with our CSP representation. Initial tests showed virtually no interchangeability in any of the planning problems tested. Occasionally, a `NOTHING` and a `noop` would be marked as interchangeable, but never any "real" actions that form steps in a plan. The first attempt to explain this appealed to the fact that planning problems seldom include unnecessary objects or actions. Generally, anything placed in the problem by the domain designer is there for a specific reason.

We modified the rocket problem that was introduced in chapter 2 to include extra, unneeded rockets. Figure 4.2 reproduces an abridged definition of the rocket domain with the addition of rockets `r3` and `r4`.

The goal level of the CSP contains two variables: one for `(at alex paris)` and

```
(:action move
:parameters (?r ?from ?to)
:precondition (and (has-fuel ?r) (at ?r ?from) (rocket ?r) (place ?from)
                   (place ?to))
:effect (and (at ?r ?to) (not (has-fuel ?r)) (not (at ?r ?from))))

(:action unload
:parameters (?c ?r ?p)
:precondition (and (at ?r ?p) (in ?c ?r) (rocket ?r) (cargo ?c) (place ?p))
:effect (and (at ?c ?p) (not (in ?c ?r))))

(:init (at r1 london) (at r2 london) (at alex london)
       (at r3 london) (at r4 london) (at jason london)
       (has-fuel r1) (has-fuel r2) (rocket r1) (rocket r2)
       (has-fuel r3) (has-fuel r4) (rocket r3) (rocket r4)
       (place london) (place paris) (place jfk) (cargo alex)
       (cargo jason))

(:goal (and (at alex paris)
            (at jason jfk)))
```

Figure 4.2: *Extended rocket problem.*

one for `(at jason jfk)`. Looking at the effects list for action `unload`, we see that this is the action which can update the location of cargo. Both goals can be achieved by applying the unload operator to any of the four rockets and the appropriate person (recall that we are searching from the goal level backwards, so our choice at this point determines into which rocket we need to load which person). Since two of the rockets are superfluous, we might expect that DNPI could tell `jason` to treat rockets `r1`, `r2`, and `r3` as a single meta-rocket while `alex` is assigned to use rocket `r4`. This would give us the bundling that we desire.

Unfortunately, this does not happen. Look again at the definition of the `unload` operator in figure 4.2. Notice that the precondition list refers to two facts about the parameter `?r`. One of them, `(rocket ?r)` corresponds to typing information, and our preprocessing method (see section 3.3) can eliminate this. However, the other fact `(at ?r ?p)` cannot be ignored. This precondition demands that before applying

`(unload jason r1 jfk)` we must establish the fact `(at r1 jfk)`, while if we want to apply `(unload jason r2 jfk)` we must establish the fact `(at r2 jfk)`. These facts correspond to different variables in the CSP.

In section 4.2 we pointed out that in order for two values of some variable to be bundled together by DNPI, they must have the exact same effect when used to forward check a partially instantiated solution. Since each action has its own set of preconditions and effects that refer to specific objects in the planning world, no two actions will be constrained with the exact same set of variables. This is a result of the fact that PDDL requires all objects to be explicitly named, making them distinct.

GraphPlan generates fully instantiated actions when creating the planning graph and not action templates. Hence, each action will refer to its own particular set of objects in its preconditions and effects and thus DNPI will never be able to mark any "real" actions as interchangeable, unless a planning domain contains, for some reason, an action that is simply an alias for another.

Returning to the gripper domain introduced in the beginning of this section, we see now that DNPI cannot help to simplify this problem, given the particular CSP encoding of the planning problem that we use. Each ball has its own identity and has its own facts to record its location. Consequently, despite the tremendous similarity between `(get ball1 left)` and `(get ball2 left)`, we are powerless to take advantage of this using DNPI.

4.4 Conclusion

The kind of symmetry that exists in planning problems cannot be detected or taken advantage of using DNPI. Even though we can intuitively see relations between different actions in our own minds, the explicit naming of objects in PDDL planning

domains prevents DNPI from being effective.

Chapter 5

Using decomposition methods

This chapter covers our examination of a particular decomposition method used with Constraint Satisfaction Problems. Section 5.1 provides some background information on decomposition. Section 5.2 describes the method in which we decompose planning problems represented as CSPs. Our experiments are discussed in section 5.3.

5.1 Background

It is sometimes possible to more efficiently solve a problem by decomposing that problem into a set of independent subproblems. Each subproblem can be independently solved, and those solutions can be combined to form a solution to the original problem. Since the subproblems are independent, a decomposition strategy can lead very naturally to solvers that run in parallel. This is one simple method of improving the performance of a solver as measured in wall time.

In this chapter, we investigate the applicability of the tree clustering algorithm of Dechter and Pearl [7] to our CSP representation of planning problems. This is a method of restructuring a CSP to make solution extraction less costly.

5.2 Decomposing a planning problem

The constraint graphs that are created by converting a planning problem to a CSP exhibit structure that makes decomposition quite simple. Recall that the CSP is created from the planning graph maintained by GraphPlan. Just as the planning graph is divided into *layers*, so too is the CSP. While the planning graph has alternating layers of facts and actions, the variables in the CSP only represent facts, with actions making up the domain values of the variables.

Additionally, the constraints in the CSP can only be of three types. The mutex constraints (between facts and between actions) can only exist between variables in the same layer. Activation constraints can only exist between one variable and another variable that is either one layer above, or one layer below the first. Thus a variable in the CSP at layer l_i can only interact with other variables at layer l_j when

$$|l_i - l_j| \leq 1. \tag{5.1}$$

Figure 5.1 illustrates the structure of a CSP that models a planning problem as well as how such a CSP can be split into independent subproblems. In the top of the figure, we see the original CSP, where the variables that are vertically aligned represent facts that belong to the same layer. All constraints are represented as lines, without regard to the particular type of the constraint. All intra-level constraints are mutex constraints, and the constraints that link variables in different layers are activation constraints. To split the CSP, we need only observe that the constraints force the CSP variables to partition themselves. And equation 5.1 tells us that we can group interacting variables with little work, determining the subproblem(s) to which a variable belongs using only each variable's layer property.

We can start at the highest level n in the planning graph and form a new CSP

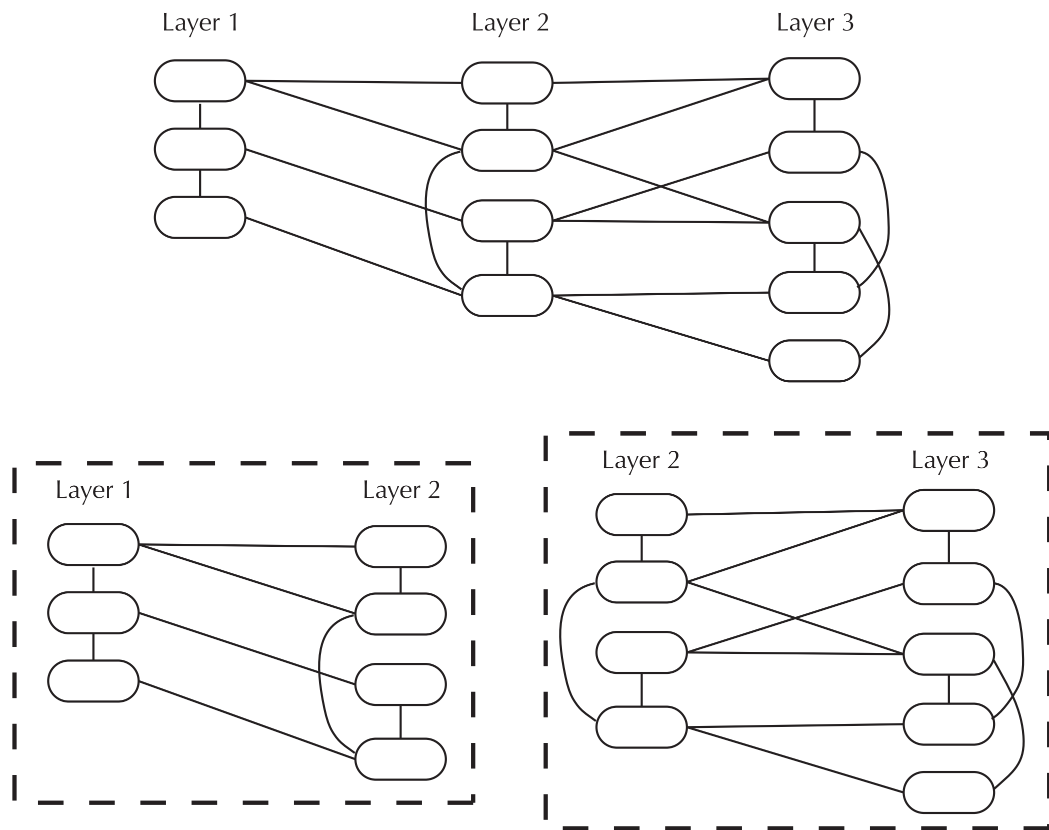

Figure 5.1: Top: *Original CSP.* Bottom: *CSP split into independent subproblems.*

Table 5.1: *Results of decomposition*

CSP	#CC full CSP all soln.	#CC all subproblems
rocket problem	406	24844
bulldozer problem 1	73696	3197887

with the variables one layer lower (i. e. at level $n-1$). Only the constraints between this subset of variables are kept. We can again form a new CSP, this time using the variables in layer $n-1$ and $n-2$. This continues until we reach the layer containing the initial conditions. Thus a CSP representing an n-layer planning problem will be decomposed into $n-1$ subproblems.

5.3 Experiments

To find out if this decomposition strategy is useful in the context of planning, we wrote a simple function to split a planning CSP. The function returns a collection of CSPs that represent the subproblems, forming a chain. Each of these new CSPs is represented in the same way as a full planning CSP, so the same solution extraction methods that are used to solve the full CSPs can be applied to the subproblems.

If we want to solve the subproblems simultaneously, we do not want one subproblem to have to wait for solution information from another before attempting to find its own solution. Hence, we will need all solutions to each subproblem, and these partial solutions can then be joined to form solutions to the entire planning problem. We use the DNPI mechanism discussed in chapter 4 to solve the subproblems. Although there is no symmetry for DNPI to take advantage of, DNPI does no more work than FC when all solutions are sought.

Table 5.1 shows some of the results of testing the decomposition technique. The first column lists the name of the planning problem, the second column lists the

number of constraint checks used by DNPI to find all solutions to the original CSP. The third column lists the sum of the number of constraints checks required to solve all of the subproblems. Also worth noting is that, while column two represents the work required to get all solutions, column three represents only the work required to solve the subproblems – more work would be required to turn these partial solutions into full solutions.

Only two results are presented in table 5.1 because the cost is prohibitively high. In both cases, the number of constraint checks required to solve all subproblems is more than 40 times the number of constraint checks required to solve the original problem. And bear in mind that when decomposing a problem additional work must be performed to find a full solution. The two tested problems, `rocket problem` and `bulldozer problem 1`, are both "easy" problems, but the amount of work required to merely solve the subproblems is already becoming prohibitive. The code necessary to find full solutions from the partial solutions of subproblems was not completed, as it is clear that this decomposition of planning problems is notably unhelpful.

The reason that this decomposition performs worse is that the planning graph is a compact representation of *many* planning problems. For example, in the rocket domain that has been used several times as an example, the planning graph that contains a solution for the goal `(and (at jason jfk) (at alex paris))` would also contain plans that solve `(and (at jason paris) (at alex paris))`, `(and (at jason jfk) (at alex jfk))`, `(and (at jason paris) (at alex jfk))`, and so forth. Consequently, the *lower* subproblems that do not include any information about the problem's goals will contain many, many solutions that have nothing to do with the problem that we actually want solved.

An attempt to "fix" the decomposition strategy might suggest solving the *highest* subproblem first (the one containing the goals) and then passing on the information

to the next lowest subproblem. However, this forces us to serialize computation of the solutions to the subproblems, and then using the tree decomposition we would solve each layer in order to propagate facts backwards. This corresponds to using forward checking with the default GraphPlan variable ordering.

5.4 Conclusion

This particular tree decomposition of the CSPs that represent planning problems is not an effective method of increasing the performance of the solver. The decomposition causes the subproblems to lose information about the goals of the problem being solved. Consequently, the subproblems exhibit excessive numbers of solutions that will need to be discarded when full solutions are computed.

However, the fact that the tree decomposition ends up acting like forward checking with the default variable ordering helps to explain why that particular static ordering was so powerful. Had we been unaware of that particular ordering, we would have seen an improvement in using the tree decomposition compared to our results from forward checking.

Along with the experiments with variable ordering, this helps to show the importance of using goal information as early as possible when solving a planning problem. This is, at least in part, due to the density of the planning graph. A planning graph that contains a solution to a given problem, will also contain solutions for any possible goals reachable by the actions in the planning problem.

Chapter 6

MAC

This chapter introduces the MAC algorithm and shows how we have applied it to planning problems. Section 6.1 explains MAC. Section 6.2 presents our results from using MAC to solve CSPs representing planning problems. Section 6.3 introduces an improvement to the MAC algorithm that we have discovered.

6.1 Background

Maintaining Arc Consistency (MAC) is a technique used to solve CSPs by enforcing Arc Consistency (AC) at each step in a backtracking search [20]. AC is a property of a CSP whereby for any variable V_i, each element in its domain has a *support* in the domain of any variable with which V_i is constrained. That is to say, in order for AC to hold, it must be the case that for every value d_i in the domain of V_i, we can find a value d_j in the domain of a variable V_j such that the constraint between V_i and V_j is not violated. This implies that any value of a variable V_i in a CSP known to be AC will not immediately cause backtracking, as this value must have support in all other variables that are constrained with V_i. However, because AC only enforces *2-consistency*, we may need to backtrack when a third variable is assigned a value.

Obviously, we do not expect to frequently find CSPs that are already in an AC state, and so we must use AC algorithms to propagate constraint information and prune values from variable domains until the CSP is AC [2]. This can be done in time polynomial in the size of variable domains and the number of constraints, which is not overly burdensome given the exponential complexity of search. On its own, AC is often enforced on a CSP as a preprocessing step to reduce the size of variable domains. This is, of course, intended to increase the efficiency of actually finding a solution to the CSP.

MAC takes a more active approach and continually enforces AC throughout the process of finding a consistent variable assignment for the CSP. Conceptually, this can be thought of as the most naïve backtracking search, but where an AC algorithm is called each time a variable is assigned a value. We backtrack whenever AC completely eliminates some variable's entire domain. And of course, some extra information must be maintained in order to correctly undo upon backtracking the effects of enforcing AC.

6.2 Results

Our experiments with MAC were concerned with the effects of various variable orderings on the performance of MAC. We first examine static orderings, and then turn our attention to dynamic variable orderings.

6.2.1 Static variable ordering

We tested MAC with several static variables orderings, and the results are presented in Table 6.1. For comparison, a random ordering was also used, and the results of those trials are presented in Table 6.2. The most striking result is that in all but

one case, when a problem is solvable by MAC using one static ordering, it is solvable using any other tested ordering, *even the random ordering.* This is in stark contrast to the results presented in Table 3.2 where all but the most trivial problems could only be solved by FC using the default GraphPlan ordering.

FC performed badly when using anything other than the default GraphPlan ordering, because any other ordering fails to use goal information to reduce the branching factor of search. However, the propagation inherent to MAC ensures that goal information is used, *even when search is not begun at the goal nodes.*

The experiments involving random variable orderings (summarized in Table 6.2) illustrate the great impact that variable ordering has on solution extraction time. In every tested problem save one, we observe that the standard deviation in constraint checks is the same order of magnitude as the mean number of constraint checks. Consequently, the solution extraction times recorded for each problem had a large variance as well.

The number of possible variable orderings grows factorially with the number of variables, and it is clearly infeasible to test every ordering. As previously noted, the random experiments discovered static orderings that led to faster solution extraction in a few cases. We are thus unable to choose the *best* ordering for a given problem ahead of time, but it may be possible to suggest a *good* ordering that seems to be effective most often across many problems.

Clearly, MAC does not suffer from FC's inability to handle anything other than the default variable ordering. Dynamic variable ordering was not useful in conjunction with FC because of the lack of goal information when search does not begin at the goal nodes. But the good behavior of MAC on all of the static orderings suggests that it may benefit from dynamic variable ordering, which we turn to next.

Table 6.1: *Constraint checks, number of nodes visited, and CPU time (in seconds) required to solve planning CSPs using MAC with various static orderings. LDF is largest domain first, DDR is smallest domain/degree ratio, DEG is smallest degree, GPO is the default GraphPlan ordering.*

	LDF	DDR	DEG	GPO
rocket problem				
#CC	806	4,647	12,458	830
NNV	37	37	61	37
CPU time	0.02	0.02	0.03	0.02
3 disc towers of hanoi				
#CC	246,200	1,018,725	1,043,545	119,140
NNV	218	248	238	202
CPU time	0.24	0.74	0.64	0.15
4 disc towers of hanoi				
#CC	-	-	-	-
NNV	-	-	-	-
CPU time	>180	>180	>180	>180
mystery problem 2				
#CC	18,774,822	200,739,983	52,756,781	38,098,959
NNV	838	838	838	838
CPU time	27.40	113.21	40.32	39.2
gripper problem 2				
#CC	14,132,672	1,023,732	3,693,785	1,455,606
NNV	4,431	684	601	1,973
CPU time	12.64	0.85	2.36	1.24
logistics rocket-a				
#CC	-	-	-	-
NNV	-	-	-	-
CPU time	>180	>180	>180	>180
bulldozer problem 1				
#CC	3,390,753	4,598,262	444,098	77,473
NNV	444	509	308	308
CPU time	2.15	2.68	0.36	0.18
bw-large-a				
#CC	-	-	-	112,841,652
NNV	-	-	-	6,253
CPU time	>180	>180	>180	70.02

Table 6.2: *Constraint checks, number of nodes visited, and CPU time (in seconds) required to solve planning CSPs using MAC and random variable orderings. The data presented is computed from 100 random orderings per problem. The final column tabulates the number of random orderings that exceeded the allowed 180 seconds of CPU time.*

	#CC	σ	NNV	σ	time	σ	% unsolvable
rocket problem	1,917	1,108	37.7	1.3	0.02	0	0
3 disc towers of hanoi	8,354,261	16,495,095	3,105.9	7,194.9	7.56	16.34	0
4 disc towers of hanoi	-	-	-	-	-	-	100
mystery problem 2	106,818,571	14,339,388	844.5	48.5	71.67	9.06	9
gripper problem 2	5,568,009	7,262,796	2,144.7	3,333.6	4.64	6.49	0
logistics rocket-a	74,346,951	49,795,856	1,581.1	904.6	77.05	55.42	92
bulldozer problem 1	20,160,951	30,820,424	4,502.6	7,332.1	17.10	29.47	10
bw-large-a	-	-	-	-	-	-	100

6.2.2 Dynamic variable ordering

Dynamic variable ordering (DVO) works by deciding on the fly which variable to visit next. When the current variable assignments are all consistent, and we wish to move on to a new variable, we can compute some metric value for each variable and then choose the most "promising" variable for consideration next. A common metric is the number of constraints in which a variable participates. For these experiments, we test three metrics:

1. from among the variables at the highest level in the planning graph, choose the one with the smallest remaining domain (LDF)

2. from among the variables at the highest level in the planning graph, choose the one with the smallest ratio of remaining domain to the number of constraints in which the variable participates (DDR).

3. from among the variables at the highest level in the planning graph, choose the one with the smallest ratio of remaining domain to the number of constraints the variable shares with *unassigned* future variables (DDD).

Table 6.3: *Constraint checks, number of nodes visited, and CPU time (in seconds) required to solve planning CSPs using various solution extraction methods. The first column presents results of FC using the default GraphPlan ordering, the second column presents the best performance for MAC with the associated static ordering in parentheses, the third column presents the results of MAC using dynamic least domain first variable ordering, the fourth column presents the results of MAC using dynamic domain/degree ratio variable ordering, and the final column represents the results of MAC using dynamic domain/future variables ordering.*

		FC (GPO)	MAC (*best*)	(DVO-LDF)	(DVO-DDR)	(DVO-DDD)
	rocket problem					
A	#CC	132	806 (LDF)	851	822	840
	NNV	37	37	37	37	37
	CPU time	0.01	0.02	0.02	0.02	0.02
	3 disc towers of hanoi					
B	#CC	109,087	119,140 (GPO)	148,935	70,699	97,824
	NNV	3300	202	166	164	166
	CPU time	0.16	0.15	0.15	0.11	0.14
	4 disc towers of hanoi					
C	#CC	-	-	-	-	-
	NNV	-	-	-	-	-
	CPU time	> 180	> 180	> 180	> 180	> 180
	mystery problem 2					
D	#CC	215,221	18,774,822 (LDF)	178,643,049	175,591,092	182,780,437
	NNV	838	838	838	838	838
	CPU time	10.48	27.40	108.73	100.48	105.07
	gripper problem 2					
E	#CC	532,638	1,023,732 (DDR)	1,009,514	746,619	274,541
	NNV	15,893	684	1,018	577	267
	CPU time	0.51	0.85	0.93	0.59	0.27
	logistics rocket-a					
F	#CC	-	-	4,767,109	-	-
	NNV	-	-	348	-	-
	CPU time	> 180	> 180	3.24	> 180	> 180
	bulldozer problem 1					
G	#CC	6,173	77,473 (GPO)	33,994,667	2,984,655	1,906,714
	NNV	308	308	12,482	1,590	1,003
	CPU time	0.13	0.18	28.02	2.31	1.55
	bw-large-a					
H	#CC	57,919,778	112,841,652 (GPO)	162,919,122	99,002,285	45,601,895
	NNV	169,190	6,253	4,662	3,686	1,139
	CPU time	81.32	70.02	91.87	64.68	28.89

Table 6.3 compares the constraint checks and CPU time required to solve various planning problems using FC with the default static ordering, MAC with the best static ordering noted, and MAC with dynamic ordering. In three cases (rows A,D, and G), FC is the least costly method of finding a solution. In one case (row B), MAC with DVO-DDR is the least costly method of solution extraction. In one case (row F), MAC with DVO-LDF is the only method that can find a solution. Both MAC with DVO-DDR and DVO-DDD outperform MAC with DVO-LDF in all cases except for the one problem which is only solved by MAC with DVO-LDF. In two cases (rows E and H), MAC with DVO-DDD is the least costly method of solution extraction. All of the MAC methods are able to solve the same problems that can be solved with FC.

However, MAC with DVO-LDF is able to solve one additional problem that FC and MAC with static ordering are unable to solve. In fact, MAC with DVO-LDF can solve the `logistics rocket-a` problem in a mere 3.2 seconds while the other methods grind away for 180 seconds before being interrupted without having found a solution. Given the better performance of DVO-DDR on the other tested problems, this just serves to reinforce the fact that there is no one correct variable ordering.

While FC *may* be able to solve a given planning problem faster, MAC with DVO-DDR or DVO-DDD is competitive. This comes as somewhat of a surprise as, conceptually, one tends to think that MAC involves a “lot of work” while FC is “fast and simple”. Thus it is commonly believed that the extra computation in MAC will slow it down too much for the algorithm to be useful. But the results are clear that MAC can approach the performance of FC on the problems that FC is able to solve, and MAC is able to solve additional problems that are infeasible using FC.

In order for MAC to approach the performance of FC, it needs a good variable ordering. We shall now consider the problem of how to choose a good ordering.

Table 6.4: *Size of various planning problems encoded as CSPs*

CSP	soln. level	At solution bearing level # var.	ave. dom. size	ave # cons.
rocket problem	3	37	1.9	4.6
3 disc towers of hanoi	7	164	2.9	13.5
4 disc towers of hanoi	16	566	4.0	20.9
mystery problem 2	6	435	3.1	25.0
gripper problem 2	6	116	2.8	14.7
logistics rocket-a	8	280	3.8	44.3
bulldozer problem 1	10	269	2.5	10.2
bw-large-a	12	720	5.4	82.3

6.2.3 Choosing a variable ordering

Ideally, we would like to solve each CSP using the best possible variable ordering. It is unlikely that there is an efficient way to find the *best* ordering, so instead we would like to either find a way to pick from several orderings on a per problem basis, or an ordering that works well for many problems.

If we want to choose an ordering on a per problem basis, we need some ways to characterize a CSP in order to make such a decision. Table 6.6 shows the constraint density and the tightness for the tested planning problems. Table 6.4 shows the average domain sizes and average number of constraints per variable in the tested problems.

Perhaps the first thing to consider, is why the 4 disc towers of hanoi problem is unsolvable using either FC or MAC. This problem has a comparatively large average domain size, and low constraint density. This indicates that information propagates poorly through the constraint network (owing to the low constraint density) and thus the domains experience little pruning.

In fact, the average domains reported in Table 6.4 are somewhat misleading. Recall the variable pruning introduced in Section 3.3. The pruned variables were counted when computing the average domain sizes, as the implementation does not

Table 6.5: *Average domain sizes excluding pruned variables*

CSP	# unpruned var.	ave. dom. size
rocket problem	24	2.4
3 disc towers of hanoi	83	4.8
4 disc towers of hanoi	275	7.2
mystery problem 2	247	4.7
gripper problem 2	79	3.6
logistics rocket-a	271	3.9
bulldozer problem 1	105	4.8
bw-large-a	706	5.5

physically remove the variables – it simply reduces their domains to size one and removes all constraints involving the pruned variables. Because these variables no longer have an impact on solution extraction, surely the average domain size should be computed without considering these variables.

Table 6.5 presents the average domain sizes of the planning CSPs without considering the pruned variables. In this table, we see that the 4 disc towers of hanoi problem actually has the largest average domain size of any of the tested problems. The blocks world problem `bw-large-a` has the next largest average domain size, with a large number of variables. MAC could only solve this problem using the default GraphPlan ordering. Not a single random ordering was able to solve this problem.

However, average domain size seems, at best, to merely indicate which problems *may* be unsolvable. The next most difficult problem seems to the `logistics rocket-a` problem. This problem does not have a particularly large average domain size, yet it could only be solved by MAC using DVO-LDF and 8% of the random orderings. Yet, three problems that have larger average domain sizes than the logistics problem are all solvable using any tested ordering. So it does not seem that we can draw any solid conclusions from the average domain sizes.

Let us now turn our attention to the information on constraint density and con-

Table 6.6: *Density and tightness of the solution bearing level of various planning problems encoded as CSPs*

				tightness			
CSP	soln. level	# var.	density	mean	σ	max	min
rocket problem	3	37	0.104	0.236	0.084	0.500	0.167
3 disc towers of hanoi	7	164	0.069	0.301	0.239	0.800	0.008
4 disc towers of hanoi	16	566	0.029	0.333	0.265	0.833	0.004
mystery problem 2	6	435	0.092	0.233	0.198	0.990	0.000
gripper problem 2	6	116	0.110	0.234	0.163	0.675	0.010
logistics rocket-a	8	280	0.141	0.191	0.143	0.694	0.028
bulldozer problem 1	10	269	0.031	0.333	0.278	0.790	0.012
bw-large-a	12	720	0.115	0.301	0.189	0.904	0.000

straint tightness shown in Table 6.6. Notice that these problems have low density and relatively high tightness. There is experimental evidence that indicates MAC performs well in general on problems with these characteristics, and our findings are consistent with this.

It does not seem that we have a good way to recommend a variable ordering based on any of the CSP metrics that we have collected. Perhaps then, we should suggest a standard ordering that performs well on most problems. MAC with DVO-DDR or DVO-DDD is very competitive with FC. Thus it seems that one of these dynamic orderings may be the best variable ordering to use with MAC. Clearly, the ordering is not guaranteed to be best, but it seems to perform reasonably well.

6.3 Improvement to the MAC algorithm

Although widely known and used, the MAC algorithm itself does not seem to be well described in the literature. There is a recursive implementation given by Sabin and Freuder [20], and other papers which discuss MAC seem to assume that the reader is familiar with how to implement it. Indeed, it is not terribly difficult to implement and the algorithm can be thought of as the most naïve backtracking search, but where

an AC algorithm is called each time a variable is assigned a value. We backtrack whenever AC completely eliminates some variable's entire domain. And of course, some extra information must be maintained in order to correctly undo the effects of enforcing AC upon backtracking.

A recursive algorithm is not well suited to the size of these planning problems, so an iterative implementation was needed. Algorithm 1 presents such an iterative implementation. In the course of developing the iterative implementation, we found a rather simple yet effective improvement.

The `while` loop from lines 20 to 22 in Algorithm 1 simply avoids calling the AC procedure when the next variable has a domain size of one. Because AC is enforced at each step of solution extraction in MAC, we know that any variable with a domain size of one must be consistent with all of its neighbors when that value is selected. The recursive implementation given by Sabin and Freuder does not exhibit this improvement. Given the simplicity of this improvement, it is likely that it may have been found and utilized by others, but there does not seem to be any mention of it in the literature.

Table 6.7 presents a comparison of the costs associated with solving various planning problems both with and without the above improvement. The number of nodes visited is unchanged by the optimization, as the only thing it can do is avoid making calls to the AC routine when the constraint network can already be guaranteed to be in an arc-consistent state.

The results clearly indicate the efficacy of this improvement, as none of the trials indicate less than a doubling of performance. In fact, MAC-I always uses fewer than half as many calls to the AC routine as the unimproved version of MAC. As the improvement does not take advantage of anything special about planning problems, it seems likely that it would be effective in solving general CSPs. Indeed, the amount

Algorithm 1 Improved iterative implementation of MAC

```
1: if not ac() then
2:    no solution found
3: end if
4: i = 1
5: loop
6:    consistent = 1
7:    if variables[i].nextValue() then
8:       if not ac() then
9:          consistent = 0
10:         undoac()
11:      end if
12:   else
13:      consistent = 0
14:      update domain of variables[i] as in FC
15:      undoac()
16:      i = i-1
17:   end if
18:   if consistent then
19:      i = i+1
20:      while variables[i].currentDomain.size == 1 do
21:         i = i+1
22:      end while
23:   end if
24:   if  i > variables.size then
25:      solution found
26:      break
27:   else if  current < 1  then
28:      no solution found
29:      break
30:   end if
31: end loop
```

Table 6.7: *Constraint checks and CPU time (in seconds) required to solve planning CSPs using the standard GraphPlan ordering. The first column presents results of the improved MAC while the second column presents results of the normal MAC. The final column shows the speedup.*

	MAC-I	MAC	$\frac{MAC}{MAC-I}$
rocket problem			
#CC	830	2,986	3.60
CPU time	0.02	0.02	1.00
calls to AC	2	38	19
skipped calls to AC	36	0	-
3 disc towers of hanoi			
#CC	119,140	577,905	4.85
CPU time	0.15	0.49	3.27
calls to AC	24	210	8.75
skipped calls to AC	186	0	-
4 disc towers of hanoi			
#CC	-	-	-
CPU time	> 180	> 180	-
calls to AC	-	-	-
skipped calls to AC	-	-	-
mystery problem 2			
#CC	38,098,959	78,502,074	2.06
CPU time	39.2	116.18	2.96
calls to AC	340	839	2.46
skipped calls to AC	499	0	-
gripper problem 2			
#CC	1,455,606	5,456,754	3.75
CPU time	1.24	5.08	4.10
calls to AC	627	2311	3.68
skipped calls to AC	1684	0	-
logistics rocket-a			
#CC	-	-	-
CPU time	> 180	> 180	-
calls to AC	-	-	-
skipped calls to AC	-	-	-
bulldozer problem 1			
#CC	77,473	521,952	6.74
CPU time	0.18	0.83	4.61
calls to AC	16	309	19.31
skipped calls to AC	293	0	-
bw-large-a			
#CC	112,841,652	-	-
CPU time	70.02	> 180	-
calls to AC	433	-	-
skipped calls to AC	6047	-	-

of work required to check if the next variable has a domain size of one is very small compared to the amount of work required to run an AC algorithm on a network that is already arc consistent.

6.4 Conclusion

MAC seems to be an effective tool for use in solving planning problems formulated as CSPs. Using static variable orderings, MAC is able to come near the performance of FC in solving these problems. Using dynamic variable ordering, MAC is able in several cases to surpass the performance of FC. While FC quickly gets lost if solution extraction does not begin at the goal level, MAC performs best using orderings quite different from the GraphPlan based ordering needed by FC. That is, FC only seemed to work using the default GraphPlan static ordering while MAC could find solutions using many different variable orderings.

The improvement outlined in Section 6.3 certainly helped improve MAC in the context of solving planning problems. Given the extremely low overhead of the improvement and its potential for reducing effort, we strongly believe that this improvement would carry over to solving general CSPs.

Chapter 7

Conclusion

In this thesis we have considered a method of converting an AI Planning Problem to a Constraint Satisfaction Problem. We have then turned our attention to the application of various CSP techniques to solving these planning problems.

7.1 Summarizing our contributions

We identified a variable pruning method that allows us to detect certain dual-valued variables that have no impact on the planning problem. These variables can be reduced to a single value and their constraints can safely be ignored. We found that these variables generally correspond to the sort of typing information needed by the action definitions in planning domains.

Additionally, we have shown the ineffectiveness of DNPI in the context of AI planning problems that are modeled using the CSP formulation that we studied. Because the objects in planning problems are always enumerated and actions are fully instantiated, there is no way for two actions to have completely identical impacts on future variables. This is a consequence of the formalisms used to express such problems.

We also investigated a decomposition strategy for the CSPs we create that encode planning problems. This decomposition strategy was not fruitful, as any subproblem not involving the goal information of the planning problem will generate far too many partial solutions. A planning graph contains solutions to an exceedingly large number of problems. Consequently, each subproblem will have many partial solutions to these extra problems that we are not actually attempting to solve.

Finally, we provide an iterative algorithm for MAC, as well as an enhancement to MAC that can provide a several-fold increase in performance. The existing CSP literature provides no clearly described iterative algorithm for MAC.

7.2 Future work

This work has left us with some questions that deserve investigation:

1. Can the pruning technique described in section 3.3 be used to derive type information? How does this relate to the typing information discovered by TIM [10]?

2. Is our improvement to MAC equally effective in solving general CSPs?

3. How can we take advantage of the kind of symmetry we see in solutions to planning problems? For example, the rocket problem we have used as an example throughout this thesis has the following two solutions:

```
1 : load jason r2 london
1 : load alex r1 london
2 : move r2 london jfk
2 : move r1 london paris
3 : unload jason r2 jfk
3 : unload alex r1 paris

1 : load jason r1 london
```

```
1 : load alex r2 london
2 : move r1 london jfk
2 : move r2 london paris
3 : unload jason r1 jfk
3 : unload alex r2 paris
```

The only difference between these solutions is which person rides which rocket. That is to say, one solution is a permutation of the other. This sort of situation occurs in other planning domains like the logistics domain and the gripper domain, and so it seems to be a fruitful area of research.

4. Can we effectively decide which dynamic ordering heuristic to use, based on some information about a planning problem that can be known ahead of time?

5. Can we find other ways to model planning problems as a CSP that might allow techniques such as DNPI to produce better results?

Appendix A

Implementation details

The code written for this thesis was a C++ extension of the GP-CSP planning system. GP-CSP was written in C, but it is not overly difficult to merge a C++ program with a C program. C++ was chosen over C because of certain niceties afforded by an object oriented language.

Section A.1 shows an outline of the project structure, how to build the program, and how to run it. Section A.2 discusses our representation of Constraint Satisfaction Problems in the context of C++ objects. Section A.3 presents an overview of the solution extraction methods we have implemented. Finally, section A.4 briefly discusses the problems that we used to test our planner.

A.1 Project structure

The files that we have added to GP-CSP are found in the `tundish` directory. Additionally, the top-level makefile has been altered to build these extra files and link them with GP-CSP.

- `Makefile`

- `csp.cc`
- `csp.hh`
- `dnpi.cc`
- `dnpi.hh`
- `graphplan.h`
- `generate.cc`
- `my_gp_csp.cc`
- `my_gp_csp.hh`
- `solve.cc`
- `stl/`
  - `Makefile`
  - `hash.cc`
  - `hash.hh`
  - `list.cc`
  - `list.hh`

The `stl` directory contains a linked list and a hash table written to duplicate the interface presented by the versions of these data structure's found in Java.

The project can be built using the unix `make` command. An executable named `gpcsp` will be created, that can be run as follows:

```
gpcsp -csp -o domain_file -f problem_file
```

where *domain_file* is a file containing a PDDL domain description and *problem_file* is a file containing a PDDL problem definition. The `-csp` flag tells the program to use our CSP solver instead of the traditional GraphPlan based solution extraction.

A.2 CSP representation

GP-CSP has code which automatically converts a GraphPlan planning graph to a CSP. We built a new CSP representation in C++ and were able to modify GP-CSP's conversion routine to produce a CSP formulated in our representation. By using C++, our CSP representation takes advantage of object orientation to hide certain implementation details and expose methods that more closely match the pseudocode presented for CSP algorithms. Our representation does not have computational benefits over GP-CSP's original C representation, but the abstraction makes the code easy to follow.

The CSP itself is represented by a C++ class which contains a pointer to an array of CSP variables and some bookkeeping information that is updated by solution extraction methods. The CSP class also exposes a method to systematically output the contents of all CSP variables and their domains and constraints.

The C++ class which represents CSP variables is more complicated. It needs to record:

- the domain of a variable
- the fact to which the variable corresponds
- the current value assigned to the variable
- information required by various solution extraction methods, like a list of future variables for forward checking, or the bundled domains generated by DNPI
- information required by various solution extraction methods to undo instantiations
- a list of constraints in which the variable participates

Additionally, this class exposes methods to:

- iterate through the standard domain
- iterate through the *bundled* domain
- check consistency with other variables
- undo the effects of variable assignment, needed when consistency methods backtrack

Each value in a CSP variable's domain, is represented by another class. The value class simply records information like the action to which the value corresponds and the level in the planning graph where this action is found. The value class does not expose any methods.

A.3 Solution extraction

Several solution extraction methods were written as a part of this thesis. The backtracking search and the forward checking search were implemented as described in [18], with a few modifications. The modifications were slight and just had to do with the fact that our object oriented representation cut down on the amount of global data that needed to be passed around.

Our MAC algorithm was the iterative implementation described in section 6.3. Our CSP representation was adjusted to make it closely resemble the pseudocode shown in Algorithm 1. MAC requires an arc consistency algorithm, and we use AC-3 as described in [2].

The final solution extraction method with which we experimented is DNPI. The only DNPI function exposed publicly is the solve method, which all of the solution extraction methods expose. Hence changing the solution extraction method only

requires changing one line of code. DNPI uses a kind of forward checking search, in that each time a discrimination tree is built, the associated forward checking is necessarily computed and can be cached. The difficult part is the construction of a discrimination tree, which is carried out by a separate function. This function is called when we advance to a new variable which has not yet had a discrimination tree built.

When a solution is found, all of the solution extraction methods leave the CSP in a state where all variables are assigned a value. The value is one of three types:

1. a `noop` – meaning that the fact remains true from earlier in the plan graph
2. a `nothing` – meaning that the fact is never made true
3. a "real" action – meaning that an action has been instantiated to achieve the fact

. Thus a common function is able to print solutions generated by any solution extraction method as it needs only traverse the list of variables and output those that are "real" actions. Solutions to planning problems do not list `noop`s since an execution agent does not need to intervene to achieve the effect of a `noop`.

A.3.1 Further details

The file `solve.cc` contains the following functions:

- `int mac(csp_t *C)`

  Finds a solution, if one exists, to the CSP using the MAC algorithm.

- `int csp_bt_solve(csp_t *C)`

  Finds a solution, if one exists, to the CSP using a backtracking search.

- `int csp_fc_solve(csp_t *C)`

  Finds a solution, if one exists, to the CSP using forward checking.

- `int fc_label(csp_t *C, int i)`

  Required by the function `csp_fc_solve`.

- `int fc_unlabel(csp_t *C, int i)`

  Required by the function `csp_fc_solve`.

- `int ac(csp_t *C, int level)`

  Enforces arc consistency; called by the function `mac`.

- `int revise(csp_var_t *xi, csp_var_t *xj, int level)`

  Required by the function `ac`.

- `int propagation(LinkedList *Q, int level)`

  Required by the function `ac`.

- `int undo_ac(csp_t *C, int level)`

  Required by the function `mac`.

The file `dnpi.cc` contains the following functions:

- `int DNPI::solve(csp_t *csp)`

  Finds a solution, if one exists, to the CSP using the DNPI method.

- `int DNPI::fc_label(int i)`

  Required by the function `DNPI::solve`.

- `int DNPI::fc_unlabel(int i)`

  Required by the function `DNPI::solve`.

- `void DNPI::discriminationTree(int i)`

  Required by the function `DNPI::fc_label`.

- `JDTnode* search(JDTnode *start, csp_var_t* var, void* val)`

  Required by the function `DNPI::discriminationTree`.

A.4 Test problems

The problems that we used to test our planning system were included as part of the GP-CSP distribution. The web page for GP-CSP is located at

`http://rakaposhi.eas.asu.edu/gp-csp.html`

The problems which we used were all encoded in PDDL, which most modern planning systems are built to handle.

Additionally, we encoded one domain of our own, and a problem to go with it. Figure A.1 shows the PDDL definition of the rocket domain. Figure A.2 shows the PDDL definition of a problem in this domain, namely the one where Alex wants to go to Paris and Jason wants to go to JFK.

```
(define (domain rocket)
(:requirements :strips)
(:predicates (in ?o ?r) (at ?o ?p) (has-fuel ?r)
             (rocket ?r) (cargo ?c) (place ?p))

(:action load
:parameters (?c ?r ?p)
:precondition (and (at ?r ?p) (at ?c ?p) (rocket ?r)
             (cargo ?c) (place ?p))
:effect (and (in ?c ?r) (not (at ?c ?p)))
)

(:action unload
:parameters (?c ?r ?p)
:precondition (and (at ?r ?p) (in ?c ?r) (rocket ?r)
             (cargo ?c) (place ?p))
:effect (and (at ?c ?p) (not (in ?c ?r)))
)

(:action move
:parameters (?r ?from ?to)
:precondition (and (has-fuel ?r) (at ?r ?from) (rocket ?r)
             (place ?from) (place ?to))
:effect (and (at ?r ?to) (not (has-fuel ?r)) (not (at ?r ?from)))
))
```

Figure A.1: *Domain definition for the rocket domain.*

```
;2 objects to be transported

(define (problem rocket1)
(:domain rocket)
(:objects london paris jfk r1 r2 alex jason)

(:init
(at r1 london)
(at r2 london)
(at alex london)
(at jason london)
(has-fuel r1)
(has-fuel r2)
(rocket r1)
(rocket r2)
(place london)
(place paris)
(place jfk)
(cargo alex)
(cargo jason))

(:goal (and (at alex paris)
            (at jason jfk))))
```

Figure A.2: *Problem definition for the rocket problem.*

Bibliography

[1] Amy M. Beckwith and Berthe Y. Choueiry. On the Dynamic Detection of Interchangeability in Finite Constraint Satisfaction Problems. In Toby Walsh, editor, *Proceedings of* 7th *International Conference on Principle and Practice of Constraint Programming (CP'01)*, volume 2239 of *Lecture Notes in Computer Science*, page 760, Paphos, Cyprus, 2001. Springer Verlag.

[2] Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI 2001*, pages 309–315, 2001.

[3] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642, 1995.

[4] Blai Bonet and Hector Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[5] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, 1987.

[6] Berthe Y. Choueiry and Guevara Noubir. On the computation of local interchangeability in discrete constraint satisfaction problems. In *AAAI/IAAI*, pages 326–333, 1998.

[7] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[8] Minh Binh Do and Subbarao Kambhampati. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence*, 132(2):151–182, 2001.

[9] Richard Fikes and Nils Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[10] Maria Fox and Derek Long. The automaic inference of state invariants in tim. *Journal of Artificial Intelligence Research*, 9:367–421, 1998.

[11] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl—the planning domain definition language, 1998.

[12] Robert M. Haralick and Gordon L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

[13] Subbarao Kambhampati and Romeo Sanchez Nigenda. Distance-based goal-ordering heuristics for graphplan. In *Artificial Intelligence Planning Systems*, pages 315–322, 2000.

[14] Henry A. Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. In *Proceedings of the Fifth International Conference on the Principle of Knowledge Representation and Reasoning (KR'96)*, pages 374–384, 1996.

[15] Alan Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[16] Sanjay Mittal and Brian Falkenhainer. Dynamic Constraint Satisfaction Problems. In *Proc. of AAAI-90*, pages 25–32, Boston, MA, 1990.

[17] Roger Mohr and Gérald Masini. Good old discrete relaxation. In *Proceedings of the 8th European Conference on Artificial Intelligence*, pages 651–656, 1988.

[18] Patrick Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9 (3):268–299, 1993.

[19] Peter Revesz. *Introduction to Constraint Databases*. Springer, New York, 2002.

[20] Daniel Sabin and Eugene C. Freuder. Understanding and improving the MAC algorithm. In *Principles and Practice of Constraint Programming*, pages 167–181, 1997.

[21] David E. Smith and Daniel S. Weld. Conformant graphplan. In *Proc. of AAAI-98*, pages 889–896, 1998.

[22] Peter van Beek. *CSPLIB: A library of CSP routines*. University of Alberta, `http://ai.uwaterloo.ca/~vanbeek/software/software.html`, 1994.

[23] Peter van Beek and Xinguang Chen. Cplan: A constraint programming approach to planning. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 585–590, 1999.

[24] Daniel S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.

[25] Daniel S. Weld. Recent advances in AI planning. *AI Magazine*, 20(2):93–123, 1999.