Delaying Variable Binding Commitments in Planning

Qiang Yang and Alex Y.M. Chan
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L3G1
(qyang, ymchan)@logos.uwaterloo.ca

Abstract

One of the problems with many partial-order planners is their eager commitment to variable bindings. This is contrary to their control decision in delayed-commitment of operator orderings. In this paper we present an extension to the classical partial-order planners by associating every variable with a finite domain of values. The extended planner applies constraint satisfaction routines to check for the consistency of variable bindings. With the extended planner, we perform a set of experiments to show that it is able to reduce the amount of backtracking in domains where few variable instantiations can lead to final solutions. We also investigate the frequency of CSP application to determine how often should constraint checking be performed in order to yield the best performance.

Introduction

Recently, much research effort has been devoted to the study of partial-order planning systems(Barrett & Weld 1992; Chapman 1987). Researchers have studied different methods of imposing ordering constraints on the operators in a plan. It has been shown that when appropriately used, this *least-commitment* strategy can effectively reduce a planner's search space.

Despite the theoretical advances in the handling of operator ordering constraints in a plan, relatively little has been known on how best to handle *variable bindings* in planning. Most of the existing techniques adopt an ad-hoc method of *eager* commitment. Thus, if a new operator instance α is introduced in a plan, and if α has a precondition $P(?x)$, the premise is that the variable $?x$ will be eventually bound to one of the constant objects in the domain description. If there are several objects that satisfy the predicate P initially, then when $P(?x)$ is being achieved, the variable $?x$ will be bound to a constant object in each separate branch of the search tree.

This naive approach to dealing with variable bindings has a number of drawbacks. First of all, because of the eager commitment in variable binding, if there are 100 constants that can be bound to a variable $?x$, then 100 branches of the search tree will be generated when $?x$ is instantiated. Without an intelligent facility to represent bounded domains for variables, the branching factor of the search tree must be enlarged.

In addition to the branching-factor problem, another problem known as *thrashing* in heuristic search can occur. This problem happens when the same part of the search tree fails repeatedly due to the same reason. For example, let $?x$ and $?y$ be two variables in a plan. Suppose that $?x$ is bound to a constant A first, and that the constant A is incompatible with another constant B. If $?y$ is bound to B later in a planning process, the plan will fail due to the incompatibility between A and B. However, the chronological backtracking method used in most current planning systems will simply backtrack to a level immediately above the last one. The end result of this repeated backtracking is that the part of the search tree between the binding of $?x$ and $?y$ will be found useless, and a significant amount of computation would be wasted.

A solution to the above problem would be to represent the objects which each variable can be bound to *collectively*. A method known as Arc-Consistency in Constraint Satisfaction could then be applied to find out the incompatibility between the two constants A and B much earlier than with the naive method, avoiding the computational problems mentioned above. These extensions, including the augmented plan language as well as the associated reasoning techniques based on constraint satisfaction, are the main contributions of this paper.

To be sure, mixing planning with reasoning about variable bindings has been successfully attempted by a number of practical planning systems, including MOLGEN(Stefik 1981), SIPE(Wilkins 1988), GEMPLAN(Lansky 1988) and ISIS(Fox 1987). What has not been clear from the application of these systems is one of a control issue: How much constraint reasoning should be applied to each plan? How does the problem domain features such as solution density affect the efficiency of the constraint-based methods? And how often should constraints between variables be considered during a planning process?

In this paper we present an extended plan language

where each variable is associated with a finite set of domain values. We also discuss our planning algorithm which reasons about these finite variable domains in each plan and guarantees the soundness and completeness properties. In addition, we present a set of empirical results to demonstrate the efficiency gain of the extended planner and to answer the above utility question regarding constraint satisfaction.

A Constraint-Based Planner

In this section, we review the traditional partial order planners and then present our extension using constraint-based techniques.

Traditional Partial-order Planners

A traditional partial order planner represents an operator using a three tuple, a precondition list, an add list, and a delete list. Each list represents a set of literals. In an operator description, we follow the SNLP(Barrett & Weld 1992) convention to use $?x$ to denote a variable parameter $?x$. In addition to the operators, a domain is specified using two state descriptions, the initial state and the goal state. Both states are represented using sets of literals.

```
shape(?x)
  Preconds: Object(?x)
  Adds: Shaped(?x)
  Deletes: Drilled(?x), Painted(?x)
drill(?x)
  Preconds: Object(?x)
  Adds: Drilled(?x)
  Deletes: Painted(?x)
paint(?x)
  Preconds: Object(?x)
  Adds: Steel(?x)
  Deletes: Painted(?x)
```

As an example, consider a machining domain described by Smith and Peot (Smith & Peot 1992). The task is to transform a piece of stock into a desired form, by drilling, shaping and painting. The operators of the domain are listed above. The initial state $\mathcal{I}$ describes a domain where there are 100 pieces of stocks, all of which are objects. In addition, one of the stock is made of steel: $\mathcal{I} = \{\text{Object}(S_1), \ldots \text{Object}(S_{100}), \text{Steel}(S_{50})\}$. Thus, the initial state consists of 101 literals.

The goal of the domain is to find an object $?y$ and perform operations to make it shaped, drilled and painted: $\mathcal{G} = \text{Shaped}(?y), \text{Drilled}(?y), \text{Painted}(?y)$.

For a given domain description, a traditional partial-order planner invokes a search algorithm to find a plan from the goal state backwards. A top-level description of these algorithms is shown below.

```
1  open-list := { initial-plan }.
2  repeat
3     plan := lowest cost node in open-list;
4     remove plan from open-list;
5     if Correct(plan)=True then return plan ;
6     else
7        If threats exist
8           Resolve-Threats;
9        else
10          Establish-Precondition(plan);
11       endif
12       Add successor plans to open-list;
13    endif
14 until open-list is empty;
```

In this algorithm, the procedure Correct(*plan*) is a boolean function. For a sound planner, if it returns True, then the plan must be correct. For the SNLP algorithm, it is a termination routine that checks to see if every precondition of every operator has an associated causal link, and if every causal link is safe(Barrett & Weld 1992).

If *plan* is not yet correct, then a successor generation process is initiated. The procedure Establish-Precondition(*plan*) inserts operators and constraints to achieve a precondition that has not been satisfied. Let $p(?x)$ be a precondition of an operator β in a plan. If $p(?x)$ has not yet been satisfied, then a search will be made to find all existing or new operators α that have an add-list literal $p(?y)$ such that $p(?y)$ is unifiable with $p(?x)$. For each such operator instance, the following constraints are imposed on a successor plan:

1. $\alpha \prec \beta$, where $\prec$ represents the precedence relation in the plan.
2. $?x = ?y$, where $=$ is a binding constraint. $?x = ?y$ states that $?x$ and $?y$ must be bound to exactly the same constant in all instances of the plan.

The above procedure could be extended to literals with more than one parameter and literals with constant parameter.

Extending the Domain Language

In the previous machine shop example, when achieving Object($?x$) we would like to avoid the premature commitment of $?x$ to any particular stock. That is, we could associate with $?x$ a *domain* of values, each value being a stock stated in the initial state: $\text{Domain}(?x) = \{S_1, S_2, \ldots, S_{100}\}$. To support this consideration, in every plan and every operator we associate each variable with a *domain*. A domain for $?x$ can be a set of objects. The denotation of $\text{Domain}(?x) = \{O_1, O_2, \ldots, O_n\}$ is that $?x$ can be bound to any one of O_1 through O_n. Thus, the concept of a variable domain has a disjunctive or existential meaning to it.

Extending the Planning Algorithm

With the extension of the planning domain language, the planning algorithm can be extended at two points.

The first extension is in the termination step, Step 5 of the above partial-order planning algorithm. In addi-

tion to the traditional way of checking the correctness of a plan, we must now make sure that for the set of variables in the plan, each variable can be assigned a value from its domain, such that all variable binding constraints are satisfied.

To look at this problem in more detail, consider a plan with the following set of variables and domains. The variables are $?x_1, ?x_2, and ?x_3$, all with a domain $\{a, b\}$. The constraints require that no two variables take the same value.

For this problem, there does not exist a *consistent* assignment of the three variables such that all three constraints are satisfied. What we would like to add in Step 5 is a routine that can tell us whether or not a consistent assignments exist for all variables. If not, as in the case above, then the plan should be discarded.

The algorithms that perform this task are known as *Constraint Satisfaction Algorithms.* Given a set of variables, domain values for the variables and constraints among them, a constraint satisfaction algorithm determines whether or not a consistent assignment of values to the variables can be obtained. A large number of such algorithms have been found(Kumar 1992). For simplicity, we refer to any such algorithm as CSP(*plan*). In addition, we refer to the instance of the plan in which all variables are consistently assigned a value *Instance(plan).*

An extension of Step 5 is as follows:

5. if Correct(*plan*)=True, then if CSP(*plan*)= True then *return (plan Instance(plan))*;

The second extension to the planning algorithm is in the Establish-Precondition part (Step 10). First, we add the CSP checking routine to this part so that nodes with inconsistent variable bindings can be pruned. For efficiency purposes, the frequency of CSP routine application here can be specified by the user. Furthermore, whenever a precondition $p(?x)$ is achieved by an operator α, where α adds a literal $p(?y)$, we would like to make sure that $?x$ and $?y$ have an nonempty domain intersection. Then, we would like the domain of $?x$ and $?y$ to be set to that intersection. If, however, we are using the initial state literals to achieve $p(?x)$, then we would like to set the domain of $?x$ to the intersection of objects which satisfy predicate p and the original domain of $?x$. Therefore, all possible constant bindings for $?x$ are obtained without committing to a particular one. More precisely, let β be an operator with precondition $p(?x)$, Establish-Precondition could be implemented as follows.

E1 if CSP(*plan*)= False *then* return $\emptyset$; *Comment to E1: the user specifies how often the CSP routine is applied.*
E2. for all operators α_i which added literal $p(?y)$;
E3. if Domain($?x$) $\cap$ Domain($?y$) $\neq \emptyset$ then
E4. generate a successor P_i with a constraint $?x = ?y$, where Domain($?x$) = Domain($?y$) are set to Domain($?x$) $\cap$ Domain($?y$).
E5. if the initial state contains literals $p(O_j), j = 1, .., k$,
E6. then generate a successor plan Q with Domain($?x$) := Domain($?x$)$\cap$ $\{O_j, j = 1, \ldots, k\}$.
E7. return $\{Q, P_i, i = 1, 2, \ldots\}$.

Efficiency Gain with the Extended Planner

We now discuss some of the efficiency gains from the extended planner. Suppose that at a certain planning step, we introduce an operator with a parameter $?x$ to the plan, where $?x$ can be bound to n different constants, $a1, a2, \ldots, an$. If we next insert another operator with a parameter $?y$ that can be instantiated to any of m constants $b1, b2, \ldots bm$, then the state space of a traditional partial-order planner might look like Figure 1 (A). In this search tree, the partial-order planner commits to one of $a1$ through an at one level, and then commits to one of $b1$ through bm at another level. In the worst case, there are tight constraints on their values which can only be satisfied by constants an and bm , and the entire search tree with a size of $m * n$ must be scanned.

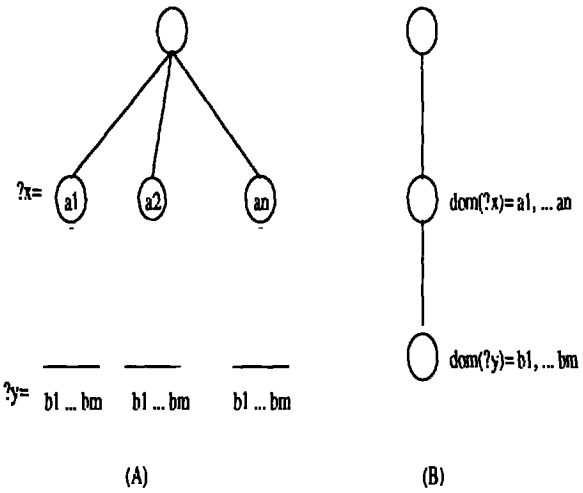

Figure 1: State spaces of the planners before and after the extension.

In contrast to the traditional approach, our extended constraint-based planner does not commit to any particular constant during planning. The domain values for both $?x$ and $?y$ are simply recorded and reasoned about at a later time. Therefore the search tree looks like the one shown in Figure 1 (B).

In order to conclude that the extended planner performs better, we must also take into account the amount of CSP processing done for each node in the search tree. In our extended algorithm, both steps 5 and E1 required CSP processing. We admit that in the worst case, this would mean that the amount of processing per node is exponential. This was the reason why Chapman(Chapman 1987) specifically required that variables in TWEAK must not have finite domains of values. However, by avoiding having finite variable domains, TWEAK (and SNLP) resorted to try out many more possible variable bindings to find one that is consistent, leading to the generation of more

nodes in the search tree.

To guarantee the soundness of the planner, full CSP reasoning must be applied at step 5. However, if we want to increase the efficiency of planning, we must determine, with respect to step E1, (1) *how much* constraint processing should be applied, and (2) *how often* should the CSP algorithms be applied.

For the *how-much* part of the question we have two choices. One is to use a polynomial-time CSP algorithm such as Arc-Consistency(Kumar 1992). Since the arc-consistency algorithms do not enforce global consistency, another alternative is to use a backtracking-based CSP algorithm such as Forward-Checking(Kumar 1992) which has good average-case performance.

The *how-often* part of the question is likely dependent on the problem domain features. One domain feature is where and how often do dead ends caused by inconsistent variable bindings occur. If the dead ends occur early in the search tree, it is more likely that variables in an arbitrary instantiation of a partial-order plan cannot be consistently bound to constants. In this case, applying CSP routines more often could detect dead ends earlier in search, leading to much improved efficiency. If the dead ends occur deep down in the search tree or there are in fact no inconsistencies, the dead end checking routine at Step E1 is likely to waste a lot of time with few nodes pruned.

In the next section, we present a set of experiments to help answer the *how-often* question while keeping the *how-much* issue fixed with an implementation. In Section 5 we discuss the implications of the results.

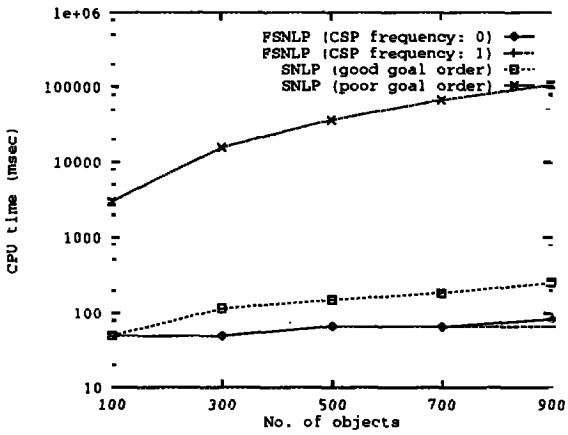

Figure 2: The Domain of Smith and Peot

Empirical Results

To test the effectiveness of our approach, we have extended the SNLP planner (McAllester & Rosenblitt 1991) which was implemented by Barrett and Weld(Barrett & Weld 1992). The new planner, which we call FSNLP, handles finite variable domains as described in Section 2. In this section, we describe our comparison of SNLP and FSNLP over a range of domains. Both planners are implemented in Allegro Common Lisp on a Sun-4 Sparc Station.

In all of the following tests, the subgoals are solved in a LIFO manner. When performing best-first search with both planners, the heuristic function is the sum of the number of unsafe causal links, the number of open conditions to be achieved as well as the number of steps in a plan. The results of the tests are plotted on two dimensional graphs, where the y-axis, displayed in log scale, shows the CPU time required for solving problems.

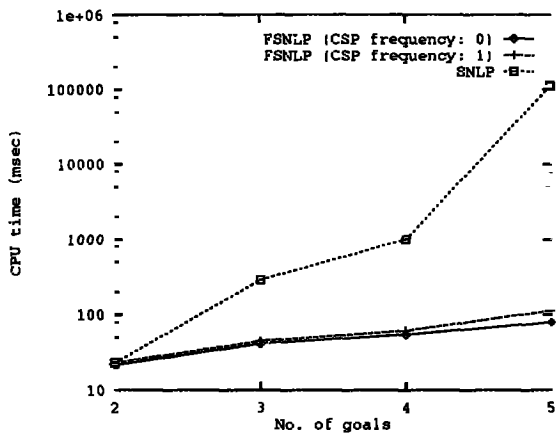

Figure 3: Exponential Domain with high solution density

The Domain of Smith and Peot

The first comparison involves the machine shop domain described in Section 2. The two planners are compared by varying the number of stocks in the initial state. To obtain an average result, it is assumed that for any n pieces of stocks s_1 through s_n the only steel one is $s_{n/2}$. To answer the *how-often* question raised in the last section, we also varied the frequency of CSP application.

Our test result appears in figure 2. In the figure, *FSNLP with CSP frequency 0* represents not applying the CSP routine at Step E1 (first step of Establish-Precondition) at all. *FSNLP with CSP frequency 1* represents applying the CSP routine for every node. *SNLP with poor goal order* stands for the SNLP planner with the goal order given in Section 2, whereas *SNLP with good goal order* represents running the same SNLP planner with an ordering of the goal suggested by Smith and Poet, whereby the planner works on the subgoal *Painted(?y)* first. In (Smith & Peot 1992),

Smith and Peot argued that this good goal order might be the key to overcome a problem with Knoblock's abstraction method.

One thing we can observe from the figure is that, compared to SNLP, FSNLP has a dramatic amount of improvement in efficiency; in fact its complexity stays constant as n increases. Another interesting phenomenon is that for this problem, the frequency of CSP application does not matter. Finally, although a good goal order can have a dramatic effect on improving the efficiency of SNLP, goal order alone still cannot completely close the gap between SNLP and FSNLP. This further demonstrates the advantage of using the constraint-based method.

Exponential Domain

One feature of the above machining domain is that the size of plans do not increase with the number of objects. In this section, we consider another domain which allows us to observe the effect of increasing plan lengths. The initial and goal states of the domain is given below.

$$\mathcal{I} = \{P(o_1), \ldots, P(o_{n-k+1}), \ldots, P(o_n), C(o_{n-k+1}), \ldots, C(o_n), \ldots, C(n_m)\}$$

$\mathcal{G} = \{G(?x_1), \ldots, G_(?x_g), C(?x_u)\}$, where $x_i \neq x_j$ for $i \neq j$ and $1 \leq u \leq g$. In this domain there are m objects, n of which satisfy predicate P, and $m - n + k$ of which satisfy C. Of the m objects k of them satisfy both predicates. The goal is to find g different objects that satisfy G. In addition, one of them must also satisfy C.

We define type O to include all objects in the domain. The operator that allows us to produce G_i from P_i is as follows [1].

achieve-G(?x), *Preconditions:* P(?x)
Adds: G(?x) *Deletes:* {}

In the tests, both the initial conditions and goal conditions were randomly permuted. We gathered average results from 10 random problems for each problem type, with a specific number of goals in $\mathcal{G}$, $g + 1$, and with a specific number (denoted by k) of objects which satisfy both predicates P_i and C.

This domain has a number of interesting features. First, we can vary the plan lengths by changing the parameter g, which controls the number of goals. Second, we can vary the density of solution by changing the number of objects that simultaneously satisfy both P_i and C. This density is represented by the ratio k/n. When the density of solution is low, few objects can satisfy the conjunctive goals $G_1(?x_l) \wedge C(?x_l)$ in the goal conditions. In this case, SNLP is expected to fail on most of the branches of the search tree. In contrast, FSNLP can avoid the *thrashing problem* by delaying the commitment to a particular object and perform much better. When the density of solution is high, and when depth-first search is used, we expect that SNLP is able to zero in onto a solution fast, whereas FSNLP would still spend extra time for CSP processing. Thus, with depth-first search and when the solution density is high, SNLP would be slightly better. The low-density results, under best-first search, are shown in Figure 3 (the solution density is only 2/10). As we can observe from the figure, SNLP performs exponentially worse than FSNLP as the number of goals increases. The results under depth-first search strategy are similar in shape.

We further varied the density of solutions in order to test the tradeoffs between commitment and the overhead of constraint reasoning. In this test the number of goals was kept constant to 5, and both planners were asked to perform depth-first search. The solution density (k/n) is varied from 0/10 to 10/10. The result of the test is shown in Figure 4. It is interesting to observe that as the solution density increases, the performance of SNLP improves. Eventually SNLP even surpassed FSNLP in efficiency when the density is 1. This has a simple explanation: when every instantiation leads to a correct solution, it does not require any addition check by CSP routines. Thus, SNLP can avoid this overhead at the high solution density region. In some other tests, we observed that both SNLP and FSNLP performed with the same order of time complexities even when the number of goals was increasing.

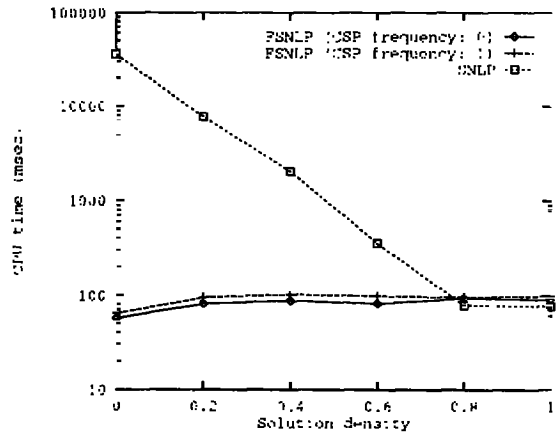

Figure 4: Exponential Domain with varying density

Variable-Sized Blocks Domain

In the above experiments the search performance of FSNLP is not very sensitive to the frequency of CSP application at Step E1. Our analysis shows that the frequency will affect FSNLP when dead ends can be

[1] This domain is similar to D^1S^1 domain described by Barrett and Weld (Barrett & Weld 1992). Our experiment is analogous to the comparison between TOCL and POCL described in their paper.

discovered at varying levels in the planner's search tree. To test this hypothesis, we used a blocksworld domain where there are a number of blocks $B_1, B_2, \ldots, B_n$ of unit size, and where there is a particular block A which can support more than one block. The task is to move all the B blocks on top of A, using the following operator:

PutonA($?x$, $?s$); *Types:* $?x$: B; $?s$: L;
Preconds: $Clear(?x), SpaceOnA(?s)$;
Adds: $OnA(?x)$; *Deletes:* $SpaceOnA(?s)$

In this definition, B is the set of blocks, and L is the set of locations available on block A, and $?s$ could be bound to a location on A. By varying the number of locations that satisfy the predicate *SpaceOnA* in the initial state, we can change where the dead end occurs. in a search tree. For the test, the number of blocks is fixed at 5, and the number of locations on top of block A varies from 2 to 5. The dead end occurs very high up in the search tree if the number of locations is 2, and deep in the search tree if the number of locations is 4. When the number of locations is 5, all 5 blocks can be put on object A at the same time on different locations, and no dead end occurs.

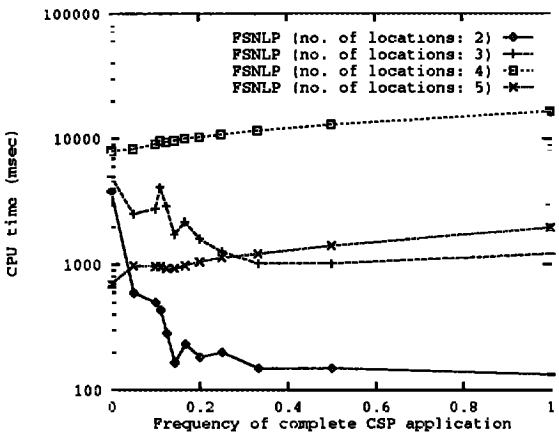

Figure 5: Performance in Variable-Sized Blocks Domain

Figure 5 depicts the results, where the *frequency of CSP application* refers to the inverse of the number of nodes along a search path between any two successive CSP applications at Step E1. The frequency is 1 if CSP is applied to every node. If CSP is not applied to any nodes at all, the frequency is 0. The figure shows that when the dead end occurs early in the search tree, for example, when the number of locations is 2 (the solid line in the figure), the CPU time consumed is in inverse proportion to the frequency of solving the CSP. In the opposite case, situation is reversed: the CPU time used is in direct proportion to the frequency of solving the constraint network, implying that it is better to apply CSP checking infrequently.

Conclusions

In this paper we presented an extension to classical partial-order planners by applying CSP algorithms. The advantage of this approach derives from delayed commitment for variable bindings which is symmetric to the advantage of partial-order planners over the total-order ones. From the experiments we observed a tremendous improvement of efficiency in general with our planner FSNLP over SNLP. We also see that the effectiveness of applying constraint satisfaction algorithms depends on where dead ends occur in the search tree. We are currently performing more extensive experiments(Chan 1994) to further validate this approach.

Acknowledgement

This work is supported by a grant from Natural Sciences and Engineering Research Council of Canada

References

Barrett, A., and Weld, D. 1992. Partial order planning: Evaluating possible efficiency gains. Technical Report 92-05-01, University of Washington, Department of Computer Science and Engineering, Seattle, WA 98105.

Chan, A. Y. 1994. Variable binding commitments in planning. Master's thesis, University of Waterloo, Department of Computer Science, Waterloo, Ont. Canada.

Chapman, D. 1987. Planning for conjunctive goals. *Artificial Intelligence* 32:333–377.

Fox, M. 1987. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling.* Morgan Kaufmann.

Kumar, V. 1992. Algorithms for constraint satisfaction problems: A survey. *AI Magazine* Spring, 1992:32–44.

Lansky, A. L. 1988. Localized event-based reasoning for multiagent domains. *Computational Intelligence* 4(4).

McAllester, D., and Rosenblitt, D. 1991. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, 634–639. San Mateo, CA: Morgan Kaufmann Publishers, Inc.

Smith, D. E., and Peot, M. A. 1992. A critical look at knoblock's hierarchy mechanism. In *Proceedings of the First International Conference on AI Planning Systems*, 307–308.

Stefik, M. 1981. Planning with constraints. *Artificial Intelligence* 16(2):111–140.

Wilkins, D. 1988. *Practical Planning: Extending the Classical AI Planning Paradigm.* San Mateo, CA: Morgan Kaufmann Publishers, Inc.