An Evaluation of the Temporal Coherence Heuristic in Partial-Order Planning[*]

Qiang Yang   Cheryl Murray

Department of Computer Science
University of Waterloo

Abstract

This paper presents an evaluation of a heuristic for partial-order planning, known as *temporal coherence*. The temporal coherence heuristic was proposed by Drummond and Currie as a method to improve the efficiency of partial-order planning without losing the ability to find a solution (i.e. completeness). It works by using a set of domain constraints to prune away plans that do not "make sense," or temporally incoherent. Our analysis shows that, while intuitively appealing, temporal coherence can only be applied to a very specific implementation of a partial-order planner and still maintain completeness. Furthermore, the heuristic does not always improve planning efficiency; in some cases, its application can actually degrade the efficiency of planning dramatically. To understand when the heuristic will work well, we conducted complexity analysis and empirical tests. Our results show that temporal coherence works well when strong domain constraints exist that significantly reduce the search space, when the number of subgoals is small, when the plan size is not too large and when it is inexpensive to check each domain constraint.

Key words: AI planning, Heuristic Problem Solving.

Address: Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1
Tel: (519) 888-4716. E-mail: qyang@logos.uwaterloo.ca

[*]The authors are supported in part by grants from the Natural Sciences and Engineering Research Council of Canada, and ITRC: Information Technology Research Center of Ontario.

1 Introduction

There has recently been a renewed interest in developing algorithms for partial-order planning. Most of the algorithms, including TWEAK[2], SNLP and its variations [8, 1, 6, 7] and ABTWEAK[14], are domain-independent in nature. They are, for the main part, aimed at addressing formal properties such as soundness, completeness and expressive power of the planning systems. To improve their efficiency, certain domain-dependent knowledge has to be utilized. One way to apply domain-dependent heuristics is to prune plans that violate certain constraints specific to a particular domain, thus reducing the size of the search space.

An important property of a pruning heuristic is its completeness, whereby if a solution exists for a planning problem, one solution can be found. Complete pruning heuristics thus always make sure that at least one solution path is retained after pruning is done.

The *temporal coherence* heuristic [4, 5] has been proposed as a complete pruning heuristic. It works by pruning away plans that violate certain *domain constraints*, *i.e.*, constraints that specify physically impossible situations in a domain. To apply the heuristic, a set of preconditions, known as *bulk preconditions*, are found for each plan. Then the domain constraints are applied to the bulk preconditions to check whether they are violated. If so, then the plan is removed from the search space. In [4, 5], Drummond and Currie provided a proof of its completeness, and reported that it could lead to improved efficiency in solving several blocks-world problems. Furthermore, the heuristic has been implemented in O-Plan [3], a successor of NONLIN[10], and was referenced in several influential works of AI planning[12, 11].

Since no complexity result, empirical or analytical, is available that shows how effective the heuristic is, we have implemented it with a version of TWEAK and performed a complexity analysis. We found that the completeness of the temporal coherence heuristic is extremely sensitive to the way a planning algorithm is implemented. Many partial-order planning algorithms work by iterating through cycles of activities. Within each cycle, they select a plan from the search space and generate successor plans as follows:

> **Method 1:** Find *one* subgoal that is currently not true. Find all operators that can achieve this goal. Generate successor plans based on the set of operators.

This method is complete (a proof is presented later in this paper), and is the basis of our implementation of the TWEAK algorithm as well as other algorithms such as SNLP[8, 1].

However, our analysis shows that to maintain completeness, temporal coherence cannot be applied to prune successor plans using the above successor generation method. Instead, it can only be applied to the following method:

> **Method 2:** Find *all* subgoals that are false or solely supported by the initial state. Find all operators that can achieve any such subgoal. Generate successor plans based on these operators.

The main difference between the two methods is that while method 1 generates successor plans based on one subgoal, method 2 is based on all false preconditions in the current plan. Using all subgoals to generate successor plans is likely to increase the branching factor of search. Since both search trees have the same depth, for method 2 to be more efficient the temporal coherence heuristic must be able to prune away a large number of successor plans. However, through experimentation and an asymptotic analysis, we have found that temporal coherence does not guarantee an improvement in search for many domains; under certain conditions, it is in fact dramatically worse to use temporal coherence with method 2 as compared to method 1 without using any heuristic at all.

This negative result prompted us to study the conditions under which temporal coherence will work well. We have found that for temporal coherence (TC) to improve the search efficiency, a number of conditions must be met. First, the strength of the domain constraints must be strong enough for TC to prune a substantial amount of the search space. Second, the number of preconditions of operators must not be too large. Third, very few preconditions in a solution plan are established by the initial state facts. Fourth, the solution plans contain a small number of operators. Finally, the domain constraints are inexpensive to check. Under these conditions, one can show that method 2 with temporal coherence can be more efficient than method 1.

The paper is organized as follows. Section 2 describes a version of TWEAK that will be used as the basis of our subsequent discussions. Section 3 introduces the temporal coherence heuristic and section 4 examines the completeness of several implementations of TWEAK with the application of the heuristic. Sections 5 and 6 present analytical and empirical results on the performance of temporal coherence. Section 7 concludes the paper.

2 A Partial-Order Planning Algorithm

2.1 Definitions

We use the TWEAK language to describe our planning problems and provide definitions for those notations used in the paper. Readers should refer to [2] for more details of the TWEAK language.

A planning domain consists of a set of operators and a set of states. Each state is described by a set of literals. The operators describe possible transitions from one state to the next. Each operator α is defined in terms of a set of precondition literals, *Preconditions*(α), and a set of effect literals *Effects*(α). We use two special operators to represent the initial and goal states, the former describing an agent's initial situation, and the latter stating the desired conditions. Operator $\mathcal{I}$ has a set of empty preconditions, and has as its effects the set of literals true in the initial situation. Likewise, the operator $\mathcal{G}$ has a precondition set identical to the set of goal literals. The effect set of $\mathcal{G}$ is empty.

A plan Π consists of a set of operators restricted by the following constraints:

1. A set of precedence constraints on operator ordering that enforces a *partial order* on the operators. If an operator α precedes β then we denote it by $\alpha \prec \beta$.

2. A set of codesignation constraints on the binding of variables of the operators that enforces pairs of variables x_i and x_j to bind to the same constant. Each codesignation

constraint between variables (or constants) x_i and x_j is denoted $x_i \approx x_j$.

3. A set of non-codesignation constraints, that forces two variables to instantiate to different constants. Each noncodesignation constraint between x_i and x_j is denoted as $x_i \not\approx x_j$.

For simplicity, we assume that the special operators $\mathcal{I}$ and $\mathcal{G}$ exist in every plan. In addition, the precedence constraints always make sure that $\mathcal{I}$ precedes all other operators and $\mathcal{G}$, and that $\mathcal{G}$ is preceded by all other operators.

Let p be a precondition of an operator α. An operator β in the same plan is called an *establisher*, if $\beta \prec \alpha$ holds, and if p is an effect of β. The triple, (β, α, p) is called an *establishment relation.*

An establishment relation can be *clobbered* by other operators. For the above relation (β, α, p), let γ be an operator such that

1. γ can be between β and α without violating the partial order and

2. γ has an effect q that can negate p.

Then γ is called a *clobberer* with the relation (β, α, p). We call the tuple

$$\langle \beta, \alpha, p, \gamma, q \rangle$$

a conflict in the plan [13].

A completion of a plan Π is a total order of the operators in which all variables are replaced by constants and all constraints in Π are satisfied. A precondition p is *necessarily true* just before an operator α, if and only if in *every* completion of the plan, there exists an operator β such that (β, α, p) is an establishment relation and there is no clobberer with the relation. The Modal Truth Criterion[2] can verify whether or not every precondition in a plan is necessarily true in $O(N^3)$ time for a plan with N operators.

A planning problem consists of an initial state, a goal state, and a set of operators. A plan is a *solution* for a planning problem if every precondition of every operator in the plan is necessarily true just before that operator.

2.2 WatTweak

A partial-order planning algorithm takes as input a set of operators, an initial state and a goal state. It attempts to find a solution plan if one exists. This is done by generating a search tree. The nodes on the search frontier of the tree are kept track of by an open-list. Initially, the open-list contains only the initial plan, with the initial state operator $\mathcal{I}$ and the goal state operator $\mathcal{G}$. The algorithm terminates when every precondition of every operator is *necessarily true* just before that operator.

The important properties of a partial-order planning algorithm include its soundness, completeness, and efficiency. An algorithm is *sound* if every solution it generates is guaranteed to reach the goal from the initial state. It is *complete* if it always terminates with a solution provided that one exists.

Input: an initial plan: the initial state operator followed by the goal state operator.
Output: a solution plan if one exists.

Algorithm WatTweak

```
1   open-list := { initial-plan }.
2   repeat
3      plan = lowest cost node in open-list;
4      remove plan from open-list;
5      if plan is a solution plan, then return plan
6      else
7         successors := generate-successors(plan);
8         add successors to open-list;
9      endif
9   until open-list is empty;
```

Table 1: WatTweak planning algorithm.

In [2], Chapman presents a high level discussion of the planner Tweak. In this paper, we consider a particular variation of Tweak called WatTweak. A top-level routine for WatTweak is presented in Table 1. In the algorithm, the `generate-successors` routine is yet unspecified. Under a breadth-first search strategy, this procedure not only determines the completeness of WatTweak, but also its efficiency. The successor generation routine (step 7) can be strengthened by heuristics that eliminate some of the successors which are "dominated" by others, thereby effectively reducing the branching factor of the search tree. The *temporal coherence* heuristic designed by Drummond and Currie was proposed as such a successor-elimination heuristic.

3 Temporal Coherence

Temporal coherence (TC) is a heuristic for pruning some plans from the search space in order to reduce the total cost of search. The heuristic avoids working on partially completed plans that don't "make sense," that is, plans that are not physically realizable. To check if a partial plan makes sense, the heuristic first determines the *bulk preconditions* of the plan, and checks them against all the *domain constraints* for the state space. The bulk preconditions are preconditions of operators in the current plan, that need to be made true in order for the rest of the plan to be executable. Domain constraints define a number of physical laws

(MOVE $1 to $2 from $3)			
Preconditions	(IsBlock $1), (IsBlock $2), (CLEAR $1), (CLEAR $2), (ON $1,$3).	Effects	(ON $1,$2), (not CLEAR $2), (CLEAR $3) (not ON $1,$3).
(MOVE $1 to Table from $2)			
Preconditions	(IsBlock $1), (IsBlock $2), (CLEAR $1), (ON $1,$2)	Effects	(ON $1,Table), (not ON $1,$2), (CLEAR $2)

Table 2: Operator definition for the blocks world domain.

in a particular application domain. If anyone of the domain constraints is violated in the current bulk preconditions, then according to TC the current plan should be pruned from the search space.

Below, we provide a more detailed description of both bulk preconditions and domain constraints.

3.1 Bulk Preconditions

The bulk preconditions of a partially ordered plan consist of all the preconditions which are not *necessarily true*, and those preconditions which are only true because they are asserted by the initial state. To illustrate this, consider the following simple example from the blocks world domain. The operators of this domain are listed in Table 2. In the parameter lists, any symbol preceded by a $ sign is an unbound variable.

The initial state is

((IsBlock A) (IsBlock B) (ON A B) (ON B TABLE) (CLEAR A)),

and goal state is ((ON B A)), as shown in Figure 1.

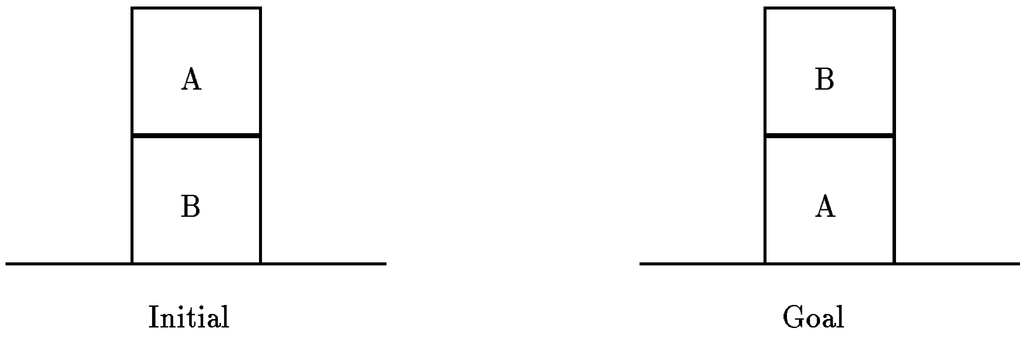

Figure 1: A Blocks World Example

For this problem, a partially completed plan might be:

$$\mathcal{I} \longrightarrow \text{(MOVE B to A from \$X)} \longrightarrow \mathcal{G}$$

The operator (MOVE B to A from $X) achieves the goal, (ON B A), and has 5 preconditions of its own

(IsBlock A), (IsBlock B), (ON B $X), (CLEAR A), and (CLEAR B).

Among those, (CLEAR B) and (ON B $X) are bulk preconditions of the plan, because they are not satisfied. The two IsBlock literals and (CLEAR A) are also bulk preconditions because they are only satisfied by the initial state.

The intuition of bulk preconditions can be explained via a solution plan to the above problem:

$$\mathcal{I} \rightarrow^{1} \text{(MOVE A to Table from B)} \rightarrow^{2} \text{(MOVE B to A from Table)} \rightarrow^{3} \mathcal{G}.$$

The plan is annotated by three positions between its operators. In the above plan, the partial plan after position 1 is the entire solution itself. The bulk preconditions for this plan are the conditions that must be true in the initial state in order for the entire plan to be executed. Similarly, the partial plan after position 2 consists of the operator (MOVE B to A from Table) followed by the goal state. In order for this partial plan to be executable, all preconditions of the operators must be true, including those asserted by the first operator, (MOVE A to Table from B), and the initial state. These are the bulk preconditions of the partial plan. Through this example, it is not hard to see that the definition of bulk preconditions is exactly the partial-order analogue of a *kernel* in the triangle table of a STRIPS plan[9].

3.2 Domain Constraints and Temporal Coherence

The domain constraints specify physical laws that cannot be violated by any state of the agent . For example ((ON $1 $2) (ON $1 $3)) might be a domain constraint for the blocks world, indicating that a block $1 cannot be on top of two different blocks, $2 and $3, at the same time. In each domain constraint, unique variable names are assumed to be bound to unique constants or variables which are necessarily non-codesignating. In the above example, if the bulk preconditions contained both (ON A B) and (ON A C), there would be a violation of domain constraints, making the bulk preconditions temporally incoherent.

As another example, consider the Towers of Hanoi domain, where there are 3 pegs and 3 disks. Let the three pegs be $\mathtt{P}_1$,$\mathtt{P}_2$, and $\mathtt{P}_3$, and let the disks be `Big`, `Medium` and `Small`. We can represent the location of the disks using literals `ONB($x)`, `ONM($x)`, and `ONS($x)`. Initially, all disks are on $\mathtt{P}_1$, and in the goal state they are on $\mathtt{P}_3$. The operators for moving the disks can be represented as shown in Table 3.

For this domain, the domain constraints specify that a disk can only be on one peg at a time, that a disk cannot simultaneously be on a peg and not on the same peg, and that it is impossible that a disk not be on any of the three pegs. For the big disk, these three constraints imply that when $\$x$, $\$y$ and $\$z$ are bound to different constants, a violation of TC occurs if the bulk preconditions match any of the following patterns:

Preconditions	*Effects*
`MoveL($x,$y)`	
`Ispeg($x), Ispeg($y), not (ONS $x), (not ONS $y), not (ONM $x), (not ONM $y), (ONB $x)`	`(not ONB $x), (ONB $y)`
`MoveM($x,$y)`	
`(Ispeg $x), (Ispeg $y), (not ONS $x), (not ONS $y), (ONM x)`	`(not ONM $x), (ONM $y)`
`MoveS($x,$y)`	
`(Ispeg $x), (Ispeg $y), (ONS $x)`	`(not ONS $x), (ONS $y)`

Table 3: Operators for the Tower of Hanoi

1. ((ONB $x) (ONB $y)),
2. ((ONB $x) (not ONB $x)),
3. ((not ONB $x) (not ONB $y) (not ONB $z)).

Our implementation of domain constraint checking iterates through the list of all domain constraints. For each constraint list, a test is made on the set of bulk preconditions using a unification algorithm. If the domain constraint unifies with a subset of the bulk preconditions, then a violation occurs. The process is repeated until a violation is found or until the list of domain constraints is exhausted. A detailed description of our implementation is provided in Appendix B.

Temporal coherence is a pruning heuristic that employs domain constraints. For a given plan, the set of bulk preconditions is first constructed. Then the algorithm is applied to the bulk preconditions. If any domain constraint is violated by the bulk preconditions, then the heuristic suggests that this plan be pruned from the search space.

The intuition of temporal coherence was explained in [4]. TWEAK is an incremental planner in that it adds one operator at a time. During each iteration, TWEAK selects a partially completed plan. It then generates a set of successor plans by adding a new operator or new constraints. For each of the successor plans, there may be a set of bulk preconditions that must be true in order for the partial plan developed so far to be successfully "executable." If so, then the bulk preconditions must satisfy the domain constraints. Thus, the domain constraints are used by TC as a sufficient condition to eliminate plans from the search space, while hopefully retaining at least one solution path to a goal.

In [4], Drummond and Currie presented a theorem showing the completeness of TWEAK with the application of TC. The theorem states that if there is a solution plan, then a temporally coherent search path exists in TWEAK's search space. Thus, if a breadth-first

Input: a plan state Π.
Output: a set of successor plans.

Algorithm Successor-Generation(Π)

```
1   Let $\mathcal{R} = \{(p, \alpha) \mid p$ is a preconditions of $\alpha\}$ be the set of pairs consisting of
    preconditions and operators in $\Pi$. Let $\mathcal{P}$ be a subset of $\mathcal{R}$;
2   For each pair $(p, \alpha)$ in $\mathcal{P}$,
       2.1 Let $\beta_j$, $j = 1, 2, \ldots, m$, be operators with an effect $p$, ;
       2.2 Let $n_j$ be a copy of plan $\Pi$ and construct an establishment relation $(\beta_j, \alpha, p)$ in $n_j$;
       2.3 For each plan $n_j$, $j = 1, 2, \ldots, m$,
          2.3.1 Find all conflicts $C_j$ with the establishment relation $(\beta_j, \alpha, p)$;
          2.3.2 Find all constraints $\{l_{j,k}, k = 1, 2, \ldots, u_j\}$, for resolving all the conflicts $C_j$;
          2.3.3 Impose each constraint set $l_{j,k}$ over a copy of $n_j$, obtaining $\Pi_{j,k}$;
3   Let *successors* be the set of all $\Pi_{j,k}$ as obtained in the previous step.
4   Return *filter-successors(successors)*;
```

Table 4: A successor generation algorithm.

search strategy is used, TWEAK will always find a solution while maintaining temporal coherence.

However, as we will demonstrate below, a critical assumption for their completeness theorem to hold is that the search space must be generated by a *particular* type of successor generation routine. Other than this particular implementation, there are many simpler ways of successor generation for which the application of TC destroys TWEAK's completeness. In the next section, we present this analysis.

4 Applying Temporal Coherence Heuristic to WatTweak

4.1 Successor Generation

Recall that in Algorithm WatTweak, the procedure `generate-successors` is left unspecified. In this section, we consider several ways of defining this procedure, and verify the completeness properties of the resulting algorithms when the temporal coherence heuristic is applied.

A generic procedure for the successor generation algorithm is shown in Table 4.1. Step 1 of the algorithm corresponds to a control point, where a decision has to be made as to what subset of pairs of precondition and operator should be selected. Once the selection is done, the preconditions will be subsequently achieved. Step 2 finds all operators β that can achieve a precondition p of an operator α, where (p, α) is a pair in subset $\mathcal{P}$ found in step 1. It then finds all conflicts with the new establishment relation (β, α, p), where each conflict consists of a clobbering operator γ with an effect q which can deny p. Step 2.3.2 finds constraints according to Chapman's MTC[2]. The constraints can be classified into three groups (of which the *white knight* constraint is a combination):

Promotion $\alpha \prec \gamma$,

Demotion $\gamma \prec \beta$,

Separation $p \not\approx \neg q$.

After the constraints are found, step 2.3.3 imposes these constraints, and obtains new successor plans. In step 4, the procedure *filter-successors* allows the application of temporal coherence to filter out some successors, before they are returned to the main planning algorithm. In particular, if temporal coherence is used, then procedure `filter-successors` will iterate through all successors, extracting the bulk preconditions of each successor plan and checking the domain constraints against all bulk preconditions. If a violation is detected for a successor plan, then the plan is removed from the successor list before it is returned.

Below, we enumerate the different ways of specifying step 1 of Algorithm `Successor-Generation`.

4.2 Without TC

We are interested in successor generation methods that give rise to complete planners. That is, if there is a solution, then one of them can be found. Without the application of TC, there is a spectrum of successor generation methods which are all complete. The spectrum depends on which set of preconditions are selected to be achieved next. The precondition set can include one precondition, all unsatisfied preconditions, or all bulk preconditions in the entire plan. We make the spectrum explicit below.

4.2.1 One-Unsat

For One-Unsat, a single pair consisting of an operator and one of its unsatisfied preconditions is arbitrarily chosen in step 1, and no filtering is done in step 4. That is, $\mathcal{P}$ is a singleton set $\{(p, \alpha)\}$, where p is a precondition of operator α that is not necessarily true. This version of WatTweak is complete; a rigorous proof is given in Appendix A. The intuition of the completeness proof is to consider the search tree generated by WatTweak with One-Unsat, and show that if a solution plan exists, then a path exists in this tree, such that

1. every node is a subset of the solution plan, and

2. the number of operators and constraints in each node monotonically increases along the path.

4.2.2 All-Unsat

An alternative successor generation method is to generate the successors based on *all* preconditions that are not necessarily true. This corresponds to setting $\mathcal{P}$ to be all pairs of (p, α) where p is a precondition of operator α that is not necessarily true. With this method, in step 2 of procedure `successor-generation`, all establishers for all unsatisfied preconditions are found. This second method of successor generation is called All-Unsat.

The All-Unsat method is clearly complete, because the set of successors generated by All-Unsat is a superset of that by One-Unsat and the latter has been shown to be complete.

4.2.3 All-Bulk

Next on the spectrum is a method based on *all* bulk preconditions. This is referred to as All-Bulk. All-Bulk is complete, since its successor set is a superset of All-Unsat, and the latter is complete.

Notice that for both All-Unsat and All-Bulk methods the branching factors of search may be much larger than One-Unsat. This is because while One-Unsat's branching factor depends only on the number of ways to achieve one precondition or goal, All-Unsat and All-Bulk , in the worst case, must multiply this by the total number of preconditions and goals in the entire plan.

4.3 With TC

We now consider the spectrum again with the application of TC. To facilitate our discussion, we do this by enumerating the entire spectrum in the opposite direction from the above. That is, we go from All-Bulk to One-Unsat.

4.3.1 TC-Bulk

The application of TC to procedure `filter-successors` eliminates temporally incoherent plans. We refer to All-Bulk with TC as TC-Bulk. Proof of the completeness of TC-Bulk has been given by Drummond and Currie[4]. Although this particular successor generation method was not made explicit in their papers, this was the one implied in their entire discussion of the completeness proof. To see this, consider an informal discussion of their proof as follows.

The basis of their completeness proof is essentially that if a plan exists, then a temporally coherent path for finding that plan must also exist in the space generated by the set of all bulk preconditions. For example, suppose that the plan is a sequence of operators $a_i, i = 0, 1, 2, ...n, n+1$, in which $a_0 = \mathcal{I}$ and $a_{n+1} = \mathcal{G}$. Then between any pair of operators in this plan, a_i and a_{i+1} the agent must find itself in a temporally coherent state. Thus, if one traces this sequence backwards from the goal, adding one operator to the inverse subplan at a time, one can always be guaranteed to be in temporally coherent states until the whole plan is reconstructed. Moreover, if a solution plan exists for a planning problem, then in the search tree generated by the All-Bulk method there is a search path that corresponds exactly to this solution plan. Thus, if a solution exists, then a temporally consistent path in All-Bulk's search tree also exists.

4.3.2 All-Unsat with TC

With All-Unsat, successors are only generated based on all unsatisfied preconditions in a plan. If TC is applied to filter successor plans, then All-Unsat becomes incomplete. To see this, consider the following example from the blocks world domain.

For this problem, the initial state is (ON B Table), (ON C Table), (ON A B), (CLEAR A), and (CLEAR C) and the goal state is (ON A B) and (ON B C) as shown in Figure 2.

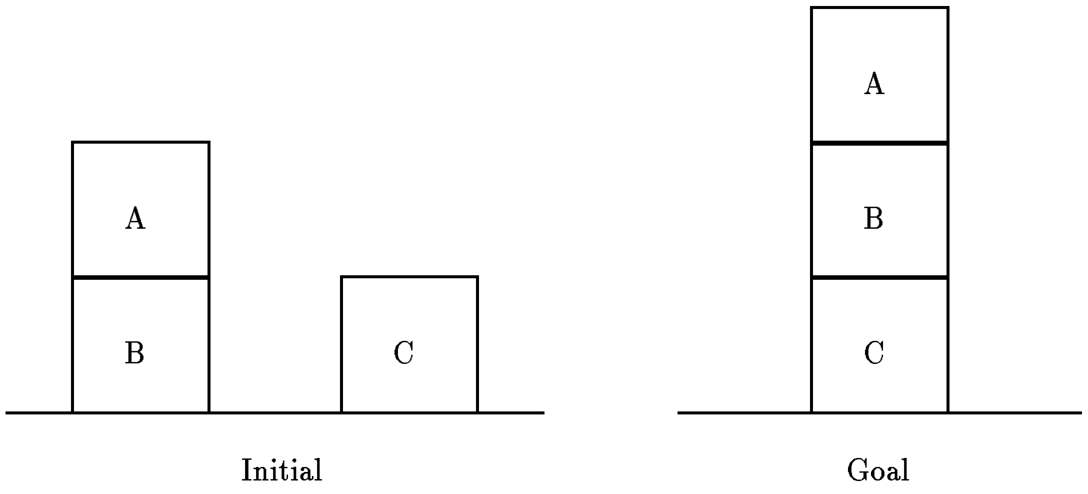

Figure 2: A blocks world example.

When the planner begins generating successors, there is only one unsatisfied precondition (ON B C), since (ON A B) is satisfied by the initial state. Thus the planner adds the operator (MOVE B to C from $X) to achieve this precondition. The bulk preconditions are now

(IsBlock A), (IsBlock B), (ON A B), (CLEAR B), (CLEAR C) and (ON B $X).

The plan after this operator is added is shown in Figure 3. This plan contains a violation of the domain constraints because B cannot be both clear and under A. There are no other possible ways to achieve (ON B C) so the planner terminates without finding a solution.

The incompleteness problem is not unique to the blocks world domain. The next Tower of Hanoi example illustrates the same incompleteness problem.

Consider a variation of the Tower of Hanoi domain in which the medium and large disks are initially located on peg 1 and the small disk is on peg 3. Suppose that the goal state is G = ((ONS P3) (ONM P3)), representing that the small and medium disks are both on peg 3.

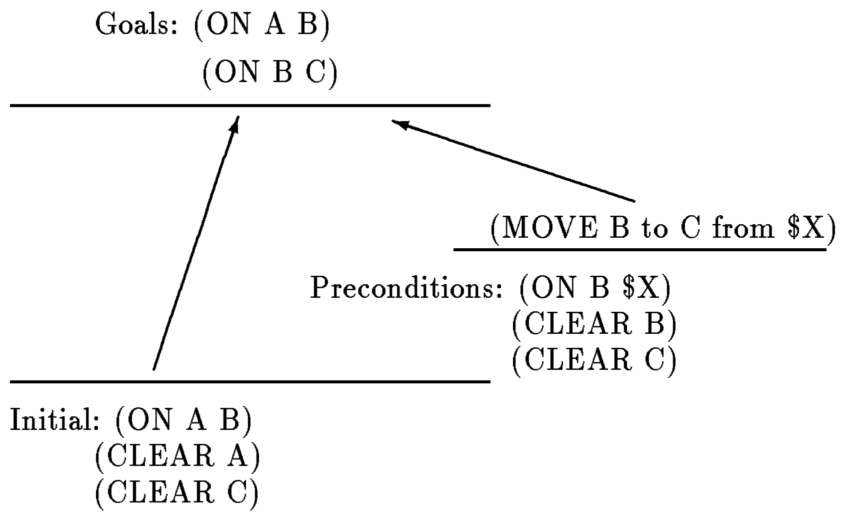

Figure 3: A blocks world plan.

Since the subgoal (ONS P3) is already true in the initial state, the planner selects the other goal (ONM P3) to be achieved first. The only operator it can add to the initial plan is (MOVEM $X P3). This operator has the following preconditions,

(Ispeg $X) (Ispeg P3) (ONM $X) (NOT ONS $X) (NOT ONS P3).

The bulk preconditions now include these preconditions as well as a goal, (ONS P3). We now have a domain constraint violation because a disk cannot be on a peg and not on a peg at the same time. Since this is the only successor generated, the planner terminates and no solution is found.

The above two examples show that TC-Bulk is incomplete. The reason behind its incompleteness can be understood as follows. In each case, there is a goal condition g which is already true in the initial state (g is (ON A B) in blocks world and (ONS P3) in Tower of Hanoi example). However, to achieve all other goal conditions, g must be temporarily violated and then restored. In the search space, this solution plan corresponds to the only search path from the root node to a goal node on which every node is temporally coherent. All other paths will eventually lead to a violation of TC. Since All-Unsat does not generate a search path that achieves a precondition already established by the initial state, the temporally coherent path is never generated. Therefore, no solution plan could be found.

A corollary of this result is that no successor generation method based on any subset of All-Unsat is complete either. This is because, if the search space of a planner does not contain a solution path, then no subset of the search space contains any solution. Thus, the corollary implies that One-Unsat wth TC is also incomplete.

Method:	**One-Unsat**	... **Some-Unsat** ...	**All-Unsat**	... **All-Unsat+** ...	**All-Bulk**
W/out TC	Complete	Complete	Complete	Complete	Complete
With TC	Incomplete	Incomplete	Incomplete	?	Complete

Table 5: A spectrum of successor generation methods.

4.4 Summary

So far, we have investigated several successor generation schemes along a spectrum of successor generation methods, and analyzed their completeness properties when the temporal coherence heuristic is applied. On one end of the spectrum is One-Unsat, which generates successor plans based on only one unsatisfied precondition. On the other end is All-Bulk, which generates successor plans based on the set of all bulk preconditions in the current plan. Somewhere in the middle of the spectrum is All-Unsat, which generates successor plans based on the set of all unsatisfied preconditions. Between One-Unsat and All-Unsat are methods based on a subset of the currently unsatisfied preconditions in a plan. These methods can be called *Some-Unsat*. Finally, between All-Unsat and All-Bulk are methods that are based on All-Unsat as well as a subset of preconditions established solely by the initial state. We call these methods All-Unsat+. The spectrum is shown graphically in Table 5. With respect to the spectrum, our analysis shows that when TC is applied no successor generation method between One-Unsat and All-Unsat inclusive is complete. On the other hand, All-Bulk with TC, *i.e.* TC-Bulk, is complete.

The question remains whether any successor generation methods between All-Unsat and All-Bulk on this spectrum would be complete with TC. These methods correspond to All-Unsat+. Recall that the only difference between All-Bulk and All-Unsat is that the latter does not generate successors for preconditions established in the initial state, while the former does. Thus, any middle point on the spectrum between these two methods is necessarily based on All-Bulk plus a subset of preconditions true in the initial state. A choice of a subset of preconditions established by the initial state must partition the set of initial state facts into distinct subsets. Only some of these subsets are used for successor generation, while others are not. We conjecture that any such choice that will make the resulting successor generation method complete with TC, is *domain-dependent* in nature. Moreover, even with a domain-dependent choice of subset, we do not expect the resulting planner to show a quantitative difference in performance from TC-Bulk. Thus, it is likely that TC-Bulk is effectively the only domain-independent method for successor generation with TC.

To conclude, without temporal coherence One-Unsat is complete, but with temporal coherence All-Bulk is likely to be the only domain-independent and complete method for successor generation. Because of this, and because of the fact that TC-Bulk was the successor generation method underlying the introduction of temporal coherence (see [4], page 346, proof of theorem 1), the evaluation of TC can be done via an evaluation of TC-Bulk. In particular, we would like to know whether TC-Bulk will always outperform One-Unsat. If not, when can it do so? This is the main issue we attempt to address next with an asymptotic analysis and empirical tests.

5 Analyzing the Utility of TC

5.1 Analytical Framework

Above we have shown that without additional domain knowledge, TC-Bulk is likely to be the only successor generation method for WatTweak to be complete with the application of temporal coherence. On the other hand, One-Unsat is complete without using any heuristic at all. Thus, to determine when it is worthwhile to apply the temporal coherence heuristic to WatTweak, we need to find out the conditions under which TC-Bulk can outperform One-Unsat. In this section, we perform an asymptotic analysis comparing the running times of TC-Bulk and One-Unsat.

For the ease of analysis, we assume that a breadth-first search method is used for planning. Search is then guided by the number of operators in each plan. Under this search strategy, the average planning cost is determined by the branching factor and depth of the search tree, as well as the amount of time spent on each plan in the search tree.

Let B be the number of ways to achieve a precondition or goal by One-Unsat. In our successor generation algorithm, this is the number of successor plans returned by Step 3 when Step 1 finds just one precondition-operator pair. Let D be the number of operators in the optimal plan for a planning problem. Since a breadth first search is used, D is also the depth of search using either One-Unsat or TC-Bulk as successor generation methods. Let T_{OU} be the average amount of time spent by One-Unsat to execute the successor generation algorithm. Then the average planning time by WatTweak using One-Unsat is

$$O(B^D * T_{OU}).$$

For TC-Bulk, let N_B be the average number of bulk preconditions in one plan. If no filtering is done using temporal coherence, then the successor generation algorithm is simply All-Bulk. On the average, the branching factor for All-Bulk is $B * N_B$, since B successor plans must be generated for each bulk precondition. With temporal coherence, a number of successor plans may be pruned. Let S be the percentage of remaining successor plans after pruning by TC. We can use S to denote the *strength* of domain constraints. The domain constraints get stronger as the value of S approaches zero. Then $B * N_B * S$ is the average branching factor of TC-Bulk. Finally, let T_{TC} be the average amount of time spent by TC-Bulk on one pass of the successor generation algorithm. Then the average planning time by WatTweak using TC-Bulk is

$$T_{TC} * (B * N_B * S)^D.$$

The notations defined in this section are summarized in Table 6.

Notations	
D	The depth of search using either One-Unsat or TC-Bulk.
B	The number of ways to achieve each precondition by One-Unsat.
T_{OU}	The average amount of time spent by One-Unsat for successor generation.
T_{TC}	The average amount of time spent by TC-Bulk for successor generation.
N_B	The average number of bulk preconditions in one plan.
S	The strength of domain constraints.

Table 6: Notations used in the analysis.

From the two formulas above, TC-Bulk will outperform One-Unsat when the following relationship among the various factors holds:

$$S < (T_{OU}/T_{TC})^{1/D}/N_B.$$

5.2 Analysis

The relative performance of the two successor generation methods depends mainly on four factors. The first factor is S, which represents the *strength* of domain constraints. For different domains, S can vary from weak to strong. In the weakest extreme no successor plan is pruned. In this case $S = 100\%$ and TC-Bulk will be equivalent to All-Bulk. With the strongest possible domain constraints, every successor plan is pruned and S is 0%. In general, the value of S is between 0 and 1, and the smaller the value of S, the stronger the domain constraints are.

The second factor is the amount of time each algorithm spent on a plan expansion. The time taken by One-Unsat, T_{OU}, is always a lower bound of T_{TC}. This is because for any given plan, the successor plans generated by One-Unsat is always a subset of All-Bulk. Moreover, TC-Bulk takes extra time to filter through the successors using domain constraints. Thus, One-Unsat takes no more time per node than TC-Bulk.

The third main factor determining the relative performance is N_B, the average number of bulk preconditions in a plan. The specific value of N_B in turn depends on a number of factors. First, if the number of preconditions for each operator is large, then during search the number of unachieved preconditions is large. Thus, N_B is likely to be large also. Similarly N_B may be large for a planning problem with a large number of goals. Second, if in the solution plan to a planning problem, a large number of preconditions or goals are established by the initial state facts, then by the definition of bulk preconditions, N_B is expected to be large. Third, if the number of operators in a solution plan is large, then the number of bulk preconditions is also likely to be large. This implies that for a given planning problem, N_B is likely to increase as the search tree gets deeper.

The fourth main factor is the depth D of the search tree. If breadth first search is used for WatTweak and if search is guided by the number of operators in a plan, then D is the total number of operators in the plan.

Let C be $(T_{OU}/T_{TC})^{1/D}/N_B$. Recall that TC-Bulk will outperform One-Unsat when $S < C$. This condition places a lower bound limit on the strength of the domain constraints. Intuitively, the formula states that for TC-Bulk to outperform One-Unsat, the domain constraints must be so strong that the remaining percentage of successors after pruning must be less than C. For this reason, C is called the *critical bound* on the strength of domain constraints.

Based on the formula we can consider the asymptotic behaviour of the time complexity as D approaches infinity. In this extreme case, the formula tends to $S < 1/N_B$. Intuitively, this means that for a very large plan, TC-Bulk is expected to outperform One-Unsat if the domain constraints are so strong that successor plans for all but one bulk preconditions are pruned.

From the above discussions, we can predict factors affecting the performance of TC-Bulk as compared to One-Unsat. TC-Bulk will perform better than One-Unsat when the strength value S is small and the critical bound value C is large. Since C depends inversely on T_{TC}, D and N_B, this happens when they all have small values. Therefore, TC-Bulk is preferred when the following conditions hold:

1. The domain constraints are strong. That is, the value of S is very small. This enables TC-Bulk to prune a large portion of the successor plans.

2. The number of preconditions of operators and the number of goals are small. This condition will reduce the value of N_B.

3. Few preconditions of operators in a plan can be established by the initial state facts. This condition will also reduce the value of N_B.

4. The solution plans have small sizes. Thus, D has a small value.

5. The domain constraints are inexpensive to check. This reduces the time spent by TC-Bulk on each plan.

The hypotheses are supported by our analyses above. In the next section, we present experiments to empirically verify the conditions.

6 Empirical Comparisons

To further support our predictions listed above, we have run experiments that show the effects of various factors on the efficiency of planning. All routines are coded in Allegro Common Lisp and run on a Sun4/sparc station. For fair comparison, we have used the breadth-first strategy for controlling search in WatTweak. In addition, our choice of which precondition/operator pair to be achieved next is based on the order in which operators are inserted into the plan; we always choose a newest subgoal from the unsatisfied or bulk preconditions.

(drive $r $loci $locj)			
Preconditions	(AT $r $loci).	Effects	(AT $r $locj) (not AT $r $loci) .

(pickup $r Money)			
Preconditions	(AT $r I) (AT Money I)	Effects	(Holding $r Money).

Table 7: Operator definition for the robot domain.

6.1 The Strength of Domain Constraints

The first set of tests are designed to explore the effect of S on the efficiency of planning. We expect that the stronger the constraints, the better the performance of TC-Bulk. When the domain constraints are strong enough TC-Bulk can outperform One-Unsat.

Our domain consists of a robot who can travel between cities. Initially the robot is located at the city `I`. The goal is to reach city `G` while holding money. To do so, the robot has to first pick up the money at `I`, then find a route to go from the initial city `I` to the destination city `G`. Between the cities I and G there is a four by four matrix of intermediate cities. Each city c_{ij}, $1 \leq i < 4, 1 \leq j \leq 4$, is connected to all adjacent cities $C_{i+1,k}$, $k = 1, 2, 3, 4$.

To move from one city to the next, the robot can use an operator `drive`. To get the money, the robot can use an operator `pickup`. The definitions of `drive` and `pickup` is given in Table 7.

The initial state is: Initial = ((AT R1 I) (AT Money I) (not Holding R1 MONEY))

The goal state is: Goal =((At R1 G) (Holding R1 Money))

An example plan of this domain is:

$$pickup(R1, Money); drive(R1, I, c_{11}); drive(R1, c_{11}, c_{21}); \ldots; drive(R1, c_{41}, G).$$

To test the effect of S on planning efficiency, we have designed four sets of domain constraints with varying degrees of strength. The weakest constraint is dc_0, which simply states that the robot R1 cannot be at two different cities at the same time. The second strongest is dc_1, which in addition to the requirement by dc_0, prunes any plan whose bulk preconditions contain $At(c_{i4})$ *and not* $Holding(R1, Money)$. That is, the robot can enter a city c_{i4} only when it is holding money. The third strongest constraint dc_2 further disallows a robot to enter a city on the third and fourth column without first holding money. The strongest constraint dc_3 prohibits the robot from entering any city except those on the first column without first holding the money.

The travel domain has the following characteristics:

1. The domain constraints are inexpensive to test. The computation involves only simple pattern matching that on no more than two literals at a time.

Planner	DC	Expanded Nodes	CPU Seconds	Branching
One-Unsat	N/A	343	23.6	4
TC-Bulk	dc_0	470	205.3	4
	dc_1	176	95.6	3
	dc_2	48	22.1	2
	dc_3	8	3.6	1

Table 8: Comparing One-Unsat with TC-Bulk in the robot domain.

2. The number of preconditions for each operator is small. In fact, the number is one for the drive operator. In the next section, we test the effect of increasing the number of preconditions on search efficiency.

3. The majority of the operators do not depend on the initial facts.

4. The number of inserted operators in the optimal solution D is six, a relatively small number

From the last two characteristics, we know that the number of bulk-preconditions N_B is no more than two. These properties ensure that the effect of domain constraint on search efficiency is isolated. Thus, with the varying strength of domain constraints, we can better determine the change in planning cost.

The CPU-time comparison of One-Bulk and TC-Bulk is shown in Table 8. As seen from the table, the CPU Time performance of TC-Bulk improves as the constraints get stronger. A direct effect of strong constraints is the decrease in branching factor of search. We also see that TC-Bulk is worse than One-Unsat in three out of four cases. TC-Bulk outperforms One-Unsat only under the strongest constraint dc_4. An interesting case is the data under the constraint dc_1, where TC-Bulk can actually have a smaller effective branching factor and need to search fewer nodes. But since it spends an extra amount of time per node, the cumulative CPU time is still worse than One-Unsat.

6.2 The Effect of Operator Preconditions

Based on our asymptotic analysis, we predicted earlier that if the number of preconditions for each operator is large then N_B is likely to be large also. This in turn will degrade the performance of TC-Bulk. To test this prediction, we augmented the above travel domain as follows. For each `drive` operator that connects cities c_{ij} to $c_{i+1,k}$ we added the following extra preconditions: $\{p1, p2, \ldots, pl\}$, where $l = 1, 2, \ldots$ is an integer value that can be varied. These literals are also added to the effects of the pickup operator.

The test results for each set additional preconditions are shown in Table 9. For all the tests in this section, the strongest domain constraint dc_3 is used. The solution length is

	TC-BULK		One-Unsat	
Preconditions	Expanded Nodes	CPU Seconds	Expanded Nodes	CPU Seconds
(p1)	58	133.3	576	175.3
(p1)(p2)	154	388.3	1050	193.7
(p1) (p2) (p3)	298	991.9	1050	266.9
(p1) (p2) (p3) (p4)		over 1200	1050	329.4
(p1) (p2) (p3) (p4) (p5)		over 1200	1050	435.7

Table 9: Comparing One-Unsat with TC-Bulk with changing preconditions. A CPU time limit of 1200 seconds is imposed.

again six for all problems. As can be seen from the table, as the number of preconditions increases, the cost of using TC-Bulk increases much faster than One-Unsat. In all but the first cases, One-Unsat is clearly more efficient than TC-Bulk in terms of CPU time. This is also expected from our analysis, for the number of preconditions is a major contributing factor to the value of N_B.

The above augmentation to the test domain used the `pickup` operator to achieve all added preconditions of the subsequent operators in a solution plan. The next test adds all additional preconditions to the initial state, and uses the original `pickup` operator. This modification verifies the prediction that if the number of preconditions which depend on the initial state facts increases in a solution plan, the complexity of TC-Bulk will increase as well. The test results are shown in Table 10, where each row presents the data corresponding to the addition of one more precondition to each `drive` operator and to the initial state. As can be seen from the table, One-Unsat exhibits an almost linear CPU cost behaviour. In contrast, TC-Bulk again approaches the CPU time limit of 1200 seconds in an exponential manner.

6.3 Two Traditional Planning Domains

The above tests used a somewhat idealistic domain, where the number of preconditions and domain constraints can be easily controlled. In this section, we present test results in two traditional domains, the blocks world domain and the Tower of Hanoi domain. In both domains, we used completely instantiated operators where all variables are replaced by constants. This helps control the explosion problem of TC-Bulk and enables the test to be performed with relative ease.

We choose the Sussman's anomaly problem for the blocks world test, which involves the movement of three blocks: A, B and C. For this problem, the initial state is one in which the block C is on A, and both A and B are on the table. The goal is to build a tower with A on B and B on C. The Tower of Hanoi problem has the standard initial and goal states,

	TC-BULK		One-Unsat	
Preconditions	Expanded Nodes	CPU Seconds	Expanded Nodes	CPU Seconds
(p1)	163	108.8	343	34.2
(p1)(p2)	882	972.4	343	38.4
(p1) (p2) (p3)		over 1200	343	40.0
(p1) (p2) (p3) (p4)		over 1200	343	43.9
(p1) (p2) (p3) (p4) (p5)		over 1200	343	47.1

Table 10: Comparing One-Unsat with TC-Bulk with changing preconditions. The additional preconditions are achieved by the initial state. A CPU time limit of 1200 seconds is imposed.

Planner	Sol Len	States Expanded	States Generated	CPU Sec	Branching	N_B
Sussman's Anomaly						
TC-bulk	3	Over 199	Over 5567	1200	28	11.5
One-Unsat	3	8	23	1.6	2.9	
Tower of Hanoi						
TC-bulk	7	Over 300	Over 3157	1200	10.5	5.6
One-Unsat	7	322	899	136.3	2.8	

Table 11: Comparing One-Unsat with TC-Bulk on a blocks world problem and the Tower of Hanoi problem.

with the objective of moving all three disks from peg 1 to peg 3. The domain constraints for both domains are listed in Appendix B.

The test results of both problems are shown in Table 11. As can be seen from the table, TC-Bulk performed much worse than One-Unsat. To analyze the reason behind the performance difference, we also recorded the average branching factors and number of bulk preconditions in each case. It is clear that for these two problems, the branching factors for TC-Bulk are 5 to 10 times higher than One-Unsat. This can be explained by the large number bulk preconditions in each case.

As discussed earlier, for TC-Bulk to perform efficiently, the domain constraints must be sufficiently strong. It seems that in these two domains, the domain constraints are not strong enough to reduce the branching factors below that of One-Unsat. To see this conclusion more precisely, we computed the critical bound on S and S itself for each test problem (see Table 12). As can be observed, the actual value of S is much higher than required by the critical bound C.

Test Problem	$C =$ **Bound on** S	S
Sussman's Anomaly	$3.1 * 10^{-6}$	0.2
Tower of Hanoi	$2.6 * 10^{-8}$	0.3

Table 12: Strengths of Domain Constraints.

	TC-Bulk			One-Unsat		
Sol Len	Nodes	CPU	Branching	Nodes	CPU	Branching
0	0	0.1	0	0	0.03	0
1	2	0.4	5	2	0.1	2
2	6	4.4	5.3	3	0.2	2
3	99.5	180.6	6.7	8.8	0.9	2.4
4	145	600	7.4	20	4.5	3

Table 13: Tower of Hanoi results.

Table 13 displays the comparison results with increasing solution lengths, on eight different configurations of initial and goal states from the Tower of Hanoi domain. These problems include all variations of the domain for which the solution lengths are no more than four. For problems with solution lengths greater than four TC-Bulk will never finish within a time limit of 600 CPU seconds. It can be seen that for these problems One-Unsat is more efficient than TC-Bulk, and as the solution length D increases, the difference between the performance of TC-Bulk and One-Unsat also widens. This observation supports our analytical result that TC-Bulk is likely to perform worse when the search depth D is large.

7 Conclusions

In this paper we have examined the temporal coherence heuristic in the context of partial-order planning. We have analyzed the heuristic along the dimensions of completeness and efficiency. Our main conclusions are summarized below:

1. The completeness of planning with the application of temporal coherence depends critically on how the successor generation routines are implemented. We have examined a spectrum of successor generation methods. Each point on the spectrum is determined by a subset of operator preconditions used for generating successor plans. On this spectrum, All-Bulk is likely to be the only domain-independent method for which the application of temporal coherence will result in a complete planner.

2. The application of TC to WatTweak does not always result in a more efficient planner. Among the cases that we have tested, TC-Bulk can be ten times worse than One-Unsat.

3. Our analytical comparison of TC-Bulk and One-Unsat shows that for TC-Bulk to be more efficient, a number of conditions must be satisfied by the planning domain. In particular, for TC-Bulk to be more efficient, the domain constraints must be strong, the number of preconditions for each operator must not be too large, few preconditions depend on the initial state facts, the solution plan has a small size, and the domain constraints themselves must be inexpensive to test. These results are further confirmed by our empirical tests.

Acknowledgement

The authors would like to thank Mark Drummond for many motivating discussions. The authors also greatly appreciate many excellent comments made by the referees of the Computational Intelligence Journal.

References

[1] Anthony Barrett and Dan Weld. 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. 41 pages.

[2] David Chapman. 1987. Planning for conjunctive goals. *Artificial Intelligence*. Elsevier, Amsterdam. 32:333–377.

[3] Ken Currie and Austin Tate. 1992. O-plan: the open planning architecture. *Artificial Intelligence*. Elsevier, Amsterdam. 52(1):49–86.

[4] Mark Drummond and Ken Currie. 1988. Exploiting temporal coherence in nonlinear plan construction. *Computational Intelligence*. National Research Council of Canada, Ottawa, Ontario, Canada. 4(2):341–348.

[5] Mark Drummond and Ken Currie. 1989. Goal ordering in partially ordered plans. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers, Inc., San Mateo, CA. Pages 960–965.

[6] Oren Etzioni, Steve Hanks, Danial Weld, Denise Draper, Neal Lesh, and Mike Williamson. 1992 An approach to planning with incomplete information. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann Publishers, Inc., San Mateo, CA. Pages 115–125.

[7] Subbarao Kambhampati. 1992. Characterizing multi-contributor causal structures for planning. In *Proceedings of the First International Conference on AI Planning Systems*. Morgan Kaufmann Publishers, Inc., San Mateo, CA. Pages 116–125.

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

[9] Nils Nilsson. 1980. *Principles of Artificial Intelligence.* Morgan Kaufmann Publishers, Inc., San Mateo, CA. 476 pages.

[10] Austin Tate. 1977. Generating project networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-77).* Morgan Kaufmann Publishers, Inc. San Mateo, CA. Pages 888–893.

[11] Austin Tate, James Hendler, and Mark Drummond. 1990. A review of AI planning techniques. In *Readings in Planning*, Morgan Kaufmann Publishers, Inc., San Mateo, CA. Pages 26–49.

[12] David Wilkins. 1988. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann Publishers, Inc., San Mateo, CA. 205 pages.

[13] Qiang Yang. 1990. An algebraic approach to conflict resolution in planning. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90).* Morgan Kaufmann Publishers, Inc., San Mateo, CA. Pages 40–45.

[14] Qiang Yang, Josh Tenenberg, and Steve Woods. 1991. Abstraction in nonlinear planning. Technical Report CS 91-65, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada N2L 3G1. 50 pages.

A Completeness of One-Unsat

In this section, we prove that One-Unsat is complete. That is, if there is a solution to a planning problem, then one of them can be found by One-Unsat.

We prove by induction that if a solution plan Π exists, then there exists a path in the search tree of One-Unsat, such that the following conditions hold:

1. For every node n on that path, $n \subseteq \Pi$, and

2. For every node n_{i-1} and its successor n_i on that path, $n_{i-1} \subset n_i$.

Condition 1 states that every node on that path is a subset of the solution plan Π. Condition 2 states that the size of a node monotonically increases along the path from the root. Since the number of elements in the solution plan Π is finite, clearly, if these two conditions hold, the path will eventually stop at the node containing Π itself.

We now inductively prove conditions 1 and 2 on level i of the search tree, where $i = 0, 1, 2, \ldots$. For the base case, let $i = 0$. The root node R of One-Unsat's search tree contains two operators, $\mathcal{I}$ and $\mathcal{G}$. The only constraint is an ordering from $\mathcal{I}$ to $\mathcal{G}$. By definition, R is a subset of every plan. Therefore, $R \subseteq \Pi$ and condition 1 holds for R. Also, R has no predecessor. Thus, condition 2 trivially holds.

For the inductive assumption, suppose that both conditions 1 and 2 hold for every node on a path up to depth k. Let n_k be a node at depth k of the path. If n_k is a solution plan, then the theorem holds trivially. If n_k is not a solution plan, then there must be some precondition of an operator α such that p does not necessarily hold just before α. The successor generation routine of One-Unsat will find all establishing operators $\beta_j, j = 1, 2, \ldots, m$ for achieving p. It then generates m copies of plan n_k, $n_1, \ldots, n_m$. For each copy n_j, it inserts the corresponding operator β_j, and imposes all constraints to remove conflicts with establishing relation (β_j, α, p) in n_j. Again all ways of removing conflicts are considered, each giving rise to a copy $\Pi_{j,l}$ of n_j, $l = 1, 2, \ldots, L_j$, in which all conflicts with the relation (β_j, α, p) are removed. The set of final successors of this step is:

$$\{\Pi_{j,l} \mid j = 1, 2, \ldots, m, \ \mathit{and}\ l = 1, 2, \ldots, L_j\}.$$

These are the successors of the node generated by ONE-UNSAT.

Recall that p is necessarily true in Π, just before α. Thus, there must be an establisher β of p in Π, and no conflicts. Since the one-step successor generation process described above considers all ways of achieving p just before α in n_k, β must be one of the establishers β_j. Furthermore, the constraints imposed on n_j remove conflicts from existing operators, by either moving the clobbering operators before β, after α, or force the conflicting effect of the clobbering operator to bind to different constants as p. Since one of the three choices in constraints must hold in the solution plan Π, these constraints must also be members of Π. In other words, there exists a successor $\Pi_{j,l}$ which is a subset of Π. This proves condition 1.

Since an extra declobbering step is done for generating successors, using existing or new establishers, each new successor $\Pi_{j,l}$ must have more constraints imposed on it as compared to n_k. Thus, condition 2 also holds. As a result, for some path of the search tree of One-Unsat, conditions 1 and 2 must both hold for every node.

B Domain Constraints for Empirical Tests

B.1 Implementation

Algorithm DC shows our implementation used for checking domain constraints. The constraints are contained in the global variable *domain-constraints*. The function, check-all-dom-constr, calls the function check-one-dom-constr for each constraint in the domain constraint list until a violation is found or until the list of domain constraints is exhausted. The value "True" is returned if a violation is found.

As specified in [4], different variable names in a domain constraint must map to different objects. This is implemented as follows. Each domain constraint consists of a list of assertions followed by a list of variable pairs. Each pair of variables in the list implies that the variables must be necessarily noncodesignating in order for the matching to be true. For example, the domain constraint for specifying that a block cannot be on two different things at the same time is (((ON \$1 \$2) (ON \$1 \$3)) ((\$2 \$3))), which is true when the two literals unify with the conditions in the bulk preconditions, and when the variables \$2 and \$3 map to different constants or variables. Similarly, the following domain constraint states that two blocks cannot be on the same thing, unless that thing is the table: (((ON \$1 \$2) (\$3 \$2)) ((\$1 \$3) (\$2 TABLE))). In this case, \$1 must necessarily not codesignate with \$3, and \$2 must necessarily not codesignate with the table.

The function check-one-dom-constr checks all possible matchings between a domain constraint and the list of bulk preconditions to determine if a violation exists. After it finds a possible violation it verifies (line 12) that it really is a violation by checking the list of necessarily non-codesignating variables. Check-one-dom-constr returns true if a violation is found.

Algorithm DC

```
1   check-all-dom-constr(bulk-precond)
2       dc-list = *domain-constraints*;
3       violation = false;
4       while (notempty(dc-list) and not violation)
5           violation = check-one-dom-constr (head(dc-list), bulk-precond, nil);
6           dc-list = tail(dc-list);
7       endwhile
8       return violation;
9   end

10  check-one-dom-constr(dc-list, bulk-precond, map-list)
```

```
        if (empty(dc-list)) then
            if map-list is a valid mapping then
                return true;
            else
                return false;
            endif
        else
            violation = false;
            pre-list = bulk-precond;
            while (notempty(pre-list) and not violation)
                if (head(prelist) is unifiable with head(dc-list) with substitution mapping) then
                    apply mapping to dc-list;
                    add mapping to map-list;
                    violation = check-one-dom-constr(tail(dc-list), bulk-precond, map-list);
                else
                    pre-list = tail(pre-list);
                endif
            endwhile
        endif
    end
```

B.2 Domain Constraints in the Robot Travel Domain

```
 Domain Constraints:

(setq *domain-constraints*
      '(
(((at $r $x)(at $r $y)) (($x $y)))
(((at $r $x)(not at $r $x)) nil)
))
```

B.3 Tower of Hanoi Domain

```
Domain Constraints:

 (setq *domain-constraints*
      '((((onb $x) (onb $y)) (($x $y)))
        (((onm $x) (onm $y)) (($x $y)))
        (((ons $x) (ons $y)) (($x $y)))
        (((ons $x) (not ons $x)) nil)
        (((onm $x) (not onm $x)) nil)
        (((onb $x) (not onb $x)) nil)
        (((not ons $x)(not ons $y)(not ons $z)) (($x $y)($y $z)($x $z)))
        (((not onm $x)(not onm $y)(not onm $z)) (($x $y)($y $z)($x $z)))
        (((not onb $x)(not onb $y)(not onb $z)) (($x $y)($y $z)($x $z)))))
```

B.4 Blocks World

```
Domain Constraints:

(setq *domain-constraints*
        '((((on $x $y) (clear $y)) (($y table)))
          (((on $x $y) (on $x $z)) (($y $z)))
          (((on $x $y) (on $z $y)) (($x $z) ($y table)))
          (((on $x $x)           ) nil)
          (((on $x $y) (on $y $x)) nil)))
```

List of Symbols

Symbol	**Meaning**
α, β, γ	Operators
Π	Plan
$\rightarrow$	precedence relation
$\approx$	codesignation constraint
$\not\approx$	noncodesignation constraint
$\mathcal{P}$	a set of pairs consisting of preconditions and operators.
D	The depth of search using either One-Unsat or TC-Bulk
B	The number of ways to achieve each precondition by One-Unsat.
T_{OU}	The average amount of time spent by One-Unsat for successor generation
T_{TC}	The average amount of time spent by TC-Bulk for successor generation
N_B	The average number of bulk preconditions in one plan.
S	The strength of domain constraints.

List of Tables

List of Figures