A Comparison of the SNLP and TWEAK Planning Algorithms*

Craig A. Knoblock
Information Sciences Institute
& Computer Science Department
University of Southern California
Marina del Rey, CA 90292
knoblock@isi.edu

Qiang Yang
University of Waterloo
Computer Science Department
Waterloo, Ont., Canada N2L 3G1
qyang@logos.uwaterloo.ca

Abstract

Most current partial-order planning systems are based on either the TWEAK or SNLP planning algorithms. Both planning algorithms are complete and correct. The SNLP algorithm distinguishes itself from TWEAK in that it is also *systematic*, so it never generates redundant plans in its search space. This paper compares the two planning algorithms and shows that SNLP's systematicity property does not imply that the planner is more efficient than TWEAK. To compare the two systems, we review the SNLP algorithm and describe how it can be easily transformed into TWEAK. Then we present a complexity analysis of each system and identify the factors that determine the performance of the systems. Finally, we present results on a set of classic planning problems, which show that the relative performance of the two systems depends on the characteristics of the problems being solved.

1 Introduction

There has been a great deal of work recently on comparing total and partial order planning systems [1, 6], but little has been done in comparing different partial order planners themselves. Partial order planners largely fall into two classes: the TWEAK type [3, 6, 9], or the SNLP type [1, 4, 5]. In this paper, we present our results in rigorously comparing the efficiency of SNLP and TWEAK.

On the surface, the two planners are quite different. However, on a careful examination one finds that they mainly differ in whether or not they protect the conditions achieved. During planning, an inserted plan step is likely to cause interactions with previously inserted steps. If a goal is achieved by one plan step, then later it could be *threatened* by other steps. A step is *protected* by removing the threats by imposing constraints on a plan. Between the two planners, TWEAK protects nothing. Thus, a given condition can be achieved more than once. SNLP, on the other hand, protects everything. This enables it to carefully partition the search space into non-overlapping parts.

The use of goal protection in SNLP prevents the planner from generating redundant plans and thereby reduces the size of the search space, however, enforcing the goal protection has a cost. In this paper, we show that neither of the two planners is always a winner. In some domains our planner based on TWEAK, which for convenience we will simply refer to as TWEAK, greatly outperforms SNLP, and in other domains, vice versa. The challenge is to identify the features of the domains where each planner is expected to perform well, so that practitioners can balance the protection methods based on the application domain.

In the following sections, we first review the SNLP algorithm and show how SNLP can be easily transformed into TWEAK. Next, we present an analysis of the two algorithms in order to identify the parameters that determine the performance of the two planners. Then, we present empirical results on a set of example problems to support the analysis.

*The first author is supported by Rome Laboratory of the Air Force Systems Command and the Defense Advanced Research Projects Agency under contract no. F30602-91-C-0081. The second author is supported in part by grants from the Natural Science and Engineering Council of Canada, and ITRC: Information Technology Research Centre of Ontario. The views and conclusions contained in this paper are the author's and should not be interpreted as representing the official opinion or policy of DARPA, RL, NSERC, ITRC, or any person or agency connected with them.

2 The SNLP Algorithm

In the planning algorithms that we consider below, we follow the notations used by Barrett and Weld [1]. A plan is a 3-tuple, represented as $\langle S, O, B\rangle$, where S is a number of steps, O is a set of ordering constraints, and B the set of variable binding constraints associated with a plan. A step consists of a set of preconditions, an add list, and a delete list. The binding constraints specify whether two variables can be bound to the same constant or not.

The core of SNLP is the recording of the *causal links* for why a step is introduced into a plan, and for protecting that purpose. If a step S_i adds a proposition p to satisfy a precondition of step S_j, then $S_i \xrightarrow{p} S_j$ denotes the causal link. An operator S_k is a threat to $S_i \xrightarrow{p} S_j$ if S_k can possibly add or delete a literal q that can possibly be bound to p. For convenience, we also refer to the pair $(S_k, S_i \xrightarrow{p} S_j)$ as a threat. In addition, we define an operator S_k to be a *positive threat* to $S_i \xrightarrow{p} S_j$, if S_k can possibly be between S_i and S_j, and S_k adds a literal q that can possibly be bound to p. Likewise, S_k is a *negative threat* if it can possibly be between S_i and S_j, and deletes a literal q that can possibly be bound to p. SNLP is a descendant of NONLIN [8], but differs in that NONLIN only considered negative threats.

The following algorithm, which is an adaptation of McAllester and Rosenblitt's Find-Completion algorithm [5] and Barrett and Weld's POCL algorithm [1], has been shown to be sound, complete, and systematic (never generates redundant plans). In the algorithm below, the notation codesignate(R) denotes the codesignation constraints imposed on a set of variable pairs R. For example, if $R = \{(x_i, y_i) \mid i = 1, 2, \ldots k\}$, then codesignate($R$) $= \{x_i = y_i \mid i = 1, 2, \ldots k\}$. Similarly, noncodesignate(R) denotes the set of non-codesignation constraints on a set R of variable pairs. The parameters of the algorithm are: *S=Steps, O=Ordering constraints, B= Binding constraints, G= Goals, T=Threats, and L=Causal links.*

Algorithm SNLP$(\langle S, O, B\rangle, T, G, L)$

1. **Termination:** If G and T are empty, report success and stop.

2. **Declobbering:** A step s_k *threatens* a causal link $s_i \xrightarrow{p} s_j$ when it occurs between s_i and s_j, and it adds or deletes p. If there exists a threat $t \in T$ such that t is a threat between a step s_k and a causal link $s_i \xrightarrow{p} s_j \in L$, then:
   - Remove the threat by adding ordering constraints and/or binding constraints using promotion, demotion, or separation. For completeness, all ways of resolving the threat must be considered.
     Promotion: $O' = O \bigcup \{s_k \prec s_i\}$, $B' = B$
     Demotion: $O' = O \bigcup \{s_j \prec s_k\}$, $B' = B$
     Separation:
     $O' = O \bigcup \{s_i \prec s_k\} \bigcup \{s_k \prec s_j\}$. Let q be the effect of s_k that threatens p and let P be the set of binding pairs between q and p. $B' = B \bigcup \sigma$, where $\sigma \in \{\alpha \mid \alpha = \text{noncodesignate}(s) \bigcup \text{codesignate}(P - s), \text{where } s \subseteq P \wedge s \neq \emptyset\}$.[1]
   - **Recursive invocation:**
     SNLP$(\langle S, O', B'\rangle, T - \{t\}, G, L)$

3. **Goal selection:** Let p be a proposition in G, and let S_{need} be the step for which p is a precondition.

4. **Operator selection:** Let S_{add} be an existing step, or some new step, that adds p before S_{need}. If no such step exists or can be added then backtrack. Let $L' = L \bigcup \{S_{\text{add}} \xrightarrow{p} S_{\text{need}}\}$, $S' = S \bigcup \{S_{\text{add}}\}, O' = O \bigcup \{S_{\text{add}} \prec S_{\text{need}}\}$, *and* $B' = B \bigcup$ the set of variable bindings to make S_{add} add p. For completeness, all ways of achieving the step must be considered.

5. **Update goal set:** Let $G' = (G - \{p\}) \bigcup$ preconditions of S_{add}, if new.

6. **Threat identification:** Let $T' = \{t \mid$ for every step s_k that is a positive or negative threat to a causal link $s_i \xrightarrow{p} s_j \in L'$, $t = (s_k, S_i \xrightarrow{p} S_j)\}$.

7. **Recursive invocation:**
   SNLP$(\langle S', O', B'\rangle, T', G', L')$.

3 The TWEAK Algorithm

The SNLP algorithm can be easily modified to implement the TWEAK algorithm. TWEAK differs from SNLP in only four ways. First, TWEAK uses a Modal Truth Criterion for the termination check [2]. This algorithm takes an $O(n^3)$ time, as compared with the $O(1)$ time termination routine of SNLP. Second, no ordering constraints are imposed when separation is used to resolve a threat, and no requirement

[1] The possible binding constraints are mutually exclusive, since systematicity requires that the search space is partitioned into non-overlapping parts.

is needed for variable binding constraints to be mutually exclusive. Third, when a plan is not correct, the MTC selects a goal by identifying a precondition of a step that is not necessarily true. Fourth, it only ensures at each step that all threats with the most recently built causal link are removed. However, after a causal link is constructed and threats removed, it can be clobbered again. In such a case, TWEAK will have to re-establish the causal link.

More precisely, the SNLP algorithm can be modified to implement the a TWEAK algorithm by modifying the steps as shown below. This algorithm requires a procedure called **mtc** that given a plan returns true if the plan is complete and otherwise returns a precondition of some step in the plan that does not necessarily hold.

Termination: If mtc($\langle S, O, B\rangle$) is true, report success and stop.

Separation: $O' = O$. Let q be the effect of s_k that possibly codesignates with p and let P be the set of binding pairs between q and p. $B' = B \bigcup \sigma$, where $\sigma \in \{\alpha \mid \alpha = \text{noncodesignate}(e), \text{where } e \in P\}$.

Goal Selection: Let p be the precondition of step S_{need} returned by the **mtc** procedure.

Threat identification:
Let $l_{new} = S_{\text{add}} \xrightarrow{p} S_{\text{need}}$, which is the causal link constructed in step 4. Let $T' = \{t | \text{for every step } s_k$ that is a *negative* threat to l_{new}, $t = (s_k, l_{new})\}$.

This implementation of TWEAK is both sound and complete. In addition, our modification requires only minimal changes to the implementation of SNLP [1], showing that both the SNLP and the TWEAK algorithms can be expressed equally elegantly.

4 Analyzing the Algorithms

Algorithm Complexities

Let *eb* be the effective branching factor and *ed* the effective depth of the search tree. In both algorithms, *eb* is the maximum number of successor plans generated either after step 2, or after step 6, while *ed* is the maximum number of plan expansions in the search tree from the initial plan state to the solution plan state. Then with a breadth-first search, the time complexity of search is $O(eb^{ed} * T_{node})$, where T_{node} is the amount of time spent per node.

We next analyze the complexity of the algorithms by fleshing out the parameters *eb*, *ed* and T_{node}. In this analysis, let P denote the maximum number of preconditions or effects for a single step, let N denote the total number of operators in an optimal solution plan, and let A be either the SNLP or TWEAK algorithm.

To expand the effective branching factor *eb*, we first define the following additional parameters. We use b_{new} for the number of new operators found by step 4 for achieving p, b_{old} for the number of existing operators found by step 4 for achieving p, and r_t for the number of alternative constraints to remove one threat. The effective branching factor of search by either algorithm is then

$$eb = \max\{(b_{new} + b_{old}(A)), r_t(A)\},$$

since each time the main routine is followed, either step 2 is executed for removing threats, or step 3–7 is executed to build causal links. If step 2 is executed, r_t successor states are generated, but otherwise, $(b_{new} + b_{old})$ successor plan states are generated.

Next, we expand the effective depth *ed*. In the solution plan, there are $N * P$ number of (p, S_{need}) pairs, where p is a precondition for step S_{need}. Let $f(A)$ be the fraction of the $N * P$ pairs chosen by step 3. For each pair (p, S_{need}) chosen by step 3, step 6 accumulates a set of threats to remove. Let $t(A)$ be the number of threats generated by step 6. Finally, let v be the total number of times any fixed pair (p, S_{need}) is chosen by step 3. Then we have

$$ed(A) = f(A) * N * P * t(A) * v(A).$$

For SNLP, each pair (p, S_{need}) must be visited exactly once. Therefore, $f(\text{SNLP}) = 1$ and $v(\text{SNLP}) = 1$. Also, SNLP examines every causal link in the current plan in step 4. The average time complexity for SNLP is:

$$O(\max(b_{new}+b_{old}(\text{SNLP}), r_t(\text{SNLP}))^{N*P*t(\text{SNLP})}*N*P).$$

In TWEAK, $f(\text{TWEAK}) \leq 1$, and can be much smaller than one, while $f(\text{SNLP}) = 1$. Pednault [7] noted that using an existing action to achieve a goal that is already true in the current plan will needlessly increase the size of the search tree. This is precisely what SNLP does by requiring causal links for all preconditions. However, since TWEAK does not protect any past causal links, a precondition can be visited twice. Therefore, $v(\text{TWEAK}) \geq 1$. $t(\text{TWEAK})$, however, should be much smaller than $t(\text{SNLP})$, since TWEAK only declobbers the most recently constructed causal link, and only negative threats are considered. Thus the number of threats is much smaller. Finally, TWEAK's termination routine takes $O((N * P)^3)$ time per node. In all, the complexity of TWEAK is:

$$O(\max(b_{new}+b_{old}(\text{TWEAK}), r_t(\text{TWEAK}))^{eb}*(N*P)^3$$

where $eb = f(\text{TWEAK}) * N * P * t(\text{TWEAK}) * v(\text{TWEAK})$.

Comparison

We now compare the complexity of the SNLP and TWEAK. The branching factor b_{new} is the same in both algorithms. The number of ways to resolve each threat, r_t, for SNLP will be greater than or equal to r_t for TWEAK. However, the difference will be small for domains where literals have a small number of parameters[2]. However, $b_{old}(\text{SNLP})$ will be the same or less than $b_{old}(\text{TWEAK})$ since SNLP will avoid redundant plans, which can arise from simple establishment. Since most domains have a small number of parameters, the branching factor for SNLP will generally be less than or equal to the branching factor for TWEAK. Similarly, the time per node T_{node} for SNLP will be strictly less than the time per node for TWEAK.

On the other hand, the effective depth of SNLP will usually be greater than the effective depth of TWEAK. $f(\text{TWEAK})$ is generally smaller than one, while $f(\text{SNLP})$ is exactly one since a causal link must be built for each precondition. If all the goals are already true in the initial state, then $f(\text{TWEAK}) = 0$, while $f(\text{SNLP}) = 1$. $t(\text{TWEAK})$ will also be a lower bound on $t(\text{SNLP})$ since TWEAK only declobbers the most recently constructed causal link. However, $v(\text{TWEAK})$ will be greater than or equal to $v(\text{SNLP})$ since TWEAK may have to establish a particular condition multiple times in the planning process. This factor will be determined by the number of negative interactions that arise during planning. If this number is not too large, then TWEAK will search to a smaller depth than SNLP.

The analysis shows that the relative performance of SNLP and TWEAK depends on the combination of the actual parameters in a planning problem. Factors that work strictly in the favor of SNLP are b_{old}, T_{node}, and v. Factors that work strictly in the favor of TWEAK are r_t, t and f.

Systematicity

SNLP is systematic, which means that no redundant plans are generated in the search space. It is possible to find an example where TWEAK does generate redundant plans, thus TWEAK is not a systematic planner. However, a planner that is systematic is not necessarily more efficient. The systematicity property reduces the branching factor by avoiding redundant plans, which is achieved by protecting against both the negative and positive threats. This increases the factor t, a multiplicative factor in the exponent. Thus, SNLP reduces the branching factor at a price of increasing the depth of search. As a result, one can get a systematic, but less efficient planning system.

[2]For domains where the number of parameters in a literal is large, r_t for TWEAK could be exponentially smaller than SNLP

5 Empirical Results

In this section we report experimental results on seven sample domains and problems. The domains and example problems were taken from the domains distributed with the SNLP implementation developed by [1]. Each of these problems was run in SNLP [1] and a version of TWEAK that we modified from SNLP. The problems are solved using a breadth-first search on the solution length. The results are shown in Table 1.

The analysis in the previous section can be used to predict the performance of a planning algorithm for a given class of domains. We conjecture that two important features of a domain that determine the performance of the two algorithms are the *number of positive threats* and *number of negative threats.* These features are important because the major difference between the two algorithms is the way they handle threats. TWEAK only avoids some negative threats while SNLP protects against all positive and negative threats. If there were no threats at all in a domain, then the two planners would perform approximately the same. Thus, the level of threats distinguishes where each planner will work well.

The table confirms our conjecture that neither system will always perform better than the other. In all of these problems there appear to be many positive and negative threats. The performance on the individual problems depends on the combination of these two competing features. On the Schedworld, Ferry, and Molgen problems, the combination of the negative and positive threats balances out and both planners perform about the same. On the Blocksworld, Tower of Hanoi, and Fixit problems, the positive threats appear to dominate resulting in significantly better performance in TWEAK. In contrast, on the Monkey domain SNLP performs significantly better than TWEAK. In this case, the negative threats appear to dominate, which results in a much higher branching factor for TWEAK due to the many redundant plans that must be considered.

There are some clear trends in terms of the parameters of the search that confirm our previous analysis. First, as predicted by the analysis, the branching factor for SNLP is the same or lower on all the problems except the Tower of Hanoi and on this problem

Domain	Time (sec)		Nodes		Avg BF		Avg Depth		T_{node}(ms)	
	SNLP	Tweak	SNLP	Tweak	SNLP	Tweak	SNLP	Tweak	SNLP	Tweak
Blocksworld	1.8	0.68	217	73	2.05	2.09	7.54	5.98	8.3	9.3
TOH	3.21	0.6	345	50	1.49	1.47	11.63	6.71	9.3	12.0
Schedworld	42.39	39.97	2438	1934	3.85	3.92	9.3	7.67	17.4	20.7
Ferry	6.39	5.54	767	686	1.34	1.63	16.53	13.44	7.7	8.1
Molgen	18.84	23.58	1589	1683	1.36	1.47	17.87	17.54	11.9	14.0
Monkey	4.51	24.9	532	1625	1.29	1.83	13.25	10.5	8.5	15.3
Fixit	40.82	14.62	2245	668	2.19	2.41	8.93	7.23	18.2	21.9

Table 1: Comparison of SNLP and TWEAK Algorithms on Example Problems

they are essentially the same. Second, the results on the time per node also support the analysis in that SNLP has a lower cost per node than TWEAK on every problem. Third, the depth of the search in TWEAK is smaller than SNLP on every problem. Thus, the values of v on these problems turned out to be much less significant than the values of f and t. Since SNLP performed consistently better in terms of branching factor and time per node and TWEAK performed consistently better on the search depth, the overall performance depends on the combination of the specific values for these parameters.

6 Conclusion

This paper presented a detailed comparison and analysis of the SNLP and TWEAK planning algorithms. The detailed complexity analysis is important for two reasons. First, it shows that the systematicity property of SNLP does not always reduce the overall search of a planner because it has the side-effect of increasing the search depth, which can degrade performance. Second, it provides the foundation for predicting the conditions under which different planning algorithms will perform well. As the results show, SNLP performs better than TWEAK when there are many negative threats and few positive threats. And TWEAK performs significantly better than SNLP in the opposite case.

In future work, we plan to study the combination of many negative and many positive threats in order to predict the performance of the two systems on this class of problems. To pursue this analysis we plan to study several domains in more depth and perform detailed empirical measurements on the individual parameters of the complexity analysis.

References

[1] Anthony Barrett and Dan Weld. Partial order planning: Evaluating possible efficiency gains. Technical Report 92-05-01, University of Washington, Department of Computer Science and Engineering, 1992.

[2] David Chapman Planning for Conjunctive Goals. Artificial Intelligence, volume 32, pp. 333-377, 1987.

[3] Steve Chien. *An Explanation-based Learning Approach to Incremental Planning.* PhD thesis, University of Illinois, Urbana, IL., 1990.

[4] Oren Etzioni, Steve Hanks, Daniel Weld, Denise Draper, Neal Lesh, and Mike Williamson. An approach to planning with incomplete information. submitted for publication, University of Washington, Department of Computer Science and Engineering, 1992.

[5] David McAllester and David Rosenblitt. Systematic nonlinear planning. In *Proceedings of the 9th AAAI*, Anaheim, CA, 1991.

[6] Steve Minton, John Bresina, and Mark Drummond. Commitment strategies in planning: A comparative analysis. In *Proceedings of the 12th IJCAI*, Sydney, Australia, 1991.

[7] Edwin P.D. Pednault Toward a Mathematical Theory of Plan Synthesis. Ph.D. Thesis, Department of Electrical Engineering, Stanford University, Stanford, CA, 1986.

[8] Austin Tate Generating Project Networks. IJCAI77, pp. 888-893, 1977.

[9] Qiang Yang, Josh Tenenberg, and Steve Woods. Abstraction in nonlinear planning. University of Waterloo Technical Report CS 91-65, 1991.