Two Approaches to Semi-Dynamic Disjunctive Temporal Problems

Peter J. Schwartz and Martha E. Pollack

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI, USA
pschwart@eecs.umich.edu, pollackm@eecs.umich.edu

Abstract

This paper considers two approaches to improving efficiency and stability in semi-dynamic Disjunctive Temporal Problems (DTPs). A semi-dynamic DTP is a sequence of DTPs (Stergiou and Koubarakis 1998) in which each problem is a restriction of the problem before. We consider three basic types of DTP restrictions—tightening the bound of an existing constraint, adding a simple constraint, and adding a disjunctive constraint. The techniques we use for solving semi-dynamic DTPs come from the literature on dynamic Constraint Satisfaction Problems (CSPs) and include nogood recording (Schiex and Verfaillie 1993) and oracles (van Hentenryck and Provost 1991). Experimental results show that nogood recording improves efficiency but hurts stability, whereas oracles improve efficiency even more while also improving stability. The performance of nogood recording and oracles used in combination is not significantly different than the performance of oracles alone.

Introduction

The Disjunctive Temporal Problem (DTP) is a very expressive temporal constraint formalism (Stergiou and Koubarakis 1998). This expressivity comes at the cost of NP-complete complexity (Dechter, Meiri, and Pearl 1991), so a great deal of prior research has focused on heuristic approaches to solving DTPs (Stergiou and Koubarakis 2000, Tsamardinos and Pollack 2003, Armando et al. 2004). For many applications, however, it may be necessary to solve a *sequence* of DTPs in which each problem is very similar to the one before. Moreover, in at least some applications, it is much more common for one DTP in the sequence to be a restriction, rather than a relaxation, of the previous DTP in the sequence.

Execution-monitoring systems, such as Autominder (Pollack et al. 2003) and the Remote Agent (Muscettola et al. 1998), provide one example. Consider the Autominder system, a schedule maintenance and execution-monitoring system that issues reminders to keep a user on schedule. Autominder represents a user's schedule as a DTP, which is updated whenever time passes, an action is executed, or a new action and its constraints are added to the schedule. After each such change, Autominder must check the consistency of the new DTP. As long as the user neither retracts any constraints from the schedule nor executes an action that violates the schedule, each DTP in the sequence will be a restriction of the one before.

Plan generation systems that model complex temporal constraints, such as DT-POP (Schwartz and Pollack 2004) and TGP (Smith and Weld 1999), provide another example. DT-POP is a partial-order planner that can generate plans that include complex temporal relationships over multiple goals and actions, each of which may involve arbitrarily many time points. DT-POP represents all of these temporal relationships as DTPs, so each time a partial plan is modified to repair a flaw, DT-POP must solve a new DTP to determine whether the resulting plan is consistent. Whether adding an ordering constraint to repair a threat adding a causal link or new action to repair an open condition, the DTP is modified by adding or tightening constraints. As a result, the DTP of each partial plan in the search tree is a restriction of the DTP of its parent.

In this paper, we explore the question of solving *semi-dynamic DTPs*—sequences of DTPs in which each is a restriction of the previous one. We consider three types of basic DTP restrictions, which are jointly sufficient to represent any of the changes to a DTP described in the examples above. We employ two techniques that have been previously used for solving dynamic CSPs (sequences of finite-domain CSPs): nogood recording and oracles.

We are interested in how these techniques affect search efficiency as well as solution *stability*—i.e., the extent to which the solution to one DTP is similar to the solution to the previous DTP. As pointed out by Verfaillie and Schiex (1994, page 307), instability in a dynamic CSP "…may be unpleasant in the framework of an interactive design or a planning activity, if some work has been started on the basis of the previous solution." The same can be said of dynamic DTPs.

In the next section, we define DTPs and describe the standard approaches to solving them. We then introduce and formally define the semi-dynamic DTP. After that, we describe nogood recording and oracles, and we show how they can be applied to semi-dynamic DTPs. We then describe a set of experiments that compares these two techniques, both alone and together, and present results

showing their effect on efficiency and stability. We conclude with a discussion of the results and ideas for future work.

The Disjunctive Temporal Problem

The DTP is a generalization of the Simple Temporal Problem (STP). An STP (Dechter, Meiri, and Pearl 1991) is a pair $<V, C>$, where V is a set of temporal variables that represent time points and C is a set of simple temporal constraints over the time points of V. Each simple temporal constraint of C has the form $x - y \leq b$, where x and y are time points of V and b is a real number. Such a constraint is interpreted as "x follows y by no more than b units of time." An STP is often represented as a *d-graph*, in which each time point is a vertex and each constraint is a directed edge weighted by its bound. An STP is consistent iff its d-graph contains no negative cycles, so if $|V| = n$, then the STP can be solved in $O(n^3)$ time with a standard all-pairs shortest path algorithm such as Floyd-Warshall or Bellman-Ford (Cormen, Leiserson, and Rivest 1990).

A DTP is also a pair $<V, C>$, where V is a set of temporal variables that represent time points, but C is a set of *disjunctive* temporal constraints over the time points of V. Each disjunctive temporal constraint of C has the form $c_1 \vee c_2 \vee \ldots \vee c_m$, where each c_i is a simple temporal constraint. A disjunctive temporal constraint is satisfied if at least one of its component simple temporal constraints is satisfied. A DTP can express constraints that an STP cannot. For example, if a schedule contains two actions, A and B, that cannot overlap, a DTP can express this with the constraint "A ends before B begins, or B ends before A begins."

There are two predominant methods for solving a DTP. The first is to convert the DTP into a Satisfiability (SAT) problem. This method is used in TSAT++ (Armando et al. 2004), which is currently the fastest DTP solver. The second is to convert the DTP into a finite-domain CSP, as explained below. This method is used in Epilitis, which was the fastest when it was developed several years ago (Tsamardinos and Pollack 2003). Very little research has considered dynamic SAT problems (Hoos and O'Neill 2000), whereas much research has already been done on dynamic CSPs (van Hentenryck and Provost 1991, Schiex and Verfaillie 1993, Verfaillie and Schiex 1994). For this reason, the current work builds on the CSP method for solving DTPs.

To transform a DTP into a finite-domain CSP $<V, D, C>$ (called a *meta-CSP*), one has each variable of V represent a single disjunctive temporal constraint of the DTP, where the domain of each variable $v \in V$ is the set of component simple temporal constraints. The constraints C of the meta-CSP express the requirement that any combination of simple temporal constraints that is chosen as an assignment of the values as a solution of the meta-CSP must form a consistent STP.

For example, suppose we are given a DTP with time points $\{w, x, y, z\}$ and these constraints:

$c_1 = (z - y \leq 2)$
$c_2 = (w - y \leq 5) \vee (w - x \leq 7)$
$c_3 = (y - x \leq -3) \vee (y - w \leq -6)$

This DTP contains three disjunctive temporal constraints, so the corresponding meta-CSP will contain three variables. The domain of each variable in the meta-CSP is the set of component simple temporal constraints, so the domain of c_1 is $\{(z - y \leq 2)\}$, the domain of c_2 is $\{(w - y \leq 5), (w - x \leq 7)\}$, and the domain of c_3 is $\{(y - x \leq -3), (y - w \leq -6)\}$.

After this transformation, the DTP can be solved with any number of standard CSP techniques. Epilitis, the fastest DTP solver that uses this meta-CSP transformation, uses five different techniques to improve efficiency. Three of these techniques—forward checking, conflict-directed backjumping, and nogood recording—come from the finite-domain CSP literature. The other two techniques—removal of subsumed variables and semantic branching—are made possible by the fact that the temporal constraints are linear inequalities. See (Tsamardinos and Pollack 2003) for details on Epilitis.

The Semi-Dynamic DTP

As mentioned above, we define a semi-dynamic DTP to be a sequence of (static) DTPs, where each DTP in the sequence is a restriction of the previous one. The current work considers three types of restrictions in DTPs:

1. **tighten:** The bound b of a temporal constraint $x - y \leq b$ is reduced. When a bound is tightened, it is possible that some of the assignments in which it participates used to be valid but now create a negative cycle in the d-graph. The constraint that is tightened corresponds to a value of the meta-CSP, so tightening the bound of a constraint in the DTP has the effect of tightening a constraint in the meta-CSP.

2. **add STC:** A simple temporal constraint is added to the DTP. This corresponds to adding a variable to the meta-CSP with a domain size of one. The meta-CSP solver has no choice but to include the assignment of this value to this new variable in any potential solution that it considers.

3. **add DTC:** A disjunctive temporal constraint is added to the DTP. This corresponds to adding a variable to the meta-CSP with a domain size greater than one. The meta-CSP solver can choose any of the values in the domain of the new variable.

Some might argue for the inclusion of a fourth type of restriction—the removal of a disjunct from a disjunctive temporal constraint. In the meta-CSP, this would correspond to the removal of a value from the domain of a variable. It is possible to achieve the same effect by tightening the bound of the disjunct to negative infinity; even though the value is still present in the meta-CSP, it

could not participate in any solution and would be immediately pruned by forward checking. For this reason, we only consider the three types of restrictions listed above.

We can now formally define the semi-dynamic DTP:

Definition 1. A **semi-dynamic DTP** is a pair $<P_0, C>$, where P_0 is a static DTP, and C is a sequence of changes, each of which is one of the three types of restrictions listed above. If C contains n changes, then a semi-dynamic DTP solver must solve the sequence of DTPs $P_0, P_1, \ldots, P_n$, where each DTP P_i is created by restricting P_{i-1} according to change c_i, for $1 \leq i \leq n$.

The current work only considers semi-dynamic DTPs in which each change is a single restriction. In some systems, it might make more sense to perform several restrictions between subsequent DTPs. For example, if an event e is executed exactly t units of time after another event f, an execution-monitoring system can represent this by adding two simple temporal constraints to the schedule: $(e - f \leq t)$ and $(f - e \leq -t)$. Both of these constraints can be added before the new DTP is solved. Semi-dynamic DTPs in which each change is a set of restrictions remains an open avenue for future research.

We consider two techniques that have been shown to improve the performance of dynamic CSPs, adapting them to semi-dynamic DTPs. The next section describes nogood recording, and the section after that describes oracles.

Nogood Recording

Nogood recording (NGR) has been shown to be an effective technique for improving both static and dynamic CSPs (Schiex and Verfaillie 1993). Intuitively, a nogood is a partial assignment of variables that cannot be extended to a complete solution. Formally, a nogood of CSP $<V, D, C>$ (where V is the set of variables, D is the set of respective variable domains, and C is the set of constraints) is a pair $<A, J>$. A is an assignment to a subset of the variables of V, and J is a subset of V such that the constraints over J prevent A from participating in any solution (J is called the justification of the nogood).

As a CSP solver searches for a solution, it records a nogood each time it finds a dead end. As search continues, the CSP solver checks each partial assignment it considers against each nogood it has recorded. If it finds a nogood $<A, J>$ such that A is a subset of the partial assignment being considered, then that partial assignment can be pruned immediately. If the CSP solver is also using conflict-directed backjumping, then the justification J tells the solver how far it can safely backtrack.

In order to maximize the pruning power of each nogood, the assignment should be made minimal. This is accomplished by applying two basic properties of nogoods. First, if $<A, J>$ is a nogood, then $<A{\downarrow}J, J>$ is also a nogood, where $A{\downarrow}J$ is the projection of assignment A onto the variables of justification J. In other words, we can safely remove any variable in A that does not participate in the constraints over J. Second, if all possible extensions of an assignment A along a particular variable x have been recorded as nogoods $<A \cup \{x \leftarrow v_i\}, J_i>$ for all values v_i in the domain of x, then $<A, \cup_i J_i>$ is also a nogood. In other words, since any solution must assign a value to every variable, and since there is no valid extension of A that assigns a value to x, we know that A cannot be extended to a complete solution. During search, this second property makes it possible to record a nogood when backtracking over a non-leaf that is a generalization of all of its children combined.

Suppose we are solving a CSP with variables w, x, y, and z, all of which have the domain $\{1, 2, 3\}$. Say our current assignment is $\{w \leftarrow 1, x \leftarrow 2, y \leftarrow 3, z \leftarrow 1\}$, but this assignment is found to violate a constraint over w, x, and z. We could record the nogood $<\{w \leftarrow 1, x \leftarrow 2, y \leftarrow 3, z \leftarrow 1\}, \{w, x, z\}>$, but the first property of nogoods tells us that we can instead record the smaller nogood $<\{w \leftarrow 1, x \leftarrow 2, z \leftarrow 1\}, \{w, x, z\}>$. This nogood has greater pruning power because it applies no matter what value is assigned to y. Now say that, after further search, we discover that $<\{w \leftarrow 1, x \leftarrow 2, z \leftarrow 2\}, \{w, x, z\}>$ and $<\{w \leftarrow 1, x \leftarrow 2, z \leftarrow 3\}, \{w, x, z\}>$ are also nogoods. At this point, we have seen that the assignment $\{w \leftarrow 1, x \leftarrow 2\}$ extended with any assignment to z leads to a dead end, so the second property of nogoods tells us that we can record the nogood $<\{w \leftarrow 1, x \leftarrow 2\}, \{w, x, z\}>$. This nogood has more pruning power because it applies no matter what value is assigned to z.

Epilitis was the first system to apply NGR to static DTPs. The only difference between applying NGR to finite-domain CSPs and DTPs comes when finding the justification of a nogood at a leaf dead end. In a CSP, the constraints are given in some explicit description, but in the meta-CSP of a DTP, the constraints are given implicitly that any assignment must form a consistent STP. An STP is inconsistent iff its d-graph contains a negative cycle, so the justification of a nogood in a DTP is found by simply identifying the time points involved in the negative cycle, and then identifying the meta-CSP variables whose assignments created that negative cycle.

In practice, the number of nogoods that are recorded can grow to an unmanageable size, so a few basic techniques can be used to prevent this from happening. First, as the reader may have noticed while looking at the previous example, when the second property of nogoods is applied, the new nogood that is generated is a generalization of the nogoods that were combined to create it. Whenever the second property is applied, all of the nogoods that were combined to create the new one can be removed from the set of nogoods without losing any pruning power. Second, the more variables in a nogood's assignment, the fewer assignments they apply to, so nogoods with large assignments have little pruning power. It is therefore common practice to select a size limit and only record nogoods whose assignment is within this limit. Tsamardinos and Pollack (2003) suggest a nogood size

limit of 10 for Epilitis, and we follow their suggestion in our implementation.

The application of NGR to dynamic problems is straightforward. Nogoods are recorded while searching for a solution to one problem instance and then applied while searching for a solution to the next problem instance. If the problem is relaxed, some of the nogoods may no longer be valid and would have to be removed from the set. The current work, however, only considers semi-dynamic DTPs in which no relaxations are permitted, so we do not need to deal with this case. A restriction only removes solutions of a problem instance, so any partial assignment that could be safely pruned by a nogood before the restriction can be safely pruned after.

Oracles

Oracles (van Hentenryck and Provost 1991) have also been applied to dynamic finite-domain CSPs. During CSP backtracking search, all partial assignments that were tested before the first solution was found must have been pruned. After a restriction to the CSP, those same partial assignments can be immediately pruned from the new search space. An oracle records the path to the first solution so the next search can try to repeat it, bypassing the pruned portion of the search space.

Formally, we define an oracle for a problem as a triple <*A*, *O*, *P*>. *A* is the solution assignment of the *previous* problem in the sequence, *O* is the order in which variables were chosen to find this solution, and *P* is the set of values that were tested and pruned from the domain of each variable along the way. If the current problem resulted from a restriction, then all of the values in *P* can be safely pruned again without being tested. If the solver is also using conflict-directed backjumping or NGR, then a justification must be recorded along with each pruned value of *P*.

Given a CSP <*V*, *D*, *C*> and an oracle <*A*, *O*, *P*>, backtracking search proceeds as follows. First, a variable *var* is selected according to *O*. Each value in the domain of *var* that appears in *P* can be pruned immediately without search. Then the value assigned to *var* according to *A* is chosen. This process continues until either a new solution is found or the value that should be chosen according to *A* causes an inconsistency. In the latter case, the oracle is abandoned from that point in the search. From then on, variables and values are selected according to the solver's heuristics, and no more values are pruned without first being searched. If a solution is found, a new oracle can be generated from the search path to guide the next search.

An oracle can be generated directly from a search tree once the first solution is found. Consider the search tree shown in Figure 1. The nodes that are crossed out represent partial assignments that were tested and pruned, the black nodes represent partial assignments along the search path to the solution, and the white nodes represent partial assignments that have not been tested yet. The oracle is enclosed in a dotted line.

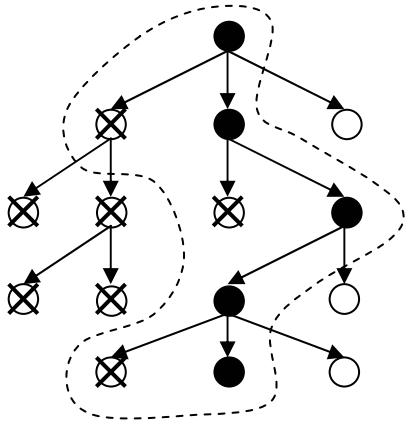

Figure 1. An oracle from a CSP search tree.

The pseudo-code for solving a CSP with an oracle appears in Figure 2. The **Oracle-Search** procedure takes three arguments: the CSP <*V*, *D*, *C*> that is currently being solved, the oracle <*A*, *O*, *P*> from the previous problem that is guiding the current search, and the oracle <*A'*, *O'*, *P'*> that is being created to guide the next search. If the given CSP is consistent, **Oracle-Search** returns the new oracle (which contains the solution assignment); otherwise, **Oracle-Search** returns FAIL.

Oracle-Search is a recursive procedure, so it begins by checking if all of the variables have been assigned, and, if so, it returns the new oracle (line 1). Otherwise, it selects the next variable to be assigned. If the old oracle is still being followed, **Oracle-Search** follows the variable ordering *O* from the previous search, and all of the values that were tested and pruned during the previous search can be pruned immediately without being tested in the current search (lines 3-6). If the old oracle has been abandoned, **Oracle-Search** falls back on a heuristic variable selection method (lines 7-8). Either way, the new variable choice is appended to the variable ordering of the new oracle (line 9).

Now **Oracle-Search** tests the values in the working domain of the chosen variable until one is found that leads to a solution or all of them have failed. The first step is to choose a value to test. If the old oracle is still being followed, **Oracle-Search** starts by testing the value that was assigned to the chosen variable in the previous solution assignment (lines 11-12). If the old oracle has been abandoned, **Oracle-Search** falls back on a heuristic value selection method (lines 13-14). The chosen value is removed from the working domain of the chosen variable (line 15).

Once the variable and value have been chosen, the new binding is added to the current assignment and tested against the constraints (lines 16-17). If none of the constraints are violated, **Oracle-Search** calls itself recursively (line 18). In this recursive call, the new oracle has been modified to reflect the new variable and value choices, along with any other values that have been pruned from the chosen variable's working domain. If the chosen

value leads to a solution, then that solution (which is the new oracle) is returned immediately (line 19).

If the chosen value does not lead to a solution, or if the new assignment failed the constraint check, then the value is pruned from the working domain of the chosen variable (line 20). The fact that this value has been pruned is recorded in the new oracle (line 21). Since the value from the previous solution assignment is always tested first, that value must have failed by this point. This means that the oracle cannot lead the solver directly to a solution, so the oracle is abandoned (line 22). From this point on, **Oracle-Search** will behave exactly like a standard CSP backtracking search algorithm, except that it will continue to build the new oracle to guide the next search. If all of the values in the working domain of the chosen variable have failed to lead to a solution, then **Oracle-Search** returns FAIL (line 23).

```
Oracle-Search(<V, D, C>, <A, O, P>, <A', O', P'>)
// <V, D, C> is the CSP being solved
// <A, O, P> is the oracle from the previous problem
// <A', O', P'> is the new oracle being created
1  if all variables assigned, then return <A', O', P'>
2  P" ← P'
3  if still following old oracle <A, O, P>, then
4     choose next variable var according to O
5     prune all values from D[var] that appear in P[var]
6     P"[var] ← P[var]
7  else (old oracle has been abandoned)
8     choose next variable var according to heuristic
9  let new ordering O" be O' appended with var
10 while working domain D[var] is not empty
11    if still following old oracle <A, O, P>, then
12       choose value val according to A[var]
13    else (old oracle has been abandoned)
14       choose value val according to heuristic
15    remove val from D[var]
16    let new assignment A" be A' ∪ {var←val}
17    if A" does not violate constraints C then
18       sol ← Oracle-Search(<V, D, C>, <A, O, P>,
             <A", O", P">)
19       if sol ≠ FAIL then return sol
20    prune val from D[var]
21    P"[var] ← P"[var] ∪ {val}
22    abandon the old oracle
23 return FAIL
```

Figure 2. A CSP solver that uses an oracle to guide search.

We can make several observations about the behavior of this algorithm. If the solution of the previous problem is still a valid solution after the restriction, then the oracle will lead the solver directly to it. If a new variable has been added to the problem, the oracle will cause the solver to try to extend the previous solution by assigning the new variable each of the values in its domain before it backtracks to test any other assignments. If the solution of the previous problem is no longer a valid assignment after the restriction, then the oracle will still force the solver to use as much of the previous solution as possible.

The use of oracles can be applied directly to the meta-CSPs of a semi-dynamic DTP. As with nogoods, a relaxation could invalidate some of the justifications that are recorded with the pruned values of the oracle, so these values would have to be removed from the pruned set before the next search was to begin. The current work, however, only considers restrictions, so we do not need to deal with this case.

Experimental Comparison

We compare the efficiency and stability of four algorithms for solving semi-dynamic DTPs. Since nogood recording and oracles operate on CSPs and Epilitis is the fastest algorithm to solve DTPs using the meta-CSP transformation, we use Epilitis as the underlying static DTP solver of all of our algorithms.

As stated earlier, TSAT++ is currently the fastest static DTP solver, but the publicly available implementation of TSAT++ only outputs four pieces of information after solving a DTP: the consistency of the given DTP, the total time TSAT++ used to determine consistency, the search time, and the number of constraint checks performed. It does not output a solution of the DTP when it is found to be consistent. This makes it impossible to reuse solutions or to measure the stability of solutions of subsequent DTPs in the sequence, so a fair comparison to TSAT++ is not possible at this time. We are presently working with the creators of TSAT++ to get an implementation that does output solutions.

The four algorithms we tested are as follows:

1. **Naïve:** The first algorithm is a naïve algorithm that repeatedly applies Epilitis to solve each DTP in the sequence from scratch. Even though this algorithm (as well as all of the others) uses nogood recording as a method to enhance the efficiency of Epilitis, the nogoods learned during one search are forgotten when the DTP changes.

2. **NGR:** The second algorithm is exactly the same as the first algorithm, except that all of the recorded nogoods are applied to all subsequent DTPs in the sequence.

3. **OR:** The third algorithm forgets all of its recorded nogoods when the DTP changes just like the naïve algorithm does, but this algorithm generates an oracle after each DTP is solved and applies it when solving the next DTP in the sequence.

4. **Both:** The fourth algorithm combines both techniques. It applies recorded nogoods to all subsequent DTPs, and it generates an oracle after solving one DTP and applies it to the next.

As pointed out in the section on oracles, if the previous solution is still a valid assignment after the restriction, an oracle will guide the search to test the previous solution

first and then extend it if a new meta-CSP variable has been added to the problem. To facilitate better comparison, we apply this same technique to both of the algorithms that do not use oracles. Before starting a new search, all of the algorithms check whether the previous solution is still valid, or if it can be extended when a new variable has been added to the meta-CSP. A new search is performed only if the previous solution is not (and cannot be extended to) a solution of the current problem.

To compare the algorithms, we generated four sets of 50 semi-dynamic DTPs. Each semi-dynamic DTP consists of a consistent initial DTP (P_0) and a sequence of restrictions (C). The first set contains only tighten changes, the second set contains only add STC changes, the third set contains only add DTC changes, and the fourth set contains all three types of changes chosen from a uniform distribution. The sequence of changes terminates when either the DTP becomes inconsistent (because more restrictions will always produce inconsistent DTPs) or the DTP has been changed 50 times.

The initial DTP of each dynamic problem is generated randomly based on the parameters of the most difficult problems considered in (Tsamardinos and Pollack 2003). Each of the initial DTPs in our experiments contains 30 time points and 180 disjunctive temporal constraints. This is a ratio of 6 disjunctive constraints to each time point, which Tsamardinos and Pollack found to be a phase transition point where DTPs are the most difficult to solve. Each disjunctive temporal constraint is the disjunction of 2 simple temporal constraints, and the bound of each simple constraint is chosen randomly with uniform probability from the integers of [-100, 100].

A tighten change is generated by randomly selecting a simple constraint of one of the disjunctions and reducing its bound by a random integer amount in [0, 100]. An add STC change is generated in the same manner as the disjunctive constraints of the initial DTP, except that the new constraint contains only 1 disjunct instead of 2. An add DTC change is generated in exactly the same manner as the disjunctive constraints of the initial DTP. For all types of restrictions, the set of time points in the DTP does not change.

Each type of change affects the DTP to a different degree. A tighten change only has a probability of 0.5 of tightening a disjunct that is part of the solution from the previous problem, and even if it does, it is still possible that the bound is not reduced enough to invalidate the previous solution. An add STC change forces the solution to include one particular simple constraint, so it is more likely to invalidate the previous solution. An add DTC change forces the solution to include a new simple constraint, but the particular simple constraint can be chosen from the set of (two) disjuncts in the new disjunctive constraint. Because of the inherent differences among the types of changes, the semi-dynamic DTPs generated by applying them differ in both the average number of changes that are made before the problem becomes inconsistent and the percentage of solutions that can be reused for each algorithm. Table 1 shows the differences among the four problem sets. The values in Table 1 demonstrate that the average number of changes and percentage of reused solutions can both vary greatly with the change type, but vary little among the different algorithms.

change type	**avg number of changes before inconsistent**	**% solutions reused**			
		Naïve	**NGR**	**OR**	**Both**
tighten	39.6	91.91	92.32	92.27	92.22
add STC	5.1	62.75	61.57	61.66	61.57
add DTC	16.0	82.75	82.38	82.20	81.75
all	13.4	78.81	78.81	80.06	79.25

Table 1. Average number of changes per problem and percentage of solutions reused for semi-dynamic DTPs with different types of restrictions.

We ran each of the four algorithms on all of the problems and measured the efficiency and stability of each algorithm. Efficiency was measured in two ways: the CPU time and the number of nodes (i.e., partial assignments) tested while solving each semi-dynamic DTP. Stability is measured as the percent of variable bindings from the previous solution that match the new solution. Recall that a new search is only necessary when the previous solution cannot be reused or extended. If all of the same variable bindings from the previous solution match the new solution, then the stability is 100%; however, in the case of 100% stability, a new search is not necessary, so these cases are not included in the averages that are reported. Thus, the stability values reported are lower than they would be had the solution-reuse cases been incorporated into the averages. Tables 2, 3, 4, and 5 show the results of tighten, add STC, add DTC, and all changes, respectively. The algorithms were all implemented in Java and the experiments were run on a 3.0 GHz Pentium 4 machine with 1 GB of memory.

	Naïve	**NGR**	**OR**	**Both**
time (sec)	37.60	19.38	11.91	12.86
nodes	27,777	14,041	9,808	10,116
stability	75.50	73.60	79.20	79.40

Table 2. Efficiency and stability on tighten restrictions.

	Naïve	**NGR**	**OR**	**Both**
time (sec)	13.44	11.99	9.79	10.39
nodes	9,778	8,863	7,808	8,230
stability	73.46	67.15	75.10	75.80

Table 3. Efficiency and stability on add STC restrictions.

	Naïve	NGR	OR	Both
time (sec)	39.14	28,69	21.86	22.09
nodes	27,882	20,596	16,452	16,309
stability	76.80	74.18	80.40	80.17

Table 4. Efficiency and stability on add DTC restrictions.

	Naïve	NGR	OR	Both
time (sec)	16.48	12.39	9.45	10.25
nodes	12,037	8,953	7,770	8,285
stability	73.30	71.67	76.10	76.32

Table 5. Efficiency and stability on all restrictions.

Discussion

Tables 2-5 show that for every type of restriction, nogood recording helps efficiency but hurts stability relative to the naïve approach. NGR helps efficiency because the nogoods allow earlier pruning in DTPs that appear later in the sequence. NGR can hurt stability because Epilitis counts the number of nogoods in which each value participates as part of its variable and value selection heuristics, so the additional nogoods can steer the DTP solver to a very different part of the search space. Even if the number of nogoods for each value was not incorporated directly into the variable and value selection heuristics, the additional nogoods from the previous search would still allow forward checking to prune some values early. Since most heuristics count the number of values in the working domain of each variable, and since the additional nogoods would remove some of these values, we would still expect the heuristics to guide the search to a different part of the search space, leading to a reduction in stability.

Tables 2-5 also show that for every type of restriction, the use of oracles resulted in better efficiency and stability than either the naïve approach or nogood recording. On one hand, an oracle can improve efficiency by pruning part of the search space and reusing part of the previous solution. On the other hand, an oracle might have hurt performance by leading the solver to part of the search space where it will not find a solution. Apparently, oracles help performance much more than they hurt it: they improve performance in all cases, most dramatically reducing CPU time to less than one third of the naïve approach when tightening bounds (see Table 2). Oracles improve stability by starting the search from the previous solution, generally increasing stability by 2 to 3 percent over the naïve approach. This is in contrast to nogood recording, which generally decreased stability by roughly the same amount.

It is interesting to compare the performance of the combination of oracles with nogood recording against the performance of oracles alone. We had expected that the combination of techniques, when compared to oracles alone, would result in equivalent stability (because the oracles would still guide the beginning of the search) but improved efficiency (because the nogoods would allow earlier pruning and provide more informed heuristics when the oracle is abandoned). Tables 2-5 show us that, overall, the performance of the combination of techniques was nearly equivalent to the performance of oracles alone in both stability (as we had expected) and efficiency (as we had *not* expected). We conjecture that the efficiency of oracles alone versus oracles with nogood recording is so similar because the stability resulting from oracles is so high (generally in the range of 75 to 80 percent); this means that only a relatively small number of variables in the meta-CSP need to be reassigned, so the additional pruning and heuristic power gained through the use of nogoods has little opportunity to influence efficiency. Additional experimentation is necessary to determine whether or not this is true.

Conclusions

This paper is an initial attempt to improve efficiency and stability in semi-dynamic DTPs. We have presented two approaches and examined their effectiveness. Our initial experimental results show that oracles are more effective than nogood recording at improving both efficiency and stability. We hope to continue our research by running more experiments that vary the number of time points and the ratio of time points to disjunctive temporal constraints, and by comparing the techniques on real data.

There are many possible ways to extend this work. This work only investigates restrictions on DTPs, so one obvious extension is to consider the problem of fully dynamic DTPs, which include both restrictions and relaxations. We already touched briefly on how relaxations can complicate both nogood recording and oracles. Another extension would be to look at other dynamic CSP algorithms and apply them to dynamic DTPs. For example, the local changes algorithm of (Verfaillie and Schiex 1994) strives to reuse as much of the previous solution as possible, so it might be able to improve stability, but possibly at the cost of efficiency.

Acknowledgements

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA), through the Dept. of the Interior, NBC, Acquisition Services Division, under Contract No. NBCH-D-03-0010, and on work supported by the Air Force Office of Scientific Research, under Contract No. FA9550-04-1-0043.

References

Armando, A., Castellini, C., Giunchiglia, E., and Maratea, M. 2004. A SAT-Based Decision Procedure for the Boolean Combination of Difference Constraints. *Proceedings of SAT04*.

Cormen, T., Leiserson, C., and Rivest, R. 1990. *Introduction to Algorithms*. Cambridge, Mass.: MIT Press.

Dechter, R., Meiri, I., and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49:61-95.

Hoos, H. and O'Neill, K. 2000. Stochastic Local Search Methods for Dynamic SAT—an Initial Investigation, Technical Report TR-00-01, Computer Science Department, University of British Columbia.

Muscettola, N., Nayak, P., Pell, B., and Williams, B. 1998. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence* 103(1-2):5-48.

Pollack, M.E., Brown, L., Colbry, D., McCarthy, C.E., Orosz, C., Peintner, B., Ramakrishnan, S., and Tsamardinos, I. 2003. Autominder: An Intelligent Cognitive Orthotic System for People with Memory Impairment. *Robotics and Autonomous Systems* 44:273-282.

Schiex, T., and Verfaillie, G. 1993. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools* 3(2):187-207.

Schwartz, P., and Pollack, M.E. 2004. Planning with Disjunctive Temporal Constraints. *ICAPS-04 Workshop on Integrating Planning into Scheduling*.

Smith, D., and Weld, D. 1999. Temporal Planning with Mutual Exclusion Reasoning. *Proceedings of the 16th International Joint Conference on Artificial Intelligence*.

Stergiou, K, and Koubarakis, M. 1998. Backtracking Algorithms for Disjunctions of Temporal Constraints. *15th National Conference of Artificial Intelligence (AAAI-98)*.

Stergiou, K, and Koubarakis, M. 2000. Backtracking Algorithms for Disjunctions of Temporal Constraints. *Artificial Intelligence* 120:81-117.

Tsamardinos, I., and Pollack, M. 2003. Efficient Solution Techniques for Disjunctive Temporal Reasoning Problems. *Artificial Intelligence* 151(1-2):43-90.

van Hentenryck, P., and Provost, T.L. 1991. Incremental Search in Constraint Logic Programming. *New Generation Computing* 9:257-275.

Verfaillie, G., and Schiex, T. 1994. Solution Reuse in Dynamic Constraint Satisfaction Problems. *Proceedings of AAAI-94*.