Anytime, Complete Algorithm for Finding Utilitarian Optimal Solutions to STPPs

Bart Peintner and **Martha E. Pollack**
Computer Science and Engineering
University of Michigan
Ann Arbor, MI 48109 USA
{bpeintne, pollackm}@eecs.umich.edu

Abstract

We present a simple greedy algorithm and a novel complete algorithm for finding utilitarian optimal solutions to Simple Temporal Problems with Preferences. Unlike previous algorithms, ours does not restrict preference functions to be convex. We present experimental results showing that (1) a single iteration of the greedy algorithm produces high-quality solutions, (2) multiple iterations, bounded by the square of the number of constraints, produce near-optimal solutions, and (3) our complete, memory-boundable algorithm has compelling anytime properties and outperforms a branch-and-bound algorithm.

Introduction

This paper presents a novel algorithm for finding utilitarian optimal solutions to Simple Temporal Problems with Preferences (STPPs). We show that the algorithm is complete, memory-boundable, has compelling anytime properties, and outperforms a branch-and-bound algorithm.

Recent planning and scheduling applications, e.g. (Pollack *et al.* 2002; Muscettola *et al.* 1998), have used Temporal Constraint Satisfaction Problems (TCSPs) (Dechter, Meiri, & Pearl 1991) to model and reason about temporal constraints that restrict when events are scheduled. Given a set of events and temporal constraints, TCSP algorithms find schedules: time assignments to each event that satisfy the constraints. The difficulty of finding such assignments depends on the expressive power of the constraints: the simplest form is a tractable subclass called Simple Temporal Problems (STPs).

STPs define only which assignments are acceptable; they cannot express that some assignments may be more preferable than others. For example, consider a Mars rover performing experiments under tight power restrictions: for two schedules that satisfy all constraints, the one requiring less time, and thus consuming less power, may be preferable.

To address this limitation, recent work has extended TCSPs to represent preferences (Khatib *et al.* 2001; 2003; Peintner & Pollack 2004; Morris *et al.* 2004). Preferences are represented by adding a preference function to each constraint, which defines the value of each schedule with respect to that constraint. Adding preferences changes the task from finding *any* legal schedule to finding an *optimal* schedule. Previous work focuses on two types of optimality, both defining a schedule's value as an aggregation of individual constraint values. The first type, called maximin or Weakest Link Optimality, defines a schedule's value as the *lowest* value of all individual constraints. While this aggregator allows efficient algorithms (Khatib *et al.* 2003; Peintner & Pollack 2004), it is not always appropriate. A second type, called utilitarian optimality, defines a schedule's value as the *sum* of all individual constraint values. To date, algorithms for finding utilitarian optimal solutions to STPs with Preferences (STPPs) have required convex preference functions (Morris *et al.* 2004).

In this paper, we describe the Greedy Anytime Partition algorithm for STPPs (GAPS), an iterative algorithm that does not restrict preference functions and exhibits appealing properties that make it suitable for planning and scheduling:

Anytime Finds solutions that average over 80% of optimal after a single iteration, and up to 99% of optimal after m^2 iterations, where m is the number of constraints. Also performs comparably to a previous algorithm for STPPs that handles only convex preference functions.

Complete Finds optimal solution in time that compares favorably to a branch-and-bound algorithm.

Memory-boundable Allows caller to define trade-off between space and anytime performance.

We begin by presenting a small example and background concepts. Then we describe our algorithms, report empirical results, and conclude with future work.

Example

We describe a very simple example based on the Mars rover domain (Morris *et al.* 2004) in which two events need to be optimally scheduled: the start- and end-time of a single experiment (events S and E). The experiment must begin some time after the instrument it requires becomes available (event A, set to time 0). Although the experiment can start immediately once available, it is preferable that some time separates A and S to allow the instrument to cool. This preference is expressed as C1 in Figure 1(a); the horizontal axis represents the difference between A and S, while the vertical axis represents the preference of each temporal difference.

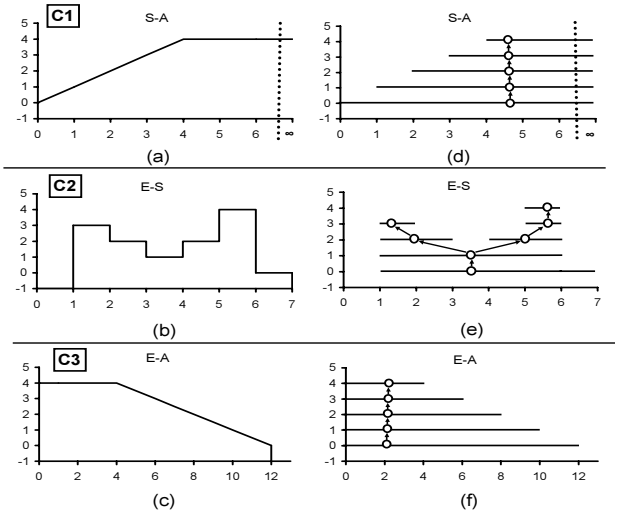

Figure 1: Example preference functions and their projections.

The scientific value of the experiment increases irregularly with time spent running it (E-S), but is mitigated by the significant power usage of the instrument. The net value is expressed by the preference function in Figure 1(b). Finally, since other experiments can begin once the current experiment ends, it is preferred that the experiment finish as early as possible. Figure 1(c) expresses this relationship.

A legal schedule in this example is one in which $E < 12$ and $1 \leq E - S < 7$; that is, one in which all preference functions map to a nonnegative value. For a schedule $\{A = 0, S = 3, E = 6\}$, constraints C1, C2, and C3 have respective preference values 3, 1, and 3, for a utilitarian value of 7. An optimal schedule, of value 10, is $\{A = 0, S = 4, E = 5\}$.

Background

An STP is a pair $\langle X, C \rangle$, where X is a set of events, and C is a set of temporal constraints of the form: $x - y \in [a, b]$, $x, y \in X$ and $a, b \in \Re$. An STP *solution* is a schedule satisfying all constraints. An STP is *consistent* if at least one solution exists. The worst-case complexity of consistency-checking algorithms is $O(|X|^3)$ time.

Because STPs only include hard constraints, and not preferences, they force the knowledge engineer to reason about a trade-off when specifying constraints: specifying more preferred values may yield better solutions but may instead over-constrain the problem; specifying less narrow bounds may reduce solution quality but makes it more likely that a solution will be found. Allowing preference functions lets the knowledge engineer avoid reasoning about this trade-off.

To extend an STP to an STPP, each constraint is assigned a preference function that maps a temporal difference between the constraint's events to a *preference value*, a quantitative measure of the value's desirability. STPP constraints have the form: $\langle x - y \in [a_k, b_k], f_k : t \in [a_k, b_k] \rightarrow \{0, \Re+\} \rangle$.

As noted above, once preference functions are added, the challenge becomes that of finding an *optimal* schedule. We evaluate a schedule with an objective function that aggregates the values from each constraint's preference function. In this paper, we use the *sum* function, which results in *utilitarian* optimal schedules.

The only previous algorithm for finding utilitarian solutions to STPPs is that of Morris *et al.* (2004). They found optimal solutions using linear programming and showed that their algorithm, WLO+, found solutions averaging 90% of optimal, and did so significantly faster than the LP approach, despite having been designed for another type of optimality. However, both WLO+ and the LP formulation restrict preference functions to be convex; thus it cannot handle preferences functions like that shown in Figure 1(b). Without this restriction, the problem is NP-hard.

Algorithms

Our algorithms borrow the idea of projecting preference functions onto hard STP constraints, allowing standard, polynomial-time algorithms for solving STPs to be leveraged (Khatib *et al.* 2001). Rather than operating directly on soft constraints, i.e., constraints with a preference function, our algorithms first convert them into *preference projections*, which represent each soft constraint using a set of hard STP constraints, i.e., those without preference functions.

To obtain preference projections, we first discretize the preference function range into a set of real values A, called a *preference value set* ($\{0, 1, 2, 3, 4\}$ in our example). Then, we project each constraint in the STPP to each level $l \in A$.

The projection of soft (STPP) constraint C_k to preference level l is a list of hard (STP) constraints representing the intervals that satisfy C_k at preference level l or higher. The right column of Figure 1 shows the preference projection for each constraint in our example. Each horizontal line segment in Figure 1(d)-(f) is an element of the preference projection. For example, the line segment at preference level 3 in Figure 1(f) denotes that if $0 \leq E - S \leq 6$, then the preference value for constraint $C3$ will be 3 or greater.

We identify each projected constraint with the 3-tuple $\langle k, l, i \rangle$, where k is the soft constraint from which it is derived, l is its preference level, and i is an index that distinguishes it from other projected constraints at the same level.

Definition 1 *A* **preference projection** *for constraint* $C_k = \langle x - y \in [a_k, b_k], f_k \rangle$ *over preference value set* $\mathbf{A}$*, is the set* $\mathcal{P}_k = \{\mathcal{P}_k[l] : l \in \mathbf{A}\}$*, where the projection at level* l *is* $\mathcal{P}_k[l] = \{C_{\langle k,l,1 \rangle}, C_{\langle k,l,2 \rangle}, \ldots, C_{\langle k,l,n \rangle}\}$*, and (1)* $C_{\langle k,l,p \rangle} = \langle x - y \in [a_p, b_p] \rangle$*, (2)* $b_p < a_{p+1}$ *for* $1 \leq p < n$ *and (3)* $\bigcup_{p=1}^{n} [a_p, b_p] = \{t | f_k(t) \geq l\}$.

In Figure 1(d)-(f), circles and arrows show how each preference projection forms a tree. The root of the tree is always a single constraint at preference level 0. A constraint's child is always one level higher and represents an interval that is a subset of the parent constraint's interval. We refer to all descendants of a constraint h as $des(h)$.

Given a set of preference projections, a hard STP called a *component STP* can be formed by selecting one STP constraint from each STPP constraint's preference projection.

Definition 2 *A* **component STP** *is a set of STP constraints,* $S = \{C_{\langle 1,l_1,i_1 \rangle}, \ldots, C_{\langle m,l_m,i_m \rangle}\}$*, where* m *is the number of STPP constraints.* S*'s utilitarian value is* $u(S) = \sum_{i=1}^{m} l_i$.

The component STP $ROOT = \{C_{\langle 1,0,1\rangle}, \ldots, C_{\langle m,0,1\rangle}\}$ contains the root of each constraint's projection tree and is the most relaxed component STP; if $ROOT$ is inconsistent, then the STPP has no solution. Conversely, the most restrictive and preferable component STP contains constraints from the highest level of each tree; if consistent, no other consistent schedule for that STPP will have a higher value.

Given these extremes, we can cast our search for an optimal schedule as a search for the highest-valued component STP H, with value $u(H)$. Because the preference values are discretized, $u(H)$ may be up to $m * \delta_A$ less than the value of the optimal schedule, where δ_A is the maximum distance between preference values in the preference value set A. When the preference functions are step functions and A contains a value for each step value, as in Figure 1(b), the error is 0. However, for smooth functions, such as that in Figure 1(c), the search may produce slightly suboptimal answers.

STPP_Greedy

Figure 2 presents STPP_Greedy, a simple algorithm for quickly finding a high-quality solution to an STPP. The algorithm searches the space of component STPs, starting with the lowest-valued, $ROOT$, and repeatedly improving its value by replacing a single constraint with one of its children. The main function, `replaceAConstraint`, picks a constraint, replaces it with one of its children, and returns the child's identifier (the 3-tuple $\langle k, l, i\rangle$) or a failure flag ($k = -1$) if no replacement is possible. If the replacement by a child leads to an inconsistent component STP, a greedy decision is made: the constraint is restored, marked as "finished" and never again chosen. In our implementation, we reduce the time spent on consistency checking (in line 5) by using the `AC-3cc` solver (Cervoni, Cesta, & Oddi 1994).

STPP_Greedy solution quality depends on the quality of the heuristic that chooses which child will replace its parent. We designed and studied several heuristics, but given space limitations, we describe only the one that performed best: the *1 step assignment window reduction* (1AWR) heuristic.

1AWR uses the "event assignment windows" maintained by AC-3cc to estimate which of all possible children will constrain the network least when propagated. The assignment windows indicate the lower and upper bound of possible assignments to each event. For example, after propagating all constraints in the $ROOT$ component STP of our example, AC-3cc would report the windows: $\{A = [0, 0], S = [0, 11], E = [1, 12]\}$. For each child, 1AWR mimics the first iteration of AC-3cc to determine how much the assignment windows of the child's two events will shrink. The child with the minimal reduction is chosen. For example, if $C3$'s child is chosen, the first step in the propagation would reduce E's interval from $[1, 12]$ to $[1, 10]$, an assignment window reduction of 2. We later show that with 1AWR, STPP_Greedy finds solutions that average above 80% of the optimal value.

STPP_Greedy($ROOT$)
1. IF $ROOT$ is inconsistent, RETURN Ø
2. cSTP ← $ROOT$
3. DO
4. $\quad\langle k, l, i\rangle \leftarrow$ `replaceAConstraint`(cSTP)
5. $\quad$IF $k \neq$ -1 AND cSTP is inconsistent
6. $\quad\quad$`markAsFinished`($C_{\langle k,l,i\rangle}$)
7. $\quad\quad$cSTP[k] ← `parentOf`($C_{\langle k,l,i\rangle}$)
8. $\quad$END IF
9. WHILE $k \neq$ -1

RETURN cSTP

Figure 2: The STPP_Greedy algorithm.

GAPS

We now show how to use STPP_Greedy to achieve an anytime, complete, and memory-boundable algorithm for finding the utilitarian-optimal solutions to unrestricted STPPs. Our algorithm, called the Greedy Anytime Partition algorithm for STPPs (GAPS), can be understood as combining pruning, greedy search, and a "divide and conquer" strategy.

At a high level, GAPS is very simple: it starts by running STPP_Greedy to find a greedy solution G (i.e. a component STP). Then, GAPS uses G to partition the entire STPP search space into $n+1$ smaller subproblems, n of which will be placed in a priority queue to be recursively solved later with GAPS, and one of which will be pruned. After this first iteration, GAPS repeats these steps on a subproblem removed from the queue, keeping track of the best solution found by each call to STPP_Greedy. When the queue empties, the best solution will be the optimal solution.

The interesting element of GAPS is how it partitions the STPP search space using the component STP G. The goal of the partition is to isolate a part of the search space that contains only component STPs of utilitarian values less than or equal to G's. Such a partition can be pruned from the search, because no solution better than G exists in that space.

We will illustrate the partition operation, which we call a *Greedy Partition* (GP), using Figure 3. Each large box in Figure 3 represents a single STPP subproblem using a set of projection trees — one for each preference projection in the problem. The trees in problem (S_0), which are labeled C1, C2, and C3, are the same trees as in Figure 1(d)-(f), i.e., the preference projections for our example problem. All other large boxes represent subproblems of (S_0) that are produced during the GP: those on the bottom row are subproblems of (S_0) that will eventually be placed in the priority queue; (S_1^3) is a subproblem of (S_0) that will be pruned; and the rest are intermediate subproblems that arise during the partition.

An STPP subproblem is a subset of the component STPs in the original STPP. In Figure 3, a component STP is *not* included in a subproblem if at least one of its components is grayed. For example, consider the boxed elements of (S_0), which represent the component STP G found by running STPP_Greedy on the original problem. G exists in all subproblems on the top row, but not in those in the bottom row, where one of G's components is grayed out in each.

The GP operation consists of m *single projection partitions* — one for each preference projection in the problem. In our example, the first will partition (S_0) into (S_0^1) and (S_1^1) using the C1 projection. To achieve the split, it first copies all preference projections *other than* C1 (i.e., C2 and C3) into the new subproblems (S_0^1) and (S_1^1). Then, projection C1

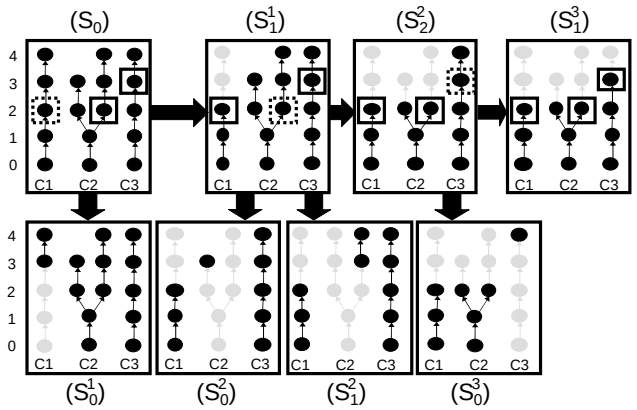

Figure 3: Greedy Partition operation used in GAPS.

is split into two pieces: subproblem (S_1^1) inherits nodes with values less than or equal to G's value in C1 (2 in our example), while (S_0^1) inherits nodes with values greater than G's value. The result is a partition in the mathematical sense: all component STPs in (S_0) that contain C1 constraints above G's value are in (S_0^1), and all others are in (S_1^1). Therefore, the optimal solution will exist in either (S_0^1) or (S_1^1).

GP's second single projection partition splits (S_1^1) into subproblems (S_0^2), (S_1^2), and (S_2^2) using projection C2. Subproblem (S_2^2) inherits the subtree consisting of constraints at and below G's level, while the rest is split into two parts, (S_0^2) and (S_1^2), because the C2 constraints above the greedy solution form two trees. Finally, the subproblem (S_2^2) is split into (S_0^3) and (S_1^3) based on constraint C3.

After the three steps, problem (S_0) is partitioned into the set of subproblems $\{(S_0^1), (S_0^2), (S_1^2), (S_0^3), (S_1^3)\}$. The optimal solution is guaranteed to reside in exactly one of these.

We can formalize this process with two definitions:

Definition 3 *A **single projection partition** of a subproblem $S = \{\mathcal{P}_k : 0 \leq k < m\}$ with respect to a single constraint $C_{\langle k,l,p\rangle}$ is a set of subproblems, $\Upsilon(S|C_{\langle k,l,p\rangle}) = \{S_0, S_1, \ldots, S_n\}$, where n is the size of $H = \{C_{\langle k,r,p\rangle} \in \mathcal{P}_k : r = l+1\}$, H_i is the i^{th} element of H and*

- $S_n = S \backslash \mathcal{P}_k \cup \{C_{\langle k,r,p\rangle} \in \mathcal{P}_k : r \leq l\}$
- $\forall_{i<n} S_i = S \backslash \mathcal{P}_k \cup H_i \cup des(H_i)$.

The first bullet refers to the subproblem placed on the top row of Figure 3, while the second bullet refers to those added to the bottom row. Using set algebra, it can be easily proved that single projection partitions are true partitions; hence, no component STP is "lost" in the process.

Definition 4 *A **Greedy Partition** of a subproblem $S = \{\mathcal{P}_k : 0 \leq k < m\}$ with respect to component STP $G_{[0,m]} = \{C_{\langle k,l,p\rangle} : 0 \leq k < m\}$ is a set of subproblems $\Psi(S|G_{[0,m]})$, which is recursively defined as $\Psi(S|G_{[i,m]}) = \Upsilon(S|G_i) \backslash S_n \cup \Psi(S_n|G_{[i+1,m]})$, where the base case is $\Psi(S_n|G_{[m,m]}) = S_n$.*

Applying Definition 4 to our example, S refers to (S_0) and $\Psi(S|G_{[0,m]})$ produces $\{(S_0^1), (S_0^2), (S_1^2), (S_0^3), (S_1^3)\}$.

GAPS(STPP**)**
1. S ← `project`(STPP)
2. ssQ ← priority queue of subspaces, initially empty
3. best ← Ø
4. REPEAT
5. G ← `STPP_Greedy`(S)
6. IF u(G) > u(best) best ← G
7. `addToQueue`(ssQ, `GreedyPartition`(S, G))
8. S ← `getNextSS`(ssQ);
9. WHILE (ssQ not empty)

RETURN best

Figure 4: The STPP algorithm GAPS.

Once the Greedy Partition is made, the last subproblem created can be pruned. In our example, the search space (S_1^3) consists only of component STPs with value $u(G)$ (2+2+3=7) or less, since at each stage of its formation, we kept only the projected constraints at or below G's level. Thus, because no component STP in the space can exceed the value of G, (S_1^3) can be pruned from the search. Only subproblems (S_0^1), (S_0^2), (S_1^2), and (S_0^3) are inserted into the queue to be fully searched in later iterations.

Figure 4 presents the core GAPS algorithm[1]. We have discussed every part except for line 8, which retrieves the next subproblem from the priority queue to prepare for the next iteration. The choice of which subproblem to solve next impacts both anytime performance and space requirements. By default, we prioritize based on an upper bound, which is the highest valued component STP (consistent or not) in the subproblem. This priority function ensures that subproblems with the most potential are explored first. Unfortunately, this strategy allows the set of queued subproblems to grow quite large. An alternative is to recursively explore each child of a subproblem before moving on to the next (similar to a depth-first search). Since the maximum number of recursive partitions for a search space is $m*(|A|-1)$ and each partition can produce at most $m+(n-1)$ children (where m is the number of constraints and n is the maximum i for any $C_{\langle k,l,i\rangle}$), the maximum number of subproblems queued is $m^2 * (|A|-1)$ (n disappears because $n \ll m$).

Of course, the two strategies can be combined: given a bound of M nodes, the first $M-(m^2*(|A|-1))$ nodes can be inserted using the upper bound priority function. Anytime the number of nodes in memory exceeds this threshold, new subproblems are inserted using the depth-first priority function. Anytime the number of nodes drops below this threshold, upper bound priority is resumed. Therefore, GAPS's caller can effectively manage a space/time trade-off.

GAPS is complete because of two properties. First, the set of subproblems produced by GP is a true partition, meaning, every component STP in the original problem exists in one of the new subproblems. Second, every component STP that is pruned is guaranteed to have value equal to or less than the greedy solution. Thus, the optimal solution is never pruned. A formal proof is omitted due to space restrictions.

[1]Figure 4 hides many details needed for practical implementation. For example, subproblems can be stored compactly in a tree.

Empirical Evaluation

To test GAPS, we randomly generated STPPs and compared the solution quality and running time of GAPS to a branch and bound (BB) variant, called the Russian Doll algorithm (Dechter 2003). We compared to BB because it is a popular approach for solving constraint optimization problems and because it is complete, anytime, and handles non-convex functions. We also tested GAPS against WLO+ on STPPs with convex preference functions.

For the STPP problem, BB works by maintaining a partial STP: it starts with the root of the first preference projection, then adds the root from the next projection, and so on until one of three possibilities occur: (1) the partial STP becomes inconsistent; (2) an upper bound calculation proves that the partial STP cannot extend to a STP with greater value than the current best; or (3) the partial STP contains an element of every projection (a full STP). In the third case, the full STP is stored as "best". In all cases, the algorithm backtracks by removing the last element from the STP then trying a new projected constraint from the previous STPP constraint.

The upper bound calculation after adding an element from the i^{th} projection is made by adding the value of the partial STP (containing constraints 0 through i) to the upper bound of constraints $i+1$ to $m-1$ (there are m constraints). The naive upper bound calculation simply uses the level of the highest-valued constraint for each projection. However, we do better by modifying the well known Russian Doll technique, which first calculates the optimal solution for STPP constraint $m-1$, then calculates the optimal solution for STPP constraints $m-2$ to $m-1$, and so on until the optimal solution is found. This dynamic programming technique ensures that after assigning constraint i, the upper bound calculation for constraints $i+1$ to $m-1$ is optimally tight.

Setup

We investigated three variations of the GAPS algorithm: GAPS-1, which runs only a single greedy iteration; GAPS-m^2, which runs m^2 iterations; and GAPS, which runs until completion or until a 10-minute time threshold is reached. The same threshold was applied to BB, which returns the best solution found when the threshold is met.

We ran five tests: the first fixed the number of events to 10 and varied the constraint density from 1 to 3.5, where density is the ratio of constraints to events. The second test fixed the constraint density at 2 and varied the number of events from 5 to 12 (limited by the efficiency of the complete algorithms). To measure how the algorithms fared with larger problems, a third test varied the number of events from 10 to 90 but omitted BB.

The first three tests operated on networks restricted to semi-convex preference functions. A semi-convex function f is one in which the set $\{x : f(x) > l\}$ forms at most a single interval for any level l. This restriction ensured that all preference projection trees were chains, as in constraints C1 and C3 in our example. We made this restriction because there is particular interest in this restriction in previous STPP work (Khatib *et al.* 2001). To show GAPS performs well without this restriction, our fourth test repeated the second, but produced networks that contained, on average, about 1 branch in the projection tree for every 5 intervals.

The shapes of the preference functions in Tests 1-4 were determined by a *reduction factor* chosen from an interval [*redLB*, *redUB*]. This factor represents the fraction of a projected constraint's interval that is covered by its child constraints. A factor close to 1 results in functions with steep slopes; one close to 0 results in shallow slopes and fewer levels. All tests used the interval [.5, .9] for the reduction factor during generation. We arrived at this value by performing a set of experiments that compared the solution quality of STPP_Greedy using several different reduction factor intervals. For the interval [.5,.9], STPP_Greedy fared the *worst*. The maximum number of preference levels was 10.

The fifth test compared WLO+ solution quality to the solution quality produced by GAPS when run for an equal amount of time. The preference functions in the randomly generated STPPs were convex (as required by WLO+) and piece-wise linear; the number of linear segments ranged from 3 to 6. We tested STPPs with 100 events and constraint density ranging from 1 to 7.

The third test averaged the results from 100 trials, while the others averaged 200 trials. All algorithms were written in Java and run on a Pentium 4 3Ghz WindowsXP machine.

Results

Figure 5 shows the results of Tests 1-4. The plots show the average normalized utilitarian value for the solutions from each algorithm (the left axis) and the running times in seconds for some of the algorithms (the right axis of tests 1 and 3). The normalized utilitarian value is computed by dividing the utilitarian value by the number of constraints. The value itself (shown on the left axis) has little significance; the relative values among algorithms interest us most.

First we discuss the relative performance of GAPS and BB. In Test 1, both had equal solution quality for small problems, but GAPS finished more quickly. As density increased, both required similar time, but BB's solution quality fell quickly relative to GAPS. Thus, for problems in which both found the optimal solution, GAPS found it faster, and for harder problems GAPS finds better solutions than BB in a given amount of time. These trends also hold in Tests 2 and 4, although we omit run-times for readability.

Comparing Tests 2 and 4, we can see that although allowing unrestricted functions increased the variance, the general relationship between GAPS and BB changes only slightly, with BB faring slightly worse in the unrestricted case. In summary, GAPS exceeded BB in both optimal solution time and anytime performance.

Next, we discuss the anytime performance of GAPS using GAPS-1 and GAPS-m^2. All four tests show that GAPS achieves most of its gains in the initial iterations. GAPS-1 found solutions averaging $> 80\%$ of GAPS's value, while GAPS-m^2 averaged between 96.5% and 99%. Not shown is the result that running m iterations averaged $> 90\%$.

Test 3 shows the running time for GAPS-1 and GAPS-m^2. GAPS-1 needed about one second for STPPs with 50 events and 100 constraints, while GAPS-m^2 required 100 seconds. This shows running time does not scale directly

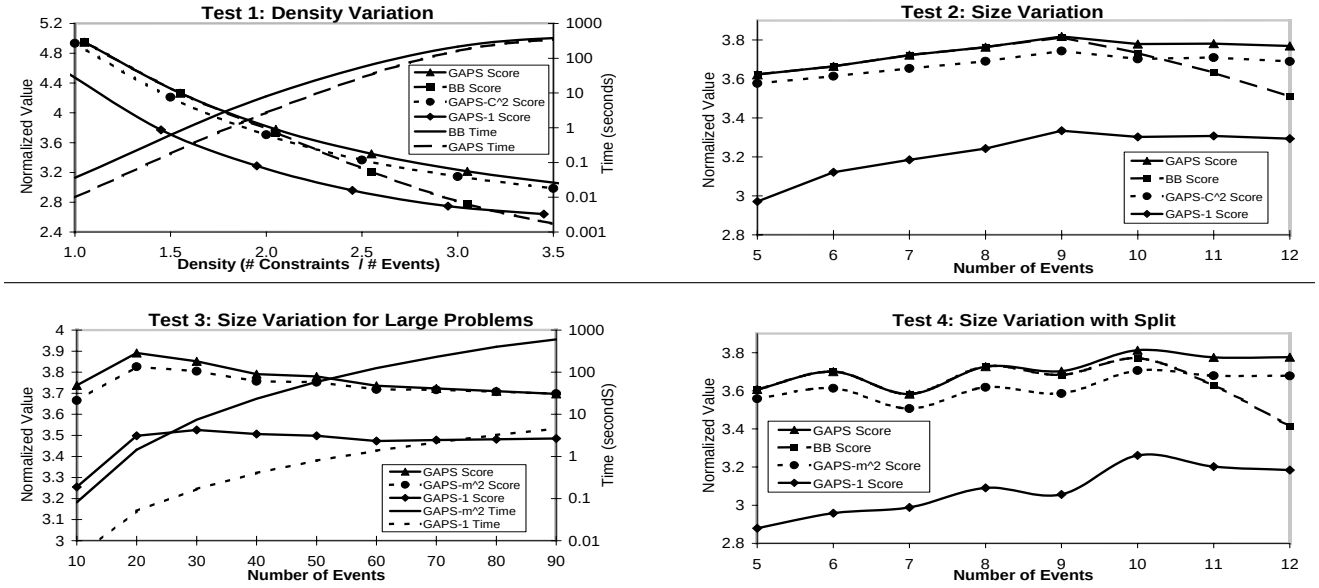

Figure 5: Solution quality and running times for the first four tests.

with the number of iterations: as expected, running time of each iteration decreases with the size of the subproblem.

The fifth test showed that GAPS found slightly better solutions than WLO+ when run for equal time. The following table shows average solution quality as the number of constraints increases:

# Constraints	100	200	300	400	500	600	700
WLO+	89	142	195	251	310	371	430
GAPS	87	147	203	260	319	379	437

The most important result lies in the anytime performance of the GAPS algorithm. That solutions 80-99% of optimal can be found so quickly is important for planning and scheduling applications that use STPPs.

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHD030010 and the Air Force Office of Scientific Research under Contract No. FA9550-04-1-0043. Any opinions, findings, conclusions or recommendations are those of the authors and do not necessarily reflect the view of DARPA, the Department of Interior-National Business Center, or the United States Air Force.

Conclusion

This paper presented a novel algorithm for finding utilitarian optimal solutions to Simple Temporal Problems with Preferences (STPPs). We showed that the algorithm is complete, memory-boundable, has compelling anytime properties, and outperforms WLO+ and a branch-and-bound algorithm.

In the near future, we plan to use GAPS as the core of an algorithm for finding the optimal utilitarian solutions to Disjunctive Temporal Problems with Preferences, a more expressive model than STPPs. We are also working to incorporate non-temporal constraints into the STPP model.

References

Cervoni, R.; Cesta, A.; and Oddi, A. 1994. Managing dynamic temporal constraint networks. In *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems*, 13–18.

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

Dechter, R. 2003. *Constraint Processing.* San Francisco, CA 94104: Morgan Kaufmann.

Khatib, L.; Morris, P.; Morris, R.; and Rossi, F. 2001. Temporal constraint reasoning with preferences. *17th International Joint Conf. on Artificial Intelligence* 1:322–327.

Khatib, L.; Morris, P.; Morris, R.; and Venable, K. B. 2003. Tractable pareto optimal optimization of temporal preferences. *18th International Joint Conference on Artificial Intelligence* 1:1289–1294.

Morris, P.; Morris, R.; Khatib, L.; Ramakrishnan, S.; and Bachmann, A. 2004. Strategies for global optimization of temporal preferences. In *10th International Conf. on Principles and Practice of Constraint Programming*, 408–422.

Muscettola, N.; Nayak, P. P.; Pell, B.; and Williams, B. C. 1998. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence* 103(1-2):5–47.

Peintner, B., and Pollack, M. E. 2004. Low-cost addition of preferences to DTPs and TCSPs. In *19th National Conference on Artificial Intelligence*, 723–728.

Pollack, M. E.; McCarthy, C. E.; Ramakrishnan, S.; Tsamardinos, I.; Brown, L.; Carrion, S.; Colbry, D.; Orosz, C.; and Peintner, B. 2002. Autominder: A planning, monitoring, and reminding assistive agent. *7th International Conf. on Intelligent Autonomous Systems* 7.