

# Solving Overconstrained Temporal Reasoning Problems

Matthew Beaumont, Abdul Sattar, Michael Maher, and John Thornton

School of Computing & Information Technology,  
Griffith University,  
Nathan, QLD 4111, Brisbane, AUSTRALIA

**Abstract.** Representing and reasoning with temporal information is an essential part of many tasks in AI such as scheduling, planning and natural language processing. Two influential frameworks for representing temporal information are: interval algebra and point algebra [1, 8]. Given a knowledge-base consisting of temporal relations, the main reasoning problem is to determine whether this knowledge-base is satisfiable, i.e., there is a scenario which is consistent with the information provided. However, when a given set of temporal relations is unsatisfiable, no further reasoning is performed. We argue that many real world problems are inherently overconstrained, and we can not just ignore them, we must address them. This paper investigates approaches for handling overconstrainedness in temporal reasoning. We adapt a well studied notion of *partial satisfaction* to define *partial scenarios*: an optimal partial solution. We propose two reasoning procedures for computing an optimal partial solution to a problem or a complete solution if it exists.

## 1 Introduction

Temporal reasoning is a vital task in many areas, such as planning [2], scheduling [5] and natural language processing [6]. Currently the main focus of research has been on how to represent temporal information and how to gain a complete solution from a problem. How the information is represented depends on the type of temporal reasoning that is needed.

There are two different ways in which we can reason about a temporal problem. The reasoning method that is chosen depends on the information available. When a problem is presented with only qualitative information, that is information about how events are ordered with other events, Qualitative Temporal Reasoning is performed. From the sentence "Fred drank his coffee while he ate his breakfast" we can only gather information about the relative timing of the two events. On the other hand information can be presented in as quantitative information, that is information about when certain events can or do happen. For example, Fred ate his breakfast at 7:35am and drank his coffee at 7:40am. For this paper we only deal with qualitative information.

Currently research has only been aimed at finding a complete solution or determining that a problem has a solution [1, 8, 4]. If the problem is not solvable

then only an error is provided. However in many situations simply determining that the problem has no solution is not enough. What is needed in this situation is a partial solution, where some of the constraints or variables have been weakened or removed to allow a solution to be found.

While there has been no research on finding a partial solution to an overconstrained temporal reasoning problem, there has been research done on finding partial solutions to overconstrained constraint satisfaction problems (OCSP). One such approach is Partial Constraint Satisfaction [3]. Partial Constraint Satisfaction takes an overconstrained problem and obtains a partial solution by selectively choosing variables or constraints to either remove or weaken. This is done in such a way as to minimize the total number of variables or constraints that are removed or weakened and leads to an optimal partial solution.

In this paper we define two methods for finding a solution to an overconstrained Temporal Reasoning problem. The first method uses a standard brute force approach with forward checking/pruning capabilities. The second method also uses a brute force strategy but replaces forward checking/pruning with a cost function that can revise previous decisions at each step of the search. Both methods provide the ability to find an optimal partial solution or a complete solution if one exists.

In sections 2 and 3 we give the relevant background information for both Temporal Reasoning and Partial Constraint Satisfaction. Section 4 introduces both methods and explains in detail how they work. We also present some preliminary experimental results in Section 5.

## 2 Interval and Point Algebra

The way in which qualitative temporal information is represented plays a key role in efficiently finding a solution to the problem or determining that no solution exists. Two representation schemes are Allen's Interval Algebra [1] and Vilain and Kautz's Point Algebra [8].

Interval algebra (IA) represents events as intervals in time. Each interval has a start and an end point represented as an ordered pair  $(S, E)$  where  $S < E$ . The relation between two fixed intervals can consist of one of the 13 atomic interval relations. The set of all 13 atomic interval relations is represented by  $I$  and is shown in table 1.

To represent indefinite information about relations between non-fixed intervals can be achieved by allowing relations to be disjunctions of any of the atomic relations from the set  $I$ . By allowing disjunctions of the 13 atomic relations we can construct the set  $A$  containing all  $2^{13}$  possible binary relations including the empty relation  $\emptyset$  and the no information relation  $I$ . To complete the algebra Allen also defined 4 interval operations over the set  $A$ : intersection, union, inverse and composition. The operations and their definitions are shown in table 2.

A temporal problem expressed with IA can be easily be shown as a temporal constraint graph [4]. In a temporal constraint graph nodes represent intervals

Relation	Symbol	Semantics	Symbol	Relation
X before Y	<		>	Y after X
X meets Y	m		mi	Y meet-by X
X overlaps Y	o		oi	Y overlapped-by X
X during Y	d		di	Y contains X
X starts Y	s		si	Y started-by X
X finishes Y	f		fi	Y finished-by X
X equals Y	=		=	Y equals X

**Table 1.** The set  $I$  of all 13 atomic relations.

Operation	Symbol	Formal Definition
Intersection	$\cap$	$\forall x, y \ x A_1 \cap A_2 y \text{ iff } x A_1 y \wedge x A_2 y$
Union	$\cup$	$\forall x, y \ x (A_1 \cup A_2) y \text{ iff } x A_1 y \vee x A_2 y$
Inverse	$\sim$	$\forall x, y \ x \sim y \text{ iff } y A x$
Composition	$\circ$	$\forall x, y \ x (A_1 \circ A_2) y \text{ iff } \exists z \ x A_1 z \wedge z A_2 y$

**Table 2.** The 4 interval operations and their definitions.

and the arcs between nodes are labeled with interval relations. Such a graph can easily be represented as a matrix  $M$  of size  $n * n$  where  $n$  is the number of intervals in the problem. Every element of the matrix contains an interval relation from the set  $A$  of all possible interval relations with two restrictions. For the elements  $M_{ii}$  the interval relation is always = and  $M_{ji} = M_{ij}^{-1}$ .

One of the key reasoning tasks in IA is being able to determine if a problem is satisfiable. A problem is satisfiable if we can assign a value to each interval's start and end point such that all the interval relations are satisfied. Satisfiability can be determined by the use of the path-consistency method [1]. The method simply keeps computing for all  $a, b, c$  of the matrix  $M$ :

$$M_{ac} = M_{ac} \cap (M_{ab} \circ M_{bc})$$

until there is no change in  $M_{ac}$ . A matrix  $M$  is said to be path-consistent when no elements are the empty set  $\emptyset$  and there is no change. However as shown by Allen [1] path-consistency does not imply Satisfiability for interval algebra. Infact determining Satisfiability for IA is NP-Hard [8] and a backtracking algorithm must be used with path consistency to determine satisfiability.

Point Algebra (PA) differs from IA in that events in time are only represented as a point instead of an interval. By representing events as points, the relations between events are reduced to three possibilities  $\{<, =, >\}$ . The set  $P = \{\emptyset, <$

,  $\leq$ ,  $=$ ,  $>$ ,  $\geq$ ,  $\neq$ ,  $?$  contains every possible relation between events. The relation  $? = \{<, =, >\}$ , means no information is known about that relation.

The advantage of PA is that it is more computationally attractive than IA in that the path-consistency method ensures Satisfiability [8]. It is also possible to encode some relations from IA into PA [8]. However a major disadvantage is that expressive power is lost by representing time as points.

While the full set of interval relations  $A$  is NP-Hard in computing Satisfiability there exists subsets of  $A$  that require only polynomial time. The  $SA_c$  subset defined by Peter Van Beek and Robin Cohen [7] contains all the relations from  $A$  that can be converted to PA. Another popular subset is the ORD-Horn maximal subset  $H$  which contains all the relations from  $A$  that provide Satisfiability for the path-consistency method [4]. The ORD-Horn subset also includes all the relations in  $SA_c$  such that  $SA_c \subset H$ .

### 3 Partial Constraint Satisfaction

Partial Constraint Satisfaction (PCS) [3] is the process of finding values for a subset of the variables in a problem that satisfies a subset of the constraints. A partial solution is desirable in several cases:

- The problem is overconstrained and as such has no solution.
- The problem is computationally too large to find a solution in a reasonable amount of time.
- The problem has to be solved within fixed resource bounds.
- The problem is being solved in a real-time environment where it is necessary to be able to report the current best solution found at anytime.

There are several methods that can be used to obtain a partial solution [3]:

1. Remove variables from the problem.
2. Remove constraints from the problem.
3. Weaken constraints in a problem.
4. Widening a domain of a variable to include extra values.

Removing a variable from the problem is a very drastic approach to obtain a partial solution. By removing a variable, all the constraints associated with that variable are also removed. Conversely if, when removing constraints, a variable is left with no constraints, then this is effectively the same as removing that variable. Weakening a constraint to the point where that constraint no longer constrains the variable effectively removes that constraint from the problem. From this we can see that methods 1 and 2 are really special instances of method 3. The fourth method however has no relation to the other methods. If a variable's domain is widened to the extent that it includes all possible values, the constraints on that variable can still make it impossible to assign a value to that variable. So even if the domain of a variable is widened to include all possibilities it is still not the same as removing that variable.

No matter the method that is chosen to find a partial solution there is still the question of what constitutes an optimum partial solution. The most common and easy way is to simply count the number of variables/constraints removed or the number of domains/constraints weakened. The solution that provides the minimal count is then considered optimal. An optimal solution with a count of 0 equates to a fully consistent solution. The cost of obtaining an optimal solution is equal to the count in this instance.

To gain a partial solution the most common form of algorithms used are backtracking algorithms. Conventionally a backtracking algorithm will backtrack as soon as an inconsistency is encountered. However for PCS the algorithm continues and records the fact that an inconsistency was encountered. When a partial solution is found the cost of that solution is the number of inconsistencies that were recorded. The algorithm then backtracks and continues to find alternate solutions with potentially lower costs.

## 4 Temporal Constraint Partial Satisfaction

Temporal reasoning is often a vital task for many applications. However whilst current temporal reasoning algorithms are relatively fast and efficient they have one major problem. Many applications, such as scheduling, require a solution to the problem presented even when the problem is overconstrained. Applying the current temporal reasoning algorithms will only identify that the problem is indeed overconstrained and as such has no solution. So the problem is that whilst the algorithms detect the overconstrained nature of the problem, they do not provide any form of a partial solution. To solve this problem we introduce two algorithms for finding partial solutions.

### 4.1 Method 1

The first method uses a standard branch and bound search with forward checking/pruning to gain an optimal partial solution. The branch and bound algorithm used is slightly modified from the one presented earlier.

The algorithm starts by initializing a dummy network such that all relations in this dummy network are the relation  $I$ . This dummy network is then passed to the branch and bound algorithm and the search begins.

First a relation is chosen, this relation is then divided into two sets, a consistent set  $CS$  and an inconsistent set  $IS$ . The set  $CS$  contains only relations that appear in both the original relation and what remains in the dummy networks relation. For example, if the original had the relation  $\{<, m, mi, s\}$  and the dummy relation  $\{<, mi, f, fi, >\}$  then the set  $CS$  would be  $\{<, mi\}$ . The set  $IS$  contains the rest of the relations not in  $CS$ , which would be  $\{f, fi, >\}$ . After this is done, a single relation is chosen first from the set  $CS$  and instantiated in the dummy network. The Path Consistency algorithm is then called to propagate the effects of this instantiation. In the event that the branch and bound algorithm backtracks to this point or the Path Consistency call fails, another

atomic relation is chosen. If all relations from the set  $CS$  have been tried then atomic relations are chosen from the set  $IS$ . However when a relation from the set  $IS$  is chosen a cost count is incremented to reflect that a relation was chosen in conflict with the originally desired relations.

If the Path Consistency call was successful then another relation is chosen and the process begins again. At anytime if the cost of the current path exceeds the current best cost then backtracking occurs to a point where the cost is lower than the best cost and processing begun again. When all relations are exhausted the best result is returned as the optimal solution.

**Input:**       Original: The original network  
                   Dummy: A dummy network  
                   Cost

**Method1**

```

1. Begin
2.   If Cost >= BestCost then backtrack
3.   If PathConsistent(Dummy) fails then backtrack
4.   If there are still relations to process in Dummy then
5.     begin
6.       get next relation (X, Y) from Dummy
7.       CS = Dummy[X, Y] ∩ Original[X, Y]
8.       IS = Dummy[X, Y] − CS
9.       for all i in CS do
10.        begin
11.          instantiate Dummy[X, Y] to i
12.          Method1(Original, Dummy, Cost)
13.        end
14.       for all i in IS do
15.        begin
16.          instantiate Dummy[X, Y] to i
17.          Method1(Original, Dummy, Cost + 1)
18.        end
19.       end
20.     else
21.       begin
22.         Record Dummy as the best solution found so far
23.         BestCost = Cost
24.       end
25.   End

```

**4.2 Method 2**

The second method, much like the first, uses a branch and bound algorithm to control the search. However, unlike the first method, no forward checking/pruning is done as it is incompatible with how this method finds solutions. For this

method the cost is only computed at the end of a search path. At each step in the search path an approximate cost is found based on how many relations need to potentially be changed to make the network consistent. With this approximate value a decision is made as to whether to proceed on this path or abandon it. This requires two additional algorithms, a Real Cost algorithm and an Approximate Cost algorithm.

**Approximate Cost Function** At each level of the search it is required that we be able to judge the cost of partially explored solution. The ApproximateCost function finds an approximate cost that is always equal to or less than the real cost of the partially explored solution. The reason for using an approximate cost function instead of finding the real cost is that until all relations are atomic it is almost impossible to find an absolute cost at that point. The reason for this is that finding every inconsistency at this point would require a separate NP-Hard search and then an additional search to find the best cost.

To calculate the approximate cost the first step is to determine a lower bound of how many triples are inconsistent. A triple is a set of any three nodes from the problem. To determine if a triple  $T = (A, B, C)$  is inconsistent, we determine if:  $M_{AC} \cap (M_{AB} \circ M_{BC}) \neq \emptyset$ . It is enough to test the path  $(A, B, C)$  to determine an inconsistency. Computing  $(B, C, A)$  and  $(B, A, C)$  is unnecessary due to the fact that if the path  $(A, B, C)$  is consistent then there is an atomic relation  $X$  in  $M_{AB}$  and  $Y$  in  $M_{BC}$  that make some or all atomic relations in  $M_{AC}$  consistent. Now if we take the composition of  $M_{AC}$  and the inverse of  $Y$ , the resulting allowed relations will include  $A$ . This is because given any three atomic relations  $N, P, Q$  that are path consistent then  $Q \in (N \circ P)$ ,  $N \in (Q \circ P^{-1})$  and  $P \in (N^{-1} \circ Q)$ .

When a triple is determined as inconsistent it is added to a list that records all inconsistent triples. Each time a triple is added to the list a count for each relation in the triple is incremented. The counts for all relations are stored in an occurrence matrix  $O$ , with each element of  $O$  starting at 0. For example, if the triple  $(A, B, C)$ , is inconsistent then  $O_{AB}$ ,  $O_{BC}$  and  $O_{AC}$  are all incremented by 1 to represent the fact that each of those relations occurred in an inconsistency.

**Input:** Network M  
**Output:** A List containing all inconsistent triples  
 A matrix O recording the occurrence count for each relation

**DetermineInconsistencies**

1. **Begin**
2.     **For** A = 1 to size(M) - 2 **do**
3.         **For** B = A + 1 to size(M) - 1 **do**
4.             **For** C = B + 1 to size(M) **do**
5.                 **begin**
6.                     **If**  $(M_{AC} \cap (M_{AB} \circ M_{BC})) = \emptyset$  **do**
7.                         **begin**
8.                             add (A,B,C) to List
9.                             increment  $O_{AB}$ ,  $O_{BC}$  and  $O_{AC}$  by 1

```

10.         end
11.     end
12. return (List, O)
13. End

```

Once the list of inconsistencies is determined the list is then processed to find an approximate number of relations to weaken to remove all inconsistencies. In simplified terms the algorithm simply takes a triple from the inconsistency list and tries each relation one at a time, effectively performing a brute force search. However there are some special circumstances which allows the algorithm to be more efficient.

The first situation occurs when every relation in a triple occurs only once. In this case it does not matter which relation is chosen as no other triple will be removed from the list. In this case the cost is incremented by 1 and processing continues. Lines 9-13 of the following code handle this case.

The second situation is when a triple is chosen that contains a relation that has already been selected. In this case the occurrence matrix is reduced by 1 for each relation in the triple and the cost remains the same. Lines 14-20 of the following code handle this case.

The last case is when a triple contains some relations that only occur once. In this case the relations that only occur once are simply ignored as choosing them will affect no other triples and therefore provide no possibility of offering a lower approximate cost. Line 23 is used to check for and handle this case.

**Input:**       List of inconsistencies  
                   An occurrence matrix  $O$   
                   Cost  
                   BestCost

**ApproxCost**

```

1.  Begin
2.  If Cost >= BestCost then backtrack
3.  If there are no more triples left in List then
4.  begin
5.      BestCost = Cost
6.      backtrack
7.  end
8.  get and remove the next triple (A,B,C) from List
9.  If  $O_{AB}$  and  $O_{AC}$  and  $O_{BC}$  all equal 1 then
10. begin
11.     ApproxCost(List, O, Cost + 1, BestCost)
12.     backtrack
13. end
14. If  $O_{AB}$  or  $O_{AC}$  or  $O_{BC}$  <= 0 then
15. begin
16.     decrement all three relations in the occurrence matrix O by 1

```

```

17.   ApproxCost(List, O, Cost, BestCost)
18.   increment all three relations in the occurrence matrix O by 1
19.   backtrack
20. end
21. For each relation  $R$  in {AB, AC, BC} do
22. begin
23.   If  $O_R \neq 1$  then
24.   begin
25.     TVal =  $O_R$ 
26.     decrement  $O_{AB}, O_{AC}, O_{BC}$  by 1
27.     set  $O_R$  to 0
28.     ApproxCost(List, O, Cost + 1, BestCost)
29.     increment  $O_{AB}, O_{AC}, O_{BC}$  by 1
30.      $O_R =$  TVal
31.   end
32. end
33. End

```

The ApproximateCost function is responsible for calling DetermineInconsistencies and then passing its results to ApproxCost.

**Input:** Network M  
CurrentBestCost the current BestCost value

**Output:** Cost  
**ApproximateCost**

```

1. Begin
2.   (List, O) = DetermineInconsistencies(Network)
3.   BestCost = CurrentBestCost
4.   ApproxCost(List, O, 0, BestCost)
5.   return BestCost
6. End

```

**Real Cost Function** At the end of a search, when all relations are atomic, the real cost of that solution can be determined. Unlike the ApproximateCost algorithm, RealCost returns not only a cost but also a consistent network. The question arises however of why it is not possible to use the ApproximateCost algorithm to determine the real cost when all relations are atomic? When presented with the network in Figure 1 it is possible for ApproximateCost to work out a minimal cost that does not provide a consistent network. In this network the cost of solving it is 2, however if the relations chosen are R(A,D) and R(B,C) then this still does not provide a solution as no value can be assigned to those relations together to make them consistent.

To handle this problem it is necessary to perform a full PathConsistency check at the end of a search. Furthermore it is also necessary to include relations

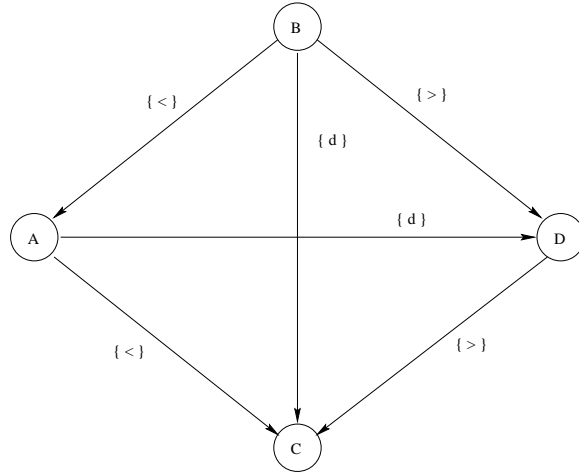


Fig. 1.

with an occurrence of 1 in the search, which impacts the performance greatly. Another problem can also arise that is related to the above network and that occurs when relation  $R(A,D)$  and  $R(B,C)$  are both the relation  $I$ . In this case the real cost algorithm will never find a solution since no inconsistencies are reported by the DetermineInconsistencies algorithm. This problem is handled by allowing the search path to extend into these relations and thus allowing real cost to function properly. Unfortunately this also results in an large increase in the search space.

B Finding the real cost of a network is similar to finding the approximate cost in that we process a list of inconsistencies to find the least number of relations to change to remove all inconsistencies. However we can no longer make use of all the special circumstances used in the approximate cost algorithm and some extra processing is also required to verify that the solution found is consistent.

The only special circumstance that can be kept is when one of the relations in a triple has an occurrence of 0 or less. Like before these triples are ignored as they are already solved. However we must record any relation in that triple that has an occurrence greater than 0 in the Removed list. This is due to the possibility that we may remove a relation from consideration that needs to be weakened to gain the optimal cost. Lines 10-18 of the following code handle this circumstance.

All other triples are processed normally and relations that have an occurrence of 1 are also treated the same as other relations. Since the solution found is required to be consistent it is possible that selecting one relation over another, where both have an occurrence of 0, could result in the final solution still being inconsistent. All relations that are considered here are marked as occurring in the search path. Lines 19-32 of the following code process this situation.

When there are no more triples left the relations that occur in the Removed list and are marked are then removed from the Removed list. The Removed list now only contains those relations that have absolutely no chance of being selected at an earlier stage. The Removed list is then passed to the ProcessRemoved algorithm which is responsible for finding the final cost and solution. Lines 3-8 of the following code handle this situation.

**Input:**        Network M  
                  List of inconsistencies  
                  An occurrence matrix O  
                  NewNet a place to store the best solution  
                  Cost  
                  BestCost  
                  Removed a list of relations

**RCost**

1. **Begin**
2.   **If** Cost  $\geq$  BestCost then **backtrack**
3.   **If** there are no more triples left in List **then**
4.   **begin**
5.     remove all the relations from Removed that have been marked
6.     ProcessRemoved(M, NewNet, Removed, Cost, BestCost)
7.     **backtrack**
8.   **end**
9.   get and remove the next triple (A,B,C) from List
10. **If**  $O_{AB}$  or  $O_{AC}$  or  $O_{BC} \leq 0$  **then**
11. **begin**
12.    decrement all three relations in the occurrence matrix O by 1
13.    add the relations that have an occurrence  $> 0$  to Removed
14.    RCost(M, List, O, NewNet, Cost, BestCost, Removed)
15.    remove the relations added to Removed
16.    increment all three relations in the occurrence matrix O by 1
17.    **backtrack**
18. **end**
19. **For** every relation  $R$  in {AB, AC, BC} **do**
20. **begin**
21.    TRel =  $M_R$
22.    TVal =  $O_R$
23.    decrement  $O_{AB}$ ,  $O_{AC}$ ,  $O_{BC}$  by 1
24.    mark all three relations (AB, AC, BC)
25.    set  $O_R$  to 0
26.    set  $M_R$  to the relation  $I$
27.    RCost(M, List, O, NewNet, Cost + 1, BestCost, Removed)
28.     $M_R = \text{TRel}$
29.    increment  $O_{AB}$ ,  $O_{AC}$ ,  $O_{BC}$  by 1
30.     $O_R = \text{TVal}$
31.    unmark all three relations

32. **end**
33. **End**

The process of marking a relation is an incremental mark and is not simply a boolean value. Any relation that is marked indicates that it has no possibility of being excluded from a search.

When there are no more triples in *List* the function *ProcessRemoved* will be called to handle a rare occasion which could otherwise result in the best cost not being found. The problem occurs when a triple is removed where one of the relations has an occurrence of 0 or less. This makes it possible for a relation that should be weakened to gain the best cost to be excluded from a search. The *ProcessRemoved* algorithm initially checks to see if the current solution is consistent, if it is then the relations in the *Removed* list are not processed. If the solution is not consistent then one or more of the relations in the *Removed* list need to be weakened to allow a solution. Line 6 checks consistency by calling *PathConsistent* which checks that the supplied network is path-consistent.

**Input:**       Network M  
                   NewNet a place to store the best solution  
                   Removed a list of removed triples  
                   Cost  
                   BestCost

**ProcessRemoved**

1. **Begin**
2.   **If** Cost  $\geq$  BestCost **then backtrack**
3.   **If** Removed is empty **then**
4.    **begin**
5.     TemporaryNetwork = M
6.     **If** PathConsistent(TemporaryNetwork) does not fail **then**
7.     **begin**
8.      BestCost = Cost
9.      NewNet = TemporaryNetwork
10.     **end**
11.     **backtrack**
12.    **end**
13.    get and remove the next relation R from Removed
14.    ProcessRemoved(M, NewNet, Removed, Cost, BestCost)
15.    **If** Cost  $<$  BestCost - 1 **then**
16.     **begin**
17.      set  $M_R$  to the relation *I*
18.      ProcessRemoved(M, NewNet, Removed, Cost + 1, BestCost)
19.      restore  $M_R$  to previous relation
20.     **end**
21.    Add relation R back to Removed
22. **End**

RealCost is similar to ApproximateCost in that it is really an interface to the functions that perform the main work.

**Input:** Network M  
CurrentBestCost  
**Output:** BestCost  
NewNet a consistent network

**RealCost**

1. **Begin**
2. (List, O) = DetermineInconsistencies(M)
3. BestCost = CurrentBestCost
4. set list Removed to empty
5. RCost(M, List, O, NewNet, 0, BestCost, Removed)
6. **return** (Cost, NewNet)
7. **End**

## 5 Experimental Results

In this section we present the preliminary results we have obtained by implementing the algorithms discussed and testing them with generated problems. The test problems were generated using Nebel's temporal reasoning problem generator [4]. The experiments were conducted on a Pentium 3 733 Mhz processor with 256 megabits of RAM running the Linux operating system. A label size (average number of atomic relations per relation) of 3 and 100 test cases were used for all experiments. Each graph uses a different degree, the degree of a problem is a percentage value indicating how many relations are unknown, a degree value of 1 indicates that all the relations in the problem are known whereas a degree of .25 indicates that only 25 consistent problem which has a consistent solution and a random problem which may or may-not contain a consistent solution. The Y axis for each graph represents the average time a set of problems took and uses a logarithmic scale. The X axis (k) shows the number of events used in a problem.

The results of the four graphs show a trend where Method1 generally performs better at lower degrees and Method2 performs better at higher degrees. This is to be expected since Method1 naturally benefits from having more unknown relations unlike Method2 where each additional unknown relation has a big impact on it's search space. Also at lower degrees there is a higher probability that the generated problem will be consistent. Both algorithms perform well when the problem is consistent which is beneficial. It is also quite evident that Method1 scales in a relatively predictable fashion as opposed to Method2 which is shown to be scaling in a worse fashion.

Overall the preliminary results would tend to indicate that Method1 is the better algorithm due to it's predictable nature and better scaling. Whilst Method2

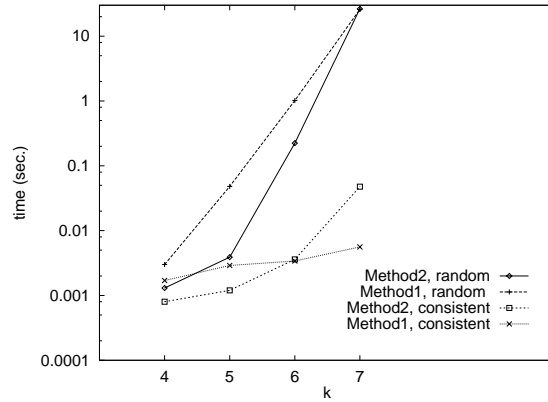


Fig. 2. Graph A: Degree = 1

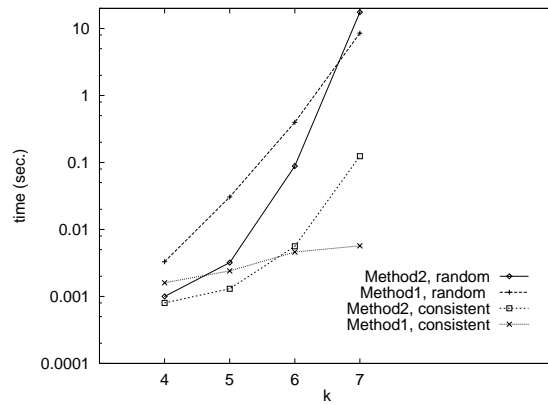


Fig. 3. Grap B: Degree = .75

often outperforms Method1 in these results it is clearly evident that as k gets bigger Method1 will begin to outperform Method2 due to the way both algorithms scale. Analysing the raw data shows that in some cases Method2 takes an extremely long time to find a solution which affects the average result. This would seem to indicate that Method2 runs more favorably on specific sorts of problems whilst Method1 generally performs equally on all sorts of problems.

## 6 Conclusion and Future Work

Finding a partial solution to a Temporal Reasoning problem has to date not been investigated until now. In this paper we outlined two algorithms that can be used in finding a solution to a TPCS problem. Both algorithms are garunteed

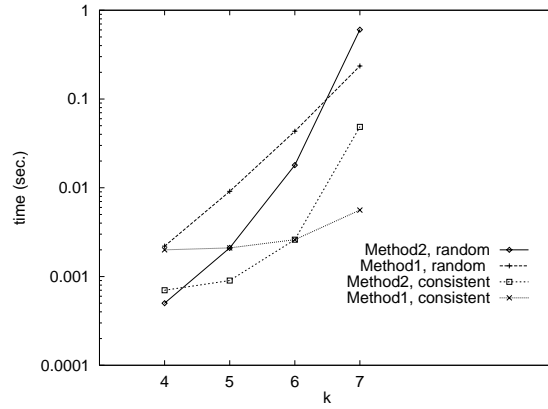


Fig. 4. Graph C: Degree = .5

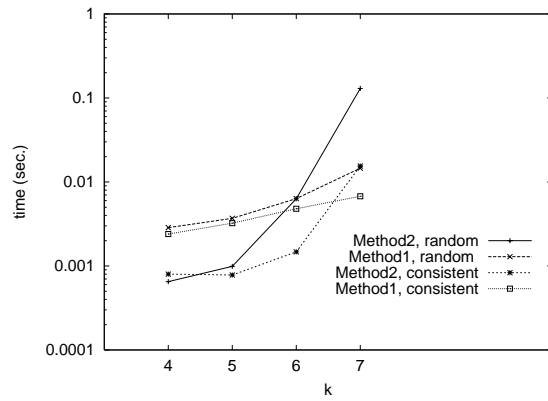


Fig. 5. Graph D: Degree = .25

to find the optimal partial solution, optimal being the least number of relations violated.

The preliminary experimental results show that finding a solution with a tradition algorithm is only practicle on very small sized problems and is thus not very usefull in the real world. The results also showed that Method1, while sometimes being slower than Method2, was much more consistent in finding solutions and is probably the overall superior algorithm due to it scaling better as k gets bigger.

For future work we will be extending the experimental results as well as investigation ways in which to improve the performance of both algorithms. One such improvement would be the use of ordering hueristics which should improve both algorithms and also has the potential to make Method2 more consistent. We will also be investigating Local Search algorithms to gain partial solutions.

Whilst Local Search algorithms do not guarantee an optimal solution they can find a relatively good solution in a short amount of time.

## References

1. J. Allen. Maintaining knowledge about temporal intervals. *Communication of the ACM*, 26(11):832–843, 1983.
2. J. Allen and J. Koopman. Planning using a temporal world model. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 741–747, Karlsruhe, W.Germany, 1983.
3. Eugene Freuder and Richard Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1):21–70, 1992.
4. B. Nebel. Solving hard qualitative temporal reasoning problems: Evaluating the efficiency of using the ord-horn class. *Constraints Journal*, 1:175–190, 1997.
5. M. Poesio and R. Brachman. Metric constraints for maintaining appointments: Dates and repeated activities. In *Proceedings of the 9th National Conference of the American Association for Artificial Intelligence (AAAI-91)*, pages 253–259, 1991.
6. Fei. Song and Robin. Cohen. The interpretation of temporal relations in narrative. In *Proceedings of the 7th National Conference of the American Association for Artificial Intelligence (AAAI-88)*, pages 745–750, Saint Paul, MI, 1988.
7. P. van Beek and R. Cohen. Exact and approximate reasoning about temporal relations. *Computational Intelligence*, 6:132–144, 1990.
8. M. Vilain and H. Kautz. Constraint propagation algorithms for temporal reasoning. In *Proceedings of the 5th National Conference in Artificial Intelligence (AAAI-86)*, pages 377–382, Philadelphia, PA, 1986.