

Managing Dynamic Temporal Constraint Networks

Roberto Cervoni, Amedeo Cesta, Angelo Oddi

IP-CNR
National Research Council of Italy
Viale Marx 15
I-00137 Rome, Italy
amedeo@pecs2.irmkant.rm.cnr.it

Abstract

This paper concerns the specialization of arc-consistency algorithms for constraint satisfaction in the management of quantitative temporal constraint networks. Attention is devoted to the design of algorithms that support an incremental style of building solutions allowing both constraint posting and constraint retraction. In particular, the AC-3 algorithm for constraint propagation, customized to temporal networks without disjunctions, is presented, and the concept of dependency between constraints described. The dependency information is useful to dynamically maintain a trace of the more relevant constraints in a network. The concept of dependency is used to integrate the basic AC-3 algorithm with a sufficient condition for inconsistency detection that speeds up its performance, and to design an effective incremental operator for constraint retraction.

Introduction

The problem of maintaining consistency among quantitative temporal constraints plays a significant role in the problem solving architectures that cope with realistic situations. Dealing with quantitative time has always been an important issue both in planning [VER83, BEL86, DEA87] and in scheduling [RIT86, LEP87]. Dean's TMM [DEA87, 89] represents the most comprehensive investigation in the management of a data base of temporalized information. The formal aspects of the temporal problem drawn from that approach are dealt with in [DAV87] and [DEC91].

In this paper we concentrate our attention on the management of quantitative temporal constraint networks and in particular on the problem of dynamic management of those constraints. The problem consists of considering the possibility of both incremental constraint posting and constraint retraction in the network. A typical scenario for such problems is the flexible creation of schedules of realistic dimension. To be useful to such a task, a temporal information manager should: (a) be able to accept the incremental refinement style for building solutions that most of the current architectures adopt; (b) be able to offer the possibility of retracting previous choices, e.g. to offer

support for making a resource available if some new high priority request comes into play; (c) maintain a network consistent by performing quick update operations --because the consistency checking is executed quite often in the problem solving cycles; (d) allow the maximal flexibility in the specification of temporal constraints. For instance, it should offer a support to recent scheduling approaches that do not specify an exact time for the start of the activities, but just an interval of possibilities [MUS93, SMI93].

A good principle in the attempt of satisfying these requirements seems to be the utilization of algorithms that: (a) work incrementally like the problem solver does; (b) use all the information contained in the previous status of the network and the computations previously performed to compute a new scenario; (c) try to circumscribe the modifications caused by a single change within a sub-area of the network.

In investigating these issues we have followed the way traced by Davis [DAV87] and the constraint satisfaction tradition, focusing on the use of arc-consistency (AC) algorithms. Those algorithms are used in most cases because they offer a good trade-off between space and time complexity. After introducing, in Section 2 and 3, the main features of the arc-consistency specialized to the temporal case, we add information in the network, namely the dependency, that can be easily computed in the AC framework (Section 4) and which is useful to achieve the goals we are pursuing: designing dynamic algorithms, exploiting locality, etc. In particular, we show how the additional dependency information endows the system with both properties for quick inconsistency checking (Section 5) and a powerful tool for deleting constraints (Section 6).

Quantitative Temporal Networks

A basic definition of temporal constraint networks can be the following:

Definition 1 - (Temporal Network) - A temporal network is a directed graph whose nodes represent time points and whose arcs represent distance constraints between time points.

Nodes represent points on a time line in which some change happens (e.g. start-time or end-time of events), while arcs, that we consider always as directed, are able to

uniformly represent both activities' duration and distance constraints between distinct activities and events.

In a temporal network, both nodes and arcs are labelled with a pair of variables. Nodes are labelled with the pair $[lb, ub]$ where lb represents the earliest possible start time for the time point, given the constraints in the network, and ub represents the latest possible start time. Arcs are labelled with the pair of constants $[d, D]$, where d and D represent the respectively minimal and maximal duration of the constraint (i.e., the user can specify the duration of a constraint dc in the form: $d_{dc} \leq duration_{dc} \leq D_{dc}$). A particular node in the graph, named *reference-point*, has the pair of constants $[0, 0]$ as a label, and represents the beginning of the considered temporal horizon. The possible values for the nodes' pairs vary from 0 to h , where h is the length of the temporal horizon considered in the particular problem. In the CSP terminology this means that Node Consistency is guaranteed setting the initialization $[0, h]$ to the time-points inserted in a network. Such a unary constraint is referred in any case to the *reference-point* -- as done in [DEC91].

The pair $[lb, ub]$ of a node identifies an interval of legal values for the time variable ti representing the occurrence of the event. Given a set of constraints (arcs) and their durations, the pairs $[lb, ub]$ associated with the nodes of the network are calculated with respect to the reference point using a constraint propagation algorithm. Such an algorithm should verify the existence of solutions: it has to verify the property of temporal consistency of the network by eliminating the values not compatible with the constraints from the temporal variables.

Definition 2 - (Temporal Consistency) - A connected temporal network is said to be temporally consistent if and only if an assignment of values to the temporal variables exists that is compatible with all the constraints in the network.

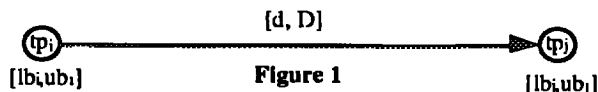
The problem we are interested in consists of computing the bounds associated with the nodes when the network is modified by either adding or removing constraints. It is worth noting that, for each time-point, we are interested in having the bounds lb and ub that not only satisfy all the constraints in the network, but also represent the minimal and maximal distance between the time-point and the *reference-point*. Moreover, we are interested in performing the previous computation working just on the part of the network actually affected by the modification.

The reader familiar with [DEC91] can recognize the problem described above as an instance of the *Simple Temporal Problem* (STP). In that paper, the problem of checking consistency of a given network of constraints is proved to be polynomial in the number of nodes N when the STP assumptions hold.

An Incremental *Propagate* Operator

While a problem solver is creating a schedule or a temporalized plan, the possible ways of adding constraints to the temporal network that memorize the partial solution

are: (a) shrinking the interval $[lb, ub]$ associated with an arc; (b) adding a new arc between any two nodes of the network; (c) adding a new node and one or more arcs that connect it to the rest of the network. Case (c) can be reformulated in terms of cases (a) and (b) considering the new time-point as unconstrained before the modification. We speak of *propagation sources* to refer to constraints' modifications that require the network revision -- the effects of each modification should be *propagated* through the network.



Given an arc (dc) between two time-points, as shown in Figure 1, the intervals of the two time-points should be consistent wrt the distance constraint represented by the arc. This means that from the interval associated with tpj an algorithm should cut out those values that are inconsistent with the values in the interval related to tpj via the constraint $[d, D]$ and *vice versa*.

Definition 3 - (Arc-Consistency) - A connected temporal network is said to be arc-consistent if and only if for any network constraint dc , with bounds $[d, D]$, between two time-points tpi and tpj , for any $ti \in [lbi, ubi]$ ($tj \in [lbj, ubj]$) there always exists a value $tj \in [lbj, ubj]$ ($ti \in [lbi, ubi]$), such that tj and ti satisfy the inequality: $d \leq tj - ti \leq D$.

To actually check for arc-consistency on a single arc we use the following property:

Proposition 1 - The simple temporal network in Figure 1 is arc-consistent if and only if the bounds lbi, lbj, ubi, ubj satisfy the relations:

- a) $lbj := \max[lbj, lbi + d]$; b) $ubj := \min[ubj, ubi + D]$;
- c) $lbi := \max[lbi, lbj - D]$; d) $ubi := \min[ubi, ubj - d]$;
- e) $lbi \leq ubi$; f) $lbj \leq ubj$;

To check the consistency of the whole network we use the *propagate* algorithm shown in Figure 2.

```

Procedure propagate (propagation-sources);
var queue: QueueType; fail: Boolean;
    current-arc: ArcType;
begin
    fail ← False; queue ← propagation-sources;
    while (not-empty queue) and (not fail) do
        begin
            current-arc ← Pop(queue);
            if revise (current-arc)
            then fail ← True
            else queue ← queue ∪ {arc | arc ≠ current-arc
                and arc connected to the nodes whose intervals
                have been modified by revise};
        end
    end

```

The algorithm has an AC-3 shape [MAC77] and continuously uses the procedure *revise* that implements Proposition 1 (i.e., it updates bounds according to expressions $a-d$ and check conditions $e-f$). A particular property of the *revise* operator, useful in what follows, is stated by the following proposition:

Proposition 2 The *revise* algorithm updates the bounds by using at most one of *a* and *c* (*b* and *d*), where *a*, *b*, *c*, and *d* are the expressions in Proposition 1.

The *propagate* algorithm is notoriously correct with respect to the arc-consistency definition [MAC77].

Correctness and Completeness of the Algorithm

The *propagate* algorithm checks arc-consistency on all the arcs belonging to the sub-network affected by a modification. But we are interested in verifying the existence of a solution to the Simple Temporal Problem, that is in checking for consistency according to Definition 2. The equivalence between the two types of consistency (the one practically checked and the one formally defined) is given by Theorem 1:

Theorem 1 - A temporal network is arc-consistent if and only if it is also temporally consistent.

Because of Theorem 1, the *propagate* algorithm is correct and complete. It recognizes as consistent only networks in the situations stated by Definition 2. It is worth noting that completeness holds when the Simple Temporal Problem hypothesis holds, that is when no disjunction is allowed on any constraint. Here, we consider as crucial the property of performing complete inferences in the temporal module for at least two reasons: (a) because in a problem solving architecture only the decision making module should be responsible for cutting the search space using domain dependent heuristics, while the temporal module should simply guarantee a service (checking consistency); (b) because of efficiency issues: dealing with disjunction is a computationally difficult problem [DEC91] and identifying practical approaches to deal with disjunction is still an open issue (e.g., see [SCH92, SCW93]).

Complexity of the Algorithm

A characterization of the worst case of the *propagate* algorithm can be obtained in terms of the maximum number of arcs inserted in the working queue when one or more source arcs are added. The choice is justified because such a maximum is directly proportional to the number of updating actions performed (such operations are the crucial part of the algorithm). From [MAC85], it is well known that AC-3 in the general case has complexity $O(d^3E)$ where *d* is the cardinality of the set of values a node may assume, while *E* is the number of arcs in the network. In our case the expression becomes $O(hE)$ if we consider that, being the temporal interval an ordered set of values, the *revise* operation can check the single constraint with constant cost rather than with a cost dependent on the cardinality of the sets associated with the nodes. This implies that the complexity of AC-3 in the temporal case loses a factor h^2 wrt the general case shown in [MAC85].

Lastly, it should be clear the advantage (in terms of computational time and space) of the *propagate* algorithm wrt PC-2 proposed in [DEC91] for the STP. The complexity of PC-2 is $O(N^3)$. The improvement is due to

the fact that arc-consistency works on arcs that are $O(N^2)$ in a complete graph. Moreover, the average complexity is reduced when the graph is not complete -- this happens very often in real schedules. Also in terms of space the AC is better because does not require an array for the distances between any couple of nodes as requested in the PC algorithms. To be fair it should be noted that PC-2 gives the user more information than the one strictly necessary to the resolution of the STP, because that algorithm calculates also the minimal network -- the distance among any pair of nodes.

Defining Dependency-Chains

As said before, the pair $[lbi, ubi]$ on a given time-point tp_i represents the minimal and maximal possible distance of the temporal variable ti wrt the **reference-point**. It is worth noting that the two distances are determined by the network's more constrained paths and the network's constraints can be partitioned in two sub-sets:

- *the active constraints* at least belonging to one of the most constrained path in the network -- i.e., active constraints cause a bound on a time-point.
- *the inactive constraints* not satisfying the previous property.

To represent the set of active constraints in a given configuration of the network we slightly modify the *propagate* algorithm. In particular, each time the bounds of a node are modified, we memorize in the node the arc that caused the modification, distinguishing between the modification of the lower and of the upper bounds -- this is done with a straightforward change of the *revise* procedure. The two pieces of information are stored in each node and called *Dependencies*: in particular, the *Lower-bound (Upper-bound) Dependency* is the arc that last modified the lower (upper) bounds of the time-point. As a consequence, starting from any node and following the dependency pointers of each given type, it is possible to cover two distinct paths (sequences of arcs) to the **reference-point**. We call these paths *Dependency-Chains*.

To clarify the definition of dependency, we introduce the functions *Last-mod-lb* (tp_j) and *Last-mod-ub* (tp_j).

Such functions, when applied to any node of the network, return respectively a reference to the last arc that has changed the lower/upper bound of the node. If the lower bound (upper bound) of the node interval has never been modified then $Last-mod-lb(tp_j)=NIL$ ($Last-mod-ub(tp_j)=NIL$). These functions allow to maintain a trace of critical paths, composed by distance constraints representing arcs that act on their extreme nodes.

Definition 4 - (Active/Inactive Arc) - Let $\langle tp_i, tp_j \rangle$ be an arc between the time-point tp_i and the time-point tp_j . The arc is said: (a) *active wrt upper-bounds* if either $Last-mod-ub(tp_j)=\langle tp_i, tp_j \rangle$ or $Last-mod-ub(tp_i)=\langle tp_i, tp_j \rangle$ hold; (b) *active wrt lower-bounds* if either $Last-mod-lb(tp_j)=\langle tp_i, tp_j \rangle$ or $Last-mod-lb(tp_i)=\langle tp_i, tp_j \rangle$ hold; (c) *inactive* otherwise.

Definition 5 - (Dependency-chain) - We define *Lower-bound* (or *Upper-bound*) *Dependency-chain* a non directed path in a temporal network whose arcs are active wrt lower-bounds (upper-bounds).

It is worth noting that, as a consequence of Proposition 2, for any arc $\langle tp_i, tp_j \rangle$ the following properties necessarily hold:

$$Last-mod-lb(tp_i) \neq Last-mod-lb(tp_j)$$

$$Last-mod-ub(tp_i) \neq Last-mod-ub(tp_j)$$

unless both of them are equal to NIL. Such a property causes that, even if the path of a dependency-chain can be non directed wrt the distance constraints direction, it is always directed wrt dependency pointers direction.

Given the definitions, a theorem can be proved:

Theorem 2 - Given a consistent temporal network in which all nodes but the reference-point satisfy the inequality:

$$Last-mod-lb(tp_i) \neq NIL \text{ (} Last-mod-ub(tp_i) \neq NIL \text{)}$$

then all the lower bounds (upper bounds) dependency-chains form a spanning tree of the network.

The set of dependency-chains produces two spanning trees over the network, one for the lower-bound and one for the upper-bound dependencies.

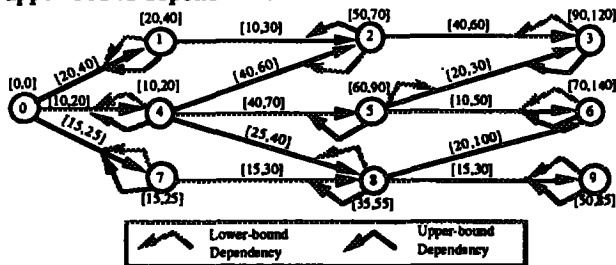


Figure 3

Examples of the previous definitions can be found in the temporal network depicted in Figure 3. We can see that the only inactive constraint is the arc $\langle 8, 6 \rangle$, while $\langle 5, 3 \rangle$ is active both wrt lower and upper-bounds, the $\langle 4, 8 \rangle$ is active wrt lower-bounds, $\langle 7, 8 \rangle$ is active wrt upper-bounds. Focusing the attention on node 5, we can see that it has a lower-bound dependency chain given by the path $\langle 0, 4 \rangle$, $\langle 4, 2 \rangle$, $\langle 2, 3 \rangle$, $\langle 5, 3 \rangle$ that is responsible for its lower bound 60, while the upper-bound dependency-chain $\langle 0, 4 \rangle$, $\langle 4, 5 \rangle$ determines the node's upper-bound 90.

As said before, the updating of dependency-chains caused by incremental modification of the network is performed by the *propagate* algorithm each time a node interval has been changed by the propagation process. The complexity of the *propagate* does not change because the only added cost is a constant factor internal to the *revise*, needed to update the two *Last-mod* functions. A special visit to the network is not necessary.

Improving Propagation Efficiency

A first aspect that can be addressed using the dependency-chains is the quick detection of inconsistency while inserting new constraints. Chances are that the propagate

algorithm detects an inconsistency only after a long sequence of update operations on closed cycles. In particular, the presence of inconsistency due to update operations on a closed path may be particularly dangerous as shown in Figure 4 and pointed out in [DAV87, page 318]. The example shows the updating process activated by the insertion of a constraint $[301, 1000]$ between nodes 1 and 3. Propagating the effects of the new constraint involves a closed (non-oriented) path. The *propagate* continuously modifies the lower bounds of nodes' intervals with step 1 (e.g., the lower bound of node 1 is set to 1, 2, 3 ...). As a consequence, an inconsistency is detected only after a number of iterations given by the length of the smallest nodes' intervals on the closed path. A "cycle" of the propagation algorithm is not necessarily given by an oriented cycle in the temporal graph, but it can be generated also by a closed non-oriented path. If the algorithm is cycling, then it is inevitable that a condition $lb > ub$ eventually happens. Early recognition of cycles allows the propagation to be promptly stopped.

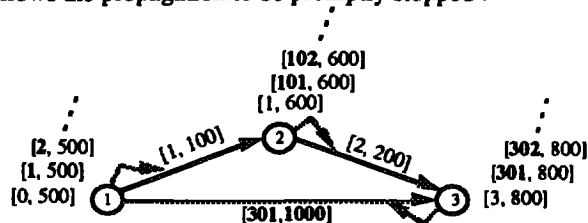


Figure 4

It is worth noting that in the network of Figure 4 the dependency-chain is closed. Using previous Theorem 2 we formulate the following corollaries:

Corollary 1 - The existence of a cycle in a dependency-chain is a sufficient condition for the inconsistency of a temporal network.

Corollary 2 - Given a consistent network and a set of new constraints, if a dependency-chain contains a cycle then at least one of the new constraints is contained in the cycle.

The two properties suggest a modification of the *propagate* that allows the algorithm to stop the propagation in the critical case of Figure 4. The new version of the algorithm is close to the previous one, the difference consisting of a check performed on the arcs inserted in the working queue. If the current arc is a propagation source then a *check-cycle* function checks if the arc belongs to a closed dependency-chain. If it does then an inconsistency is suddenly detected and the update process is stopped. To verify the existence of a closed dependency-chain in a network with N nodes has a cost $O(N)$. In fact, in the worst case the algorithm visits all the arcs belonging to a chain, i.e., $(N - 1)$ arcs. The additional cost is balanced by the conspicuous advantages shown by the experimental evaluation of the average case.

The experimentation has been carried out using a random network generator we have implemented. The generator incrementally builds a network by posing new constraints. The constraints are generated using a pseudo-

random number generator on a network consisting of a given number of nodes. In the present set of experiments, the network randomly generated is characterized by a ratio between the number of arcs and the number of nodes given by 5. The particular ratio is justified by the shape of the networks actually generated in a problem solving architecture we are working with, but analogous results have been obtained for different values of the ratio.

As done for the worst-case analysis, we choose the number of the local propagation operations as a measure of performance. Using the total number of arcs affected by propagation, we obtain a measure independent of a particular hardware. The average numbers are calculated on a total of 300 (three hundred) experiments generated by using different random seeds. The results represent a first experimental evidence of the usefulness of the property. In Table 1 a comparison is shown between the basic version of the propagate algorithm and the one which uses the sufficient condition stated in Corollary 2. Aarts and Smith have incorporated our cycle detection in the operational environment described in [AAR94] obtaining an actual improvement in the overall cost of the propagation. It should be noted that the property refers to a structural property of the dependency graph and it holds even when multiple propagation sources are active.

Table 1: comparison between the basic version of the propagate algorithm and the one which uses the sufficient condition stated in Corollary 2.

Edges (Time points)	250 (50)	500 (100)	1000 (200)	2000 (400)	4000 (800)
#OP-N	480	1232	1456	3950	8456
#OP-CD	57	110	137	353	462

#OP-N: number of revise operations in the basic propagation algorithm
 #OP-CD: number of revise operations when the algorithm uses the cycle detection

A Remove-Constraint that Exploits Locality

The *remove-constraint* algorithm can be used in all the cases in which the temporal network is modified by relaxing some of the constraints. In those cases, we perform just an updating of the sub-network affected by the deletion using a propagation process similar to the one in the insertion case. This process does not require checking for consistency because removing constraints means relaxing the problem. If the initial network is consistent, it will be also consistent after the deletion of the constraint, because while the insertion of constraints in general narrows the number of solutions for the temporal problem, the retraction widens the number of solutions.

The dependency information allows the temporal manager to "remember" which constraint caused the more recent modification of the nodes' bounds. Using that information we can distinguish two cases:

- If the removed constraint does not belong to any dependency-chain (it is an *inactive* arc), its deletion does not modify the distances among nodes and does not require update operations.
- If the constraint belongs to a dependency-chain (it is *active* wrt either lower-bounds or upper-bounds or both), it determines the present bound intervals of at least one of its extreme nodes. In this case propagation is necessary to restore previous bounds.

The distinction is interesting because it points out a sub-case in which no computation is needed. The property is exemplified in Figure 3 by the retraction of constraint <8,6>. All the bounds in the network are invariant wrt that retraction.

Also in the cases that require propagation, the dependency information results useful to restrict the propagation in a sub-network. Being *dc* the arc to remove, the sub-network influenced by its retraction contains the time-point tp_i such that $last-mod-lb(tp_i)=dc$ and all the time-points connected to the dependency-chains that can be covered from tp_i moving forward wrt the **reference-point** (i.e., all the time-points in a dependency sub-tree having tp_i as root).

We explain this property with reference to Figure 3. If we retract the constraint <4, 8> the nodes affected by the modification are #8 and #9 because their lower bounds depend on the removed arc. The key idea is that a sub-tree of the dependency graph has remained isolated from the reference point. A straightforward algorithm to re-establishing a consistent situation consists of covering the sub-tree and, for each time-point in it, inserting all the incident arcs in a queue, and applying the usual propagation algorithm to that queue. In the example, the algorithm covers the sub-tree containing nodes #8 and #9 and inserts in the queue the arcs <7, 8>, <8, 6> <8, 9> because one of them will be reconnected to a lower-bound dependency-chain. Each time point in the sub-tree should have the lower bound recomputed since their current bound is wrong. So the bound is relaxed to the default value 0 in the case of lower-bound dependency deletion (*h* in upper-bound dependency deletion). The usual propagation is applied to the arcs in the queue, without checking for consistency. In the worst case the *remove-constraint* algorithm has the same complexity of the propagate, but it also has the same property of localization of the propagation to a subnetwork, that improves its performance in the average case.

Procedure *remove-constraint* (constraint)

```

var arc: ArcType; sub-net-arcs: QueueType;
    tpi, tp-mod-lb, tp-mod-ub: TimePointType;
begin
    sub-net-arcs ← NIL;
    if <constraint is active wrt lower/upper-bounds>
    then begin
        tp-mod-lb ← tpi s.t. last-mod-lb(tpi)=constraint;
        tp-mod-ub ← tpi s.t. last-mod-ub(tpi)=constraint;
        delete (constraint);
        if (tp-mod-lb > NIL) then sub-net-arcs ←
            sub-net-arcs ∪ sub-net-arc-lb(tp-mod-lb);
        if (tp-mod-ub > NIL) then sub-net-arcs ←
            sub-net-arcs ∪ sub-net-arc-ub(tp-mod-ub);
        update-sub-network (sub-net-arcs);
    end
    else delete (constraint);
end.
```

Figure 5

Figure 5 shows the *remove-constraints* algorithm. The procedure *delete* physically deletes the arc from the network. The subnetwork to be updated is decided by the

procedure *sub-net-arc-lb* that checks the lower-bound dependencies, and by *sub-net-arc-ub* that checks the upper-bound dependencies. Figure 6 shows the *sub-net-arc-lb*, the body of the *sub-net-arc-ub* being analogous but the relaxation of the time-point's upper-bound to *h*. The actual update of the network is performed by the *update-sub-network* procedure. This procedure is very similar to the *propagate*, but the queue is initialized with the result of the previous analysis (*sub-net-arc* procedures) and consistency is not checked.

```

Procedure sub-net-arc-lb(tp)
  var sub-net-arcs: QueueType; tpi: TimePointType; arc: ArcType;
  begin
    for-each arc in {lower-bounds dependency-chains that
                      start from tp forward wrt *reference-point*}
    do begin
      tpi ← time-point s.t. Last-mod-lb (tpi) = arc;
      <relax tpi's lower bound to 0>;
      sub-net-arcs ← {arc} ∪ sub-net-arcs
    end;
  return sub-net-arcs
end

```

Figure 6

Table 2 shows the average cost of a deletion in network of increasing dimensions. The comparison is done wrt a previous version of the algorithm called *global-remove-constraint* that does not use the dependency information but updates all the bounds in the time-point to $[0, h]$ and then re-computes the new values applying the *update-sub-network* to the whole network. From the data in Table 2, we have a confirmation that the (random) deletion of a constraint from the network requires a small number of updating information (around 10% of the total number of arcs). It is worth remembering that applying PC-2 to this problem is not convenient because the cost is bound to N^3 .

Table 2: average cost of a deletion in networks of increasing dimension

Edges (Time points)	250 (50)	500 (100)	1000 (200)	2000 (400)	4000 (800)
#OP-L	115	255	425	935	1190
#OP-G	1401	4132	10281	27311	73925

#OP-L: number of revise operations in the remove-constraint algorithm
 #OP-G: number of revise operations in the global-remove-constraint algorithm

Conclusions

In the paper we have discussed the application of AC-3 to the Simple Temporal Problem, and proposed the integration of the basic algorithm with the dependency information. The dependency pointers are stored preserving the $O(hE)$ complexity of AC-3 specialized to the temporal case. The dependency information represents the concept of "causality of modifications" in the network. We have shown how such a concept is useful to define a structural property of the network that allows an inconsistency to be efficiently recognized and to design a *remove-constraint* operator that works in an incremental way and exploits the locality of the modification. The main results of our study is the analysis of the conditions for dynamically maintaining consistent information in a temporal network. We have used algorithms that, exploiting simple structural properties of the constraints, allow incremental computations, use previous computation

and exploit locality when possible. Such properties are particularly useful when a problem solving architecture deals with tasks of relevant dimensions as in the case of scheduling problems or problems that require the integration of planning and scheduling. As shown by the proposed experimental results, the number of computations required by a modification is a small fraction of the number of constraints in the network.

Due to space limits the current presentation is concise. A longer paper containing proofs and further experimental results is available from the authors.

Acknowledgements. The authors would like to thank Luigia Carlucci Aiello, Daniela D'Aloisi and the AIPS-94 anonymous reviewers for useful comments and suggestions. The authors were partially supported by CNR under "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo". Amedeo Cesta is also partially supported by CNR under Special Project on Planning.

References

[AAR94] Aarts, R.J., S.F. Smith, A High Performance Scheduler for an Automated Chemistry Workstation, to appear in *Proceedings of ECAI 94*, Amsterdam, The Netherlands, 1994.

[BEL86] Bell, C., Tate, A., Using Temporal Constraints to Restrict the Search in a Planner, AI Applications Institute, AIATR-5, University of Edinburgh, Scotland, 1986.

[DAV87] Davis, E., Constraint Propagation with Interval Labels, *Artificial Intelligence*, 32, 1987, 281-331.

[DEA87] Dean, T.L., McDermott, D.V., Temporal Data Base Management. *Artificial Intelligence*, 32, 1987, 1-55.

[DEA89] Dean, T.L., Using Temporal Hierarchies to efficiently maintain large temporal databases. *Journal of the ACM*, 36, 4, 1989, 687-718.

[DEC91] Dechter, R., Meiri, I., Pearl, J., Temporal constraint networks. *Artificial Intelligence*, 49, 1991, 61-95.

[LEP87] Le Pape, C., Smith, S.F., Management of Temporal Constraints for Factory Scheduling. Technical Report CMU-RI-TR-87-13, Carnegie Mellon University, June 1987.

[MAC77] Mackworth, A. K., Consistency in Networks of Relations, *Artificial Intelligence*, 8, 1977, 99-118.

[MAC85] Mackworth, A. K., Freuder, E. C., The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence*, 25, 1985, 65-74.

[MUS93] Muscettola, N., Scheduling by Iterative Partition of Bottleneck Conflicts. *Proc. 9th IEEE Conference on AI Applications*, Orlando, FL, 1993.

[RIT86] Rit, J.F., Propagating Temporal Constraints for Scheduling, *Proc. of AAAI-86*, Philadelphia, PA, 1986.

[SCH92] Schrag, R., Boddy, M., Carciofini, J., Managing Disjunction for Practical Temporal Reasoning. *Proceedings of KR '92*, Morgan Kaufmann, 1992.

[SCW93] Schwalb, E., Dechter, R., Coping With Disjunctions in Temporal Constraint Satisfaction Problems, *Proceedings of AAAI-93*, Washington, DC, 1993.

[SMI93] Smith, S.F., Cheng, C., Slack-Based Heuristics for Constraint Satisfaction Scheduling, *Proceedings of AAAI-93*, Washington, DC, 1993.

[VER83] Vere, S.A., Planning in Time: Windows and Durations for Activities and Goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol.PAMI-5, No.3, May 1983, 246-276.