Temporal Reasoning Problems and Algorithms for Solving Them

1 author:

Léon Planken
Delft University of Technology
9 PUBLICATIONS **151** CITATIONS

Temporal Reasoning Problems and Algorithms for Solving Them

Léon Planken
Student number 9654215

November 2, 2007

Contents

Preface

This document is a survey of literature on the subject of temporal reasoning problems. We hope it will serve several functions.

For the reader with a general background in computer science, it may serve as an introduction to the field of temporal reasoning problems and the current state of research in this area.

For ourselves, writing this survey has been a good way to consolidate the knowledge we have obtained by studying the literature. The document will also serve as a foundation to refer back to when conducting our own future research.

Finally, the survey will serve to expose as yet unexplored, blank areas on the map of this field of research. We will stumble upon a few of these while discussing the various subjects; in our own future research, we hope to fill in some blank spots.

Léon Planken, August 2007

Chapter 1

Introduction

The subject we discuss in this document concerns problems involving reasoning about time. Examples of this class of problems are:

- the scheduling of arrivals, departures and ground handling on airports;
- logistics of supply chains, taxi companies et cetera; and
- applications where agents must plan their activities autonomously or in co-operation with one another.

Several branches of temporal reasoning exist.

- *Qualitative* versus *quantitative problems*

  Qualitative problems can only specify how two events must occur in relation to each other; for example, "event a must occur before event b" is a qualitative constraint.

  In contrast, quantitative problems allow for constraining durations and time gaps; for example, "event a must occur between 20 and 50 minutes before event b" is a quantitative constraint.

  Note that every qualitative constraint can trivially be converted into a quantitative one by setting the lower and upper bounds on the time difference to zero and infinity, respectively.

- *Time points* versus *intervals*

  For problems in the former category, the fundamental elements of reasoning are instants in time without any duration. To describe an event having a non-zero duration, two time points can be used to denote the beginning and the end of the event. Constraints describe the relation of

time points to one another without discerning between time points delimiting an event with a duration versus those denoting instantaneous events.

In the latter category of problems, every fundamental item of reasoning is an interval with some (possibly zero-length) duration. Constraints describe the relation between intervals, and allow for description of such relations as events overlapping, one event occurring while another takes place, and one event finishing before another starts.

Some formalisms allow for a mixture of intervals and time points as fundamental elements.

In this survey, we limit ourselves to quantitative problems involving time points.* Research on temporal reasoning started in the 1980s with, for example, Allen's interval algebra [All83] and Vilain's and Kautz's point algebra [VK86]. The work that can be seen to have started the branch of quantitative time-point problems is the 1991 publication by Dechter, Meiri and Pearl. In this work, they proposed the Simple Temporal Problem (STP) and the Temporal Constraint Satisfaction Problem (TCSP) [DMP91]; in 2000, Stergiou and Koubarakis proposed the formalism that became known as the Disjunctive Temporal Problem (DTP), thereby giving birth to another important member of the family. These problems have at their root the general theory of constraint satisfaction problems (CSPs). Many concepts and techniques from general CSP literature such as forward checking, conflict-directed backjumping, and no-good recording have also been successfully applied in the field of temporal reasoning problems, as we will show.

In our discussion of these problems, we will focus on the following issues:

- formal definitions of the problems;
- concepts related to the problems that are used in algorithms;
- formal complexity analyses of the problems;
- a representative selection of available algorithms for solving the problems, pointing out the strengths and weaknesses of each and contrasting the differences between them.

*Problems of other types can always be recast into this form; we have already seen that this is the case for qualitative versus quantitative constraints. Representing relations between intervals with only time points as fundamental elements is less straightforward, but still quite feasible with the expressive power of the DTP, as the reader shall see.

We will start our discussion, in Chapter 2, with the Simple Temporal Problem. As its name suggests, this formalism can be used to represent only a modest selection of problems that are relatively easy to solve. Starting by discussing only this subset has the advantage that the reader is gently introduced to the concepts underlying temporal reasoning, but more importantly, being able to solve problems of this type is at the heart of solving its more difficult cousins that we will describe next.

Chapter 3 introduces both the TCSP and the DTP, which are closely related to each other and far more expressive than the STP; however, the reverse of the medal is that they are also much harder to solve. We will show that all current algorithms for tackling the TCSP and the DTP rely on fast methods for solving the STP.

Finally, in Chapter 4, we will give the conclusions to this literature survey. We will first concisely recapitulate the theory discussed, and then point out interesting blank spots on the map we have drawn of this field of research, briefly discussing our plans for future expeditions.

In the appendices, we will include proofs of some theorems we give in the main text. These theorems concern theoretical results that were not taken from existing literature but were contributed by ourselves.

Chapter 2

The Simple Temporal Problem

2.1 Introduction

In this chapter, we introduce a simple method for reasoning about temporal information, called the *Simple Temporal Problem* (STP). Though the scope of the problems that can be represented with the STP is not very wide, it still suffices in many cases; more importantly, there exist very efficient algorithms for solving it.

To illustrate this problem, we will first discuss an example and show how to represent it as an STP. Then, we will define the problem more formally and discuss some of its properties that can be used when solving it. Next, we give a representative list of the available algorithms for tackling STPs and establish the complexity of the problem theoretically. The remainder of the chapter is devoted to a more detailed discussion of each of these algorithms.

2.2 Example

Before formally defining the STP, we first present an example that is based on the one that provided in the original publication by Dechter, Meiri and Pearl that proposed the STP [DMP91]. We will use this example in the remainder of this chapter to illustrate the operation of the algorithms we will describe.

> John goes to work by car, which takes 30–40 minutes; Fred goes to work in a carpool, which takes 40–50 minutes. Today, John left home between 7:10 and 7:20, and Fred arrived at work between 7:50 and 8:10. We also know that John arrived at work after than Fred left home, but not more than 20 minutes later than he left.

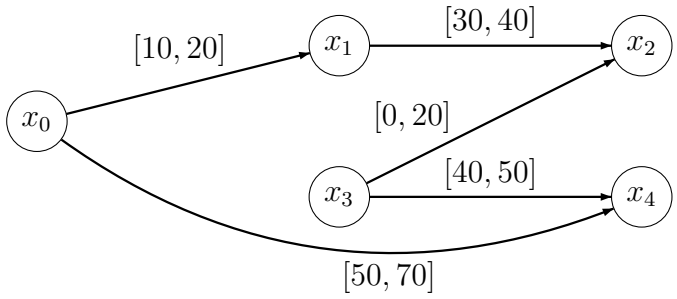


Figure 2.1: An example STP instance

We can associate a variable with each event in this short story. Let us say that x_1 and x_2 represent John leaving home and arriving at work, respectively; x_3 and x_4 denote the same events for Fred. We also need a temporal reference point to be able to refer to absolute time points; this is denoted by x_0 will stand for seven o'clock this morning. The network representation of the STP for this example is given in Figure 2.1.

2.3 Definitions

In this section, we will formally define the Simple Temporal Problem (STP). We start by defining which form an STP instance takes and giving an interpretation of this definition, and defining a solution to an STP instance. Then, we will define additional properties of the STP on which the algorithms described in Section 2.5 are based.

2.3.1 Problem representation and solutions

An instance of the Simple Temporal Problem (STP) consists of a set of time-point variables $X = \{x_1, \ldots, x_n\}$ representing events, and a set of binary constraints over the variables, $C = \{c_1, \ldots, c_m\}$, bounding the time difference between two events. Every constraint $c_{i \to j}$ has a weight $w_{i \to j} \in \mathbb{Z}$ corresponding to an upper bound on the time difference, and thus represents an inequality $x_j - x_i \leq w_{i \to j}$.* Two constraints $c_{i \to j}$ and $c_{j \to i}$ can then be combined into a single constraint $c_{i \to j} : -w_{j \to i} \leq x_j - x_i \leq w_{i \to j}$ or, equivalently, $x_j - x_i \in [-w_{j \to i}, w_{i \to j}]$, giving both upper and lower bounds. If $c_{i \to j}$ exists and $c_{j \to i}$ does not, this is equivalent to $x_j - x_i \in [-\infty, w_{i \to j}]$.

*Rational weights can easily be recast as integer weights by multiplying each with the least common denominator. Real weights are outside the scope of this discussion; as Dechter et al. note [SD97], rational weights always suffice in practice.

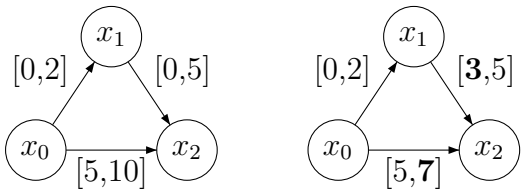


Figure 2.2: An STN and its corresponding minimal network

In this text, we will sometimes use $c_{i\to j}$ to stand for the upper bound only, and sometimes to represent both upper and lower bounds; our intention will always be clear from context.

A *solution* to an STP instance schedules the events such that all constraints are satisfied. Formally, it is represented by a tuple $\tau = \langle x_1 = b_1, \ldots, x_n = b_n \rangle$ of assignments of values to all variables. An instance is called *consistent* if at least one solution exists. Note that since the weights are integers, a consistent instance always has an integer-valued solution.

Using the concepts defined so far, time differences between two events can easily be expressed, but we have as yet no way to denote absolute time constraints such as "event x_{42} must occur between 1 September 2007 and 31 December 2007". These could be represented with unary constraints like $x_{42} \in [t_1, t_2]$, where t_1 and t_2 are suitable integer representations of 1 September 2007 and 31 December 2007, respectively; however, this has the disadvantage of having two types of constraints, unary and binary. The usual way to represent constraints of this type is to introduce a *temporal reference point*, denoted by x_0 or z, that represents some agreed-upon epoch. This way, the problem representation is nicely uniform, consisting only of variables and binary constraints between them. In the remainder of this text, we will seldom treat the temporal reference point specially, except where required for examples.

An additional advantage of introducing the temporal reference point is that a STP instance can then easily be depicted as a graph, in which nodes represent variables and weighted arcs represent constraints. When represented this way, an STP instance is also called a Simple Temporal Network (STN); we have already seen an STN representation in Figure 2.1 in the previous section. Because the STP and its network representation are so closely related, we will sometimes use the terms STP and STN interchangeably.

2.3.2 Minimal network

The constraints given in an STP instance may not accurately represent the allowed time difference between two events. We will illustrate this with

an example. See the network on the left-hand side of Figure 2.2; in this STN, $c_{0\to 2}$ allows event x_2 to occur up to 10 minutes after x_0. However, the transitive closure of $c_{0\to 1}$ and $c_{1\to 2}$ yields an actual upper bound of 7 minutes; a similar argument holds for $c_{1\to 0}$ and $c_{0\to 2}$. This method of tightening is called *path consistency* in general constraint satisfaction literature, and we will also refer to it by that name. The right-hand side of Figure 2.2 depicts a network for which every constraint has been tightened as much as possible without invalidating any solutions; this is called the *minimal network*.

The minimal network has the desirable property that solutions can be extracted from it in a backtrack-free manner. For the first variable, any value at all can be picked, yielding the first partial solution; for example, the value of the temporal reference point can be set to 0. Any partial solution can be extended by picking a new variable and instantiating it to a value that satisfies all constraints with from already instantiated variables; the minimality of the network guarantees that such a value can always be found. For this reason, calculating the minimal network is often equated with solving the STP.

An important property of the STP is that it is consistent if and only if it contains no cycles with negative total weight. It is easy to see why a negative cycle yields inconsistency: the transitive closure (summation) of the constraints making up the cycle require that an event occur before itself. For example, the two inequalities $x_2 - x_1 \leq -2$ and $x_1 - x_2 \leq 0$ form a negative cycle; their summation yields $x_2 - x_2 \leq -2$ which is clearly inconsistent. If there is a negative cycle, the minimal network is also undefined; calculating it would require that constraints along that cycle be tightened over and over again. In contrast, the absence of a negative cycle means that the shortest path between each two nodes in the STN is well-defined and that a minimal network can be calculated. As we have seen, a solution can then easily be extracted, which proves that the absence of negative cycles implies consistency.

Having defined the terminology relevant to the STP, we will now describe its complexity.

2.4 Complexity

Perhaps surprisingly, none of the relevant literature gives a formal complexity class for the STP; the respective authors only establish P as an upper bound by giving polynomial algorithms for solving the problem. In this section, we will first give a short overview of the complexity of some available algorithms; then, we include several new theorems which delimit the complexity of the

STP more rigidly.

We should start by noting that there are three possible definitions for *solving the STP*:

1. deciding consistency;
2. finding a valid instantiation of time-point variables; or
3. calculating the minimal network

These are listed in order of increasing difficulty; the first is implied by the second, which in turn is implied by the third.

We will now list the complexity of the algorithms we will describe later in this chapter. Some of these algorithms only determine consistency, while others calculate the minimal network. When describing the complexities and the algorithms themselves, n and m will be used to denote the number of variables and the number of constraints, respectively. We also use $W^*(d)$, a measure called the *induced graph width*, which depends on the ordering d of the variables chosen but never exceeds the maximum vertex degree of the graph, i.e. the maximal number of constraints that a single variable participates in.

Authors	Type	Complexity	Section
Floyd & Warshall	Minimal	$\mathcal{O}(n^3)$	2.5.1
Bellman & Ford	Consistency	$\mathcal{O}(n \cdot m)$	2.5.2
Johnson	Minimal	$\mathcal{O}(n^2 \log n \ + \ m \cdot n)$	2.5.3
Dechter et al.	Consistency	$\mathcal{O}(nW^*(d)^2)$	2.5.4
Bliek & Sam-Haroud	Minimal	$\mathcal{O}(n^3)$	2.5.5

We now give new theorems which more rigidly define the complexity of the STP. These theorems are proven in Appendix A.

Theorem 2.1 *Deciding consistency of an STP instance is* NL*-hard.*

Theorem 2.2 *Calculating the minimal network for, or deciding consistency of an STP instance is a member of* NC^2.

Theorem 2.3 *For constraint weights polynomially bounded by the number of variables, deciding consistency of an STP instance is a member of* NL.

Theorem 2.4 *For constraint weights polynomially bounded by the number of variables, calculating the minimal network for an STP instance is a member of* NL.

Algorithm 1: Floyd's and Warshall's APSP algorithm

```
1 for k ← 1 to n do
2 |  ∀i∀j : w_{i→j} ← min(w_{i→j}, w_{i→k} + w_{k→j})
3 end
```

It holds that $\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{NC}^2 \subseteq \mathsf{P}$.

The complexity class NL contains those problems that can be solved by a non-deterministic algorithm in logarithmic space; an interpretation of the NL-completeness result could be that a randomised approach may prove effective, since it is known in complexity theory that NL is equivalent to the class of problems solvable in logarithmic space by a type of probabilistic algorithms.

NC^2 problems can be solved by a parallel algorithm in $\mathcal{O}(\log^2 n)$ time using a polynomial amount of processors; these problems can therefore be said to be efficiently parallelisable. To our best knowledge, neither of these facts is made use of in current literature.

2.5 Solution techniques

Having established the complexity of solving the STP, we now move on to describing a selection of available algorithms for solving it.

2.5.1 Floyd's and Warshall's algorithm

This algorithm for calculating all-pairs-shortest-paths (APSP) on a graph was first published in 1962 [Flo62][War62] and computes the shortest distance between all pairs of vertices, or finds any negative cycle if it exists, in time $\mathcal{O}(n^3)$. We include it as Algorithm 1.

The algorithm runs a loop of n iterations, for $1 \leq k \leq n$. In each iteration, the algorithm computes for each pair (i, j) (including $i = j$) the shortest distance from node x_i via x_k to x_j and updates the weight $w_{i\to j}$ if the new value is less than the original value. After n iterations, all $w_{i\to j}$ have been set to their minimal values. The initial value of all $w_{i\to i}$ (the weight of the virtual edge from a node to itself) is taken to be zero; if it is ever to be changed to a negative value, a negative cycle has been detected and inconsistency can be concluded.

In Figure 2.3, we show the result of applying the APSP algorithm to our example problem from Section 2.2. The constraints $c_{0\to 4}$ and $c_{3\to 2}$ have been tightened, which means that we now know that our original information can

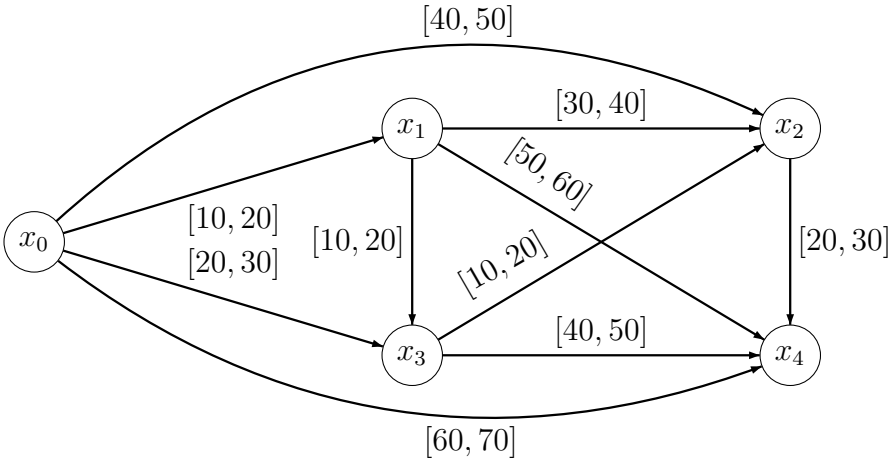


Figure 2.3: The minimal network after APSP

be refined: Fred must have arrived at work at 8:00 at the earliest, and John arrived at work at least 10 minutes after Fred left home.

We also note that the graph is now complete, which gives us additional information. For example, we know from constraint $c_{0\to 2}$ that John arrived at work between 7:40 and 7:50, and from constraint $c_{0\to 3}$ that Fred left his home between 7:20 and 7:30.

Note that APSP is equivalent to enforcing the *path consistency* property from constraint satisfaction literature.

Definition 2.1 (Path consistency) *A pair of constraints* $(c_{i\to j}, c_{j\to k})$ *is* path consistent *(PC) if all instantiations of* x_i *and* x_k *that satisfy* $c_{i\to k}$ *can be extended to an instantiation of* x_j *that satisfies both* $c_{i\to j}$ *and* $c_{j\to k}$.

An STP instance $\mathcal{S}$ *is path consistent if all of its constraints are.*

2.5.2 Bellman's and Ford's algorithm

This algorithm is based on publications by Bellman and Ford, in 1958 and 1962 respectively [Bel58][FF62]. It calculates single-source shortest paths to all vertices, instead of all-pairs shortest paths. It is similar to Dijkstra's algorithm [Dij59], but unlike the latter, it can deal with negative edge weights. It cannot be used to calculate the minimal graph, but it can be used to determine consistency.

If the graph contains a negative cycle, the algorithm will detect this in the final for loop. The reason for this is that in a negative cycle, the distance matrix will keep being updated and is never finished, so to speak; when the

Algorithm 2: Bellman's and Ford's algorithm

Input: Weighted directed graph $G = \langle V, A \rangle$, vertex $v_{\text{origin}} \in V$
Output: Distance matrix D

```
1  for i ← 1 to n do                    /* initialise distance matrix */
2  |  D[v_i] ← ∞
3  end
4  D[v_origin] ← 0
5  repeat n times
6  |  foreach (v_i, v_j) ∈ A do
7  |  |  D[v_j] ← min(D[v_j], D[v_i] + w_{i→j})
8  |  end
9  end
10 foreach (v_i, v_j) ∈ A do
11 |  if D[v_j] > D[v_i] + w_{i→j} then return INCONSISTENT
12 end
13 return D
```

updates of the distance matrix stop, there will always be an edge in a negative cycle that fails the condition in line 10.

2.5.3 Johnson's algorithm

Published in 1977 [Joh77], this algorithm may yield improved performance over Floyd's and Warshall's algorithm when run on sparse graphs.

The algorithm adds a new node v_0 with zero-weight edges to all other nodes $v_i \in V$ and then runs Bellman's and Ford's algorithm to compute the shortest paths from v_0 to all others, finding any negative cycles in the process. Then, the algorithm associates a value $h(v_i)$ with each original node $v_i \in V$; this value is equal to the shortest path from v_0 to v_i. These values are used to reweight the edges: $w'_{i \to j} = w_{i \to j} + h(v_i) - h(v_j)$. This reweighting scheme has two important properties: (i) all weights are now positive, and (ii) except for their total weight, the shortest paths are invariant.

Since the graph now no longer has negative edge weights, Dijkstra's algorithm [Dij59] can be used repeatedly to determine the shortest path from each node to all other nodes. These are then corrected again with the weights $h(v)$ to yield the shortest paths with the original edge weights. Note that in the absence of any negative edge weights, $h(v) = 0$ for all $v \in V$; i.e., no edge reweighting takes place.

Algorithm 3: Directed path consistency

for $k \leftarrow n$ **to** 1 **do**
  $\forall i < k, \forall j < k : w_{i \to j} \leftarrow \min(w_{i \to j}, w_{i \to k} + w_{k \to j})$
end

2.5.4 Directed path consistency

This method for determining consistency without calculating the minimal network was described in the publication by Dechter et al. that introduced, among other things, the STP [DMP91].

The algorithm assumes that there is a total ordering $\prec$ over the set of variables. We number the variables $\{x_1, \ldots, x_n\}$ such that $x_i \prec x_j$ if and only if $i < j$. The ordering has no impact on the soundness of the algorithm, but does influence its performance, as we will show below.

The algorithm iterates over k from n down to 1 and for all $i, j < k$ performs the same operation as APSP. Inconsistency can again be concluded as soon as any $w_{i \to i}$ drops below zero. The resemblance to Floyd's and Warshall's algorithm, i.e. to undirected path consistency, is clear. The sole difference is that undirected PC considers all pairs (i, j) throughout all iterations, whereas directed PC only considers those i and j less than k.

When running the algorithm, $w_{i \to j}$ will trivially remain unchanged if either $w_{i \to k}$ or $w_{k \to j}$ is infinite, i.e. if there is no constraint between v_i and v_k or between v_k and v_j. This means that these pairs (i, j) can be ignored by the algorithm, and explains why the ordering of the variables is significant for performance. In the next section, we will see that careful choice of this ordering will result in high efficiency.

2.5.5 Partial path consistency and the $\triangle$STP algorithm

In 2003, Xu and Choueiry [XC03c] proposed a new algorithm for solving the STP. They based their algorithm on a publication by Bliek and Sam-Haroud [BSH99], which introduces a new type of path consistency, called *partial path consistency* (PPC).

Standard path consistency is defined on complete graphs: to make a constraint graph PC, edges between all pairs of variables are considered and updated, even those that do not correspond to constraints that are explicitly defined by the problem and thus correspond to the universal constraint. Enforcing PC on an STP instance corresponds to calculating all-pairs-shortest-paths, as we have noted in Section 2.5.1. The property of partial path consistency is instead defined for chordal graphs, and considers no other edges than

Algorithm 4: PPC

Input: A chordal STN $\mathcal{S} = \langle V, A \rangle$

$Q \leftarrow A$
until Q *is empty* **do**
  $(v_i, v_j) \leftarrow$ an arc from Q
  $Q \leftarrow Q \setminus \{(v_i, v_j)\}$
  foreach v_k *such that* $(v_i, v_k) \in A$ *and* $(v_k, v_j) \in A$ **do**
    $w_{i \to j} \leftarrow \min(w_{i \to j}, w_{i \to k} + w_{k \to j})$
    if $w_{i \to j}$ *has changed* **then**
      $Q \leftarrow Q \cup \{(v_i, v_k), (v_k, v_j)\}$
    end
  end
end

those in the graph. Thus, if a constraint graph is triangulated by adding some fill edges (representing universal constraints) until each cycle of size greater than 3 has a chord, partial path consistency can be enforced. Since the triangulated graph generally contains far less edges than the complete graph, especially so for sparse constraint graphs, enforcing PPC is often far cheaper than enforcing PC.

Bliek and Sam-Haroud have proven [BSH99] that for problems with convex constraints, PPC is equivalent to PC. A constraint c is *convex* if the following proposition holds:

$$\forall x \forall y \forall z : x \leq y \leq z \wedge \text{satisfies}(x, c) \wedge \text{satisfies}(z, c) \rightarrow \text{satisfies}(y, c)$$

Informally, this means that for a constraint to be convex, if any single variable satisfies it for any two values x and z, it must also satisfy it for any value y in between.

Since STP constraints take the form of a single interval, it is easy to see that they are indeed convex. As we recall that making a STP path consistent amounts to solving the APSP problem, partial path consistency is an attractive method for tackling the STP. We include this algorithm as Algorithm 4.

In Figure 2.4, we show the result of applying the PPC algorithm to our example problem from Section 2.2; compare also with Figure 2.3. Again, the constraints $c_{0 \to 4}$ and $c_{3 \to 2}$ have been tightened, but only two new constraints have had to be added to the original problem to triangulate it, as opposed to five to complete it when the APSP algorithm was applied. Instead of the triangulation used here, which added edges representing $c_{0 \to 2}$ and $c_{0 \to 3}$ we

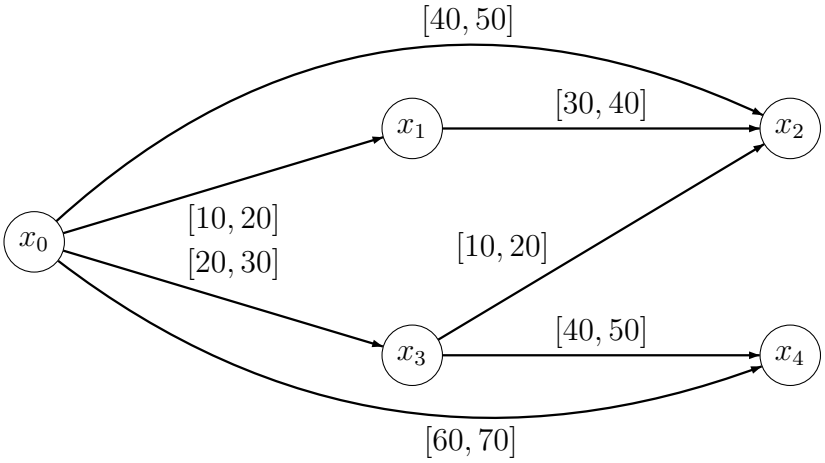


Figure 2.4: The result of applying PPC

could have chosen to add any pair of edges that were not already present and have a node in common.

Xu and Choueiry [XC03c] first realised that enforcing the PPC property suffices for solving the Simple Temporal Problem and implemented an efficient version of the algorithm, called $\triangle$STP. Their algorithm mainly differs from Algorithm 4 in that it considers the graph as being composed of triangles rather than edges, which saves some constraint checks. Even though DPC sometimes required less processing time in their tests, they stated that their algorithm performed better than any other solving method known at the time; their justification for this statement is that unlike $\triangle$STP, DPC does not compute the minimal network.

To the best of our knowledge, $\triangle$STP still represents the state of the art for this task; however, we will show in Appendix C that enforcing DPC along a carefully chosen variable ordering will always outperform $\triangle$STP for determining consistency. This ordering results in a merger between DPC and PPC that we shall call partial directed path consistency (PDPC).

2.6 Summary

In this chapter, we have seen that the Simple Temporal Problem allows for a concise and clear representation of temporal information. By means of calculating the minimal network, queries about the temporal information encoded in the instance can be answered efficiently.

We presented several algorithms for calculating this minimal network; the most used of these are Floyd's and Warshall's all-pairs-shortest-path

algorithm, well-known in graph literature, and the efficient $\triangle$`STP` algorithm, based on the property of partial path consistency from constraint satisfaction literature.

To determine whether an STP instance is consistent, i.e. whether there exists any solution at all, the minimal network need not be calculated. For this task, which corresponds to determining whether the graph contains a cycle with negative total weight, the directional path consistency algorithm and Bellman's and Ford's algorithm can be used.

In the next chapter, we will see that there are temporal problems for which the limited expressiveness of the STP does not suffice; however, the extensions we will discuss are built upon the same basic idea. Also, the fact that the STP can be solved quite efficiently proves to be very useful.

Chapter 3

The TCSP and the DTP

3.1 Introduction

The Simple Temporal Problem can represent only a rather limited scope of problems, which is of course why it can be solved so efficiently. If there are several alternative ways to perform an action, or one has to express that two events may occur in any order but must not overlap, the STP formalism no longer suffices. One has to make use of a formalism that allows for *disjunctions* to be modelled.

The STP formalism can easily be extended by allowing the constraints to take the shape of a disjunction (or union) of multiple temporal intervals instead of just a single interval. This extension has first been described as the *Temporal Constraint Satisfaction Problem* in the publication by Dechter, Meiri and Pearl [DMP91] that also defined the STP. Stergiou and Koubarakis [SK00] further expanded on the idea of disjunction with what they called the Deciding Disjunctions of Temporal Constraints problem, which is usually referred to as *Disjunctive Temporal Problem*.

By allowing disjunctions of temporal intervals, the scope of addressable problems widens greatly, at the cost that these problems are far harder to solve. In fact, it is in general infeasible to solve them for large instances: they are NP-complete. What exactly constitutes a 'large instance' differs from case to case, but real-world problems usually fall in this category.

In this chapter, we start by expanding our example to include disjunctions, thereby giving an idea of the increased scope of problems that can be addressed. Next, we give formal definitions of the problems and describe their complexity. Then, we discuss some methods for *preprocessing* problem instances; not part of the solving proper, they serve to reduce the search space before search starts and thus improve performance of the solving pro-

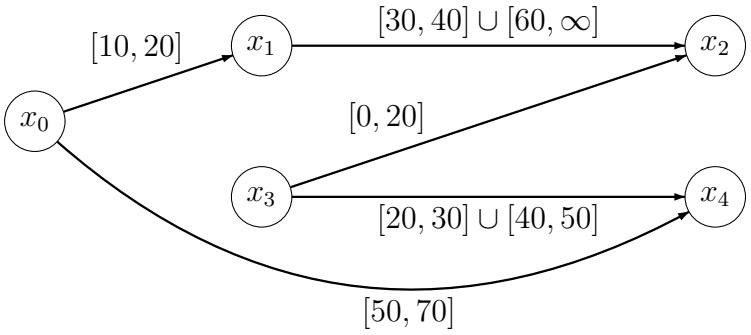


Figure 3.1: An example TCSP instance

cess. After this, we present a selection of algorithms for tackling the TCSP; since this problem has a simpler expression than the DTP, the algorithms also tend to be more straightforward. Finally, we present algorithms for the DTP. It is in this area that the most recent advances have been made.

3.2 Examples

We extend the example from Section 2.2. First, we extend the STP to a TCSP by giving both John and Fred some extra options, again taken from [DMP91]. John has the choice to take a bus, increasing his travel time to at least an hour, instead of taking his car; Fred may decide to take his own car instead of joining the carpool, thereby saving exactly 20 minutes.

The result of these choices is the TCSP instance shown in Figure 3.1. We see that the arcs representing $c_{1\to 2}$ and $c_{3\to 4}$ are now labelled by two intervals each.

Now, on to the DTP. In a DTP instance, we can easily express rules like the following. Suppose that John and Fred share a single parking lot at their office, and that they therefore have agreed never to both come to work with their cars on the same day. To incorporate this rule, we first have to transform the TCSP instance into a DTP, which is straightforward, as we will show in Section 3.4. After transformation, we add the following constraint:

$$x_1 - x_2 \leq -50 \ \vee \ x_3 - x_4 \leq -35$$

This constraint says that John or Fred, or both John and Fred, must have a travel time exceeding a threshold of respectively 50 and 35 minutes, thus limiting their choice of transportation.

3.3 Definition

3.3.1 The Temporal Constraint Satisfaction Problem

In a Temporal Constraint Satisfaction Problem* (TCSP), a constraint $c_{i \to j}$ between two time points x_i and x_j can be expressed as a union of intervals:

$$x_j - x_i \in I_{ij1} \cup \cdots \cup I_{ijn} = [a_{ij1}, b_{ij1}] \cup \cdots \cup [a_{ijn}, b_{ijn}]$$

This allows the time difference between the two time points to assume a value from any of the intervals, instead of from just a single interval. The network representation of the TCSP is similar to the STN; the only difference is that arcs are not labelled by a single interval, but by a union of intervals.

3.3.2 The Disjunctive Temporal Problem

In the Disjunctive Temporal Problem (DTP), every constraint is a disjunction of inequalities, each involving two time points:

$$(x_{j_1} - x_{i_1}) \leq w_{i_1 j_1} \vee \cdots \vee (x_{j_n} - x_{i_n}) \leq w_{i_n j_n}$$

The most important difference with the TCSP is that more than two different time points may participate in a single DTP constraint, whereas the TCSP constraints are binary. Conversion from the TCSP to the DTP is therefore much more straightforward than vice versa, and for the same reason, there is no simple network representation of the DTP: arcs are now no longer sufficient to represent DTP constraints, as they involve only two variables.

3.3.3 Object-level and meta-level

The TCSP and the DTP are both usually solved with some flavour of backtracking search. An instance can be tackled in two ways:

- In the *object-level* approach, values are assigned to all time points while making sure that all constraints are satisfied;
- In the *meta-level* approach, an interval or an inequality is selected for each constraint, while making sure that the resulting STP, which is called a *component STP* or a *labelling*, remains consistent.

*Note that not all authors adhere to this nomenclature; they often reserve the phrase "temporal constraint satisfaction problem", being rather wide in scope, to refer to *any* CSP that deals with time. Since all constraints in the TCSP are binary, the abbreviation bTCSP is sometimes used instead (e.g. [Mof06]). However, in this document, we will use the abbreviation TCSP to denote this binary problem only.

All published algorithms use the latter approach, in which efficiently solving the STP is clearly of prime importance.

A note on terminology. In the meta-level approach, the original object-level constraints take on the role of variables, an instantiation of which corresponds to the selection of an interval (in the case of the TCSP) or an inequality (in the case of the DTP), and thus corresponds to a component STP. No explicit meta-level constraints exist; the single implicit meta-level constraint is satisfied if and only if an instantiation corresponds to a consistent component STP.

We now summarise the terminology used in the remainder of the chapter, mostly conforming to the nomenclature adopted by Tsamardinos and Pollack [TP03]:

- 'time point', 'node' and 'vertex' refer to an object-level variable, denoted by x_k.
- 'variable' refers to a meta-level variable (i.e. an object-level constraint), denoted by c_i.
- 'value' and 'arc' refer to a single inequality or interval as used in a component STP; the jth value for c_i is denoted by c_{ij}, and an arc from time point x_i to time point x_j is denoted by $c_{i \to j}$.*
- 'instantiation' refers to a selection of a meta-value for a meta-variable, or a set of such selections.
- 'domain' refers to the set of (remaining) valid instantiations for a meta-variable; for c_i, this is a set $\{c_{i1}, \ldots, c_{in}\}$.

3.4 Complexity

Deciding consistency of an instance is NP-complete for both the TCSP and the DTP. Membership in NP is obvious. Dechter et al. [DMP91] include a proof of hardness for the TCSP in the publication which defines the problem, using a reduction from THREE-COLOURABILITY, which we reproduce here almost verbatim.

Theorem 3.1 *Deciding consistency for the TCSP is* NP*-hard.*

*It may seem confusing here that we adopt such similar denotations c_{ij} and $c_{i \to j}$ for different concepts; however, the intended meaning will always be clear from context.

Proof Reduction from THREE-COLOURABILITY. Let $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$ be a graph to be coloured; we construct a TCSP T in the following way. First, we define a temporal reference point x_0. Then, for each node $v_i \in V$, we introduce a time point x_i and a constraint $c_{0\to i} : x_i - x_0 \in \{1, 2, 3\}$, where the set $\{1, 2, 3\}$ corresponds to the three allowed colours. Finally, for each edge $\{v_i, v_j\} \in E$ we add a constraint $c_{i\to j} : x_j - x_i \in [-\infty, -1] \cup [1, \infty]$, which ensures that the time points representing connected nodes take different values. Hence, T is consistent if and only if G is three-colourable. □

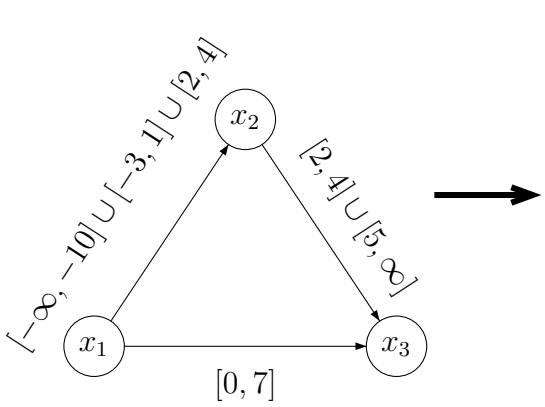
The complexity of calculating the minimal network for the TCSP is only given as FNP-hard in the literature; Schwalb and Dechter write that determining minimality of a given network is NP-hard [SD97].

Indeed, the minimal network may in some cases be super-polynomial (if not quite exponential) in the size of the original instance; see the discussion on fragmentation in Section 3.5.1 below. However, for weights polynomially bounded in the number of time points, calculating the minimal network becomes feasible in FNP. This follows from an argument similar to the one we make in Appendix A.2.2 on the complexity of calculating the minimal network for the STP.

Curiously, Stergiou and Koubarakis nowhere explicitly establish the complexity of deciding consistency for their DTP. An instance of the TCSP can however trivially be translated into an instance of the DTP by means of splitting each constraint c_i (having $|c_i|$ intervals) into $|c_i| - 1$ DTP constraints with two inequalities each, plus at most 2 singleton DTP constraints, as we show in Figure 3.2. This results in a DTP instance with size differing by no more than a constant factor* from the original TCSP instance, and also shows that deciding consistency for the DTP is as hard as it is for the TCSP.

Both the TCSP and the DTP belong to the equivalence class of NP-complete problems and can therefore be said to have the same complexity. Though it may not be expected at first sight, we show in Appendix B that there also exists a linear transformation back from the DTP to the TCSP. Since these problems are so closely related, the simplest algorithms for solving them also show a close kinship, as we shall see in Sections 3.6 and 3.7. First, we will discuss methods that can be used to enhance the efficiency of search.

*Despite the fact that the DTP instance I' has more constraints than the TCSP instance I, the reader can verify that a reasonable representation of both instances yields $|I'| = \mathcal{O}(|I|)$.



$$
\begin{array}{rcl|rcl}
x_2 - x_1 \leq -10 & \vee & x_1 - x_2 \leq 3 \\
x_2 - x_1 \leq 1 & \vee & x_1 - x_2 \leq -2 \\
x_2 - x_1 \leq 4 & & \\
\\
x_1 - x_3 \leq 0 & & \\
x_3 - x_1 \leq 7 & & \\
\\
x_2 - x_3 \leq -2 & & \\
x_3 - x_2 \leq 4 & \vee & x_2 - x_3 \leq -5
\end{array}
$$

Figure 3.2: Transformation from TCSP to DTP

3.5 Preprocessing

Before initiating the actual solving procedure of an instance, some form of preprocessing can be performed to prune the search space, thereby making the solving itself more efficient.

In this section, we first describe preprocessing methods for the TCSP, which are all based on the *path consistency* (PC) property, which we defined in Section 2.5.1. Because DTP constraints are in general of higher than binary order and disjunctive in nature, the path consistency method cannot be usefully employed to them; however, we shall describe several preprocessing methods of a different nature.

3.5.1 Path consistency

To show how path consistency can be enforced on TCSP instances, we need to define the composition operation.

Definition 3.1 (Composition) *The* composition *of two binary TCSP constraints, denoted $c_1 \odot c_2$, corresponds to the set*

$$\{r \mid \exists s, t : s \in c_1 \wedge t \in c_2 \wedge r = s + t\}$$

See Figure 3.3 for an example of this operation. Informally, $c_1 \odot c_2$ can be associated with a "summation" of the two constraints.

Proposition 3.2 *$c_{i \to j}$ and $c_{j \to k}$ are path consistent if and only if $c_{i \to k} \subseteq c_{i \to j} \odot c_{j \to k}$.*

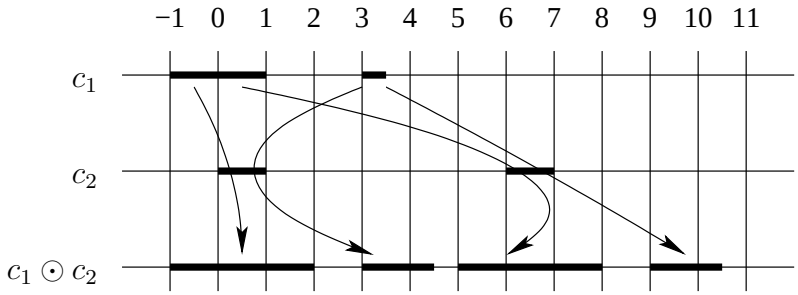


Figure 3.3: Composition

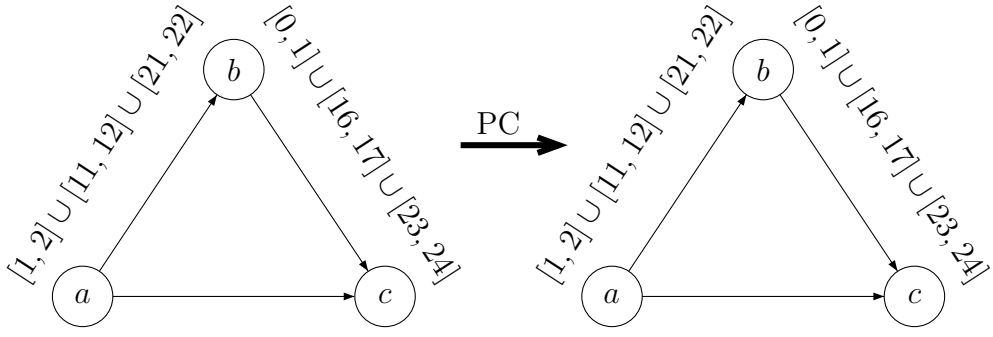


Figure 3.4: The PC property and fragmentation

Algorithm 5: Path consistency

```
for k ← 1 to n do
    ∀i∀j : c_{i→j} ← c_{i→j} ∩ (c_{i→k} ⊙ c_{k→j})
end
```

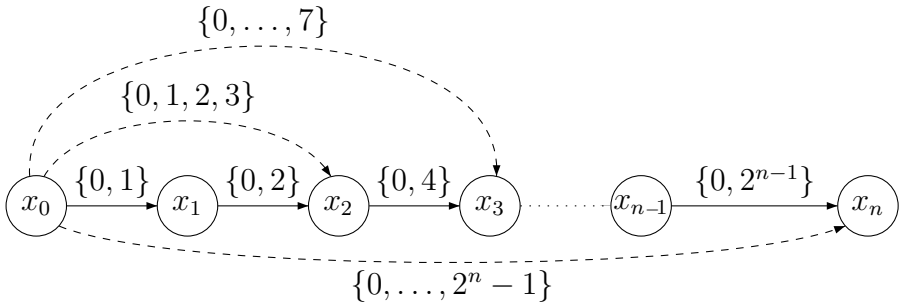


Figure 3.5: Exponential fragmentation

Algorithm 5 can be used for enforcing path consistency on a TCSP instance. Note the great similarity to APSP (Algorithm 1); indeed, the two algorithms are identical when applied to an STP instance.

For the result of enforcing PC on a constraint, see Figure 3.4. All object-level values from the domain of $c_{a\to c}$ in the left-hand side of the figure that are not consistent with the constraints $c_{a\to b}$ and $c_{b\to c}$ have been removed in the right-hand side domain. Note that the number of intervals labelling $c_{i\to j}$ has more than tripled; this phenomenon is called *fragmentation*.

Fragmentation can cause the number of intervals labelling a single constraint to explode, possibly even assuming exponential proportions; we show this in Figure 3.5. In this figure, we use sets of two integers as constraints, which are of course unions of two singleton intervals. The solid arcs represent original constraints; the dashed arcs are added by enforcing PC. After applying PC, the longest arc holds 2^n intervals, which is exponential in the number of time points.

3.5.2 Upper-lower tightening

To avoid the problem of fragmentation, several approximations of PC have been proposed. These will never cause the number of intervals labelling an edge to increase, and therefore can be enforced in polynomial time. One of these approximations is *upper-lower tightening (ULT)* [SD93], in which only the upper and lower bound imposed by each constraint are updated in each iteration, and all internal interval boundaries are ignored.

Algorithm 6: Upper-lower tightening

Input: TCSP $\mathcal{T} = \langle X, C \rangle$
Output: Tightened TCSP $\mathcal{T}'$

```
1  repeat forever
2  |  C' ← ∅
3  |  foreach constraint c_i ∈ C do
4  |  |  c'_i ← [min c_i, max c_i]                    /* circumscribe c_i */
5  |  |  C' ← C' ∪ c'_i
6  |  end
7  |  ⟨X, C'⟩ ← Solve-STP(⟨X, C'⟩)
8  |  C'' ← ∅
9  |  foreach constraint c'_i ∈ C' do
10 |  |  if c'_i = ∅ then return INCONSISTENT
11 |  |  c''_i ← c_i ∩ c'_i
12 |  |  C'' ← C'' ∪ c''_i
13 |  end
14 |  if C'' = C then return ⟨X, C''⟩      /* fixed point reached */
15 |  else C ← C''
16 loop
```

This approach comes down to circumscribing a TCSP as an STP by taking the upper and lower bound for each constraint, thus summarising each constraint with a single interval. For the resulting STP, the minimal network can then be calculated by an algorithm like APSP or $\triangle$`STP`; the resulting interval for each constraint in this minimal network is then intersected with the original TCSP constraint. This intersection operation could eliminate some of the original intervals, leading to a different circumscribing STP, in which case the process is repeated. Upper-lower tightening is included as Algorithm 6.

The ULT algorithm repeats this procedure until an inconsistency is discovered by the STP solver or a fixed point is reached. If during some iteration, line 11 eliminates no intervals from any constraint, it is not hard to see that the next circumscribing STP is identical to the current minimal STP, after which the algorithm will terminate. Clearly then, this method requires solving an STP at most k times, where k is the total number of intervals in the original TCSP instance.

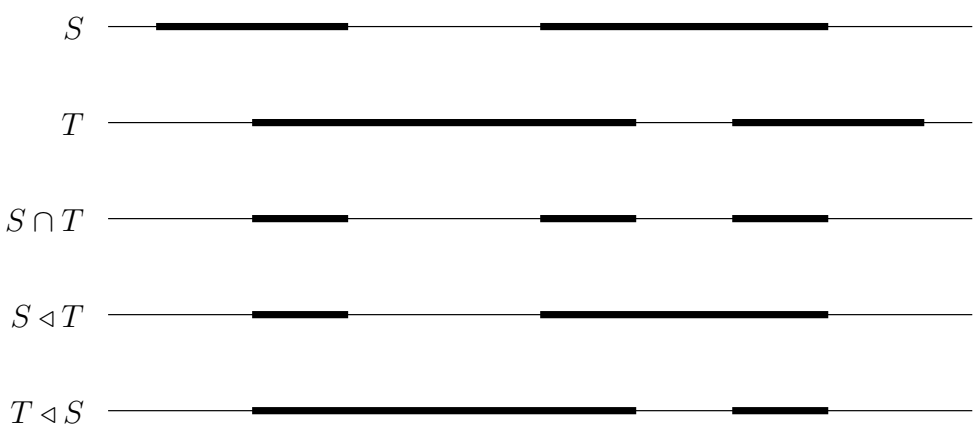


Figure 3.6: Standard versus loose intersection

3.5.3 Loose path consistency

A more sophisticated algorithm for approximating PC is *loose path consistency (LPC)* [SD97]. This method hinges on the insight that fragmentation is caused by the intersection operation, so it employs an operation called *loose intersection* instead.

Definition 3.2 (Loose intersection) *Let S and T be unions of intervals, $S = I_1 \cup \cdots \cup I_n$ and $T = J_1 \cup \cdots \cup J_m$. Now, define I'_i as follows:*

$$I'_i = \begin{cases} \emptyset & \text{if } I_i \cap T = \emptyset \\ [\ \min(I_i \cap T)\ ,\ \max(I_i \cap T)\ ] & \text{otherwise} \end{cases}$$

The loose intersection of S and T, denoted $S \triangleleft T$, is then equal to $\bigcup_{i=1}^{n} I'_i$.

Note that loose intersection is a non-commutative operation and in general $S \triangleleft T \neq T \triangleleft S$. Informally, $S \triangleleft T$ is equal to S with some of its intervals shrunk or eliminated; see Figure 3.6. It follows directly from the definition that loose intersection will never yield a number of intervals larger than the number of intervals in the left-hand operand, which solves the problem of fragmentation.

The loose path consistency operation computes

$$\forall i < j : c'_{i \to j} \leftarrow \bigcap_{\forall k} (c_{i \to k} \odot c_{k \to j})$$

and assigns $c_{i \to j} \leftarrow c_{i \to j} \triangleleft c'_{i \to j}$.

The authors discuss several variations of the LPC algorithm, of which the one they identify as having the best trade-off between effectiveness and

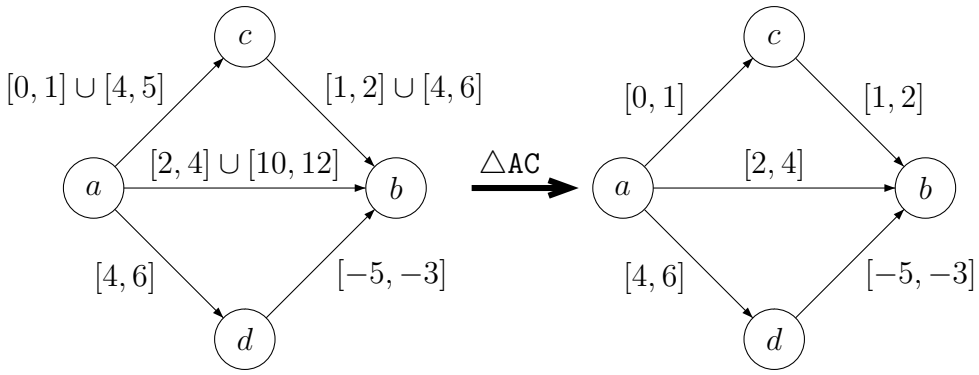


Figure 3.7: The △AC algorithm in operation.

efficiency uses a partial approach reminiscent of PPC; it only considers triangles of which at least two constraint edges were present, i.e. non-universal, in the original constraint graph.

3.5.4 △Arc-consistency

The final preprocessing method for the TCSP that we include was described by Xu and Choueiry [XC03a]. It considers the TCSP at the meta-level and is derived from the *arc consistency* property in CSP literature.

Definition 3.3 (Arc-consistency) *A constraint c involving variables X_c is* arc-consistent *if for each variable $x \in X_c$ having domain $d(x)$, an instantiation of x to any value in $d(x)$ can be extended to an instantiation for all variables in X_c such that c is satisfied.*

TCSP constraints are trivially arc-consistent; however, △AC considers a set of meta-TCSP constraints that correspond to the set of all triangles in the original TCSP. The meta-variables participating in this meta-constraint correspond to the edges in the triangle, and the domain of each of these meta-variables consists of the intervals labelling the edge.

Now, for each (meta-)value in the domain of each (meta-)variable, it is checked whether it is supported by values in the domain the other two variables of the (meta-)constraint. Such support automatically implies reciprocal support to each of the two supporting values. If support does not exist, the value is deleted from its domain, and other values that were supported by this value are re-checked for still having support from some other value.

As an example, consider the small network in Figure 3.7. If we first consider triangle $\{a, b, c\}$, the three lower intervals mutually support each

other, as do the three higher ones. Looking at triangle $\{a, b, d\}$, we see that the interval $[10, 12]$ in the domain of constraint $c_{a\to b}$ has no support, since $c_{a\to d} \odot c_{d\to b} = [-1, 3]$, which has an empty intersection with $[10, 12]$. In turn, removing this interval from $c_{a\to b}$ causes the higher intervals of constraints $c_{a\to c}$ and $c_{c\to b}$ to lose their support. The resulting network is shown on the right-hand side of the figure.

Because the algorithm considers the meta-level, it makes an all-or-nothing choice: either an interval is removed wholesale or it remains unchanged, but it is never tightened. Xu and Choueiry cite the publication that proposed the loose path consistency (LPC) algorithm six years prior to their research [SD97], but very selectively indeed: they ignore the LPC algorithm itself and only mention the weaker upper-lower tightening procedure (ULT), which had been published even longer before [SD93]. In their conclusions, they state that an interesting direction for future research would be a merger of $\triangle$AC with ULT, completely overlooking the fact that the LPC algorithm is just that (and more).

3.5.5 Preprocessing the DTP

The structure of the DTP makes the general path consistency method impractical; however, some preprocessing can still be done. We will list the approach taken by three of the algorithms described in the next section.

Stergiou's and Koubarakis's algorithm

Stergiou and Koubarakis [SK00] first proposed the DTP formalism along with a solver which we will discuss in Section 3.7.1. They also described some preprocessing rules:

- If the domain of a (meta)variable c_i contains an inequality that is subsumed by a variable c_j with just one inequality in its domain, c_i can be deleted. The reason for this is that satisfying c_j implies satisfying c_i.

  As an example, take these disjunctions:

  $$\begin{aligned} c_1 &: \quad x_2 - x_1 \leq 4 \vee x_2 - x_3 \leq 2 \\ c_2 &: \quad\quad\quad x_2 - x_1 \leq 3 \end{aligned}$$

  Clearly, if c_2 is satisfied, the first inequality in c_1 is also true, satisfying c_1. Therefore, c_1 can be disregarded during solving.

- If each value in the domain of a variable c_j subsumes a value from the domain of some variable c_i, the latter can be deleted; this is just a generalisation of the previous rule.

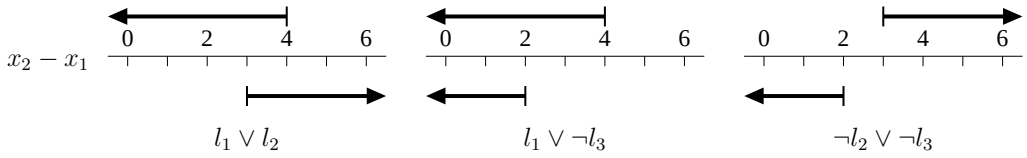


Figure 3.8: Preprocessing rules in the TSAT algorithms

For example, c_3 can be disregarded when solving the following set of disjunctions, because in any solution, either c_{41} or c_{42} must be satisfied, the former subsuming c_{31} and the latter subsuming c_{33}; c_3 is satisfied in both cases.

$$\begin{aligned} c_3: &\quad x_3 - x_2 \leq 9 \vee x_1 - x_4 \leq -2 \vee x_3 - x_4 \leq 6 \\ c_4: &\quad x_3 - x_2 \leq 4 \vee x_3 - x_4 \leq 5 \end{aligned}$$

- If the domain of a variable c_i contains a value c_{ik} that is inconsistent with a some value in a singleton domain, c_{ik} can be deleted from the domain of c_i. If this was the last value in c_i's domain, the entire DTP is inconsistent.

  We again provide an example. The second inequality of c_5 is inconsistent with c_6 and can be pruned:

$$\begin{aligned} c_5: &\quad x_5 - x_2 \leq 3 \vee x_3 - x_5 \leq 1 \\ c_6: &\quad x_5 - x_3 \leq -2 \end{aligned}$$

  If there were also a variable $c_7 : x_2 - x_5 \leq -8$, this would cause c_5's domain to become empty, making the DTP inconsistent.

TSAT and TSAT++

In the TSAT and TSAT++ algorithms, described in Sections 3.7.3 and 3.7.6, additional preprocessing is necessary, because the propositional satisfiability solver that is used to generate instantiations is unaware of the underlying inequalities. To avoid the generation of trivially inconsistent instantiations, all pairs of literals (i.e. individual inequalities in constraints) are checked for consistency. Since a pair of literals can only be inconsistent if it represents a negative cycle, it is clear that only pairs which each involve the same two time points need be considered.

As an example of preprocessing, consider the following literals:

$$\begin{aligned} l_1: &\quad x_2 - x_1 \leq 4 \\ l_2: &\quad x_1 - x_2 \leq -3 \\ l_3: &\quad x_2 - x_1 \leq 2 \end{aligned}$$

In the preprocessing step, the three possible pairings of these literals are considered and one of three possible conclusions is drawn, resulting in a new constraint. For the literals just listed, we depict this process in Figure 3.8. We represent the difference $x_2 - x_1$ with a number line; the literals are represented by arrows over and under this line to show where they are true. Below the number line, we show the conclusions that are drawn by the preprocessing step, which results in three new constraints:

$$l_1 \vee l_2 \quad \wedge \quad l_1 \vee \neg l_3 \quad \wedge \quad \neg l_2 \vee \neg l_3$$

Note that TSAT included an earlier version of this preprocessing method, which only considered the right-hand case and thus yielded just a single new constraint.

While current algorithms consider only pairs of literals, this method could easily be extended to consider larger sets, thereby further pruning the search space. However, some limit must be imposed, because the amount of sets of size i is of order $\mathcal{O}(n^i)$ in the worst case; the amount of clauses added by preprocessing is thus also of this size. An interesting topic for future research might be to find out at which i the overhead becomes prohibitive.

3.6 Solving the TCSP

3.6.1 Standard backtracking

The publication by Dechter, Meiri and Pearl [DMP91] that defined the STP and the TCSP also included a simple algorithm for calculating the minimal network of the TCSP. We paraphrase their recursive backtracking procedure as Algorithm 7. It is invoked with the TCSP $\mathcal{T} = \langle X, C \rangle$ and an empty labelling $\mathcal{S} = \langle X, \emptyset \rangle$ as input.

For the procedure *Solve-STP*, Dechter et al. use the APSP algorithm; for *Consistent-STP*, they use directed path consistency.

3.6.2 Improvements

Xu and Choueiry [XC03b] gave some improvements to the basic algorithm by Dechter et al.

Less consistency checks Xu and Choueiry note that a component STN without cycles is necessarily consistent, and that adding an edge to a consistent STP can only result in inconsistency if a new cycle is formed by this addition.

Algorithm 7: Backtracking algorithm for solving the TCSP

Input: TCSP $\mathcal{T} = \langle X, C \rangle$, component STP $\mathcal{S} = \langle X, C' \rangle$
Output: Minimal network M

```
1  if C = ∅ then
2  |  return Solve-STP(S)
3  else
4  |  c_{i→j} ← first value in C
5  |  C ← C \ {c_{i→j}}
6  |  M ← ∅
7  |  foreach interval I in c_{i→j} do
8  |  |  c'_{i→j} ← I
9  |  |  C' ← C' ∪ {c'_{i→j}}
10 |  |  if Consistent-STP(⟨X, C'⟩) then
11 |  |  |  M = M ∪ Solve-TCSP(T = ⟨X, C⟩, S = ⟨X, C'⟩)
12 |  |  end
13 |  |  C' ← C' \ {c'_{i→j}}
14 |  end
15 |  return M
16 end
```

The call to *Consistent-STP* in line 10 of Algorithm 7 is therefore only necessary if the addition of $c'_{i\to j}$ to the component STP $\mathcal{S}$ formed a new cycle.

Edge-ordering heuristic A general rule of thumb for ordering search is the "fail first principle": if the algorithm has entered a dead-end branch of the search tree and must backtrack, the sooner the better. The basic algorithm selects the edges in no particular order (line 4 of Algorithm 7); Xu and Choueiry give precedence to edges that participate in many triangles.

Preprocessing with $\triangle$AC Xu and Choueiry devised a preprocessing step that eliminates those (meta-)values from (meta-)variables that have no support in some triangle; in other words, for each variable $c_{i\to j}$, only those values are kept which can be extended to consistent instantiations of all triangles in which $c_{i\to j}$ participates.

We described this preprocessing step in more detail in Section 3.5.4.

Using $\triangle$STP for solving component STPs One of the most obvious improvements is the application of the highly efficient $\triangle$STP algorithm for

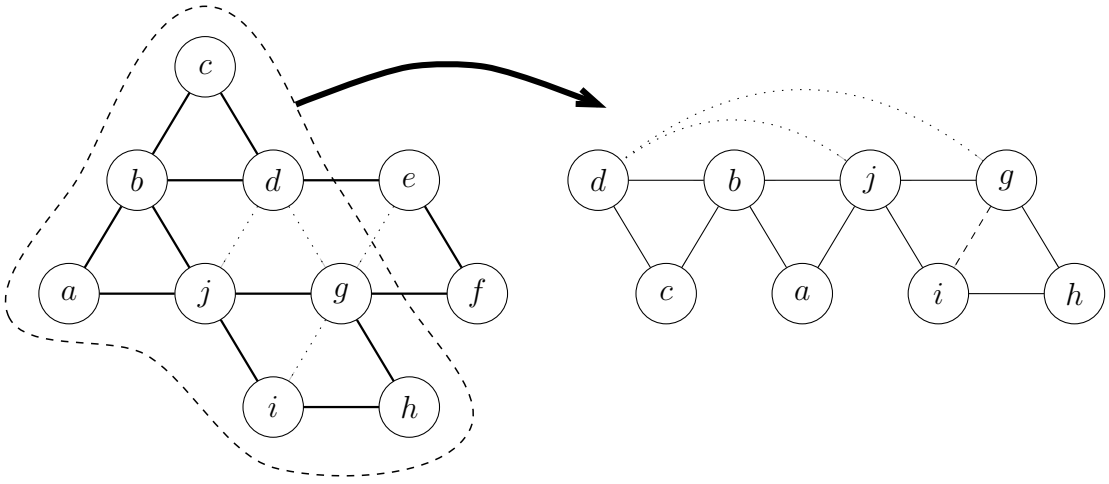


Figure 3.9: Effect of triangulation plan

determining consistency and calculating the minimal network of component STPs, in lines 2 and 10 of Algorithm 7, respectively.

For dense STPs, DPC is sometimes more efficient than $\triangle$STP for determining consistency, but Xu and Choueiry have demonstrated that this point is moot. The reason for this is that the backtracking algorithm requires building up the component STP step by step; thus, most of the time, the STP being processed is much sparser than the actual constraint graph of the TCSP being solved.

Precomputing a triangulation of the constraint graph Since the edge-ordering heuristic is run beforehand and the edges are instantiated in a fixed order, the triangulation of the constraint graph can also be done only once, before the actual solving starts.

Xu and Choueiry devised two triangulation plans. The first is to build up the constraint graph step by step, and to compute a triangulation at every step; the second is to compute only the triangulation of the full TCSP constraint graph and to induce from that the triangulated graph for each level of the search.

In Figure 3.9, we depict a constraint graph for a TCSP on the left-hand side; solid edges denote constraints specified by the problem and dotted edges denote edges added by triangulating the constraint graph. Let us assume that at some point during backtracking search, all edges but those involving nodes e and f are instantiated. On the right-hand side, we show the constraint graph as it has been instantiated at this point in search, where solid edges again denote original constraint edges. Triangulating the solid right-hand

graph yields only one extra edge $\{g, i\}$ (dashed in the figure), but when we induce the triangulation from the already triangulated left-hand graph, we also get the edges $\{d, g\}$ and $\{d, j\}$ (dotted in the figure). The latter method thus requires the $\triangle$STP algorithm to deal with two more triangles; the same solution is attained with more calculation.

Xu and Choueiry conclude that to improve performance of the algorithm, the best choice is the second triangulation plan: triangulate the constraint graph once for each level in the search tree.

3.7 Solving the DTP

Having covered the TCSP, we now turn our attention to the DTP, which as the reader will remember allows constraints to involve more than two variables.

Like Dechter et al., Stergiou and Koubarakis [SK00], who first defined the DTP formalism, also included a basic algorithm for solving it, along with several improvements; we discuss their method in Section 3.7.1.

Their basic algorithm was improved upon by Oddi and Cesta [OC00] (Section 3.7.4) and again by Tsamardinos and Pollack [TP03] (Section 3.7.5). Interspersed between the original publication by Stergiou and Koubarakis and the improvements of Oddi and Cesta was the TSAT algorithm by Armando, Castellini and Giunchiglia [ACG00], described in Section 3.7.3. They approached the problem from the perspective of propositional satisfiability (SAT). They later teamed up with Maratea [ACGM05][ACG$^+$05] and developed TSAT++ , discussed in Section 3.7.6, which improves upon their previous results; to our best knowledge, this algorithm still represents the current state of the art.

3.7.1 Stergiou's and Koubarakis' algorithm

The basic algorithm described by Stergiou and Koubarakis [SK00] has at its base the incremental directed path consistency (IDPC) algorithm by Chleq [Chl95]. As its name suggests, this algorithm enforces directed path consistency incrementally for each constraint instantiated. Recall that DPC can be used to decide consistency of the STP. Thus, enforcing DPC incrementally rather than starting anew for every component STP encountered during search is desirable for a backtracking algorithm, in which a component STP is built up step by step.

Stergiou and Koubarakis make a few adjustments to Chleq's IDPC al-

Algorithm 8: Incremental directed path consistency (IDPC)

Input: A directed component STP $G = (V, A)$ and a new arc $x_m - x_l \leq w_{l \to m}$.

Output: CONSISTENT if the new arc has been consistently added to G; INCONSISTENT otherwise.

```
1  initialise all entries in Π to FALSE
2  Q ← {max_≺({x_l, x_m})}
3  Π[x_l][x_m] ← TRUE
4  A ← A ∪ (x_l, x_m)
5  while Q ≠ ∅ do
6  |  x_k ← max_≺(Q)
7  |  Q ← Q \ {x_k}
8  |  forall x_i, x_j ≺ x_k, i ≠ j such that
   |  {(x_i, x_k), (x_k, x_j)} ⊆ A ∧ (Π[x_i][x_k] ∨ Π[x_k][x_j]) do
9  |  |  if (x_i, x_j) ∉ A ∨ w_{i→j} > w_{i→k} + w_{k→j} then
10 |  |  |  w_{i→j} ← w_{i→k} + w_{k→j}
11 |  |  |  A ← A ∪ {(x_i, x_j)}
12 |  |  |  if w_{i→j} + w_{j→i} < 0 then return INCONSISTENT
13 |  |  |  Q ← Q ∪ {max_≺({x_i, x_j})}
14 |  |  |  Π[x_i][x_j] ← TRUE
15 |  |  end
16 |  end
17 end
18 return CONSISTENT
```

gorithm. Unfortunately, some of these changes introduced errors.* In the version of IDPC we include here, as Algorithm 8, we have corrected these errors.

IDPC is called as a subroutine in a backtracking algorithm similar to the original TCSP algorithm by Dechter et al., with the following changes (line numbers refer to Algorithm 7):

- Instead of selecting an interval (in line 7), an inequality is selected;

*These errors include:

- Added arcs (x_l, x_m) must satisfy $x_l \prec x_m$; arcs in the reverse direction cannot be added. Thus, the STP that is built up is a tree and is trivially consistent.
- The comparison in line 9 was $w_{i \to j} < w_{i \to k} + w_{k \to j}$.
- The comparison in line 12 was $w_{i \to j} - w_{j \to i} < 0$.

- The original algorithm calculates the minimal network. For the DTP, the concept of "minimal network" is undefined; instead, the algorithm returns each individual satisfying instantiation of (meta-)variables.

- As mentioned, the call to *Consistent-STP* in line 10 is implemented by Chleq's IDPC algorithm instead of standard DPC.

3.7.2 Improvements

Stergiou and Koubarakis propose several improvements to their basic algorithm, well known in general CSP literature (e.g. [HE80], [Pro93], [FD95]), that improve efficiency by pruning the search space.

Preprocessing Before the search itself starts, a preprocessing step detailed in Section 3.5.5 is performed. This step prunes the search space by removing those constraints that are trivially satisfied and those inequalities that are trivially inconsistent with any solution.

Backjumping When a dead-end is encountered (the check in line 10 of Algorithm 7 fails for all values in a domain), the original algorithm backtracks just one step; however, the cause of the problem may lay farther in the past. The technique of *backjumping*, also known as conflict-directed backtracking, skips those variables that play no role in the current inconsistency, and immediately reverts to the most recent variable that does. This avoids fruitlessly instantiating the variables in between to other values.

Forward checking When some variable is instantiated, this may not lead to a direct inconsistency, but it may remove all remaining values in the domain of some variable farther down the line. The technique of *forward checking* runs each (meta-)variable instantiation by the domains of all remaining variables, eliminating those values which are invalidated; if the domain of some variable becomes empty, the instantiation is reverted.

Variable ordering The standard algorithm imposes no particular order on the instantiation of the variables. The ordering can be determined statically, before the search starts, as Xu and Choueiry do for the TCSP; but Stergiou and Koubarakis propose a dynamic ordering scheme. They use the *minimum remaining values* heuristic, giving precedence to those variables that have the least values left, again adhering to the "fail first principle". This technique is most successful when combined with forward checking, which causes domains

Algorithm 9: TSAT

Input: DTP $\mathcal{D} = \langle X, C \rangle$, component STP $\mathcal{S} = \langle X, C' \rangle$

if $C = \emptyset$ **then return** Solve-STP($\mathcal{S}$)
else if $\emptyset \in C$ **then return** FALSE /* empty clause */
else if $\exists\{l\} \in C$ **then** /* unit clause */
  return TSAT(simplify($\mathcal{D}$,$C' \cup \{l\}$), $\langle X, C' \cup \{l\} \rangle$)
else
  choose $l \in c \in C$ /* branch */
  return TSAT(simplify($\mathcal{D}$,$C' \cup \{l\}$) , $\langle X, C' \cup \{l\} \rangle$) $\vee$
    TSAT(simplify($\mathcal{D}$,$C' \cup \{\neg l\}$), $\langle X, C' \cup \{\neg l\} \rangle$)
end

of variables to shrink in a dynamic fashion that cannot be anticipated before search starts.

3.7.3 TSAT

This algorithm was proposed in 2000 by Armando, Castellini and Giunchiglia [ACG00]. It makes use of the power of available solvers for the propositional satisfiability (SAT) problem, which is used to generate *valuations* for a set of clauses, where each clause corresponds to a (meta-)variable. A valuation sets at least one literal from every clause to true, which corresponds to an instantiation of the variables, and yields a component STP. If this turns out to be consistent, a solution has been found; if it is inconsistent, search backtracks, asking the propositional satisfiability solver to come up with a new valuation. We include the TSAT algorithm as Algorithm 9.

Before the algorithm starts the actual solving of the instance, it performs a preprocessing step that was detailed in Section 3.5.5. When preprocessing is done, the search is started with the DTP instance $\mathcal{D} = \langle X, C \rangle$ and an empty component STP $S = \langle X, \emptyset \rangle$. The instance $\mathcal{D}$ consists of a set of time points X and a set of (meta-)variables or clauses C, with the domain of each variable $c \in C$ modelled as a set of inequalities (or *literals*, in the SAT terminology). The algorithm recursively selects variables to be instantiated until it reaches one of the base cases: either no variables remain or some variable's domain is empty. In the former case, the resulting STP instance is checked for consistency; if it is inconsistent, search backtracks, otherwise a solution has been found and the algorithm terminates. In the latter case, an inconsistency has been deduced, so search also backtracks.

To reach one of these base cases, the algorithm must recursively instanti-

ate variables. Whenever the domain of some variable (clause) holds just one value (literal), it is instantiated (*unit propagation*). Otherwise, a value that has the greatest number of occurrences in the smallest domains is selected to be instantiated; this heuristic was first proposed by Böhm in his winning entry in a 1991/1992 SAT competition organised at the University of Paderborn [BB93]. At first glance, application of this rule to the DTP would appear almost useless because identical inequalities seldom appear multiple times in different domains. However, the preprocessing step introduces clauses of length 2 that contain negations of original inequalities; these will usually appear more often, which explains the suitability of the heuristic.

In Algorithm 9, a variable is instantiated by the *simplify* function. In its simplest form, this function does the following:

- it removes all variables whose domain contains the value to be instantiated, since the constraints they represent are satisfied by the current component STP;
- it removes the negation of the inequality to be instantiated from the domains of all variables, since this is now no longer available to be consistently instantiated.

The forward-checking version of the function does one more thing:

- it checks all remaining values in all domains for consistency with the current component STP (including the newly added inequality), and removes all values that are found to be inconsistent.

This is a much more expensive operation than the other two, but its costs are empirically justified.

The main difference between the original algorithm by Stergiou and Koubarakis and TSAT is that the latter performs what is called *semantic branching*; if a literal is found to be inconsistent with the current component STP, the SAT approach causes its negation to be added instead, which further prunes the search space, whereas the former let this information go to waste. This is one of the foremost reasons for TSAT's improved performance: it is almost two orders of magnitude faster than Stergiou's and Koubarakis' algorithm.

3.7.4 Improvements by Oddi and Cesta

Oddi and Cesta [OC00] improved upon Stergiou's and Koubarakis' algorithm by including semantic branching and disregarding constraints that are

already satisfied in the current component STP, both of which are also used in the TSAT approach.

A new improvement is the use of incremental full path consistency (IFPC) [MH86] instead of incremental directional path consistency (IDPC). Though both require quadratic time in the worst case, maintaining IFPC is slightly more expensive than IDPC in practice; the extra cost is justified because performing a forward check, which is done many more times than adding a new edge and re-enforcing PC/DPC, can be done in constant time on a fully PC graph, while it requires linear time on a DPC graph. The trade-off is not entirely clear-cut, though, and requires further investigation [TP03].

As a fourth and final improvement, Oddi and Cesta found that forward checking need not be performed every time. They show that a value $c_{i \to j}$ for some variable need only be checked if the weight $w_{j \to i}$ of the reverse arc in the APSP matrix maintained by IFPC has changed since the last check. They call this technique *incremental forward checking*; it saves some constraint checks, but does not prune the search space.

3.7.5 Epilitis (Tsamardinos and Pollack)

In 2003, Tsamardinos and Pollack published their study on the DTP [TP03]. Beside an algorithm of their own, called Epilitis, it included a thorough overview of existing techniques for tackling the DTP. All previous solvers only included a subset of these techniques; Epilitis uses all of them, sometimes in a more advanced implementation.

A new technique that Tsamardinos and Pollack bring to bear on the DTP is *no-good recording* or *no-good learning*, well-known in general CSP literature (e.g. [SV94]). Informally, without no-good recording, a search algorithm will make the same mistake over and over again. In principle, a no-good can be recorded whenever the current instantiation of (meta-)variables cannot be extended any further, i.e. whenever search must backtrack. This happens when the domain of some variable C_i consists only of values that would yield a negative cycle in the component STP when chosen for instantiation. The *justification* J of the no-good to be recorded, also called the *culprit set*, then consists of the variables whose current values yield the other edges in these negative cycles; the no-good itself is defined as the pair $\langle \mathcal{A}, J \rangle$, where $\mathcal{A}$ is the subset of the current instantiation ("assignment") limited to variables in J.

Once a no-good is recorded, it is checked whenever all variables in its justification are first instantiated. The amount of no-goods that can potentially be learnt during the solving process is theoretically exponential in the size of the problem instance and could have a prohibitive impact on performance. It

variable	value	arc
C_1	c_{11}	$x_2 - x_1 \leq -20$
C_2	c_{22}	$x_3 - x_2 \leq 10$
C_3	c_{31}	$x_1 - x_3 \leq 15$
C_4	c_{43}	$x_4 - x_2 \leq -5$
C_5	c_{51}	$x_3 - x_4 \leq 15$
C_6	c_{62}	$x_5 - x_3 \leq 30$
C_7	c_{71}	$x_1 - x_2 \leq 15$
C_7	c_{72}	$x_1 - x_3 \leq 5$
C_7	c_{73}	$x_1 - x_4 \leq 10$
C_7	c_{74}	$x_1 - x_5 \leq -10$

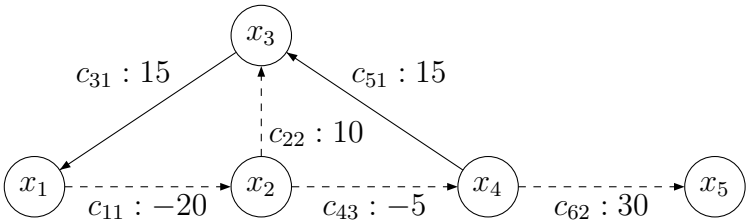


Figure 3.10: No-good recording

is for this reason that Epilitis only records no-goods involving up to a limited number of variables; the authors found a bound of 10 to give good results.

The justification J of a no-good is also used for determining which variable to backjump to; whether or not a no-good is actually recorded, search resumes at the deepest variable in J, since any intermediate variables cannot have played a part in the inconsistency discovered. As search progresses, multiple recorded no-goods can be combined into a single new one; a detailed description of this technique is however outside the scope of this discussion.

See Figure 3.10 for an example of no-good recording. The bottom of the picture depicts a "snapshot" of the component STN at some point during search, corresponding to the instantiation of the variables C_1 through C_6, shown in the top half of the table.

The next variable to be instantiated is C_7; however, all values in its domain, $\{c_{71}, \ldots, c_{74}\}$, shown in the bottom half of the table, would correspond to arcs in the STN yielding a negative cycle. We conclude that the search has reached a dead end, and at this point a no-good can be inferred.

The justification J of the no-good consists of those instantiated variables that participate in at least one of the negative cycles, so in this case, $J = \{C_1, C_2, C_4, C_6\}$. The corresponding arcs are shown dashed in the graph and give $\mathcal{A} = \{c_{11}, c_{22}, c_{43}, c_{62}\}$. The algorithm can now record the no-good

$\langle \mathcal{A}, J \rangle$. Search backjumps to C_6, the deepest variable in J; not much of a jump, really, since it skips no variables. If the domain of C_6 contains no more options after c_{62}, a new dead end has been found and the no-good can be pruned to $\langle \{c_{11}, c_{22}, c_{43}\}, \{C_1, C_2, C_4\} \rangle$, after which a little jump back to C_4 is made for search to continue.

No-goods also play a role in the heuristic that is used to determine which variable to instantiate, and to which value. Like Stergiou's and Koubarakis' algorithm, Epilitis adheres to the fail first principle by employing the *minimum remaining values* heuristic, always instantiating the variable that has the least valid values left in its domain. If there is a tie for selecting this variable, it is broken by selecting the variable C_i with the value c_{ij} in its domain that is inconsistent with the largest amount of other remaining values; the next tie-breaker criterion is the amount of no-goods that c_{ij} appears in. When a variable has been selected, the order in which values are selected to instantiate it to uses the same metric, but reversed, which helps finding a solution (if it exists in the current search branch) as soon as possible.

The use of no-goods by Epilitis is effective; however, it surprisingly still suffers from two obvious weaknesses:

- In the algorithm as presented, the justification recorded for no-goods can only grow during search. In the example given above, after backtracking again for C_6, we asserted that the new no-good has as its justification the set $\{C_1, C_2, C_4\}$; however, this is not according to the specification of Epilitis, which never shrinks justifications and only removes assignments from $\mathcal{A}$. This means that the actual no-good recorded is $\langle \{c_{11}, c_{22}, c_{43}\}, \{C_1, C_2, C_4, C_6\} \rangle$.

  The implication is that checking no-goods is postponed, until all variables in its justification, some of them irrelevant, have been instantiated. Hence, the search space is not pruned as much as it could be.

- When a no-good $\langle \{c_{11}, c_{22}\}, \{C_1, C_2\} \rangle$ has been recorded, this causes the algorithm to avoid simultaneously instantiating C_1 and C_2 to these values again during the remainder of the search; we coin the term "syntactic no-good learning" for this behaviour.

  However, this type of learning does not preclude selecting values for other variables with the same effect on the constraint graph of the component STP. Referring to Figure 3.10 for the interpretation of c_1 and c_2, it would be sensible to forbid *any* instantiation that simultaneously sets $w_{1\to 2}$ and $w_{2\to 3}$ to values smaller than -20 and 10, respectively. This type of learning, which we propose to call "semantic

no-good recording", can easily be accomplished by the addition of new disjunctions, in the case of this example $x_2 - x_1 > -20 \vee x_3 - x_2 > 10$.

3.7.6 TSAT++

This algorithm was first published in 2004 [ACGM05][ACG+05], and to our best knowledge, it still represents the current state of the art, outperforming the previously fastest algorithm, Epilitis. As its name suggests, it improves on the authors' previous algorithm, TSAT, incorporating recent advances in the field of propositional satisfiability and previously untapped techniques from the constraint satisfaction literature. We list the most important improvements here.

Firstly, backjumping and no-good recording are incorporated. Unlike Epilitis, TSAT++ adds the recorded no-goods as new clauses, which is no doubt one of the reasons for its better performance. The method used by TSAT++ to bound the number of no-good constraints learnt dynamically during search also differs from that used by Epilitis. The reader may recall that the latter records and remembers *all* no-goods during search, as long as the amount of variables featuring in them does not exceed some bound. In contrast, TSAT++ enforces no limit on the size of each individual no-good, but instead periodically re-evaluates previously learnt no-goods, giving precedence to those recently discovered and "forgetting" older ones. The reason for this is that no-goods recorded at the beginning of search may no longer be relevant in the current search branch.

One of the advances in the field of propositional satisfiability employed by TSAT++ is the *two-literal watching* data structure, taken from the Chaff solver [MMZ+01]. To keep track of which clauses have only a single available literal remaining, i.e. the ones to which unit resolution can be applied, the traditional way is to associate with each clause a counter that corresponds to the number of its invalidated literals, taking action when this counter reaches the value $n - 1$ for a clause of size n. This scheme requires that the counters of all clauses that some literal participates in be updated every time it is invalidated, and again if it becomes available again during backtracking.

The technique of two-literal watching stems from the insight that the solver is not particularly interested in the first $n-2$ literals becoming invalid. For each clause, two available literals are selected to be watched. It is easy to see that as long as both remain available, unit resolution is not applicable to such a clause. Only when one of them is rendered invalid, the clause is checked to see whether there is more than one literal still remaining. If there is, one of these is selected to replace the literal that just dropped out of the two-literal watching scheme; if there is not, the single remaining watched

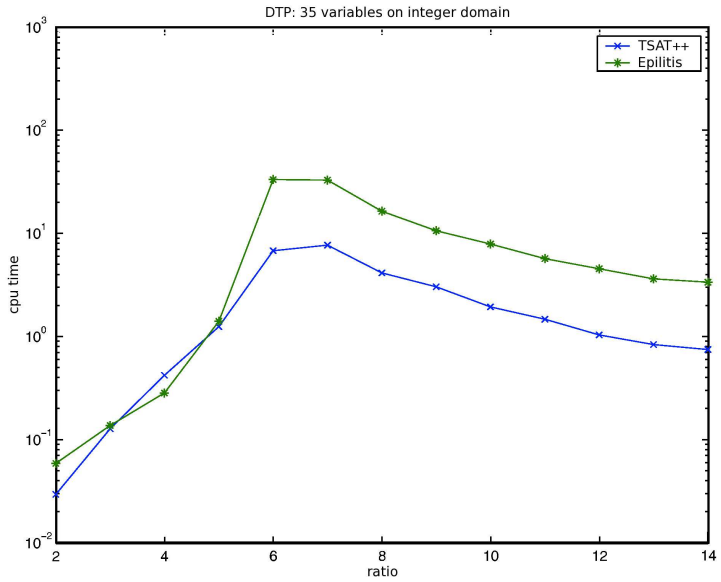


Figure 3.11: Performance of Epilitis vs TSAT++

literal is selected for unit resolution.

To compare the performance of TSAT++ to that of Epilitis, we include Figure 3.11, which was reproduced from [ACGM05]. On the horizontal axis is the ratio of constraints to time points. It can immediately be seen that TSAT ++ outperforms Epilitis by almost an order of magnitude for the majority of cases. Also interesting to note is the fact that the hardest DTP instances seem to be those which have between 6 and 7 times as many constraints as time points. This behaviour is common for DTP solvers; in fact, it was already described by Stergiou and Koubarakis [SK00], who noted that this region appears to be related to the transition from mostly consistent (underconstrained) to mostly inconsistent (overconstrained) problem instances, which lies between ratios of 4 and 7.

TSAT++ determines consistency of component STNs by running Bellman's and Ford's algorithm (BF) on them, taking the negative cycle with the least nodes as culprit set to guide backjumping and to record as no-good. The authors give the solving of component STNs relatively little attention in their discussion; they apparently run BF anew each time they check consistency, overlooking the fact that component STNs usually exhibit substantial overlap between instantiations. It is our opinion that the performance of TSAT++ could benefit greatly from some incremental consistency checker such as IFPC, as employed by Epilitis.

3.8 Summary

We have seen that the TCSP and DTP are formalisms that allow for the representation of a wide scope of problems, at the cost that they are hard to solve.

All published algorithms tackle the problems at the meta-level. At this level, original constraints are considered as variables, an instantiation of which corresponds to a component STP; if such an STP is consistent, a solution has been found. Approaching the problem in this manner means that availability of efficient algorithms to solve the STP is very important. These algorithms were described in the previous chapter.

Before the actual solving of the problem is begun, some preprocessing can be done. Preprocessing methods use some simple rules which can be used to prune the search space, thereby making the search more efficient.

The algorithms for solving the TCSP and the DTP that we discussed can be split into two categories.

The first category takes a standard CSP approach, instantiating variables one by one until either a solution has been found or a dead end is encountered. All discussed TCSP solvers fall into this category.

The second category consists of the TSAT algorithms, which abstract away the inequalities to recast the temporal problem as a SAT instance, which is then fed to a SAT solver. This solver returns an instantiation of all variables, satisfying the problem at the abstract level; this instantiation is then checked for temporal satisfiability.

Algorithms from both categories can benefit from such techniques as forward checking, backjumping and no-good recording. The best algorithm currently known is TSAT++ , of the second category.

Chapter 4

Conclusions

In this chapter, we will first give a short summary of the theory discussed in Chapters 2 and 3. We conclude by identifying interesting and promising topics for future research.

4.1 Summary

We have seen that the Simple Temporal Problem (STP) is a concise and clear method for representing temporal information, and that it can be used to efficiently answer several types of queries about that temporal information. Since the transformation of an STP instance into a directed weighted graph is an identity function, it should come as no surprise that many of the basic methods for solving the problem, such as Floyd's and Warshall's all-pairs-shortest-paths algorithm and Bellman's and Ford's single-source-shortest-paths algorithm, are taken from graph theory. However, it is interesting to note that for the STP, Floyd's and Warshall's algorithm is identical to enforcing the well-known property of path consistency (PC) from constraint satisfaction theory. The most efficient algorithm known to date for solving the STP is based on the property of partial path consistency, which is taken from the constraint satisfaction literature, but also shows close kinship to graph theory.

The extensions of the STP we discussed are the Temporal Constraint Satisfaction Problem (TCSP) and the Disjunctive Temporal Problem (DTP). Since the TCSP can also be easily represented as a graph, it shares the STP's properties of conciseness and clarity; it is however far more expressive, and therefore far harder to solve. Enforcing PC, a method that can be used to solve the STP, is also possible on TCSP instances, and may be used to attempt to prune the search space as a preprocessing step; however, it can lead

to an explosion of intervals labelling each edge, thereby inadvertently widening the search space instead. Several approximations of PC are available that are safe to use as a preprocessing step; they eliminate those intervals for which it can be concluded without search that they appear in no solution, while guaranteeing that the amount of intervals labelling an edge will never increase.

The DTP allows for even more expressiveness, at the cost of losing some clarity: since its constraints are generally of higher than binary order, it cannot be represented as a graph. This means that the preprocessing methods based on enforcing PC are not applicable to DTP instances; however, some preprocessing methods are still available, making use of simple logical rules.

All algorithms we described use a backtracking search, and work on the meta-level: they do not assign values to time points directly, but instead select intervals or inequalities and determine whether the resulting component STP is consistent. This means that the actual solving algorithms for the TCSP and the DTP are very closely related, since the same methods apply.

There are two main categories of algorithms: the CSP-based technique, for which the state-of-the-art is represented by Epilitis, and the SAT-based technique, whose state-of-the-art is TSAT++ ; currently, the latter has the edge. In both categories, techniques such as forward checking, backjumping and no-good recording are employed.

4.2 New results and future work

Throughout the text, we have presented some new results and pointed out several areas where future research could yield improvements on results from currently published research; we will sum these up in this section. We will also propose some promising new directions for research for which the literature covered by our survey may serve as a starting point. Concerning the STP, these are the following:

- We have shown on page 9 (with proofs in Appendix A) that the complexity of solving the STP is bounded below by NL and bounded above by NC^2; for polynomially bounded weights, it is NL-complete. As a consequence of membership in NC^2, we know that solving the STP can be done efficiently by a parallel algorithm; to our best knowledge, this fact has not yet been utilised at present. The implication of the NL-completeness result may be that a randomised algorithm would perform well; however, this needs further investigation as well.

- On page 15, we claimed that determining consistency of an STP instance can be done by a merger between directed path consistency (DPC) and partial path consistency (PPC); this claim is proved in Appendix C. Such a merger will always perform at least as well as PPC itself, an implementation of which is the current state-of-the-art algorithm for determining consistency of the STP. We will report on an implementation of such a new algorithm in an upcoming publication.

For the DTP (and the TCSP), further research can be done with regard to consistency checking of component STPs:

- Current algorithms for the DTP from the CSP category employ incremental path consistency algorithms such as IFPC and IDPC during their backtracking search and for forward checking; see, for example, Section 3.7.4. In an upcoming publication, we will propose a new algorithm based on an incremental version of the PPC algorithm. We strongly suspect that this algorithm will outperform one or quite possibly both existing incremental methods.

- As we stated on page 42, TSAT++ uses Bellman's and Ford's algorithm for determining consistency of component STPs; an interesting direction for future research would be to use an incremental consistency enforcing technique instead, which we expect to yield improved performance.

Finally, the two best-performing algorithms, Epilitis and TSAT++ , can each probably be made more efficient:

- In our opinion, the current state-of-the-art solver from the CSP category, Epilitis, suffers from two inefficiencies in its implementation of no-good recording, as noted on page 40:

  - It never shrinks culprit sets (justifications), thereby postponing the checking of recorded no-goods;
  - It does syntactic no-good recording, checking only those meta-variables that participated in the dead end encountered, whereas semantic no-good recording would prune the search space to a much greater extent.

  An interesting direction for future research would be whether Epilitis would outperform its nemesis TSAT++ if these inefficiencies are remedied.

- The TSAT family of algorithms performs a preprocessing step, described in Section 3.5.5 that checks all pairs of inequalities defined on the same two variables for consistency. It is our belief that for checking sets of size 3 or even 4, the resulting pruning of the search space may outweigh the penalty incurred by having to search the added clauses.

- TSAT++ was published in 2004; since then, however, the field of propositional satisfiability has not stood still. In the near future, we plan to study recent advances in SAT solving techniques and determine their applicability to the DTP.

The literature we addressed in this survey has given a perspective on the field of temporal reasoning that is far from complete, even if we limit our view to the STP/TCSP/DTP family of problems. For example, extensions exist that allow for the modelling of preferences with temporal constraints. Recall the application areas we listed in Chapter 1. In all of these, multiple parties participate, each having its own interests: airlines and ground handling companies, taxi companies or even individual taxi drivers, et cetera. Even the interests of John and Fred from our example in Section 2.2 may at times conflict.

When allowing for preferences to be modelled, one can differentiate between hard constraints and soft constraints. The former represent limits that may not be ignored; for example, one can not be in two places at the same time, nor can vehicles hold unlimited capacity. The latter represent actors' preferred way of doing things: both John and Fred would like to take their car to work and every airline wants ground handling operations for its aircraft to proceed as smoothly and efficiently as possible, without having to wait. By extending temporal reasoning problems with preferences, the goal in solving them is no longer to find any solution that satisfies all constraints specified, but instead to find a solution that must satisfy every hard constraint and cater as best as possible to the soft constraints.

It is therefore unavoidable that negotiation and fairness become issues when considering problems of this type, and one enters into the interesting fields of algorithmic game theory and mechanism design. In addition to the specific topics we already listed, we expect the general direction of our future research to be in these fields.

Appendix A

Complexity of the STP

In this appendix, we determine the formal complexity class of solving the STP. There are two possible definitions for "solving" the STP: deciding consistency or calculating the minimal network; the latter implies the former. We show that for polynomially bounded weights, the problem is NL-complete; for unrestricted weights, the problem is in NC^2. This shows that the problem is efficiently parallelisable. Surprisingly, the results hold for either definition of "solving".

A note on non-integer weights: a network with rational weights can be converted to one with integer weights by multiplying all weights by their least common multiple. The usefulness for real-valued weights (which we are not allowed to approximate with a finite decimal or binary representation) is doubtful and is outside the scope of this discussion. For these reasons, the remainder of this appendix assumes integer weights.

A.1 NL-hardness

In this section, we will show NL-hardness of STP-inconsistency. Since NL = coNL, we can then conclude that STP-consistency is also NL-hard. The same result holds for calculating the minimal network, since it implies determining consistency.

A known NL-complete problem is ST-CONNECTIVITY. This problem can be stated as follows:

> Given a directed graph $G = (V, A)$ and two vertices $s, t \in V$, does G contain a path from s to t?

This can now be reduced to STP-inconsistency in logarithmic space, as follows. We transform every vertex into a time-point variable, and we associate

every arc $(v_i, v_j) \in A$ with a constraint $c_{i \to j}$, with weight $w_{i \to j} = 0$. This leads to a trivially consistent STP, which will at least have the solution $\tau = \langle x_1 = 0, \ldots, x_n = 0 \rangle$.

Now we add the constraint $c_{t \to s}$, with weight $w_{t \to s} = -1$; if a constraint between t and s already existed, we replace the previously existing one.

If there was a path from s to t in the original graph, its total weight is trivially equal to 0. The constraint between t and s then yields a cycle with negative total weight and thus leads to inconsistency.

If the STN that results from the transformation is inconsistent, it must have a negative cycle. This cycle must incorporate the arc between t and s, because this is the only arc with negative weight. But then there must also be a path from s to t.

We conclude that STP-inconsistency is NL-hard, and hence that solving the STP is NL-hard.

A.2 NL-completeness for bounded weights

In this section, we will first show that (in)consistency for STPs with bounded weights can be decided in logarithmic space by a nondeterministic algorithm. We will again demonstrate this only for the case of determining inconsistency; the result for determining consistency then follows. We then show the same for calculating the minimal network.

A.2.1 Determining consistency

A nondeterministic algorithm can find any cycle by guessing a starting vertex, repeatedly guessing edges and verifying that the last vertex is equal to the first one. Thus, for determining inconsistency, it can guess which cycle has negative total weight and then walk that cycle, keeping a running total weight, and verify negativity when the original node is reached again.

The algorithm has limited space for determining whether the total weight of the cycle is actually negative: $\mathcal{O}(\log |I|)$, where $|I|$ is the size of the problem instance. The size of an STP instance $\mathcal{S} = \langle X, C \rangle$ depends on encoding; if we let w_{max} be the maximum absolute constraint weight, it may be $\mathcal{O}(m \log n \log w_{max})$ when the instance is encoded as a list of constraints with weights, or $\mathcal{O}(n^2 \log w_{max})$ for a matrix representation. In either case, the algorithm must have space complexity $\mathcal{O}(\log n + \log \log w_{max})$.

Calculating the total weight, which results from the summation of at most n terms of order w_{max}, uses up $\mathcal{O}(\log n + \log w_{max})$ bits of memory

space. To fit this in logarithmic space, we require $\log w_{max} = \mathcal{O}(\log n)^*$, which is attained for $w_{max} = \mathcal{O}(n^k)$ with constant k, i.e. if the maximum weight is polynomially bounded by the number of variables (and constraints). Additionally, walking the cycle requires two pointers: one to the current vertex and one to the original vertex. As each of these is logarithmic in size, we have shown that the space requirements are met.

Hence, if the value of weights is limited in this fashion, STP-(in)consistency is a member of NL and, combined with the results of the previous section, NL-complete.

A.2.2 Calculating the minimal network

This problem is not a decision problem, as above, but a function problem. In this subsection we show that this problem is FNL-complete. A function is FNL-computable if it can be calculated by a non-deterministic Turing machine with a read-only input tape, a logarithmically bounded working tape and a write-only output tape. The method we describe here makes use of the fact that deciding consistency is an NL problem.

Our very first step is to verify that the original network is consistent. If it isn't, we reject; the concept of a minimal network makes no sense for an inconsistent network.

The algorithm then iterates over all pairs (i, j) with $1 \leq i, j \leq m$. For each original constraint $c_{i\to j}$, it nondeterministically selects a new weight $w'_{i\to j} \in [-(n-1)w_{\max}\ ,\ w_{i\to j}]$, thus making sure that every new constraint $c'_{i\to j}$ is tighter than the original constraint, which is a requirement for the minimal network. The lower bound follows from a path of maximal length with all maximally negative weights; the upper bound applies if the constraint was already minimal.

If there was no original constraint $c_{i\to j}$ (put differently, if the original constraint was the universal constraint), the algorithm adds it, and selects a weight $w'_{i\to j} \in [-(n-1)w_{\max}\ ,\ (n-1)w_{\max}]$ for it.†

The space required for $w'_{i\to j}$ is then $\mathcal{O}(\log n)$ for polynomially bounded weights.

Next, the algorithm checks consistency for the original network with $c_{i\to j}$ replaced by $\bar{c}'_{i\to j} : x_j - x_i > w'_{i\to j}$, its inverse. If this network is consistent,

*The alternative, $\log w_{max} = \mathcal{O}(\log \log w_{max})$, is clearly useless.

†A minor inconvenience occurs if there is no path from x_i to x_j in the STN: in this case, the weight must remain infinite (for a universal constraint).

However, because ST-connectivity is in NL, the algorithm can check this as a subroutine; if it finds that the nodes are unconnected, it leaves the constraint universal and moves on to process the next constraint.

this implies that $c'_{i\to j}$ has been made too tight and removed some solutions from the network, so we reject.

Finally, we check consistency for the original network with $c_{i\to j}$ replaced by $\hat{c}'_{i\to j} : x_j - x_i = w'_{i\to j}$. If this network is inconsistent, $c'_{i\to j}$ is not tight enough and we reject; otherwise, we output $c'_{i\to j}$ and continue with the next constraint.

In conclusion, we have shown that the constraints we output

1. are tighter than the original constraints;
2. allow all solutions that the original network allowed;
3. cannot be tightened further without losing solutions to the problem.

In addition to the space required for determining consistency, this only requires a pointer to the constraint currently under scrutiny and its new weight $w'_{i\to j}$, for which logarithmic space suffices. Thus, calculating the minimal network is in FNL, and combined with the result from the previous section, FNL-complete.

A.3 Membership of NC^2

The discussion in the previous section shows that it is highly unlikely that solution of the STP with unbounded weights is a member of NL, since it requires the summation of those weights. We now turn our attention to the next complexity class higher up the ladder, which is NC^2.

The complexity class NC contains those decision problems that can be solved on a parallel random access machine (PRAM) by $\mathcal{O}(n^k)$ parallel processors in $\mathcal{O}(\log^c n)$ time, where c and k are constants; NC^c contains the decision problems that can be solved in $\mathcal{O}(\log^c n)$ time. It holds that $\mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{NC}^2$.

In this section we show that there exists a parallel algorithm that calculates the minimal network on a PRAM in $\mathcal{O}(\log^2 n)$ time, and that (in)consistency is thus also a member of NC^2.

We assume a PRAM with $n^3 \cdot \log(nw_{max})$ processors. The algorithm operates on the complete graph; if $w_{i\to j}$ is undefined in the problem instance, an infinite weight is assumed. The idea of the algorithm is then as follows. For every triple (i, j, k) we determine the length of all paths $x_i \to x_k \to x_j$ by calculating $w_{i\to k} + w_{k\to j}$. We set $w_{i\to j}$ to the minimum of these values, and then repeat the procedure. Every time this procedure iterates, the length of the paths taken into account doubles; in particular, after at most ${}^2\log n$ iterations, all paths have been considered. If the resulting network has a cycle

$x_i \to x_j \to x_i$ with negative weight, we conclude inconsistency; otherwise, we have calculated the minimal network.

We have $\log(nw_{max})$ processors devoted to calculating $w_{i\to k} + w_{k\to j}$ for every triple (i, j, k), which is enough to do it in parallel, in logarithmic time $\mathcal{O}(\log\log(nw_{max})) = \mathcal{O}(\log|I|)$. Determining the minimum of these values can also be done in logarithmic time by this number of processors. Hence, every iteration requires $\mathcal{O}(\log|I|)$ time to complete, which yields total time complexity of $\mathcal{O}(\log^2|I|)$.

Appendix B

Transforming DTP to TCSP

Deciding consistency of an instance of the Disjunctive Temporal Problem at first glance seems to be more complex than deciding consistency of an instance of the Temporal Constraint Satisfaction Problem. In this appendix, we show that there exists a linear translation from the former to the latter, thus demonstrating that both problems are very comparable in complexity.

Before we formally present this result, we take the following steps. First, in Section B.1, we add constraints to bound the DTP time points, making sure that this does not affect its consistency; then, having established these bounds, we show in Section B.2 how to "wrap" the DTP time points and constraints inside a TCSP instance. Finally, in Section B.3, we prove the soundness of this reduction and show that it can be performed in linear time.

B.1 Bounding the time points

Let $\mathcal{D} = \langle X, C \rangle$ be a DTP instance, with X a set of time points and C a set of disjunctions c_i of the form $c_{i1} \vee \ldots \vee c_{in}$, where each c_{ij} represents an inequality $x_l - x_k \leq w$.

We define a new time point $z \notin X$ to use as temporal reference point. Using z, we can bound the values of all time points in X by introducing a set of new constraints:

$$C_1 \quad = \quad \{x_i - z \in [0, B] \quad | \quad x_i \in X\}$$

We want to find a value for B that does not change the consistency of the problem; i.e. if the original instance $\mathcal{D} = \langle X, C \rangle$ has a solution, there must also be a solution to $\mathcal{D}' = \langle X \cup \{z\}, C \cup C_1 \rangle$. We have seen in Section 3.3.3 that on the meta-level, a solution to a DTP instance corresponds to

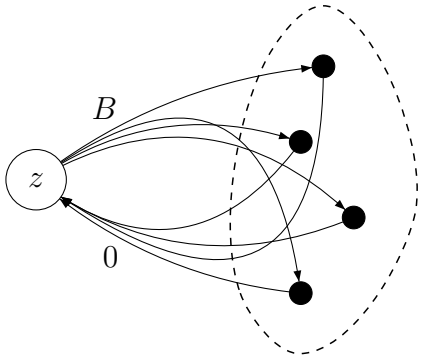


Figure B.1: Bounding the DTP time points

a consistent component STP. The constraints in C_1 are non-disjunctive; therefore, they feature in any component STP for $\mathcal{D}'$.

In Figure B.1, we depict a component STN of an original DTP instance $\mathcal{D}$ enclosed in a dashed line on the right; on the left is the new temporal reference point z, connected to all time points in X by the constraints in C_1; the entire network is then a component STN for $\mathcal{D}'$. We will now show the following:

Theorem B.1 *For $B \geq |(n-1)w_{\min}|$, where $w_{\min}$ denotes the minimum weight among all inequalities in C, no component STN for $\mathcal{D}'$ contains a simple negative cycle that includes z.*

Proof If $w_{\min}$ is positive, there are no negative edge weights in the component STN, so the result follows trivially.

Otherwise, suppose that $\mathcal{D}'$ contains a simple negative cycle $(z, x_{i_1}, \ldots, x_{i_k}, z)$ for some $k > 1$. Then, the following inequality must hold:

$$B + \sum_{j=1}^{k-1} w_{i_j \to i_{j+1}} < 0$$

which implies

$$\sum_{j=1}^{k-1} w_{i_j \to i_{j+1}} < -B \leq (n-1)w_{\min}$$

since $w_{\min}$ is negative.

The summation contains at most $n-1$ terms, which occurs if all time points in X participate once in the cycle, i.e. $k = n$. Since each of these terms is no smaller than $w_{\min}$, we conclude that the inequality cannot hold;

therefore, our assumption was false and $\mathcal{D}'$ contains no simple negative cycle involving z. □

From this, we can conclude the following:

Corollary B.2 *For $B \geq |(n-1)w_{\min}|$ as above, $\mathcal{D}' = \langle X \cup \{z\}, C \cup C_1 \rangle$ is consistent if and only if $\mathcal{D} = \langle X, C \rangle$ is.*

Proof If $\mathcal{D}$ is consistent, there exists a consistent component STN $\mathcal{S}$ for it, and it follows directly from Theorem B.1 that addition of z and the constraints C_1 preserve its consistency. Otherwise, each possible component STN for $\mathcal{D}$ contains a negative cycle; this cycle will still be present in the corresponding component STN for $\mathcal{D}'$, so inconsistency is also preserved. □

B.2 Wrapping the constraints

In this section, we will show how to transform a disjunction $c_i \in C$ from a DTP instance into a set of binary TCSP constraints. We will make use of the temporal reference point z and the bound B introduced in the previous section; specifically, we use them to define the concepts of "offside" and "onside" time points:*

Definition B.1 *A time point $t \neq z$ is called* onside *in some instantiation τ if $t - z \in [0, B]$; otherwise, it is called* offside.

Informally, only onside time points are considered to be "in play" from the perspective of the original time points X, whose playground we have already seen to be limited to an interval of $[0, B]$ around z; it follows that all time points $x \in X$ are trivially onside. In this section, we will define some extra time points to represent the individual inequalities from the DTP constraints; placing one of these time points onside will have the effect of enforcing the inequality it represents.

We consider the following disjunction $c_i \in C$:

$$c_i : \; x_{m_1^i} - x_{l_1^i} \leq w_1^i \vee \cdots \vee x_{m_k^i} - x_{l_k^i} \leq w_k^i$$

To represent this disjunction in the TCSP instance, we introduce a set of new time points $P^i = \{p_1^i, \ldots, p_k^i\}$, one for each inequality; in the remainder of this section, we will often omit the superscript i since we will consider c_i only. We will also introduce new constraints to enforce the following properties for any instantiation τ:

*The concepts *onside* and *offside* were first defined in the Laws of Football [FIF07]; according to these Laws, only football players in an onside position are allowed to score.

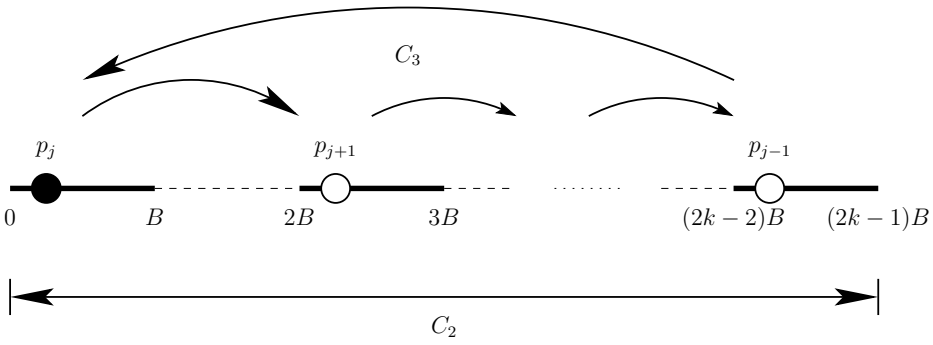


Figure B.2: The effect of constraint sets C_2 and C_3

1. all but one of the time points in P are offside; and
2. only an onside time point may influence the time points in X.

B.2.1 The offside trap

To ensure that a single time point p_j is onside, we add the following sets of binary constraints:

$$\begin{aligned} C_2^i &= \{p_j^i - z \in [0, (2k-1)B] \quad | \quad 1 \leq j \leq k\} \\ \widehat{C_3^i} &= \{p_{j_1}^i - p_{j_2}^i \in (-\infty, -2B] \cup [2B, \infty), \quad | \quad 1 \leq j_1 < j_2 \leq k\} \end{aligned}$$

The constraints in C_2 limit the range of values that the time points p_j can assume; those in $\widehat{C_3}$ ensure that none of the time points in P may take place less than $2B$ time units from eachother. Together, they achieve the goal of having exactly one time point p_j onside.

However, note that $|\widehat{C_3}| = k(k-1)/2$, which can be quadratic in the size of the original DTP. Since our goal is to produce a linear transformation, this will not do. Fortunately, we can redefine $\widehat{C_3}$ to have linear size while achieving the same effect:

$$C_3^i = \{p_{j+1}^i - p_j^i \in \{-(2k-2)B, 2B\} \quad | \quad 1 \leq j \leq k\}$$

Note that we let the subscripts wrap around in the definition of C_3 such that p_{k+1} signifies p_1. These constraints enforce the time points p_j to be ordered consecutively. If a time point p_j is offside, it takes place exactly $2B$ time units later than its precursor p_{j-1}; if it is onside, it takes place exactly $(2k-2)B$ time units *before* p_{j-1}. We pictorially represent the effect of these constraints in Figure B.2. In the situation depicted, time point p_j, shown as

a black circle, is onside, and the others, shown as white circles, are offside. The time points are positioned on a number line running from 0 to $(2k-1)B$ which denotes the difference $p_j - z$. The thick sections of the number line demarcate the values that each time point in P can assume, and the arrows represent the constraints in C_2 and C_3. The figure illustrates the following result:

Lemma B.3 *The set of all instantiations $\langle \tau_P \rangle$ of the time points in $P \cup \{z\}$ that satisfy the constraints in $C_2 \cup C_3$ has the following properties:*

1. *for each time point $p_j \in P$, there exists a set of instantiations $\langle \tau_P \rangle_j \subset \langle \tau_P \rangle$ in which p_j is the unique onside time point; and*
2. *$\langle \tau_P \rangle_j$ contains an instantiation for each value in the interval $[0, B]$ that $p_j - z$ can assume; and finally*
3. *it holds that $\langle \tau_P \rangle = \bigcup_{j=1}^{k} \langle \tau_P \rangle_j$.*

B.2.2 From inequality to binary constraint

We can now show how to translate the inequalities in c_i into binary constraints. Let $c_{ij} : x_{m_j} - x_{l_j} \leq w_j$ be an inequality featuring in the DTP constraint c_i. If p_j is onside in some instantiation τ, we want to enforce $x_{m_j} \leq p_j \leq x_{l_j} + w_j$; this is equivalent to enforcing c_{ij}, since Lemma B.3 states that $p_j - z$ is free to assume any value in the interval $[0, B]$. In contrast, if p_j is offside, it must not constrain x_{l_j} and x_{m_j} in any way. This is achieved by adding the following sets of constraints:

$$\begin{array}{rcll} C_4^i & = & \{x_{m_j^i} - p_j^i \in (-\infty, 0] \quad \mid 1 \leq j \leq k\} & (\text{i.e. } x_{m_j^i} \leq p_j^i) \\ C_5^i & = & \{p_j^i - x_{l_j^i} \in (-\infty, w_j^i] \cup (B, \infty) \quad \mid 1 \leq j \leq k\} & (\text{i.e. } p_j^i \leq x_{l_j^i} + w_j^i) \end{array}$$

This leads us to the following lemma:

Lemma B.4 *For each j, the set of instantiations $\langle \tau_j \rangle$ of the time points in $\{p_j, x_{l_j}, x_{m_j}, z\}$ satisfying the constraints in $C_1 \cup C_4 \cup C_5$ has the following properties:*

1. *if p_j is onside in some instantiation τ_j, then the values of x_{l_j} and x_{m_j} in that instantiation satisfy c_{ij}; and*
2. *the subset of $\langle \tau_j \rangle$ in which p_j is offside contains an instantiation for each value in the set $[0, B] \times [0, B]$ that the pair $(x_{l_j} - z, x_{m_j} - z)$ can assume.*

B.3 Integrating the results

Building on the results established in the previous sections, we can now state our main result.

Corollary B.5 *Let $\mathcal{D} = \langle X, C \rangle$ be a DTP instance. Define the TCSP instance $\mathcal{T} = \langle X', C' \rangle$ as follows, referring to the sets defined in the previous sections:*

- $X' = X \cup P \cup \{z\}$
- $P = \bigcup_i P^i$
- $C' = C_1 \cup C_2 \cup C_3 \cup C_4 \cup C_5$
- *for* $2 \leq j \leq 5$, $C_j = \bigcup_i C_j^i$

Then, $\mathcal{T}$ is consistent if and only if $\mathcal{D}$ is consistent.

Proof ($\Rightarrow$) If $\mathcal{D}$ is consistent, there exists a meta-instantiation of each meta-variable $c_i \in C$ to a meta-value c_{ij} yielding a consistent component STP $\mathcal{S}$. It follows from Corollary B.2 that the addition to $\mathcal{S}$ of time point z and the constraints C_1 preserve its consistency; therefore, there must exist an instantiation τ_X of the variables in $X \cup \{z\}$ such that $C \cup C_1$ are satisfied.

The meta-instantiation of each $c_i \in C$ to a meta-value c_{ij} corresponds to a selection of time points p_j^i to place onside; Lemma B.3 states that there exists an object-level instantiation τ_P of the variables in $P \cup \{z\}$ that yields this selection while satisfying the constraints in $C_2 \cup C_3$.

Finally, using z as "glue", we can combine the instantiations τ_X and τ_P into an instantiation τ of all variables in X'. For τ to be a solution to $\mathcal{T}$, it must additionally satisfy the constraints in $C_4 \cup C_5$. It follows from Lemma B.4 and our selection of onside time points p_j^i that this is the case.

($\Leftarrow$) Conversely, if $\mathcal{T}$ is consistent, there must exist a solution τ for it. Lemma B.3 states that the set of onside time points in τ has a one-to-one relationship with an instantiation to the meta-variables in C; Lemma B.4 ensures that this instantiation to C is a solution to $\mathcal{D}$. □

The reduction we have given is also very efficient.

Theorem B.6 *The transformation from $\mathcal{D}$ to $\mathcal{T}$ can be performed in linear time; it follows that the sizes of the instances are asymptotically equal.*

Proof The size of each of the sets of new variables and constraints is clearly linear in the size of the original problem instance. After the bound B has been established, which can easily be done in linear time, the construction of the new sets requires constant time for each element added to them. □

Appendix C

Partial directed path consistency

We saw in Section 2.5.5 that enforcing partial path consistency (PPC) on a STP is a very efficient way to solve it when compared to the traditional method of enforcing full path consistency (PC), i.e. calculating all-pairs-shortest-paths. The reason for PPC's improved efficiency is that it triangulates the constraint graph of the STP, which generally leads to far less edges to process than the $n(n-1)/2$ edges in the complete graph that PC has to deal with. The most efficient algorithm currently known for enforcing the PPC property is $\triangle$STP; this algorithm considers the constraint graph to be composed of triangles.

In Section 2.5.4, we saw that to determine whether an STP instance is consistent, it is not necessary to enforce full path consistency; instead it suffices to make the constraint network directionally path consistent (DPC). We also saw that the DPC algorithm is never more costly than the PC algorithm; however, its performance depends on the order in which the variables are processed.

The authors of $\triangle$STP state [XC03c] that their algorithm is in many cases faster than enforcing DPC. In this appendix, we will show that after a constraint graph has been triangulated, DPC will always outperform $\triangle$STP when processing the variables in a *simplicial elimination ordering* (sometimes also referred to as a "perfect elimination ordering").

Definition C.1 *In a graph $G = (V, E)$, a vertex $v \in V$ is* simplicial *if the set of its neighbours $N(v) = \{w \mid \{v, w\} \in E\}$ induces a clique, i.e. if $\forall\{s, t\} \subseteq N(v) : \{s, t\} \in E$.*

Let G_i be the subgraph of G induced by $V_i = \{v_1, \ldots, v_i\}$; note that $G_n = G$ for $n = |V|$. A simplicial elimination ordering *of G is an order $(v_n, \ldots, v_1)$ in*

which vertices can be deleted from G, such that every vertex v_i is a simplicial vertex of the graph G_i.

The following result was proven in [FG65]:

Theorem C.1 *A graph is chordal if and only if it has a simplicial elimination ordering.*

Enforcing PPC requires iterating over the triangles in the constraint graph at least once. We will now show that enforcing DPC along a perfect elimination ordering considers each triangle exactly once.

Theorem C.2 *Let $G = (V, E)$ be a chordal constraint graph. Enforcing DPC on G along a perfect elimination ordering $(v_n, \ldots, v_1)$ introduces no new constraint edges and considers each triangle in G exactly once.*

Proof Recall that Algorithm 3 iterates over the vertices v_k along the ordering chosen.

Let $e = \{v_i, v_j\} \notin E$, where $i, j < k$; as noted in Section 2.5.4, the algorithm will only add e to the constraint graph if both $\{v_i, v_k\} \in E$ and $\{v_j, v_k\} \in E$. Since v_k is a simplicial vertex of G_k, it follows that this is never the case.

For $i < j < k$, let $\{v_i, v_j, v_k\}$ be a triangle in G. Clearly, this triangle is only considered once, when v_k is being processed. □

We have now shown that when run on a chordal constraint graph along a simplicial elimination ordering, DPC will outperform $\triangle$`STP`. If a constraint graph is not chordal, it can be made so by running a triangulation algorithm, like $\triangle$`STP` does. For chordal graphs, a simplicial elimination ordering can be determined in linear time by running the maximum cardinality search algorithm [TY84].

We conclude that we have sketched a new algorithm that enforces a property we will call partial directed path consistency (PDPC); it relates to DPC like PPC relates to (full) PC. It is clear that this algorithm will always outperform the current state-of-the-art algorithm $\triangle$`STP` when determining consistency of an STP instance.

Bibliography

[ACG00] Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. SAT-based procedures for temporal reasoning. In S. Biundo and M. Fox, editors, *Proceedings of the 5th European Conference on Planning (Durham, UK)*, volume 1809 of *Lecture Notes in Computer Science*, pages 97–108. Springer, 2000.

[ACG+05] Alessandro Armando, Claudio Castellini, Enrico Giunchiglia, Massimo Idini, and Marco Maratea. TSAT++: an open platform for Satisfiability Modulo Theories. *Electronic Notes in Theoretical Computer Science*, 125(3):25–36, July 2005.

[ACGM05] Alessandro Armando, Claudio Castellini, Enrico Giunchiglia, and Marco Maratea. A SAT-based decision procedure for the boolean combination of difference constraints. In Holger H. Hoos and David G. Mitchell, editors, *Theory and Applications of Satisfiability Testing*, volume 3542 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2005.

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[BB93] M. Buro and H. Kleine Büning. Report on a SAT competition. *Bulletin of the European Association for Theoretical Computer Science*, 49:143–151, 1993.

[Bel58] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[BSH99] Christian Bliek and Djamila Sam-Haroud. Path consistency on triangulated constraint graphs. In *IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 456–461, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[Chl95] Nicolas Chleq. Efficient algorithms for networks of quantitative temporal constraints. In *Proceedings of CONSTRAINTS-95, First International Workshop on Constraint Based Reasoning*, pages 40–45, April 1995.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DMP91] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.

[FD95] Daniel Frost and Rina Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'95*, pages 572–578, Montreal, Canada, 1995.

[FF62] Lester R. Ford, Jr. and Delbert R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[FG65] Delbert R. Fulkerson and O. A. Gross. Incidence matrices and interval graphs. *Pacific Journal of Mathematics*, 15:835–855, 1965.

[FIF07] FIFA. Laws of the Game 2007/2008. Published by Fédération Internationale de Football Association, Zürich, Switzerland, July 2007.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[HE80] M. Harlick and G. L. Elliot. Increasing tree-search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, October 1980.

[Joh77] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

[MH86] Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225–233, 1986.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC '01: Proceedings of the 38th conference on Design Automation*, pages 530–535, New York, NY, USA, 2001. ACM Press.

[Mof06] Michael D. Moffitt. Efficient and expressive extensions of constraint-based temporal reasoning. In *Proceedings of the ICAPS-2006 Doctoral Consortium*. AAAI, June 2006.

[OC00] Angelo Oddi and Amedeo Cesta. Incremental forward checking for the Disjunctive Temporal Problem. In *ECAI 2000: Proceedings of the 14th European Conference on Artificial Intelligence*, pages 108–112, 2000.

[Pro93] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[SD93] Eddie Schwalb and Rina Dechter. Coping with disjunctions in Temporal Constraint Satisfaction Problems. In *AAAI 1993: Proceedings of the 11th National Conference on Artificial Intelligence*, pages 127–132, 1993.

[SD97] Eddie Schwalb and Rina Dechter. Processing disjunctions in temporal constraint networks. *Artificial Intelligence*, 93(1-2):29–61, 1997.

[SK00] Kostas Stergiou and Manolis Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. *Artificial Intelligence*, 120(1):81–117, 2000.

[SV94] Thomas Schiex and Gérard Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal on Artificial Intelligence Tools (IJAIT)*, 3(2):187–207, 1994.

[TP03] Ioannis Tsamardinos and Martha E. Pollack. Efficient solution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence*, 151(1–2):43–89, 2003.

[TY84] Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, August 1984.

[VK86] Marc B. Vilain and Henry A. Kautz. Constraint propagation algorithms for temporal reasoning. In *National Conference of Artificial Intelligence (AAAI'86)*, pages 377–382, 1986.

[War62] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[XC03a] Lin Xu and Berthe Y. Choueiry. An approximation of generalized arc-consistency for TCSPs. In *Working notes of the Workshop on Spatial and Temporal Reasoning (IJCAI 03)*, pages 27–32, Acapulco, Mexico, 2003.

[XC03b] Lin Xu and Berthe Y. Choueiry. Efficient techniques for searching the TCSP. In *Working Notes of the Workshop on Spatial and Temporal Reasoning (IJCAI 03)*, pages 125–134, Acapulco, Mexico, 2003.

[XC03c] Lin Xu and Berthe Y. Choueiry. A new efficient algorithm for solving the Simple Temporal Problem. In *TIME-ICTL 2003: Proceedings of the 10th International Symposium on Temporal Representation and Reasoning and Fourth International Conference on Temporal Logic*, pages 210–220, Los Alamitos, CA, USA, 2003. IEEE Computer Society.