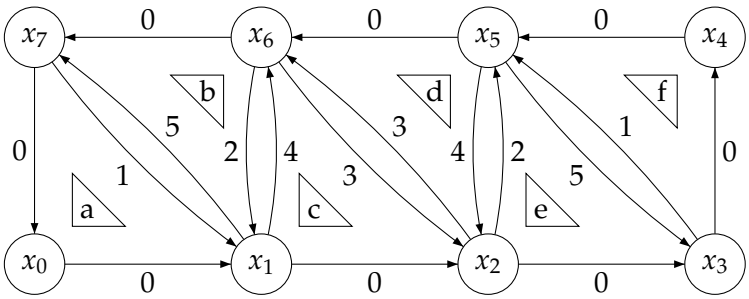
New Algorithms for the Simple Temporal Problem

Master's Thesis



L.R. Planken

New Algorithms for the Simple Temporal Problem

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

L.R. Planken
born in Alkmaar, the Netherlands

Algorithmics Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture:
A pathological STP instance for the $\triangle$STP algorithm; see Section 5.2.1.

New Algorithms for the Simple Temporal Problem

Author: L.R. Planken
Student id: 9654215
Email: L.R.Planken@tudelft.nl

Abstract

This thesis focuses on algorithms for solving the Simple Temporal Problem (STP).

It includes a theoretical analysis of the STP's complexity; discussion and comparisons of known methods for solving the STP, including the current state-of-the-art algorithm $\triangle$STP and incremental methods; an exploration into the chordal graph theory underlying $\triangle$STP; and an extension of this theory to the directed case.

Its main contribution is the proposal of new algorithms for solving the STP by enforcing partial path consistency. The new algorithms are shown to outperform the best existing algorithms; this is done by a theoretical analysis and by empirical research on a variety of test cases, in which an improvement over the older algorithms by up to an order of magnitude is demonstrated.

Thesis Committee:

Chair:	prof. dr. C. Witteveen, Faculty EEMCS, TU Delft
Supervisor:	dr. M.M de Weerdt, Faculty EEMCS, TU Delft
Member:	ir. M.J.H. Heule, Faculty EEMCS, TU Delft
Member:	dr. ir. D. de Ridder, Faculty EEMCS, TU Delft

Contents

List of Figures

Chapter 1

Introduction

In this thesis, we concern ourselves with a formalism that can be used to reason about temporal information, called the *Simple Temporal Problem* (STP). It was first proposed in 1991 by Dechter et al. [DMP91]. Though the scope of problems that can be addressed with the STP is not very wide, it still suffices in many cases. The STP has received rather widespread attention that still lasts today, with applications in such areas as medical informatics [ATMB06], spacecraft control and operations [FRCY97] and air traffic control [BW04]. Moreover, the STP appears as a pivotal subproblem in more expressive formalisms for temporal reasoning, such as the Temporal Constraint Satisfaction Problem—proposed by Dechter et al. in conjunction with the STP—and Stergiou's and Koubarakis's Disjunctive Temporal Problem [SK00].

From the word "simple", the reader may presume that solving the STP is a trivial matter. However, considering the fact that over fifteen years after its inception, it is still possible to make significant progress in the performance of solution algorithms, we feel that the opposite is true. Instead, the word "simple" should primarily be taken to refer to the great clarity, indeed simplicity, with which the STP allows temporal information to be represented.

Overview and contributions

Having briefly introduced the Simple Temporal Problem, the remainder of this introduction presents a brief overview of its treatment in this thesis; also, we globally describe our most important contributions.

In Chapter 2, we formally define the Simple Temporal Problem after first having described a motivating example. We also discuss what a solution to the STP may look like; as the reader shall see, several definitions are possible. Finally, we analyse the complexity of solving the STP, for each of the definitions of "solving" we give; by formally establishing the complexity of a problem, one may get an idea as to which approaches are suitable for tackling it. Such complexity analysis was not performed before; it is one of the contributions of this thesis.

Chapter 3 provides an inventory of available algorithms from the literature that deal with the STP. These range from general graph algorithms via constraint satisfaction algorithms to an algorithm tailored to the STP; unsurprisingly, the latter, called $\triangle$STP, represents the current state of the art. For each method presented, a brief analysis of its time complexity is given.

Then, in Chapter 4, we explore some graph theory that underlies the $\triangle$STP algorithm; we require this theoretical background when seeking improvements. In the same chapter, we propose an extension of the concept of *chordality*—previously only defined for undirected graphs—to the directed case, and explore some implications of this extension.

At this point, the stage is set for the central chapter of this thesis, in which we propose several new methods for solving the STP: Chapter 5. For each type of algorithm that was presented in Chapter 3, we propose an enhancement; further, we formally prove the soundness of these enhancements and theoretically analyse their time complexity. This chapter provides the main contributions of this thesis.

Having presented our enhancements, we evaluate their performance in comparison to that of their precursors in Chapter 6. For both our new algorithms and some previously existing ones, we test multiple heuristics; to the best of our knowledge, though some of the heuristics existed before, they have never been evaluated in this setting before. The evaluation is performed using currently available benchmark tests, randomly generated problem instances and specifically crafted problem instances of our own design. We then compare the practical performance of our algorithms against the theoretical analysis given in Chapter 5.

Finally, in Chapter 7, we present a summary, discuss our results, and indicate promising and interesting directions for future research into the interesting domain of temporal planning.

Chapter 2

Definition and complexity

In this chapter, we introduce the formalism that is the subject matter of this thesis: the *Simple Temporal Problem* (STP). To be able to make any formal theoretical statements when describing algorithms or discussing theoretical complexity, we must provide a rigorous definition; however, in the interest of a smooth introduction, we first include a small example and its translation into an STP instance.

In the latter half of this chapter, we formally establish the complexity of the STP; to the best of our knowledge, this has not been done before. A theoretical complexity analysis of a problem is often a powerful tool when deciding how to approach it. Even if we make no immediate use of its results, it may help give directions for future research.

2.1 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 also use this example in the next chapter to illustrate the operation of the algorithms we describe there.

John goes to work by car, which takes 30–40 minutes; Fred goes

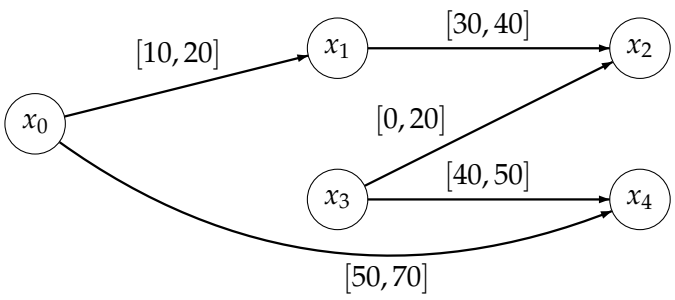


Figure 2.1: An example STP instance

> 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 Fred left home, but not more than 20 minutes later than he left.

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 and stands for seven o'clock this morning. If we represent all time intervals in minutes, the network representation of the STP for this example is given in Figure 2.1.

Now, we can assign values to the time points; if all constraints are satisfied, such an assignment is called a solution. For example, the reader can verify both from the network and from the original story that $\langle x_0 = 0,\ x_1 = 20,\ x_2 = 40,\ x_3 = 20,\ x_4 = 70\rangle$ is a solution of our example STP. Since the story is consistent, we can say the same of the STP instance.

Having given a small taste of what the STP can be used for, we now continue with a more formal definition.

2.2 Definition

In this section, we 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 what constitutes a solution to an STP instance. Then, we define additional properties of the STP on which the algorithms described in Chapter 3 are based.

2.2.1 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 time points, $C = \{c_1, \ldots, c_m\}$, 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 $x_j - x_i \in [-\infty, w_{i\to j}]$. In this text, we 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.

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

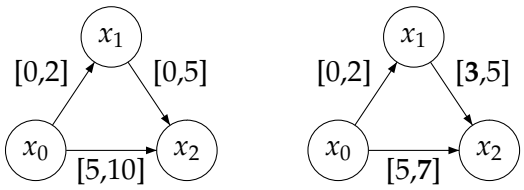


Figure 2.2: An STN and its corresponding minimal network

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 1 February 2008". These could be represented by 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 1 February 2008, 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 time points and binary constraints between them. In the remainder of this text, we seldom treat the temporal reference point specially, except where required for examples.

An additional advantage of having only binary constraints is that an STP instance can then easily be depicted as a graph, in which vertices represent time points and weighted arcs represent constraints. When represented this way, an STP instance is also referred to as 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 sometimes use the terms STP and STN interchangeably.

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 value assignments to all time points. 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. 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 then requires 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. In the remainder of this text, we refer to the problem of deciding whether an STP instance is consistent as STP-CONSISTENCY.

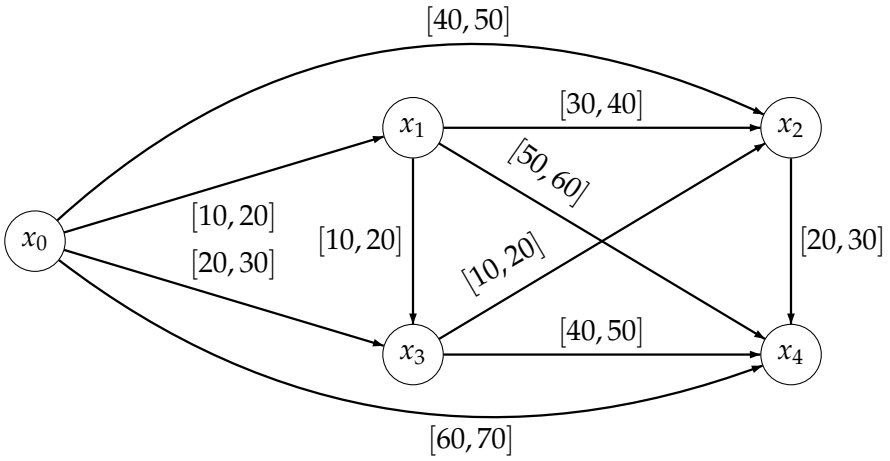


Figure 2.3: The minimal network of the example STP

2.2.2 Minimal network

The constraints given in an STP instance may not represent the actually allowed time difference between two events. We 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 (e.g. [Dec03]), and we also occasionally 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 time point, any value at all can be picked, yielding the first partial solution; an example is setting the value of the temporal reference point to 0. Any partial solution can be extended by picking a new time point and instantiating it to a value that satisfies all constraints from already instantiated time points; 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. In the remainder of this text, we refer to the problem of calculating the minimal network as STP-MINIMALITY.

If an STP instance is inconsistent, i.e. it contains 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 vertices 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 means that the absence of negative cycles implies consistency.

In Figure 2.3, we show the minimal network result of our example problem from Section 2.1. The constraints $c_{0\to4}$ and $c_{3\to2}$ have been tightened, which means that we now know that our original information can 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 $c_{0\to2}$ that John arrived at work between 7:40 and 7:50, and from constraint $c_{0\to3}$ that Fred left his home between 7:20 and 7:30.

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

2.3 Complexity

Perhaps surprisingly, none of the relevant literature gives a formal complexity class for the STP; the respective authors only establish membership in P by giving polynomial algorithms for solving the problem. In this section, we state several theorems to delimit the complexity of the STP more rigidly. For more background on complexity classes and reductions, see e.g. [Sip96].

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 are therefore interested in giving lower bounds on the complexity of STP-CONSISTENCY and upper bounds on that of STP-MINIMALITY.

We focus our attention on the following two complexity classes:

- NC is a subclass of P and contains those problems that can be solved by a parallel algorithm in logarithmic time using a polynomial amount of processors; these problems can therefore be said to be efficiently parallelisable.*
- NL is a subclass of NC and contains those problems that can be solved by a non-deterministic algorithm in logarithmic space; it is equivalent to the class of problems solvable in logarithmic space (with no time bounds) by probabilistic algorithms.

In the following sections, we use these classes to more rigidly delimit the complexity of the STP.

*The abbreviation NC was coined by Stephen Cook to stand for "Nick's class". It honours Nick Pippenger, who did extensive research on problems of this type.

2.3.1 NL-hardness

In this section, we show that deciding consistency of the STP is NL-hard.

Theorem 2.1. STP-CONSISTENCY *is* NL*-hard.*

Proof. 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?

We reduce this problem to STP-INCONSISTENCY; the desired result then follows from the fact that NL = coNL.* Note that for this reduction to be valid, it must use logarithmic memory space.†

We transform every vertex into a time-point variable, and we associate every arc $(u, v) \in A$ with a constraint $c_{u \to v}$ having weight $w_{u \to v} = 0$. This leads to a trivially consistent STP instance, which at least has the solution $\tau = \langle x_1 = 0, \ldots, x_n = 0 \rangle$. We then add the constraint $c_{t \to s}$, with weight $w_{t \to s} = -1$; if a constraint between t and s already existed, it is replaced. This reduction requires but a single pointer into the original problem instance and thus clearly satisfies the memory constraints.

Now, it holds that the STP instance is inconsistent if and only if there exists a directed path from s to t in G:

($\Leftarrow$) If there is a path from s to t in G, its total weight in the STP is trivially equal to 0. The constraint between t and s then yields a cycle with negative total weight and thus leads to inconsistency.

($\Rightarrow$) Conversely, 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 since NL = coNL, also that STP-CONSISTENCY is NL-hard. □

Since just checking consistency is already NL-hard, we know that both other problem variants listed in Section 2.3 also have this lower bound on complexity.

2.3.2 NL-completeness for bounded weights

In this section, we first show that consistency of STPs with bounded weights can be decided in logarithmic space by a nondeterministic algorithm; together with the NL-hardness result from the previous section, this means that the

*This was proven independently by Neil Immerman [Imm88] and Róbert Szelepcsényi [Sze87] in 1987, for which they shared the 1995 Gödel prize.

†Consider a class $\mathcal{M}$ of Turing machines with three tapes: read-only, write-only and read-write; the latter is logarithmically bounded, the others are unbounded. The reduction is valid only if it can be carried out by a Turing machine from $\mathcal{M}$.

Input: A weighted directed graph $G = \langle V, A \rangle$ representing an STP instance.
Output: ACCEPT or REJECT

```
1  choose s ∈ V
2  u ← s
3  sum ← 0
4  repeat
5  |  if u is a sink of G then return REJECT
6  |  choose v : (u, v) ∈ A
7  |  sum ← sum + w_{u→v}
8  |  u ← v
9  until u = s
10 if sum < 0 then return ACCEPT else return REJECT
```

Figure 2.4: Nondeterministic algorithm for STP-INCONSISTENCY

problem is NL-complete. Then, we extend this result to the calculation of the minimal network.

Theorem 2.2. *For constraint weights polynomially bounded by the number of time points,* STP-CONSISTENCY *is a member of* NL.

Proof. A nondeterministic algorithm can find any cycle by choosing a starting vertex s, repeatedly choosing an edge to walk from the current vertex, and verifying that the last vertex is equal to the first one. Thus, for solving STP-INCONSISTENCY, it can in this fashion nondeterministically select a cycle to walk, keeping a running total weight, and check negativity when the original vertex is reached again. See Figure 2.4 for the pseudocode of such an algorithm; note that the 'choose' operations in lines 1 and 6 are the only nondeterministic steps.

The space allotted to the algorithm for determining whether the cycle has negative total weight is limited: $\mathcal{O}(\log |I|)$, where $|I|$ is the size of the problem instance. The size of an STP instance $\mathcal{S} = \langle X, C \rangle$ depends on its encoding; if we let $w_{\max}$ be the maximum absolute constraint weight, we may have $|I| = \mathcal{O}(m \log n \log w_{\max})$ when the instance is encoded as a list of constraints with weights, or $|I| = \mathcal{O}(n^2 \log w_{\max})$ 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 by summing 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 $\log w_{\max} \in \mathcal{O}(\log n)$*, which is attained for $w_{\max} \in \mathcal{O}(n^k)$ with constant k, i.e. if the maximum weight is polynomially bounded by the number of time points. Additionally, walking the cycle requires three pointers

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

Input: An STP instance $\mathcal{S} = \langle X, C \rangle$
Output: The minimal network $\mathcal{S}' = \langle X, C' \rangle$

if `inconsistent`$(\mathcal{S} = \langle X, C \rangle)$ **then return** REJECT
$C' \leftarrow \emptyset$
foreach $x, y \in X,\ x \neq y$ **do**
  $c'_{x \to y} \leftarrow$ `tighten`$(c_{x \to y})$
  $C'' \leftarrow C \cup \{y - x > w'_{x \to y}\}$
  if `consistent`$(\mathcal{S}'' = \langle X, C'' \rangle)$ **then return** REJECT
  $C'' \leftarrow C \cup \{y - x = w'_{x \to y}\}$
  if `inconsistent`$(\mathcal{S}'' = \langle X, C'' \rangle)$ **then return** REJECT
  $C' \leftarrow C' \cup \{c'_{x \to y}\}$
end
return $\mathcal{S}' = \langle X, C' \rangle$

Figure 2.5: Nondeterministic algorithm for STP-MINIMALITY

s, u, v to vertices, as can be seen in Figure 2.4. Since each of these is also logarithmic in size, we can conclude that the space requirements are met.

Hence, if the weights are limited in this fashion, STP-INCONSISTENCY is a member of NL. Again, this result then transfers to STP-CONSISTENCY since NL = coNL. □

We now extend this result to the calculation of the minimal network.

Theorem 2.3. *For constraint weights polynomially bounded by the number of time points,* STP-MINIMALITY *is in* FNL.

This is not a decision problem, as above, but a function problem. We show that this problem is in FNL.* The method we describe here makes use of the result from Theorem 2.2 that deciding consistency is in NL.

Proof. We include a nondeterministic algorithm for STP-MINIMALITY in Figure 2.5. Since STP-CONSISTENCY is in NL, deciding consistency can be done as often as desired during the solving process.

The very first step of the algorithm is to verify that the original network is consistent. If it is not, the algorithm rejects; the concept of a minimal network is nonsensical for an inconsistent network. The algorithm then iterates over all pairs of time points (x, y) and nondeterministically tightens the weight of the constraint $c_{x \to y}$. This tightening operation selects a new weight $w'_{x \to y} \in [-nw_{\max}, \min(w_{x \to y}, nw_{\max})]$. In this way, every new constraint $c'_{x \to y}$

*A function is FNL-computable if it can be calculated by a non-deterministic algorithm using a logarithmically bounded amount of memory space.

is at least as tight as the original constraint, and there are no universal constraints having $w'_{x\to y} = \infty$; both are requirements for the minimal network.* The lower bound follows from a path with lowest possible total weight; the upper bounds follow from an already minimal constraint or a path with highest possible total weight, respectively. The space required for $w'_{x\to y}$ is then $\mathcal{O}(\log n)$ for polynomially bounded weights.

For $c'_{x\to y}$ to be minimal, it must satisfy two further properties:

- Its inverse, when added to the original STP instance, must yield inconsistency; if this is not the case, this means that $c'_{x\to y}$ has been made too tight and invalidated some solutions.
- Setting the distance between x and y to exactly $w'_{x\to y}$ must yield consistency; otherwise, $c'_{x\to y}$ is too lax.

These properties are checked in lines 6 and 8 of Figure 2.5, respectively. If these properties are satisfied, $c'_{x\to y}$ is indeed minimal and the algorithm moves on to the next constraint.

During operation, at all times only a single constraint and the pointers to x and y have to be kept in memory; therefore, with bounded weights, the space requirement is satisfied. We conclude that STP-MINIMALITY $\in$ FNL. □

2.3.3 Membership in 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 in $\mathcal{O}(\log^c n)$ time by a parallel random access machine (PRAM) having $\mathcal{O}(n^k)$ parallel processors, 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, which means that STP-MINIMALITY is in NC^2; this result then transfers to STP-CONSISTENCY. For more background on parallel algorithms and the PRAM model, see e.g. [JáJ92].

Theorem 2.4. STP-MINIMALITY $\in \mathsf{NC}^2$.

*An exception to the latter requirement applies if the STN is not connected; in this case, we must have $w'_{x\to y} = \infty$ for x and y in different components of the constraint graph. Because ST-CONNECTIVITY is in NL, the algorithm can check this as a subroutine; if it finds that the vertices are unconnected, it leaves the constraint universal and moves on to process the next constraint.

†L is the class of problems solvable by deterministic algorithms in logarithmic space.

Input: An STP instance $\mathcal{S} = \langle X, C \rangle$
Output: The minimal network $\mathcal{S}' = \langle X, C' \rangle$ or INCONSISTENT

```
1  foreach 1 ≤ i, j ≤ n, i ≠ j pardo
2  |  w'_{i→j} ← w_{i→j}
3  end
4  for l ← 1 to ²log n do
5  |  foreach 1 ≤ i, j, k ≤ n, i ≠ j ≠ k ≠ i pardo
6  |  |  D_ik[j] ← w'_{i→j} + w'_{j→k}
7  |  end
8  |  for m ← ²log n to 1 do
9  |  |  foreach 1 ≤ i, j ≤ n, i ≠ j and 1 ≤ k ≤ 2^{m−1} pardo
10 |  |  |  D_ij[k] ← min(D_ij[k], D_ij[2^{m−1} + k])
11 |  |  end
12 |  end
13 |  foreach 1 ≤ i, j ≤ n, i ≠ j pardo
14 |  |  w'_{i→j} ← min(w'_{i→j}, D_ij[1])
15 |  end
16 end
17 consistent ← TRUE
18 foreach 1 ≤ i, j ≤ n, i ≠ j pardo
19 |  if w'_{i→j} + w'_{j→i} < 0 then consistent ← FALSE
20 end
21 if consistent then return S' (defined by the new weights w')
22 else return INCONSISTENT
```

Figure 2.6: Parallel algorithm for STP-MINIMALITY

Proof. We assume a concurrent-read concurrent-write* parallel random access machine (CRCW PRAM). The algorithm operates on the complete graph; if $w_{i\to j}$ is undefined in the problem instance, an infinite weight is assumed. For the sake of clarity in presentation of the algorithm, we further assume that the number of time-point variables is a power of two; if it is not, dummy time points can be added that have an infinite-weight constraint with all other time points. Our approach is included as in Figure 2.6.

The algorithm maintains the current shortest known path from x_i to x_j as $w'_{i\to j}$. This measure is updated by performing the following operations:

1. for every triple (i, j, k), the algorithm determines (in parallel) the length of the path from i to k via j and stores it in $D_{ik}[j]$ (lines 5–7);

*The only place where concurrent write operations can occur is line 19; since the written value is always INCONSISTENT, we can safely assume the Common CRCW model.

2. then, for each pair (i, j), it collapses the list D_{ij} by repeatedly taking pairwise minima until the minimum entry is contained in $D_{ij}[1]$ (lines 8–12), requiring ${}^2\log n$ sequential iterations;

3. finally, the variables $w'_{i\to j}$ are updated in parallel to reflect the new shortest paths (lines 13–15).

The reader can verify that after the first iteration of the main loop, the variables $w'_{i\to j}$ contain the shortest paths from x_i to x_j via at most two constraint edges, and after the second iteration, this length has doubled to four constraint edges; in general, after the lth iteration, all paths of length 2^l have been considered. This means that ${}^2\log n$ iterations of the main loop suffice to take all possible paths into account. The entire algorithm therefore requires time $\Theta(\log^2 n)$.

Now, it remains to be shown that the algorithm needs at most a polynomial number of processors. Clearly, the parallel loops as presented in Figure 2.6 require n^3 processors; this amount suffices if we assume that taking the minimum and calculating the sum of up to n terms of size $\mathcal{O}(w_{\max})$ are unit operations. If this simplification is unjustified, multiplying the amount of processors by a factor of ${}^2\log(nw_{\max})$ (which is the bit length of the largest possible absolute weight encountered) ensures that summation can be performed in logarithmic time $\mathcal{O}(\log\log(nw_{\max}))$ and taking the minimum requires but constant time. □

2.4 Summary and conclusions

In this chapter, we gave a formal definition of the STP and formulated a motivating example. We listed three possible definitions of solving an STP instance $\mathcal{S}$, in order of increasing difficulty: (i) determining whether $\mathcal{S}$ is consistent; (ii) finding a single instantiation to all time-point variables; and (iii) calculating all minimal constraints. In this chapter and the remainder of this thesis, we concern ourself only with (i) and (iii), referring to them as STP-CONSISTENCY and STP-MINIMALITY, respectively.

To our best knowledge, none of the existing literature included a formal complexity analysis of these problems beyond implicitly establishing membership in P by providing polynomial-time algorithms. Therefore, as our main contribution in this chapter, we undertook this analysis ourselves. We proved NL-hardness and membership in NC^2 as lower and upper bounds to the complexity of both STP-CONSISTENCY and STP-MINIMALITY; furthermore, we proved that for weights polynomially bounded in the number of time points, both problems are NL-complete.

From these results, we conclude the following:

- even though STP-MINIMALITY implies STP-CONSISTENCY and is thus more difficult, the formal complexity of the two problems is identical;

- from the membership in NC^2, it follows that the solving of these problems is efficiently parallelisable;
- since it is known that NL is equivalent to RL (see e.g. [MR95], who give an algorithm for ST-CONNECTIVITY), the result for the bounded case implies that there exists a randomised approach to the STP that runs in logarithmic memory and unbounded time.

We also state some questions that may be addressed by future research:

- *How can the inherent parallelism of the STP best be exploited by solvers?*
  From the membership of the STP in NC^2, it follows that a parallel approach is viable; we demonstrated this for the abstract PRAM model. Further research may be conducted into a concrete implementation on a parallel computer with a finite amount of processors.
- *What effect does a polynomial bound on edge weights have on practical applications of the STP?*
  Our expectation is that the bounded STP is sufficient for representation of many practical cases; however, future research is required to confirm or refute this expectation.
- *Does there exist some subclass $\mathcal{C}$ of the bounded STP such that $\mathcal{C} \in \mathsf{RLP}$?*
  A positive answer to this question would imply that problems in $\mathcal{C}$ are amenable to a randomised approach requiring polynomial time and logarithmic space. Defining characteristics of $\mathcal{C}$ could regard the structure of the constraint graph, since it is known that the undirected variant of ST-CONNECTIVITY is a member of RLP.

In the next chapter, we move from the purely theoretical to a more practical perspective, by discussing known algorithms for dealing with the STP and their comparative merits.

Chapter 3

Known approaches

Having defined the STP and established the complexity of solving it, we now move on to describing a small selection of available algorithms for solving it. The algorithms we include can be classified in three groups:

- algorithms for determining consistency (Section 3.1);
- algorithms that calculate the minimal network (Section 3.2); and
- algorithms that calculate only a relevant subset of the minimal network (Section 3.3).

For the first two of these, incremental algorithms are also available in the literature; we include these in Section 3.4.

In Figure 3.1, we reproduce the STN of our example from Chapter 2, with the vertex indices renumbered to run from 1 to n; throughout this chapter, we show the constraint graphs produced by the respective algorithms when run on this STN when necessary to illustrate their operation.

By its nature, this chapter contains only few and relatively minor new contributions; however, two contributions that must be mentioned here. Firstly, to the best of our knowledge, the analysis of the time complexity of incremental directed path consistency was never published before; we present it

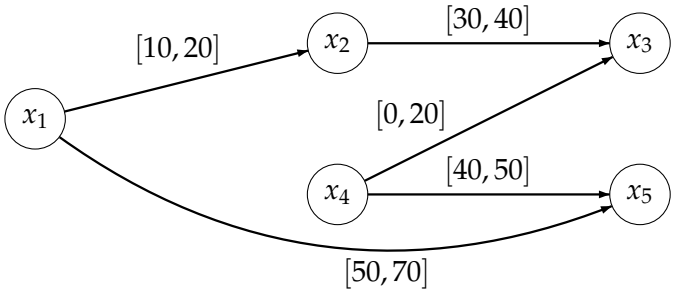


Figure 3.1: The example STN from Chapter 2

Input: Weighted directed graph $G = \langle V, A \rangle$, vertex $v_{\text{origin}} \in V$
Output: List D of distances from v_{origin} or INCONSISTENT

```
1  ∀v ∈ V : D[v] ← ∞
2  D[v_origin] ← 0
3  repeat n times
4  |  foreach (v_i, v_j) ∈ A do
5  |  |  D[v_j] ← min(D[v_j], D[v_i] + w_{i→j})
6  |  end
7  end
8  foreach (v_i, v_j) ∈ A do
9  |  if D[v_j] > D[v_i] + w_{i→j} then return INCONSISTENT
10 end
11 return D
```

Figure 3.2: Bellman's and Ford's algorithm

in Section 3.4.1. Also, for the incremental full path consistency algorithm included in Section 3.4.2, no applicable pseudocode description was found in the literature, so we wrote our own.

3.1 Determining consistency

3.1.1 Bellman's and Ford's algorithm

This algorithm, presented in Figure 3.2, is based on publications by Bellman and Ford in 1958 and 1962, respectively [Bel58][FF62]. It calculates the shortest paths to all vertices from a single source. It is similar to Dijkstra's algorithm [Dij59], but unlike the latter, it can deal with negative edge weights.

If the graph contains a negative cycle, the algorithm detects this in the loop that spans lines 8–10. The reason for this is that in a negative cycle, the distance matrix is updated time and time again, and is never finished, so to speak; when the algorithm stops updating the distance matrix, there is always an edge in a negative cycle that fails the condition in line 9. Note that an efficient implementation of this algorithm does not execute line 5 for all arcs, but only for those whose source vertex had its distance updated in the last iteration. However, the worst-case time complexity of this algorithm remains $\mathcal{O}(nm)$.

3.1.2 Directed path consistency

This method for determining consistency of an STP instance was described in the publication by Dechter et al. that introduced, among other things, the STP itself [DMP91]. This is a tailored version of the directed path consistency

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

Figure 3.3: Directed path consistency algorithm (DPC)

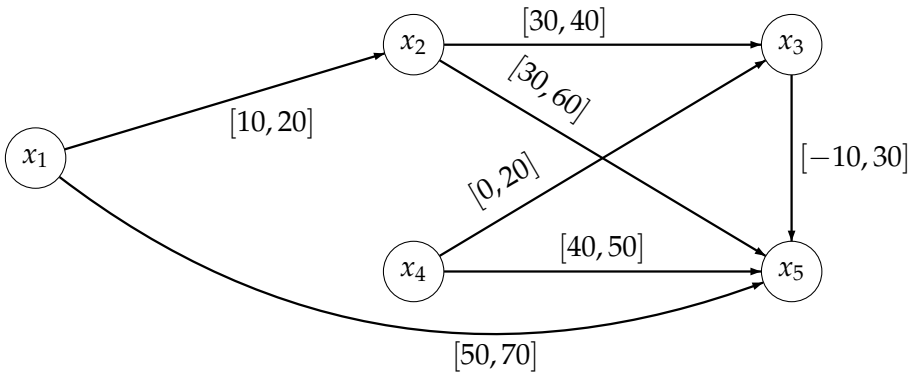


Figure 3.4: The example STN after running DPC

algorithm for general constraint satisfaction problems (see e.g. [Dec03]). In the remainder of this text, we use the abbreviation DPC to stand for directed path consistency.

The algorithm assumes that there is a total ordering $\prec$ over the set of time points. Throughout this text, without loss of generality, we number the time points $\{x_1, \ldots, x_n\}$ such that $x_i \prec x_j$ if and only if $i < j$. Note that the ordering has no impact on the soundness of the algorithm, but does influence its performance, as we show below.

See Figure 3.3. The algorithm iterates over k from 1 to n and for all $i, j > k$ (including $i = j$) computes the shortest distance from time point x_i via x_k to x_j. It then updates the weight $w_{i\to j}$ if the new value is less than the original value. The initial value of all $w_{i\to i}$ (the weight of the virtual edge from a time point 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. See Figure 3.4 for the result of applying DPC to the example problem from Chapter 2.

When running the algorithm, $w_{i\to j}$ trivially remains 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 time points is significant for performance. In Chapter 5, we show that careful choice of this ordering results in high efficiency. In general, the complexity of the algorithm is

```
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
```

Figure 3.5: Floyd's and Warshall's APSP algorithm

$\mathcal{O}(n(w^*(d))^2)$, where $w^*(d)$ is a measure called the *induced width* relative to the ordering d of the time points. For $d = (x_1, x_2, \ldots, x_n)$, we have

$$w^*(d) \;=\; \max_i \left|\{w_{i\to j} < \infty \vee w_{j\to i} < \infty \;\mid\; j > i\}\right|$$

That is, for each time point x_i, we count the number of time points x_j that appear later in the ordering and are adjacent to x_i in the constraint graph; the induced width is then equal to the maximum of these counts.

The induced width w^* of the graph itself is defined as the smallest induced width along any ordering d of the time points. Determining $w^* = \min_d w^*(d)$ is an NP-complete problem; however, as we shall see in the next chapter, fortunately there exist heuristics that yield good results in practice.

3.2 Calculating the minimal network

3.2.1 All-pairs shortest paths

The algorithm presented in Figure 3.5 for calculating all-pairs-shortest-paths (APSP) on a graph was first published in 1962 [Flo62][War62]. It computes the shortest distance between all pairs of vertices, or finds any negative cycle if it exists, in time $\Theta(n^3)$.

The algorithm runs a loop of n iterations, for $1 \leq k \leq n$. After n iterations, all $w_{i\to j}$ have been set to their minimal values; inconsistency can again be concluded as soon as any $w_{i\to i}$ drops below zero. The resemblance to directed 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. Indeed, for the STP, this algorithm is equivalent to the (full) path consistency algorithm from constraint satisfaction literature.

In Figure 3.6, we show the result of applying the APSP algorithm to our example problem from Section 2.1. This is, of course, exactly the minimal network that we already included in Chapter 2.

3.2.2 Johnson's algorithm

Published in 1977 [Joh77], this algorithm yields improved performance over Floyd's and Warshall's algorithm for sparse graphs.

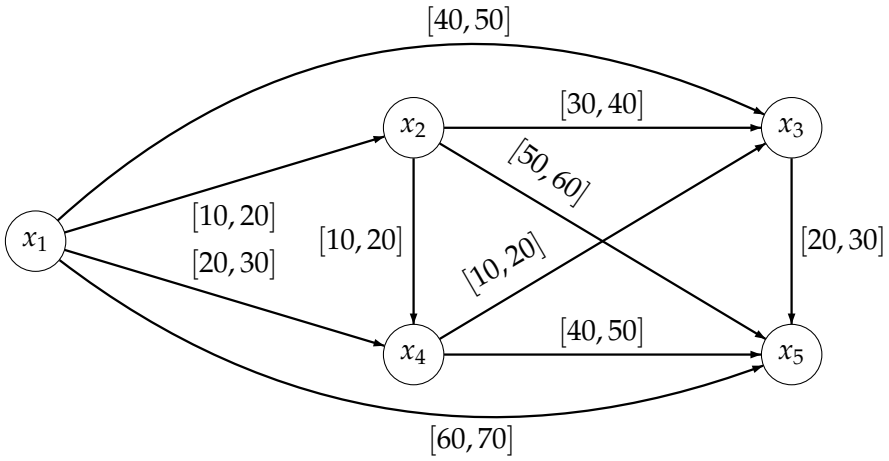


Figure 3.6: The minimal network after APSP

The algorithm adds a new vertex $v_0 \notin V$ with zero-weight edges to all other vertices $v_i \in V$ 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. The algorithm now associates a value $h(v_i)$ with each original vertex $v_i \in V$. This value is equal to the shortest path from v_0 to v_i; from the addition of the new zero-weight edges, it follows that it is never positive.

These values $h(v)$ 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 vertex to all other vertices. 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.

The time complexity of Johnson's algorithm is $\mathcal{O}(nm + n^2 \log n)$, which follows directly from the combination of a single run of Bellman's and Ford's algorithm combined with n runs of Dijkstra's algorithm.

3.3 Partial path consistency

In 2003, Xu and Choueiry [XC03] 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 (PC) is defined on complete graphs: to make a constraint graph PC, edges between all pairs of time points are considered and updated, even those that do not correspond to constraints that are ex-

plicitly defined by the problem and thus are initially universal constraints. Enforcing PC on an STP instance corresponds to calculating the minimal network, as we have noted in Section 2.2. The property of partial path consistency is instead defined for chordal graphs, and considers no other edges than those in the graph. Note that a graph is chordal if each cycle of size greater than 3 contains a chord, i.e. an edge connecting two nonadjacent vertices in the cycle. If a constraint graph is not chordal, it can be made so by the process of triangulation, which adds some fill edges (representing universal constraints). In Chapter 4, we explore these concepts in more depth. However, we can state here that the triangulated graph generally contains far less edges than the complete graph, especially so for sparse constraint graphs; hence, enforcing PPC is often far cheaper than enforcing PC.

Bliek and Sam-Haroud have proven that for problems with convex constraints, PPC is equivalent to PC if we are concerned only with the constraints that were originally present in the problem. 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.

Xu and Choueiry [XC03] first realised that PPC can be used for solving the Simple Temporal Problem and implemented an efficient version of the algorithm, called $\triangle$STP. Their algorithm considers the graph as being composed of triangles rather than edges, which saves some constraint checks in comparison to the original algorithm published by Bliek and Sam-Haroud.

We include $\triangle$STP in Figure 3.7. Note that if line 7 is executed, it enqueues T itself as well as its neighbours; line 10 makes sure that T is removed from the queue after it has been processed. In Figure 3.8, we show the result of applying the PPC algorithm to our example problem. It can be seen that the constraints $c_{0\rightarrow 4}$ and $c_{3\rightarrow 2}$ are indeed minimal, but only two new constraints have been added to the original problem to triangulate it, as opposed to the five extra constraints in the complete graph that resulted from the APSP algorithm. Instead of the triangulation used here, in which edges representing $c_{0\rightarrow 2}$ and $c_{0\rightarrow 3}$ were added, we could have chosen to add any pair of edges that were not already present and have a vertex in common.

Xu and Choueiry did not give a theoretical upper bound for the runtime of their algorithm, but an empirical study showed that it outperformed Floyd's and Warshall's all-pairs shortest paths algorithm except for very dense constraint graphs, and outperformed DPC for graphs of low to moderate density. To the best of our knowledge, $\triangle$STP still represents the state of the art for enforcing PPC.

Input: A chordal STN $\mathcal{S} = \langle V, A \rangle$
Output: The PPC network of $\mathcal{S}$ or INCONSISTENT

```
1  Q ← all triangles in S
2  while Q ≠ ∅ do
3      choose T ∈ Q
4      foreach permutation (v_i, v_j, v_k) of the time points in T do
5          w_{i→k} ← min(w_{i→k}, w_{i→j} + w_{j→k})
6          if w_{i→k} has changed then
7              if w_{i→k} + w_{k→i} < 0 then return INCONSISTENT
8              Q ← Q ∪ {all triangles in S containing v_i and v_k}
9          end
10     end
11     Q ← Q \ T
12 end
```

Figure 3.7: The △STP algorithm

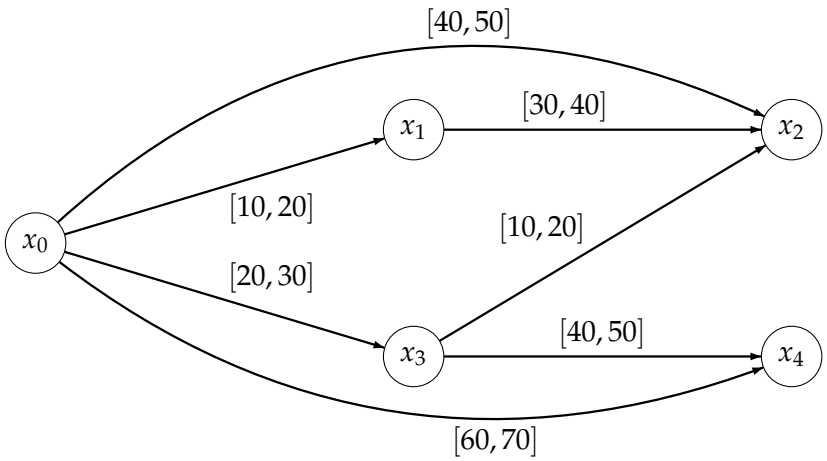


Figure 3.8: The result of applying △STP

3.4 Incremental methods

The STP appears as a pivotal subproblem of more complex temporal problems, such as the Disjunctive Temporal Problem introduced by Stergiou and Koubarakis [SK00]. The usual approach for solving these problems is to gradually build up an STP instance (called a *component STP*) and backtrack whenever an inconsistency is encountered. Beside the consistency check, maintaining minimal relations is often also very relevant as this information can be used in heuristics that guide the search process.

More concretely, at each step in the backtracking search, a single constraint is to be added to a component that is already known to be consistent (or minimal). In these cases, it is not practical to solve the entire component STP again; instead, one wants to build upon the consistency (or minimality) result that has been achieved earlier and recalculate only those constraints which have to be changed. In this section, we describe some algorithms available from literature that achieve this task.

3.4.1 Incremental directed path consistency

This method for incrementally enforcing directed path consistency (IDPC) was published by Chleq [Chl95]. We include IDPC in Figure 3.9; for consistency of presentation in this text, its form differs slightly from Chleq's original publication. In particular, the first two lines that handle trivial situations were not present in the previous version:

- either the constraint to be added immediately yields a negative cycle, and thus inconsistency is concluded (line 1); or
- the constraint to be added is no tighter than the one that was already present in the STP instance, explicit or implied, and thus no further calculations have to be performed (line 2).

If these situations do not occur, the new information must be propagated through the constraint graph. During operation, the set D contains all those edges whose weights have been adjusted, and is used to determine which edges to consider next. The set Q is in effect a priority queue which allows the algorithm to iterate over the vertices participating in the edges in D in order of the precedence relation. Chleq's original algorithm made explicit provisions for adding edges; in our presentation, we assume for brevity that nonexistent edges are represented by infinite weights. Lowering a weight from an infinite to a finite value then implicitly means that a constraint edge is added to the STP instance.

Even when updating just a single constraint in an otherwise already directionally path-consistent STP instance, the worst-case complexity of the incremental algorithm is no better than that of the "single-shot" algorithm:

Proposition 3.1. *The worst-case performance of IDPC is $\mathcal{O}(n(w^*)^2)$; for a (nearly) complete graph, where $w^* = n$, this yields $\mathcal{O}(n^3)$.*

Input: A directionally path-consistent STP $\mathcal{S} = \langle X, C \rangle$ and a new constraint $c'_{a \to b}$.
Output: CONSISTENT if $c'_{a \to b}$ has been added to $\mathcal{S}$, which is again directionally path-consistent; INCONSISTENT otherwise.

```
1  if w'_{a→b} + w_{b→a} < 0 then return INCONSISTENT
2  if w'_{a→b} ≥ w_{a→b} then return CONSISTENT
3  w_{a→b} ← w'_{a→b}
4  D ← {(a, b)}
5  Q ← {min(a, b)}
6  while Q ≠ ∅ do
7  |  k ← min(Q)
8  |  Q ← Q \ {k}
9  |  forall i, j > k, i ≠ j such that (i, k) ∈ D ∨ (k, j) ∈ D do
10 |  |  if w_{i→j} > w_{i→k} + w_{k→j} then
11 |  |  |  if w_{i→k} + w_{k→j} + w_{j→i} < 0 then return INCONSISTENT
12 |  |  |  w_{i→j} ← w_{i→k} + w_{k→j}
13 |  |  |  D ← D ∪ {(i, j)}
14 |  |  |  Q ← Q ∪ {min(i, j)}
15 |  |  end
16 |  end
17 end
18 return CONSISTENT
```

Figure 3.9: Incremental directed path consistency algorithm (IDPC)

Recall that w^*, introduced in Section 3.1.2 is the induced width of a graph.

Tsamardinos and Pollack briefly mention IDPC on page 7 of [TP03] and incorrectly state that its complexity is $\mathcal{O}(n^2)$ in the worst case.

3.4.2 Incremental full path consistency

Instead of maintaining directed path consistency, it is also possible to incrementally maintain full path consistency (IFPC), i.e. all-pairs shortest paths. To our best knowledge, this algorithm has not been presented explicitly for STPs in the available literature. Tsamardinos and Pollack [TP03] mention this approach and cite a publication by Mohr and Henderson [MH86]; however, instead of an incremental approach, this publication only presented a new "single-shot" path consistency algorithm for general constraint satisfaction problems. Instead of trying to distill an incremental algorithm for the STP from the algorithm presented by Mohr and Henderson, we present our own approach in Figure 3.10.

Two differences with the IDPC algorithm meet the eye: the algorithm

Input: A minimal STP $\mathcal{S} = \langle X, C \rangle$ and a new constraint $c'_{a \to b}$.
Output: CONSISTENT if $c'_{a \to b}$ has been added to $\mathcal{S}$, which is again minimal; INCONSISTENT otherwise.

```
1  if w'_{a→b} + w_{b→a} < 0 then return INCONSISTENT
2  if w'_{a→b} ≥ w_{a→b} then return CONSISTENT
3  w_{a→b} ← w'_{a→b}
4  I ← ∅; J ← ∅
5  forall x_k ∈ X, x_k ≠ x_a, x_b do
6      if w_{k→b} > w_{k→a} + w_{a→b} then
7          w_{k→b} ← w_{k→a} + w_{a→b}
8          I ← I ∪ {k}
9      end
10     if w_{a→k} > w_{a→b} + w_{b→k} then
11         w_{a→k} ← w_{a→b} + w_{b→k}
12         J ← J ∪ {k}
13     end
14 end
15 forall i ∈ I, j ∈ J, i ≠ j do
16     if w_{i→j} > w_{i→a} + w_{a→j} then
17         w_{i→j} ← w_{i→a} + w_{a→j}
18     end
19 end
20 return CONSISTENT
```

Figure 3.10: Incremental full path consistency algorithm (IFPC)

has a worst-case time complexity of $\mathcal{O}(n^2)$, and deciding whether the new constraint incurs inconsistency requires but constant time. The former property follows from the maximum number of iterations of the loop spanning lines 15–19. The latter property is especially desirable for application to e.g. the Disjunctive Temporal Problem, as already pointed out by Tsamardinos and Pollack; it means that *forward checking* can be done very efficiently.

For improved efficiency, we maintain sets I and J of time points; every combination of two time points, one from each of these sets, must be checked. This does not improve on the algorithm's worst-case time complexity, which remains $\mathcal{O}(n^2)$ if $I = J = X \setminus \{a, b\}$, but may help in many practical cases.

To see that this approach is sound, consider two time points x_i and x_j and assume that after addition of $c_{a \to b}$, the new shortest path between these time points is $x_i \to x_a \to x_b \to x_j$. But then, the shortest path between x_a and x_j must also follow the route $x_a \to x_b \to x_j$. In the first loop, $w_{a \to j}$ is updated to reflect this, and x_j is added J; the same holds for $w_{i \to b}$. Then, in the second loop, $w_{i \to j}$ will be set correctly. Our choice for $w_{i \to a} + w_{a \to j}$ is arbitrary; we

could also have chosen the path via x_b.

Finally, note that if the shortest path from x_i to x_j includes the updated constraint $c_{a\to b}$, it must always be of the form $x_i \to x_a \to x_b \to x_j$; since the network was already minimal before the new constraint was added, we have for any x_k that $w_{i\to a} \leq w_{i\to k} + w_{k\to a}$ and $w_{b\to j} \leq w_{b\to k} + w_{k\to j}$.

3.5 Summary

In this chapter, we explored a variety of algorithms that may be used to solve the STP. For determining consistency, we included a graph algorithm by Bellman and Ford and the directed path consistency algorithm from constraint satisfaction literature. For calculating the minimal network, two graph algorithms were included: Floyd's and Warshall's algorithm, which is equivalent to (full) path consistency from constraint satisfaction literature, and Johnson's algorithm. The latter exhibits better performance when run on sparse graphs. It is also possible to calculate only a subset of the minimal network by enforcing partial path consistency (PPC). This method considers a triangulation of the constraint graph instead of its completion, which generally contains far less edges. This explains why $\triangle$STP, an efficient implementation of PPC, is currently the state-of-the-art algorithm for calculating minimal constraints.

Finally, we discussed two incremental methods: incremental directed path consistency (IDPC) and incremental full path consistency (IFPC). Each of this methods takes as input an STP instance that already satisfies the respective path consistency property and a new constraint to be added to it, and then enforces that property anew. Somewhat surprisingly, since it enforces a stronger property, IFPC has the better worst-case complexity analysis.

Before we can propose new algorithms of our own, we take a step back in the next chapter to explore some graph theory. In this chapter, we mentioned that $\triangle$STP considers a triangulation of the constraint graph; if we hope to design a more efficient algorithm, it is necessary that we first discuss the theory behind these triangulations.

Chapter 4

Graph triangulation

In this chapter, we discuss some graph theory that underlies the current state-of-the-art STP solver △STP. By exploring graph theory on triangulation, both existing and new, we hope to find footholds for developing an improvement to this algorithm. In this chapter, we only consider the graphs themselves and abstract away from the temporal interpretation that the STP attaches to them. In the next chapter, we shift our focus back to the STP and apply the new insights gained in this chapter.

The main text of this chapter can be split into two parts. In the first part, we describe existing theory concerning undirected triangulations; in the second part, we propose an extension to the directed case. We conclude this chapter with a brief discussion of the theory and its implication to the STP.

4.1 The undirected case

In this section, we briefly state some established concepts and results for undirected graph triangulation that are readily available from the literature; we specifically mention publications by Rose [Ros72] and Kjærulff [Kjæ90]. General graph theory concepts that we make use of can be found in e.g. [Wes96].

4.1.1 Chordality

Definition 4.1. *Let $G = \langle V, E \rangle$ be an undirected graph. We can define the following concepts:*

- *If $(v_1, v_2, \ldots, v_k, v_{k+1} = v_1)$ with $k > 3$ is a cycle, then any edge on two nonadjacent vertices $\{v_i, v_j\}$ with $1 < j - i < k - 1$ is a* chord *of this cycle.*
- *G is* chordal *(also ambiguously called "*triangulated*"*) if every cycle of length greater than 3 has a chord.*

*The term "triangulation" is sometimes also used for maximal planar graphs, which do not interest us here.

Input: Undirected graph $G = \langle V, E \rangle$
Output: An ordering $\# : V \to \mathbb{N}$

```
1 L ← ∅
2 i ← 1
3 repeat
4 |   choose v ∈ V \ L for which |{{v, w} ∈ E | w ∈ L}| is maximal
5 |   #(v) ← i
6 |   i ← i + 1
7 |   L ← L ∪ {v}
8 until L = V
9 return #
```

Figure 4.1: Maximum cardinality search algorithm

- *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 $d = (v_n, \ldots, v_1)$ be an ordering of V. Also, let G_i denote the subgraph of G induced by $V_i = \{v_1, \ldots, v_i\}$; note that $G_n = G$. The ordering d is a* simplicial elimination ordering *of G if every vertex v_i is a simplicial vertex of the graph G_i.*

We then have the following result:

Theorem 4.1. *An undirected graph $G = \langle V, E \rangle$ is chordal if and only if it has a simplicial elimination ordering.*

Rose uses the concept "monotone transitive ordering":

Definition 4.2. *An ordering $(v_n, \ldots, v_1)$ is* monotone transitive *if it satisfies the following condition:*

$$\{v_i, v_j\}, \{v_i, v_k\} \in E \ \wedge \ i > j, k \ \wedge \ j \neq k \ \Rightarrow \ \{v_j, v_k\} \in E$$

It is not hard to see that this condition is satisfied exactly by the simplicial elimination orderings.

Finally, chordality checking can be done efficiently in $\mathcal{O}(n + m)$ time by the maximum cardinality search algorithm (Figure 4.1), which also produces a simplicial elimination ordering for chordal graphs (in reverse order). It labels vertices one by one, starting from an arbitrary vertex and maintaining the already labelled vertices in L; at each iteration, it labels the vertex that has the most neighbours in L.

4.1.2 Triangulation

If a graph is not chordal, it can be made so by the addition of a set of *fill edges*. These are found by eliminating the vertices one by one and connecting

all vertices in the neighbourhood of each eliminated vertex, thereby making it simplicial. If the graph was already chordal, following its simplicial elimination ordering means that no fill edges are added. In general, it is desirable to achieve chordality with as few fill edges as possible.

Definition 4.3. *Let $G = \langle V, E \rangle$ be an undirected graph that is not chordal. A set of edges T with $T \cap E = \emptyset$ is called a* triangulation *if $G' = \langle V, E \cup T \rangle$ is chordal. T is* minimal *if there exists no subset $T' \subset T$ such that T' is a triangulation. T is* minimum *if there exists no triangulation T' with $|T'| < |T|$.*

Determining a globally minimal triangulation is an NP-complete problem; in contrast, a locally minimal triangulation can be found in $\mathcal{O}(nm)$ time [Kjæ90]. Since finding the smallest triangulations is so hard, several heuristics have been proposed for this problem. Kjærulff has found that both the minimum fill and minimum degree heuristics produces good results.

The *minimum fill* heuristic always selects a vertex whose elimination results in the addition of the fewest fill edges; it has worst-case time complexity $\mathcal{O}(n^2)$. The *minimum degree* heuristic is even simpler, and at each step selects the vertex with the smallest number of neighbours; its complexity is only $\mathcal{O}(n)$, but its effectiveness is somewhat inferior to that of the minimum fill heuristic.

Either of these heuristics can be helped by decomposing the graph before triangulation (see Section 4.1.3 below), and by the following fact:

Theorem 4.2. *If there is a vertex $v \in V$ with degree $d(v) = n - 1$ (henceforth called a* hub *vertex), then G is chordal if and only if its subgraph induced by the removal of v is.*

This means that if hub vertices exist, they can be eliminated during triangulation without the need for addition of any fill edges. Hence, the min-fill heuristic always chooses to eliminate such vertices whenever they are available. With respect to the simplicial elimination ordering, we have the following result:

Theorem 4.3. *If a chordal graph $G = \langle V, E \rangle$ contains a hub vertex v, there exists a simplicial elimination ordering for G in which v comes last.*

Proof. We have from Theorem 4.2 that the graph G' induced by $V \setminus \{v\}$ is chordal; therefore, G' has a simplicial vertex w. Now, it holds that w is also a simplicial vertex of G. This result is immediate from the fact that v is a hub. Therefore, we can select w for elimination before we select v. This process can be repeated until only v remains. □

4.1.3 Decomposition

To further aid the process of triangulation, a graph can be decomposed into its biconnected components.

Definition 4.4. *Let $G = \langle V, E \rangle$ be a connected graph; then, a set of components $\mathcal{C} = \{C_1, \ldots, C_k\}$ with each $C_i \subseteq V$ is a* decomposition into biconnected components *if all of the following conditions are met:*

- *for all $C_i \in \mathcal{C}$ and all $\{v, w\} \subseteq C_i$, there exist at least two vertex-disjoint paths between v and w; further,*
- *for all $C_i \in \mathcal{C}$, $v \in C_i$ and $w \notin C_i$, no more than one vertex-disjoint path exists between v and w; and finally,*
- *all edges are contained within a single component; that is, there exist no $C_i \in \mathcal{C}$ and $\{v, w\} \in E$ such that $v \in C_i$ and $w \notin C_i$.*

Proposition 4.4. *For $i \neq j$, we have that $|C_i \cap C_j| \leq 1$. If the intersection is not empty, the vertex it contains is called a* separation vertex, *because deleting it from G would disconnect (or separate) G.*

We then have the following:

Theorem 4.5. *A graph is chordal if and only if all subgraphs induced by its biconnected components are.*

Proof. By Definition 4.4, a cycle cannot span more than a single biconnected component. From this, the result follows. □

Decomposition into biconnected components can be done in $\mathcal{O}(m)$ time by a depth-first search algorithm [Eve79].

4.2 The directed case

In this section, we propose an extension of the concepts discussed so far to directed graphs. To the best of our knowledge, such an extension has not been published before.

4.2.1 Chordality

Our extension of Definition 4.1 of chordality to directed graphs is quite straightforward; we follow the idea of Definition 4.2:

Definition 4.5. *Let $G = \langle V, A \rangle$ be a directed graph (digraph). We can define the following concepts:*

- *If $(v_1, v_2, \ldots, v_k, v_{k+1} = v_1)$ with $k > 3$ is a directed cycle (dicycle), then any arc on two nonadjacent vertices (v_i, v_j) with $1 < |j - i| < k - 1$ is a* chord *of this cycle.*
- *G is* chordal *if every dicycle of length greater than 3 has a chord, the direction of which is unimportant.*

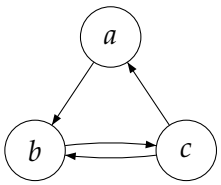


Figure 4.2: A graph with a transitive elimination ordering (a, b, c)

- *A vertex $v \in V$ is* transitive *if there is an arc from all its incoming neighbours to all its outgoing neighbours, i.e.* $\forall u, w \in V : (u, v) \in A \wedge (v, w) \in A \Rightarrow (u, w) \in A$.
- *Let G_i be the subgraph of G induced by $V_i = \{v_1, \ldots, v_i\}$; note that $G_n = G$. A* transitive 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 transitive vertex of the graph G_i.*

We then have the following result:

Theorem 4.6. *A directed graph $G = \langle V, A \rangle$ is chordal if it has a transitive elimination ordering $(v_n, \ldots, v_1)$.*

Proof. Let $(v_n, \ldots, v_1)$ be a transitive elimination ordering of G. To arrive at a contradiction, assume there exists an unchorded dicycle $(v_{i_1}, \ldots, v_{i_k}, v_{i_{k+1}} = v_{i_1})$ with $k > 3$ in G such that $(v_{i_j}, v_{i_{j+1}}) \in A$ for $1 \leq j \leq k$. Let $m = \text{argmax}\{i_1, \ldots, i_k\}$; since v_{i_m} appears in the transitive elimination ordering before any of the other vertices in the cycle, there must exist an arc $(v_{i_{m-1}}, v_{i_{m+1}}) \in A$. This is a chord of the cycle, which contradicts our premise. □

In contrast to the undirected case, the converse of this theorem does not hold. To see this, consider $G = \langle \{a, b, c\}, \{(a, b), (b, c), (c, a)\} \rangle$, i.e. a simple directed cycle of size 3. G is trivially chordal, but it has no transitive elimination ordering.

The following proposition can be verified by the reader and relates the concepts of chordality for undirected and directed graphs:

Proposition 4.7. *A digraph $G = \langle V, A \rangle$ is chordal only if the corresponding undirected graph $G' = \langle V, E \rangle$ (in which $\{v, w\} \in E \Leftrightarrow (v, w) \in A \vee (w, v) \in A$) is also chordal. The converse does not hold in general.*

The maximum cardinality search algorithm cannot be easily adapted to determine directed chordality. A reason for this is that maximum cardinality search can start in any vertex, from which the algorithm constructs an elimination ordering backwards. This works for undirected chordal graphs, for which simplicial elimination orderings can be constructed with any vertex appearing last; however, the same does not hold for directed chordal graphs. In the simple graph in Figure 4.2, vertex a is the only transitive vertex. If the

maximum cardinality search were to start in this vertex, it would fail. However, the minimum-fill heuristic can be readily adapted to the directed case and still identifies a transitive elimination ordering for chordal digraphs, if it exists.

4.2.2 Triangulation

As in the undirected case, if a directed graphs is not chordal, this property can be enforced by adding a set of fill edges. The directed case is analogous to the process described in Section 4.1.2, with the difference that instead of connecting all vertices, elimination of a vertex only connects its inbound neighbours to its outbound neighbours.

Definition 4.6. *Let $G = \langle V, A \rangle$ be a directed graph that is not chordal. A set of arcs T with $T \cap A = \emptyset$ is called a* triangulation *if $G' = \langle V, A \cup T \rangle$ is chordal.*

In addition, the definitions of minimality are analogous to those in Definition 4.3.

The heuristics for the undirected case can of course be applied unchanged to the directed case; preferably, however, they should take the directionality of edges into account. For the minimum fill heuristic, this adaptation is straightforward: for each vertex v, consider two sets of neighbours I_v and O_v, on incoming and outgoing edges respectively, and count how many arcs between all pairs $(v_i \in I_v, v_o \in O_v)$ are not yet present. For the minimum degree heuristic, our suggestion is to use as heuristic value for a vertex the multiplication of its in-degree with its out-degree. This way, sources and sinks (having respectively in-degree and out-degree zero) will be eliminated first, which is desirable because their elimination results in the addition of no fill edges.

We now state the directed counterpart of Theorem 4.2:

Theorem 4.8. *If $G = \langle V, A \rangle$ contains a vertex v that has all other vertices in its neighbourhood, disregarding directionality, i.e. $|\{w \mid (v,w) \in A \vee (w,v) \in A\}| = n - 1$, then G is chordal if and only if its subgraph G' induced by the removal of v is. We will again say that v is a* hub *vertex.*

Proof. We prove the negative statement of this theorem: G contains an unchorded directed cycle of length greater than 3 if and only if G' does.

($\Rightarrow$) Assume that G contains an unchorded dicycle of length greater than 3. If such a cycle does not involve v, the same unchorded cycle exists in G'; we show that the other case cannot occur. If it contains v, it has the form $(v, w_1, \ldots, w_k, v)$ with $k \geq 3$. However, we know that arcs exist between v and all other vertices, so the cycle has a chord, which contradicts our assumption.

($\Leftarrow$) If G' contains an unchorded cycle of length greater than 3, this same unchorded cycle must already have been present in G, because deletion of v only removed edges involving v, which cannot be a chord in this cycle. □

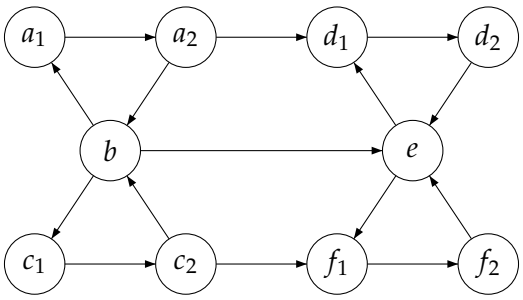


Figure 4.3: Biconnected vs. strongly connected components

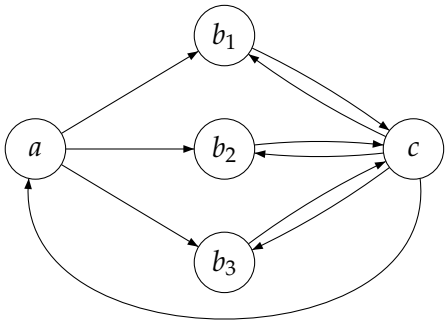


Figure 4.4: Decomposition by directed triangulation

4.2.3 Decomposition

Another property of directed graphs is that they can be decomposed into strongly connected components.

Definition 4.7. *Let $G = \langle V, A \rangle$ be a directed graph; then, a set of components $\mathcal{C} = \{C_1, \ldots, C_k\}$ with each $C_i \subseteq V$ is a* decomposition into strongly connected components *if for each C_i and each pair of vertices $\{v, w\} \subseteq C_i$, there exists a directed path from v to w and a directed path from w to v.*

Proposition 4.9. *Let $\mathcal{C} = \{C_1, \ldots, C_k\}$ be a decomposition into strongly connected components of a directed graph $G = \langle V, A \rangle$. Then, the following properties hold:*

- *$\mathcal{C}$ forms a partition of V; that is, the components are pairwise disjoint, and their union is exactly equal to V.*
- *Any cycle in G lies within a single strongly connected component.*

We now have two criteria for decomposition: biconnectivity and strong connectivity. Interestingly, interleaving these operations in an iterative fashion may result in progressively smaller subproblems which can then be solved independently.

As an example, consider Figure 4.3. Disregarding directionality of the arcs, this graph is biconnected. However, it can be decomposed into two

hourglass-shaped strongly connected components, which in turn can each be decomposed into two biconnected triangles.

Strong connectivity also has implications for the directed triangulation operation. By Theorem 4.5, it is known that in standard (undirected) triangulation, a graph is chordal if and only if all its biconnected components are chordal. For directed triangulation, we now have the following:

Corollary 4.10. *A graph is chordal if and only if all its components that are both biconnected and strongly connected are chordal.*

Proof. By Proposition 4.9, a cycle cannot span more than a single strongly connected component. From this and Theorem 4.5, the result follows. □

However, a difference is that whereas the standard triangulation operation preserves both strong connectivity and biconnectivity, the directed triangulation operation may precipitate a decomposition into biconnected components.

To see this, consider Figure 4.4. This graph is strongly connected, biconnected and chordal, and its transitive elimination ordering starts with vertex a. Elimination of a results in a graph with three biconnected and strongly connected components. This means that graphs can be further decomposed during directed triangulation, while this cannot occur in standard triangulation. When one imagines the vertices b_i to represent subgraphs instead of separate vertices, this small example extends to a much more general case and the importance of checking for strong connectivity and biconnectivity during the triangulation operation becomes clear.

4.3 Discussion

In this section, we recapitulate the matter discussed in the preceding sections and ponder some of its implications to the STP.

We have seen that a natural way to order the vertices in a chordal graph exists in the simplicial elimination ordering. In existing literature (e.g. [Dec03]), it was already known that running a directed path consistency algorithm (DPC, discussed in Chapter 3) along this ordering results in optimal efficiency; indeed, the simplicial ordering has minimal induced width w^* (ibid.). This means that finding a minimal triangulation, as discussed in this chapter, can be equated to finding a vertex ordering with minimal induced width, which was mentioned in the previous chapter; as stated, both problems are NP-complete.

Implications for the DPC algorithm can also be derived from our extension of the concept of chordality to the directed case. Because chordality on directed graphs is more specific than the same concept on undirected graphs, less fill edges may have to be added to instill chordality. In theory, this in turn leads to a smaller induced width of the transitive elimination ordering when compared to the simplicial elimination ordering, and therefore implies improved performance of the DPC algorithm.

For partial path consistency (PPC), the most important insight gained in this chapter is that there is a natural way to order the triangles in a chordal graph, which stems again from the simplicial elimination ordering. Also, for both PPC and DPC, the triangulation heuristics we described in this chapter become useful when dealing with constraint graphs that are not yet chordal.

Finally, the simplicial elimination ordering and the maximum cardinality search algorithm finds application in a new incremental method; this is presented in the next chapter along with the other implications we described, using the matter discussed here as a foundation for the new algorithms we propose.

Chapter 5

New solution methods

In this central chapter of our thesis, we propose several new approaches to the STP. Recalling the discussion in Chapter 3, the tasks performed by STP algorithms can be divided into three classes: (i) determining consistency; (ii) calculating minimal constraints; or (iii) performing either of the preceding tasks incrementally. For each of these classes, we propose a new algorithm, based on the theory presented in the previous chapter. We make theoretical claims on their performance that will be empirically evaluated in the next chapter.

5.1 Directed path consistency

The DPC algorithm was already described in Chapter 3; we now briefly recapitulate some its most important properties. Enforcing DPC can be used to determine consistency of an STP instance. This is a faster method for determining consistency than many other methods we described; however, DPC cannot be used to calculate the minimal network.

Xu and Choueiry compared their new $\triangle$STP approach to DPC and found the former outperformed the latter in most cases. However, in this section we show that using a suitable ordering of the time points, DPC in fact never performs worse than $\triangle$STP and remains an algorithm of choice when only consistency checking is required.

We show this in two steps: first, in Section 5.1.1, we take a conventional approach that was already available in literature; then, in Section 5.1.2, we improve on this approach using our new results on directed triangulation from the previous chapter.

5.1.1 Undirected chordal graphs

From general constraint satisfaction literature (e.g. [Dec03]), it is known that for chordal graphs, a simplicial elimination ordering of the vertices has minimal induced width w^*. Recalling from Chapter 3 that the time complexity of the DPC algorithm is $\mathcal{O}(n \cdot (w^*)^2)$, it follows that DPC is most efficient when run along this ordering.

In this section, we show that this ordering considers every triangle in the graph exactly once, and updates at most a single constraint in the triangle. In contrast, the △STP algorithm may update all three constraints in each triangle processed, and more importantly, it processes every triangle at least once, with equality only if the network was already minimal. Finally, recalling from Chapter 4 that the maximum cardinality search algorithm can be used to identify a simplicial elimination ordering in time linear in the amount of edges, and noting that the amount of triangles is superlinear in the amount of edges for nontrivial graphs, we can conclude that DPC outperforms △STP for chordal graphs.

This result can be extended to the general case by performing triangulation, which was already required for △STP. Since this step is identical for both algorithms, it follows that DPC consistently outperforms △STP in all cases. We note that the simplicial elimination ordering is a useful byproduct of the triangulation step, which means that running a maximum cardinality search is no longer necessary.

We now state the main results of this section.

Lemma 5.1. *STP consistency can be determined by running a directed path consistency (DPC) algorithm along a simplicial elimination ordering $(x_1, x_2, \ldots, x_n)$ of its graph representation without introducing any new constraint edges.*

Proof. In iteration k, the DPC algorithm considers the constraints between time point x_k and all its neighbours x_j with $j > k$. Because of the simplicial elimination ordering, these time points x_j induce a clique. Therefore, any constraint edge that could be added is already present. □

Theorem 5.2. *When run along a simplicial elimination ordering as in Lemma 5.1, the DPC algorithm considers every triangle in the graph representation of an STP instance exactly once.*

Proof. From Lemma 5.1, we know that no edges are added to the constraint graph; therefore, it suffices to show that all existing triangles are considered exactly once. Let $T = \{x_i, x_j, x_k\}$ be a triangle in the graph representation of an STP, and without loss of generality assume $i > j > k$. Then, T is considered exactly once, in the kth iteration of the DPC algorithm. □

5.1.2 Directed chordal graphs

We can further improve upon the result from the previous section by showing that for directed path consistency, it suffices to enforce directed chordality, which we defined in Chapter 4. The DPC algorithm can then be run along a transitive elimination ordering of the directed graph representation.

Theorem 5.3. *Let $\mathcal{S}$ be an STP instance with a directed graph representation. Then, running the directed path consistency (DPC) algorithm along a transitive elimination ordering $(x_1, x_2, \ldots, x_n)$ does not introduce any new constraint edges.*

Proof. In iteration k, the DPC algorithm considers x_k and all pairs of its neighbours $\{x_i, x_j\}$ with $i, j > k$. If there is a path $x_i \to x_k \to x_j$ (or $x_j \to x_k \to x_i$), it follows from the monotonic transitivity of x_k that there exists a constraint edge (x_i, x_j) (or (x_j, x_i), respectively), so the theorem holds in this case. Now consider a pair of neighbours $\{x_i, x_j\}$ for which such a path does not exist. Then, either $w_{i\to k} = w_{j\to k} = \infty$ or $w_{k\to i} = w_{k\to j} = \infty$, and the weight of the constraints $c_{i\to j}$ and $c_{j\to i}$ is left unchanged by the algorithm; in particular, this means that if there was no edge (i, j) or (j, i), it is not introduced. □

As we stated in Proposition 4.4, directed chordality is a weaker property than undirected chordality. It follows that less fill edges may be required for directed triangulation, resulting in a smaller induced width w^*. This means that DPC when run along a transitive elimination ordering will consistently outperform the same algorithm run along a simplicial elimination ordering.

5.2 Improved implementation of partial path consistency

In this section, we propose a new algorithm that yields major improvement in the efficiency of the partial path consistency algorithm for the STP. In the △STP algorithm, a queue of triangles is maintained; triangles are added to the queue if one of their edges' weights is updated and they are not already present in it. This means that in practice, a triangle may be processed many times. In this section, we show that there exist STP instances for which the △STP exhibits pathological behaviour; then, we propose a new algorithm that always remains well-behaved.

5.2.1 Pathological behaviour of △STP

We designed a class $\mathcal{P}$ of STPs with very simple structure for which the total amount of triangles visited by △STP may be quadratic in the amount of triangles itself. These STP instances are defined on a constraint graph consisting of a single directed cycle with all zero weight edges; this cycle is filled in with edges having carefully selected weights.

In Figure 5.1, we depict an instance of this class. This example consists of six triangles, labelled a through f for ease of reference; by repeating the pattern, the graph can be extended to an arbitrary amount of triangles. None of the constraints are initially minimal, except for the ones represented by the arcs in the zero-weight cycle; because there exists a zero-weight path between every pair of vertices, each constraint is minimal if and only if its weight is zero. Further, it is the case that every triangle, considered separately, contains three edges whose weight can be adjusted; the reader can verify this.

Theorem 5.4. *When solving STP instances from $\mathcal{P}$ consisting of t triangles, the* △STP *algorithm may require processing $\Omega(t^2)$ triangles.*

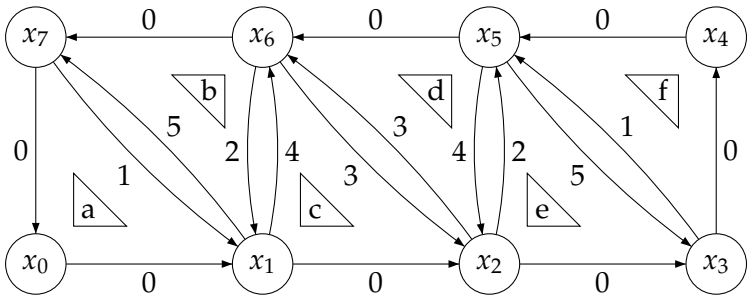


Figure 5.1: Pathological test case for △STP

Proof. We show how the STP instance from Figure 5.1 is handled by △STP; the result readily transfers to the general case. Assume that the initial queue of triangles is $Q = (a, b, c, d, e, f)$. First, triangle a is processed, and three constraints* have their weight adjusted: $w_{0\to7} \leftarrow 5$, $w_{7\to1} \leftarrow 0$ and $w_{1\to0} \leftarrow 5$. Triangle b would be added to Q, but is already contained therein. Now, triangle b is processed; again, three edges are adjusted, and triangle a is appended to the queue. This process is repeated until triangle f has been processed; at this point, all edges making up f have minimal weights, and we have that $Q = (a, b, c, d, e)$. The algorithm starts again at triangle a and proceeds to triangle e, after which $Q = (a, b, c, d)$. By now, the pattern is clear: the total number of triangles processed is $6 + 5 + \cdots + 1 = 21$. In general, for a graph on t triangles of this type, the number of triangles processed is $t(t+1)/2$, which is indeed quadratic in t. □

5.2.2 P^{3}C: our new PPC algorithm

In this section, we propose a new algorithm that enforces partial path consistency (PPC) by processing every triangle exactly twice. This means that to within a constant factor, its performance equals that of directed path consistency on an undirected chordal graph. To achieve this result, we need to know the simplicial elimination ordering, which as we stated in the previous chapter is a byproduct of triangulation.

Our new algorithm is called P^{3}C and is presented in Figure 5.2.† We can now state our main result for this section.

Theorem 5.5. *Algorithm* P^{3}C *achieves partial path consistency on consistent chordal STNs by processing every triangle in the graph exactly twice, i.e., with time complexity* $\Theta(n \cdot (w^*)^2)$. *If the instance is inconsistent, this is discovered in time* $\mathcal{O}(n \cdot (w^*)^2)$.

*Note that two of these constraints were not depicted in Figure 5.1; as usual, these are assumed to have infinite weight.

†P^{3}C (i.e. PPPC in full) could stand for Planken's PPC or Power-PPC; the author is open to any other suggestions.

Input: An STP instance $\mathcal{S} = \langle X, C \rangle$ with a simplicial elimination ordering $(x_1, x_2, \ldots, x_n)$
Output: The PPC network of $\mathcal{S}$ or INCONSISTENT

```
1  for k ← 1 to n do
2      forall i > j > k such that {i,k}, {j,k} ∈ E do
3          w_{i→j} ← min(w_{i→j}, w_{i→k} + w_{k→j})
4          w_{j→i} ← min(w_{j→i}, w_{j→k} + w_{k→i})
5          if w_{i→j} + w_{j→i} < 0 then return INCONSISTENT
6      end
7  end
8  for k ← n to 1 do
9      forall i > j > k such that {i,k}, {j,k} ∈ E do
10         w_{i→k} ← min(w_{i→k}, w_{i→j} + w_{j→k})
11         w_{k→i} ← min(w_{k→i}, w_{k→j} + w_{j→i})
12         w_{j→k} ← min(w_{j→k}, w_{j→i} + w_{i→k})
13         w_{k→j} ← min(w_{k→j}, w_{k→i} + w_{i→j})
14     end
15 end
16 return CONSISTENT
```

Figure 5.2: The P^{3}C algorithm

Proof. We first prove the time complexity, which is the easier part of our claim. The first main loop (lines 1–7) of the algorithm is just the directed path consistency algorithm; if the problem is inconsistent, this is discovered at some point during this loop. After this first leg, every triangle has been visited exactly once. The second main loop (lines 8–15) then follows the same ordering backwards and again visits every triangle exactly once. From these observations, our claims on time complexity easily follow.

To show that the algorithm is sound, we first note that after the first main loop, as a property of directed path consistency, every constraint $c_{i\to j}$ that is represented by an edge $\{x_i, x_j\}$ has been updated to the length of the shortest path from x_i to x_j in the graph induced by $\{x_k \in X \mid k < \min(i,j)\} \cup \{x_i, x_j\}$; this implies in particular that the constraints between x_{n-1} and x_n (in both directions) are minimal.

We illustrate this property of directed path consistency in Figure 5.3. In this figure, we depict a path consisting of 10 vertices. Each vertex is labelled with its position in the simplicial elimination ordering, which is also reflected in its vertical position. As a result of the simplicial elimination ordering, the edges represented by dashed lines are guaranteed to be present in the constraint graph. Assuming that the path depicted by solid lines is the shortest path (with least total weight) from x_7 to x_8, the reader can now verify that

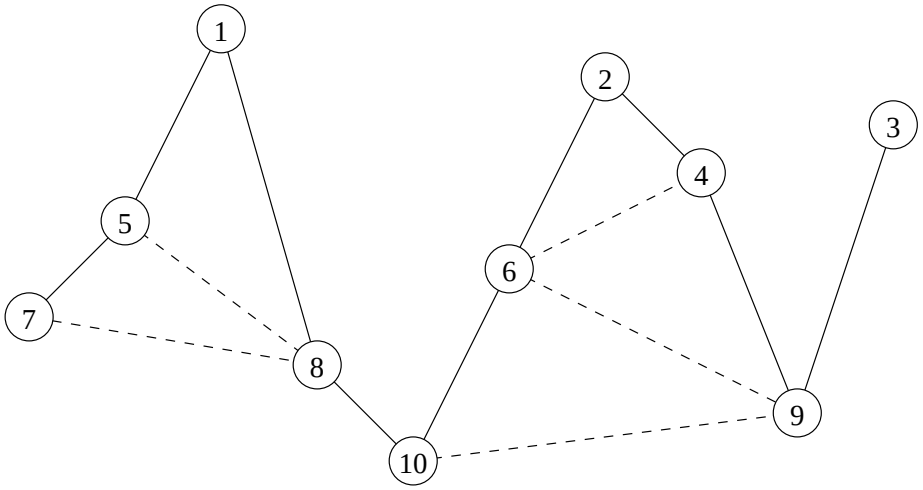


Figure 5.3: The DPC property

after the first main loop, each dashed edge has had its weight updated and can serve as a shortcut; in particular, $(x_7 \to x_8 \to x_{10} \to x_9 \to x_3)$ is now also a shortest path from x_7 to x_3. This is exactly the DPC property mentioned above.

It can now be shown by induction that after iteration k of the second main loop, all constraints in the graph induced by $\{x_i \in X \mid i \geq k\}$ are minimal. The base case for $k \geq n-1$ has already been proven; assuming that the proposition holds for $k+1$, we now show that the proposition holds for k. Consider any constraint $c_{k \to i}$ with $i > k$, and to arrive at a contradiction, assume that this constraint is not minimal after the kth iteration; i.e. after the iteration completes, there still exists some path $\pi = (x_k \to x_{j_1} \to \cdots \to x_{j_l} \to x_i)$ with total weight $w_\pi < w_{k \to i}$. We now show that this cannot occur.

By the DPC property stated above, if there appears any x_j in π with $j < k < i$, there must exist another shortest path π where each x_j has $j > k$. Therefore, we can safely assume that π satisfies this condition, and except for its first edge π lies entirely within the graph induced by $\{x_j \in X \mid j > k\}$. Now, by the induction hypothesis, $c_{j_1 \to i}$ is minimal; therefore, the shortest path can be further reduced to $\pi = (x_k \to x_{j_1} \to x_i)$. Note that because x_k appears in the simplicial elimination ordering before both x_{j_1} and x_i, the edge $\{x_k, x_i\}$ must exists. But then, we have that $w_{k \to i} \leq w_\pi = w_{k \to j_1} + w_{j_1 \to i}$ by the operations performed in the kth iteration, which contradicts our assumption. □

Having shown that the efficiency of enforcing PPC can be improved to be proportional to that of DPC, one may wonder whether the results from the previous section are also applicable to PPC. That is, the STP instance is represented as a directed graph and directed chordality is enforced on it, after which a PPC algorithm is run; our algorithm would then follow a

transitive elimination ordering instead of a simplicial elimination ordering. If the transitive elimination ordering happens to also be simplicial, which can only be the case for undirected chordal graphs, the result is, of course, identical. Otherwise, in the proof of Theorem 5.5, only the DPC property changes, because directionality of the edges becomes important. We now state a sufficient condition for minimality of constraints.

Theorem 5.6. *Let $G = \langle V, A \rangle$ be a directed chordal graph representation of an STP instance with a transitive elimination ordering $(x_1, x_2, \ldots, x_n)$. If $\pi = (x_i \to x_{j_1} \to \cdots \to x_{j_l} \to x_k)$ is the shortest path from x_1 to x_k with weight w_π, $\mathrm{P^3C}$, when run along this ordering, updates the weight $w_{i \to k}$ of (x_i, x_k) to w_π if $(x_k, x_1) \in A$.*

Proof. There is a directed cycle which consists of π joined with the edge (x_k, x_i). With regard to this cycle, directed chordality is equivalent to undirected chordality, and the result follows from Theorem 5.5. □

5.3 Incremental partial path consistency

Our final contribution in this chapter is a new incremental algorithm in the vein of IDPC and IFPC, which were presented in Chapter 3. The IPPC algorithm that is presented in this section incrementally enforces partial path-consistency on an STP instance, processing a single constraint at a time. In our approach, we assume that the new constraint added preserves chordality of the constraint graph; in practice, this may be guaranteed in several ways. If it is known on beforehand which constraints may appear and which constraints will not, the graph representing all time points and all conceivable constraint edges can be triangulated once. Alternatively, it is conceivable that chordality itself be enforced incrementally before running the incremental PPC algorithm. This matter is subject to further research and as such outside the scope of this text.

Our new incremental method is presented in Figure 5.4. Like IFPC, this algorithm has the desirable property that any inconsistency caused by the new constraint is detected in constant time; this follows from the fact that every constraint in a partially path-consistent STP instance is minimal.

The main loop is in effect similar to the second main loop from the $\mathrm{P^3C}$ algorithm in Figure 5.2; it enforces PPC in the reverse order of a simplicial elimination ordering. The algorithm maintains the set D of constraints that have been updated; in this way, during the main loop only the necessary checks are performed. Fixing a simplicial elimination ordering (line 4) in which the newly added constraint appears last can be easily done in linear time by maximum cardinality search, as discussed in Chapter 4.

The worst-case performance of this algorithm is no better than the "single-shot" PPC algorithm and remains $\mathcal{O}(n \cdot (w^*)^2)$. This suggests that IFPC remains the incremental method of choice for dense graphs, though the actual performance of IPPC may be better in practical cases. For sparse graphs, i.e.

Input: A partially path-consistent STP $\mathcal{S} = \langle X, C \rangle$ and a new constraint $c'_{a \to b}$.
Output: CONSISTENT if $c'_{a \to b}$ has been added to $\mathcal{S}$, which is again partially path-consistent; INCONSISTENT otherwise.

```
1  if w'_{a→b} + w_{b→a} < 0 then return INCONSISTENT
2  if w'_{a→b} ≥ w_{a→b} then return CONSISTENT
3  w_{a→b} ← w'_{a→b}
4  Fix a simplicial elimination ordering (x_1, x_2, ..., x_n) of S
   such that x_a = x_{n-1} ∧ x_b = x_n
5  D ← {(n − 1, n)}
6  for k ← n to 1 do
7      forall i, j > k, i ≠ j such that {i,k}, {j,k} ∈ E ∧ (i,j) ∈ D do
8          if w_{i→k} > w_{i→j} + w_{j→k} then
9              w_{i→k} ← w_{i→j} + w_{j→k}
10             D ← D ∪ {(i,k)}
11         end
12         if w_{k→j} > w_{k→i} + w_{i→j} then
13             w_{k→j} ← w_{k→i} + w_{i→j}
14             D ← D ∪ {(k,j)}
15         end
16     end
17 end
18 return CONSISTENT
```

Figure 5.4: Incremental partial path consistency algorithm (IPPC)

if $w^* \in o(\sqrt{n})$, the worst-case analysis tips in favour of IPPC. In the next chapter, we empirically evaluate the actual performance of IPPC against IFPC for graphs of different densities.

5.4 Summary

In this chapter, we presented several new approaches for tackling the STP, based on the graph theory discussed in Chapter 4. For determining consistency, we showed that running the directed path consistency (DPC) algorithm along a simplicial elimination ordering (discussed in the previous chapter) results in the best theoretical performance, with lowest induced width w^* (see Section 3.1.2).

We indicated that for the state-of-the-art $\triangle$STP algorithm, pathological problem instances can be designed, and proposed a new algorithm, P^{3}C, that does not have this disadvantage. Indeed, from the fact that P^{3}C considers each triangle in the constraint graph at most twice, whereas $\triangle$STP offers no

such guarantee, we expect our new algorithm to show better performance even in general cases.

Finally, in a logical continuation of earlier incremental methods, we proposed a new algorithm that incrementally enforces the partial path consistency property. However, the worst-case complexity analysis of this algorithm is worse than that of its main competitor, incremental full path consistency.

In the next chapter, we will put our new algorithms to the test against their precursors and empirically determine their actual performance, thus enabling us to verify the theoretical claims made in this chapter.

Chapter 6

Evaluation of new techniques

In the previous chapter, we proposed several new algorithms and gave a theoretical worst-case analysis of their performance. In this chapter, we supplement this theoretical analysis with a practical evaluation. We compared each of our new methods to all precursors in the same category that were described in Chapter 3. All algorithms were implemented in Java and were run on many different test cases of different structures and sizes.

The structure of this chapter is as follows. First, we describe the test cases we considered and explain our rationale behind our selection; then, we present and analyse the empirical results we recorded, grouped by algorithm category.

6.1 Test cases

In this section, we describe the test cases we considered for our algorithms. They can be split into three classes:

1. STP instances taken from a benchmark set;
2. randomly generated STP instances with a power law degree distribution; and
3. a specifically designed pathological test case for $\triangle$STP.

The latter of these was already discussed in the previous chapter and requires no further introduction; we discuss the others below.

6.1.1 Benchmark sets

We included three benchmark sets for SMT* solvers from SMT-LIB [RT03]. This is a collection of benchmark instances with the goal to facilitate the evaluation and comparison of these solvers and to advance the state of the art

*The acronym SMT stands for "satisfiability modulo theories"; SMT problems take the general shape of a Boolean satisfiability (SAT) problem, with each propositional literal replaced by some predicate.

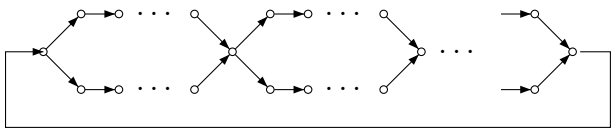


Figure 6.1: The general shape of diamond benchmark instances

in these fields. Each problem instance in these sets consists of a conjunction of clauses; in turn, each clause represents a disjunction of linear inequalities ("literals"). To solve a problem instance, an SMT solver must select for each clause a literal that must be satisfied. Such a selection is called an "instantiation"; for the benchmark sets we selected, each instantiation corresponds exactly to an STP instance. To solve an SMT problem instance, a common approach is to use a backtracking search, solving an STP instance at every step.

Our approach was to build up the constraint graph edgewise by randomly selecting a single inequality from each clause; then, we set the edge weights in such a way that the resulting STP instance was guaranteed to be consistent. We tested the following types of benchmarks:

- *DTP* benchmarks, randomly generated and used for testing solvers of the Disjunctive Temporal Problem (a special type of SMT instance);
- *Job shop* benchmarks, which represent practical scheduling problems and are well known in the planning literature (e.g. [CP89]); and
- *Diamonds* benchmarks, which are defined on very sparse graphs, the general shape of which is shown in Figure 6.1.

6.1.2 Scale-free random graphs

Scale-free graphs can be used to accurately model many real-world networks, in which many vertices have relatively small degree (number of neighbours) and a few vertices have very high degree. For example, in the Internet, many routers have relatively few connections while a few routers in the Internet backbone have very many connections. More formally, the degrees of the vertices in a scale-free graph satisfy a *power law*; that is, for some constant exponent γ, the probability $P(k)$ of a vertex having degree k is proportional to $k^{-\gamma}$.

The Barabási–Albert method [AB02] can be used to randomly generate scale-free graphs. This method starts with some relatively small seed graph on n_0 vertices, which is then evolved to a graph on n vertices by iteratively adding the remaining $n - n_0$ new vertices; each new vertex v_{new} is connected to $m \leq n_0$ already existing ones in such a way that the probability of connecting v_{new} to a vertex v_i is directly proportional to the latter's degree. This way, the evolution follows the "rich get richer" scheme: vertices that have a

type	#cases	n	m	d
DTP	60	35	156–230	12–20
diamonds	36	51–379	53–379	4
job shop	120	5–241	9–3960	4–240
scale-free	539	10–150	18–3857	4–97
pathological	40	3–350	3–1044	4

Table 6.1: Test data statistics

high degree are likely to get even more neighbours, whereas those with low degree have high probability to remain "poor".

Using this method, we generated test cases of this type for a broad sample of values for both n and m, with n varying from 10 to 150 and m varying from 2 to $\lfloor 0.9n/2 \rfloor$; we always set $n_0 = 2m$ and ensured that the seed graph was connected.

6.1.3 General form of test cases

For each test case, we made sure that the STP instance was consistent, to guarantee the mutual comparability of our test results. If we had considered inconsistent instances, the point during search at which a negative cycle was discovered would almost certainly have varied greatly between test cases and algorithms. We submit that it would be interesting to evaluate the behaviour of the algorithms on inconsistent test cases; however, for now we leave this issue as a subject for future research.

The properties of the test cases are summarised in Table 6.1, and represented pictorially in Figures 6.2 and 6.3. The table lists the amount of test cases for each type included, and the range of the number of vertices n, edges m and the graph degree d (i.e. the maximum vertex degree). Figure 6.2 displays the location of each test case in the vertices/edges plane; the line labelled "complete graph" demarcates the maximum possible amount of edges. Figure 6.3 displays the graph degree of three test cases plotted against the amount of vertices. A graph on n vertices can have at most degree $n - 1$, which is exactly the case for the job shop instances. Both the diamonds and pathological test cases have a constant graph degree of 4, and are therefore not included in this figure.

When including results of our tests, we generally set out the number of vertices on the horizontal axis; exceptions apply for the DTP test case and the scale-free graphs. For the former, all test cases have the same number of vertices, $n = 35$; therefore, we depict the number of edges on the horizontal axis instead. In the latter case, we found it in most cases more enlightening to consider a constant number of vertices and instead depict the value of the m parameter for the Barabási–Albert method (described above, in Section 6.1.2) on the horizontal axis.

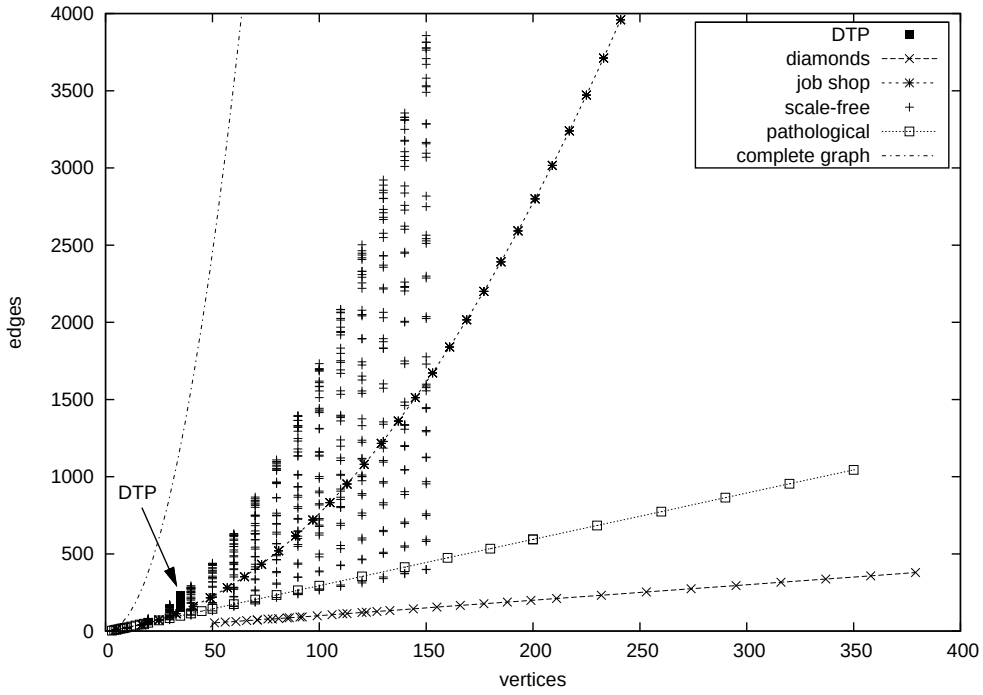


Figure 6.2: Location of test cases (vertices vs. edges)

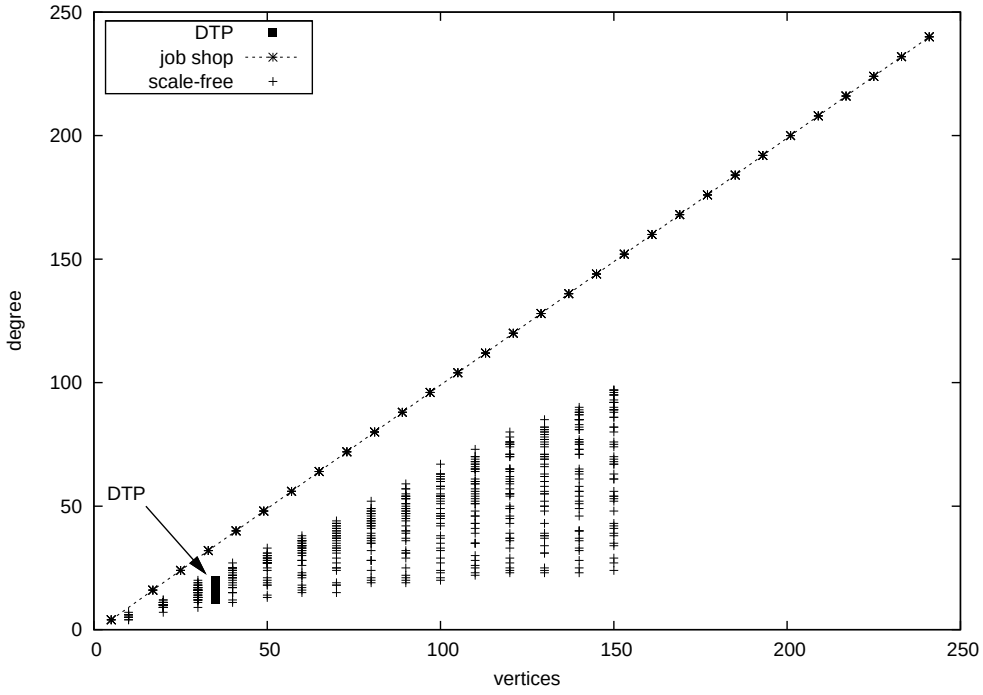


Figure 6.3: Location of test cases (vertices vs. graph degree)

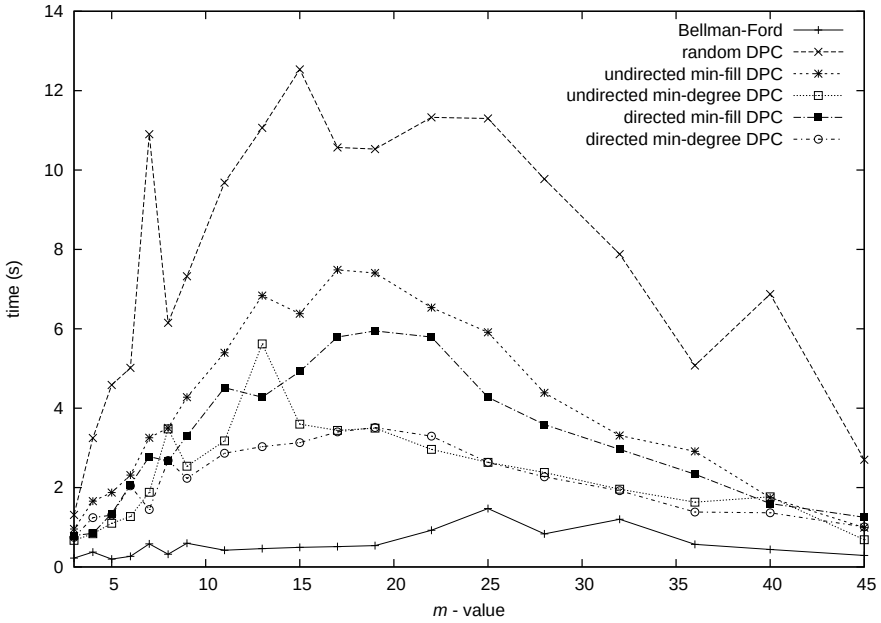


Figure 6.4: Consistency checking for scale-free graphs, $n = 100$

6.2 Consistency checking

We now discuss the result of applying consistency checking algorithms to our test cases. The tested algorithms include Bellman's and Ford's algorithm and several varieties of directed path consistency (DPC).

The first test results we describe are those on the scale-free graphs on 100 vertices, included in Figure 6.4. One of the first properties of this figure that meets the eye is the conspicuous bulge around $m = 18$ in all curves for the DPC algorithms. A likely explanation for this behaviour is that for low values of m, many vertices have so low degree that only few fill edges are added by the DPC algorithms, which means that the heuristic values need to be updated relatively infrequently. For high values of m, the graphs are dense enough that many neighbours of vertices are already connected, resulting again in the addition of few extra edges and infrequent updates of heuristic values. For the values in between, however, many vertices have a moderate amount of neighbours; but between these, fill edges must often still be added.

All these considerations do not apply to Bellman's and Ford's algorithm, which consistently outperforms the others. Among the DPC algorithms, we can note that the random ordering performs worst. Of the heuristics, the minimum fill heuristic, though theoretically producing orderings with smaller induced width w^* (see Section 3.1.2) than the minimum degree heuristic, does not yield large enough differences to justify its higher time complexity. The differences between the directed and undirected heuristics are smaller; for the minimum fill heuristic the directed version seems to have a clear edge, whereas for the minimum degree heuristic, we cannot really draw a con-

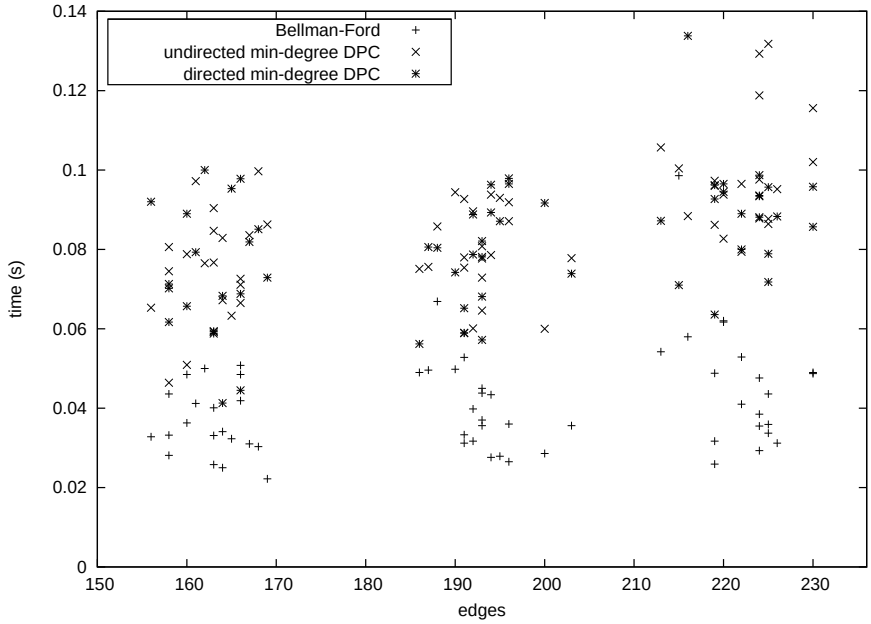


Figure 6.5: Consistency checking for DTP ($n = 35$)

clusion yet. From the results on scale-free graphs, in the interest of clarity of presentation, we only considered Bellman's and Ford's algorithm and the two minimum degree heuristics in the remainder of our tests.

The results of the benchmark tests are depicted in Figures 6.5 through 6.8.* In all cases, Bellman's and Ford's algorithm remains superior, though the results are much closer in the diamonds benchmark (Figure 6.8) than in the other cases. We venture to suggest that this is due to the very low and evenly-spread density of the diamonds benchmark; however, we relegate deeper analysis to future research. The "jumpiness" of the results for the diamonds test case may be partially explained by the relative simplicity of these problems, and their resulting sensitivity to perturbations in the testing environment; note that the time scale of the diamonds plot is two orders of magnitude smaller than that of Figure 6.7.

In Figure 6.5, a clustering of test results is visible. Recalling that the DTP test cases always contain 35 vertices, we may conclude that the number of edges in an STP instance is of far less influence on the performance of the algorithms than the amount of vertices.

The job-shop instances (Figures 6.6 and 6.7) were the hardest to solve; for these test cases, the difference between the performance of Bellman's and Ford's algorithm grew to almost an order of magnitude.

Concluding, we can safely state that Bellman's and Ford's algorithm remains the method of choice for determining consistency of STP instances.

*Note that we split the job shop results into small and large instances; between these two charts, the scale of the vertical axis multiplies by a factor of nearly 60.

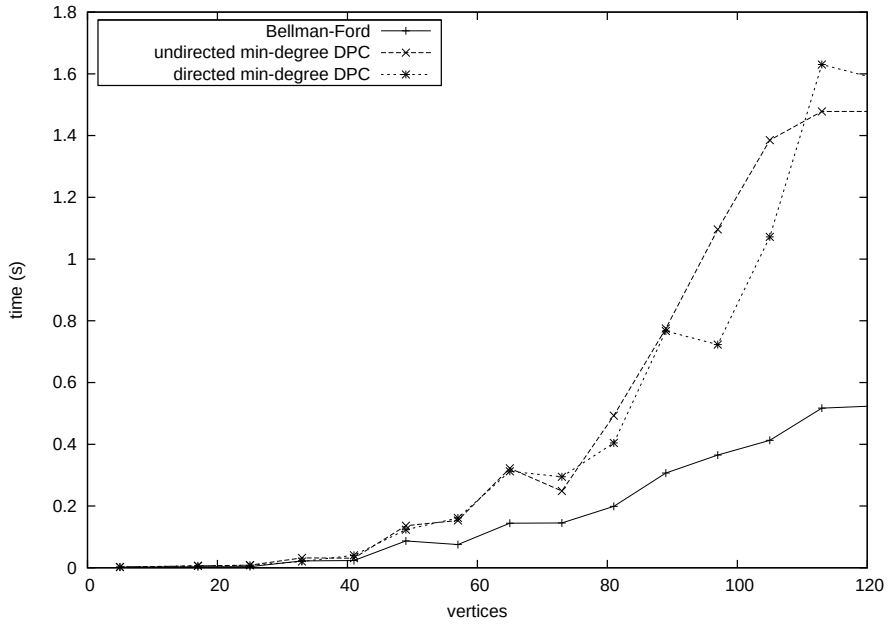


Figure 6.6: Consistency checking for job shop (small instances)

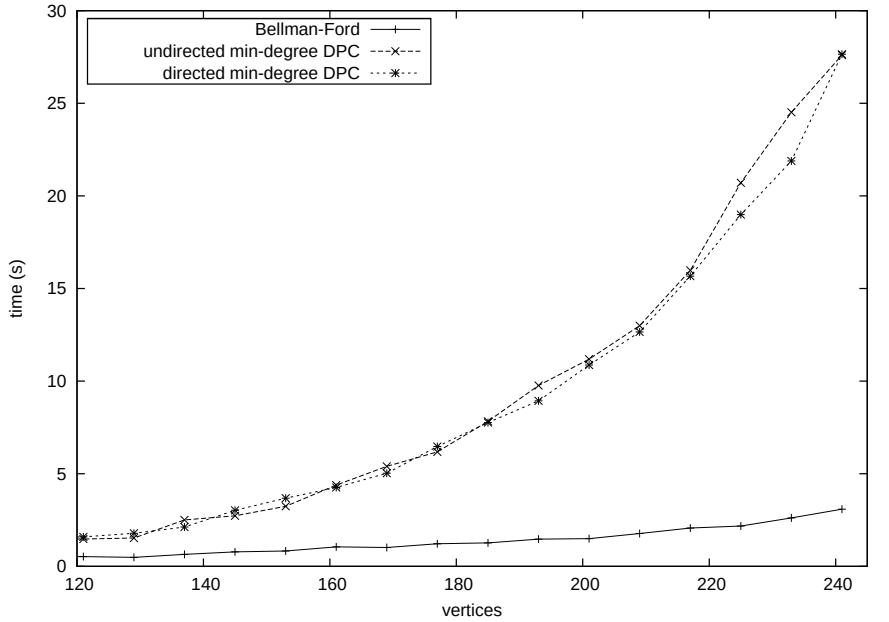


Figure 6.7: Consistency checking for job shop (large instances)

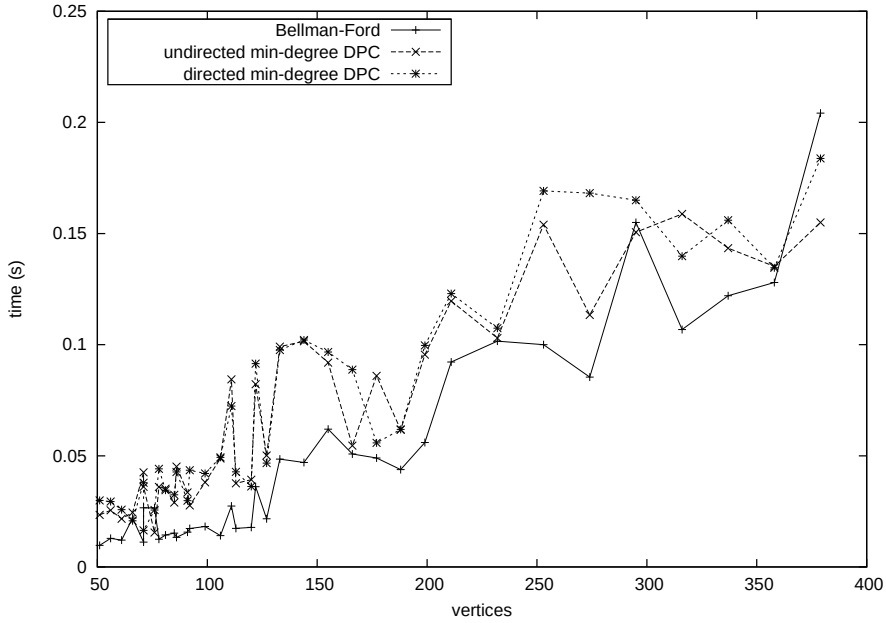


Figure 6.8: Consistency checking for diamonds benchmark

The directed and undirected graphs perform almost identically; from these results, we can discern no clear winner.

6.3 Enforcing partial path consistency

In this section, we turn to the analysis of algorithms that enforce partial path consistency (PPC), viz. △STP and our improvement P^{3}C. Solving the all-pairs shortest paths problem is equivalent to enforcing full path consistency on the STP and can thus be considered a special case of enforcing PPC; therefore, Floyd's and Warshall's algorithm and Johnson's algorithm were also included in our tests.

As in the previous section, we start with an analysis of the algorithms' behaviour on scale-free random graphs, depicted in Figure 6.9. The "bulge" that was noted in the previous section reappears, at about the same relative location; the explanation we gave for it above still applies. This time, the all-pairs shortest paths algorithms seem relatively impervious to it, though in contrast to Bellman's and Ford's algorithm, they perform worse than the other techniques. From these results, it also becomes clear that P^{3}C outperforms △STP nearly everywhere, and that the minimum-degree heuristic yields better performance than minimum-fill.

Turning to the results for the DTP instances (Figure 6.10), we again note the clustering of results, indicating that also for enforcing PPC, the number of edges in the graph is not the primary factor that influences performance. Otherwise, the same general conclusions can be derived that we stated above for

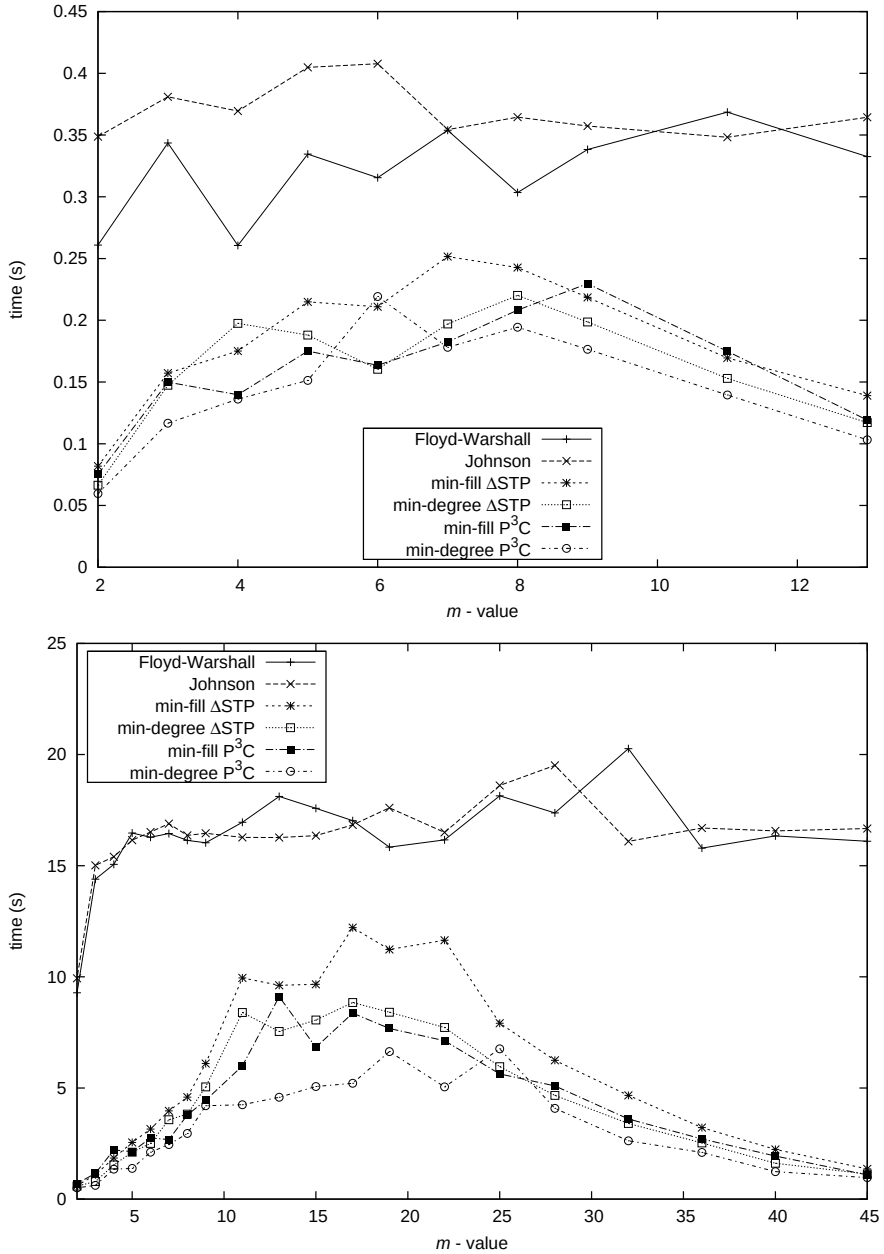


Figure 6.9: Enforcing PPC on scale-free graphs
$n = 30$ (top) vs. $n = 100$ (bottom)

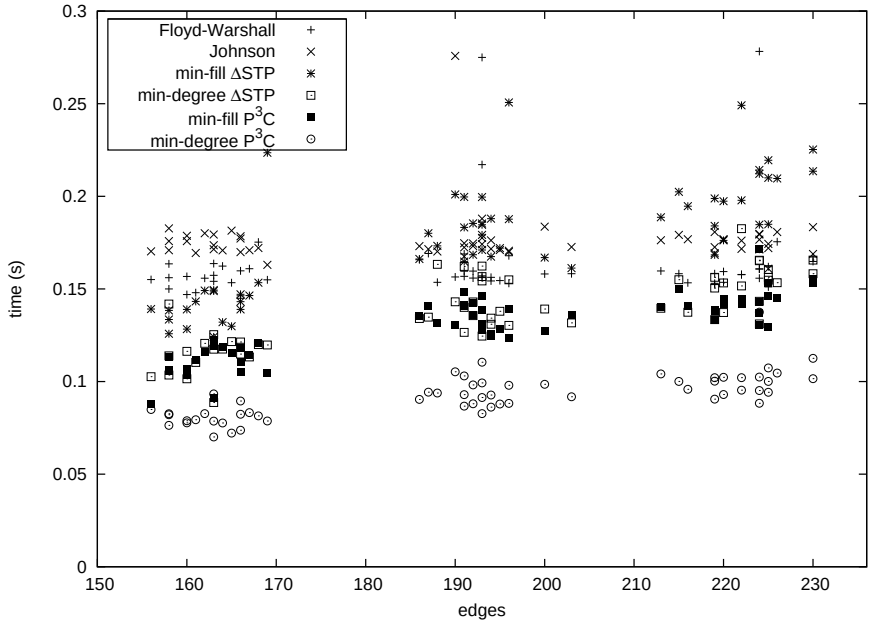


Figure 6.10: Enforcing PPC for DTP ($n = 35$)

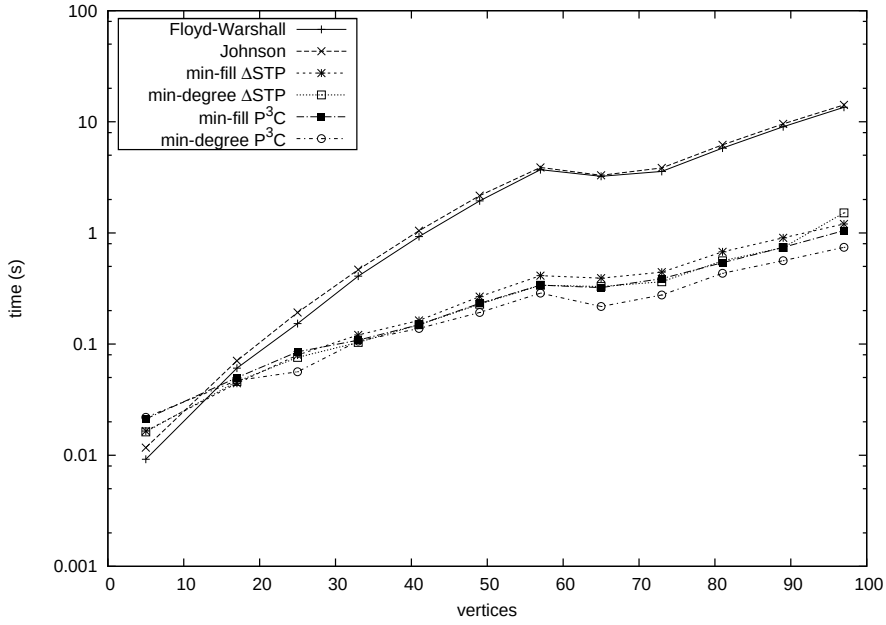


Figure 6.11: Enforcing PPC for job shop with $n < 100$ (log scale)

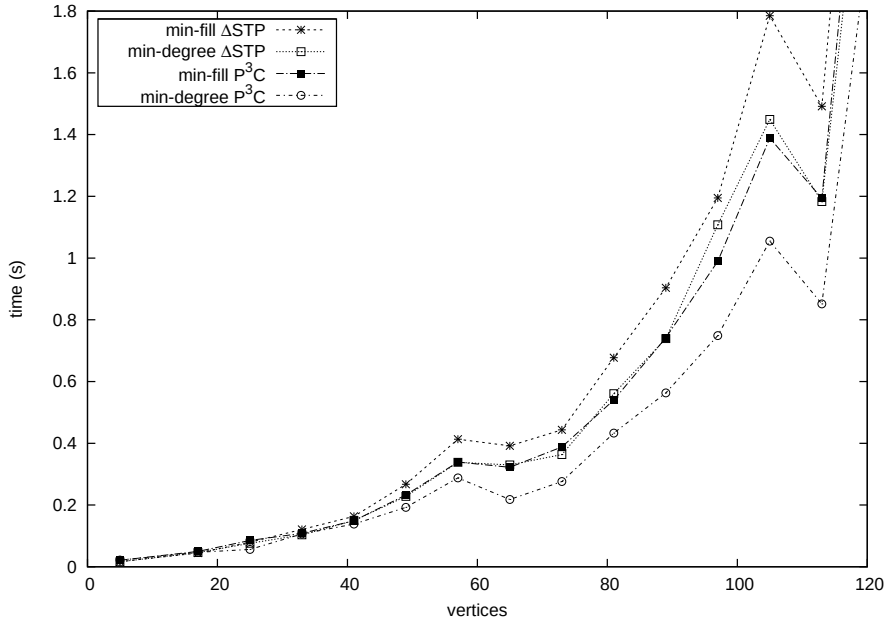


Figure 6.12: Enforcing PPC for job shop (small instances)

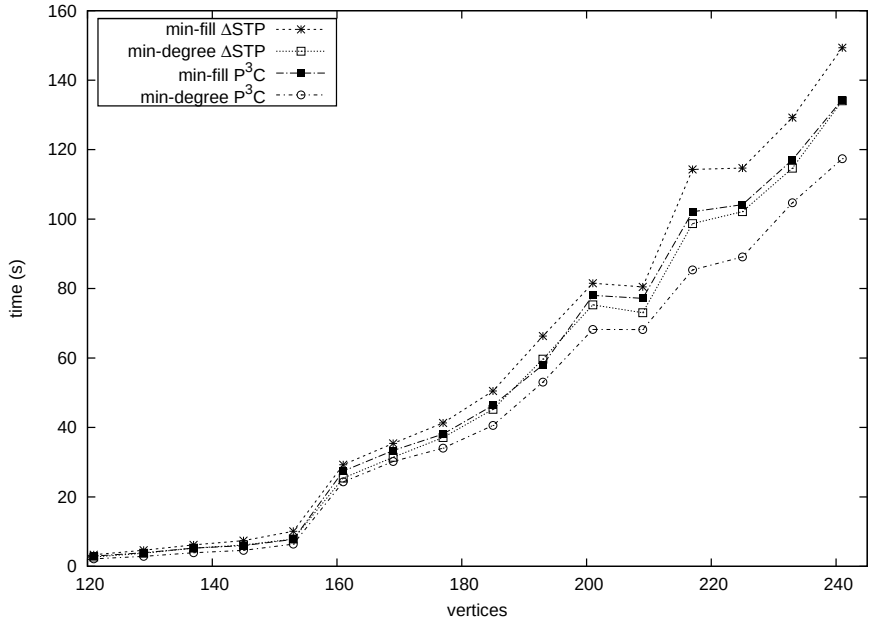


Figure 6.13: Enforcing PPC for job shop (large instances)

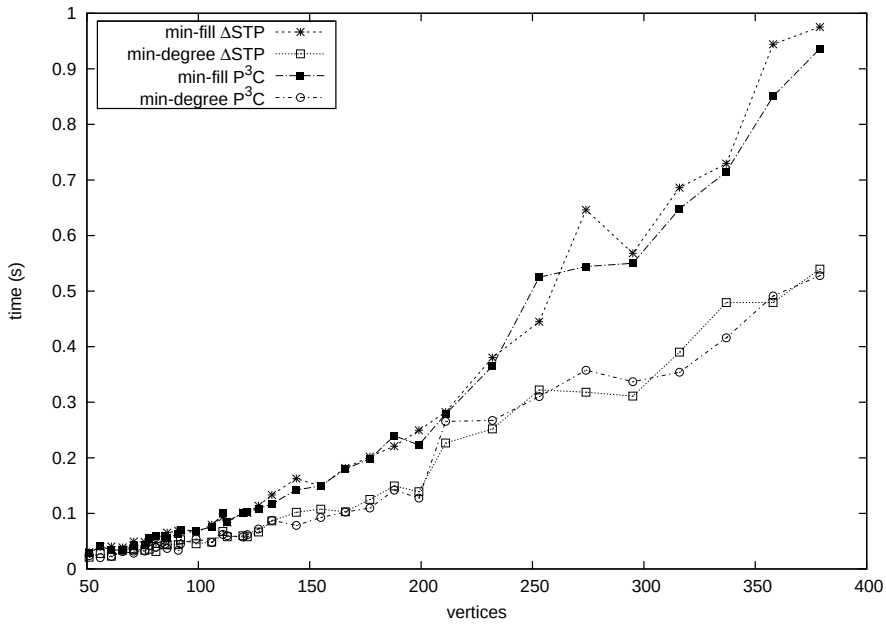


Figure 6.14: Enforcing PPC for diamonds benchmark

the scale-free graphs; the same goes for the job shop problems (Figures 6.11 through 6.13), where we omitted the all-pairs shortest paths algorithms for large instances.

With regard to the results for the diamonds instance (Figure 6.14), it may be noted that $\triangle$STP and P^{3}C exhibit comparable performance, while the difference between heuristics persists. The explanation for this again lies with the extreme sparseness of the graphs.

Finally, we discuss the behaviour of $\triangle$STP and P^{3}C on the pathological instance proposed in Section 5.2.1; for comparison, we also included Floyd's and Warshall's algorithm in the tests. The results are included in Figure 6.15. Since the constraint graphs of these STP instances are already triangulated, we omit the analysis of the triangulation heuristic. As expected, the difference in performance between P^{3}C and $\triangle$STP grows linearly with the size of the input; the difference between P^{3}C and Floyd's and Warshall's algorithm grows even faster. This is consistent with the theoretical worst-case time complexities, which for this input are linear, quadratic and cubic, respectively. The actual time spent for the instance with 350 vertices is 1.36 seconds for $\triangle$STP versus 0.15 seconds for P^{3}C; this differs by almost an order of magnitude.

6.4 Incremental solving

The third new algorithm we proposed in the previous chapter was incremental partial path consistency (IPPC). We mentioned in Section 3.4 that this method requires the constraint graph to be chordal, and that this could be en-

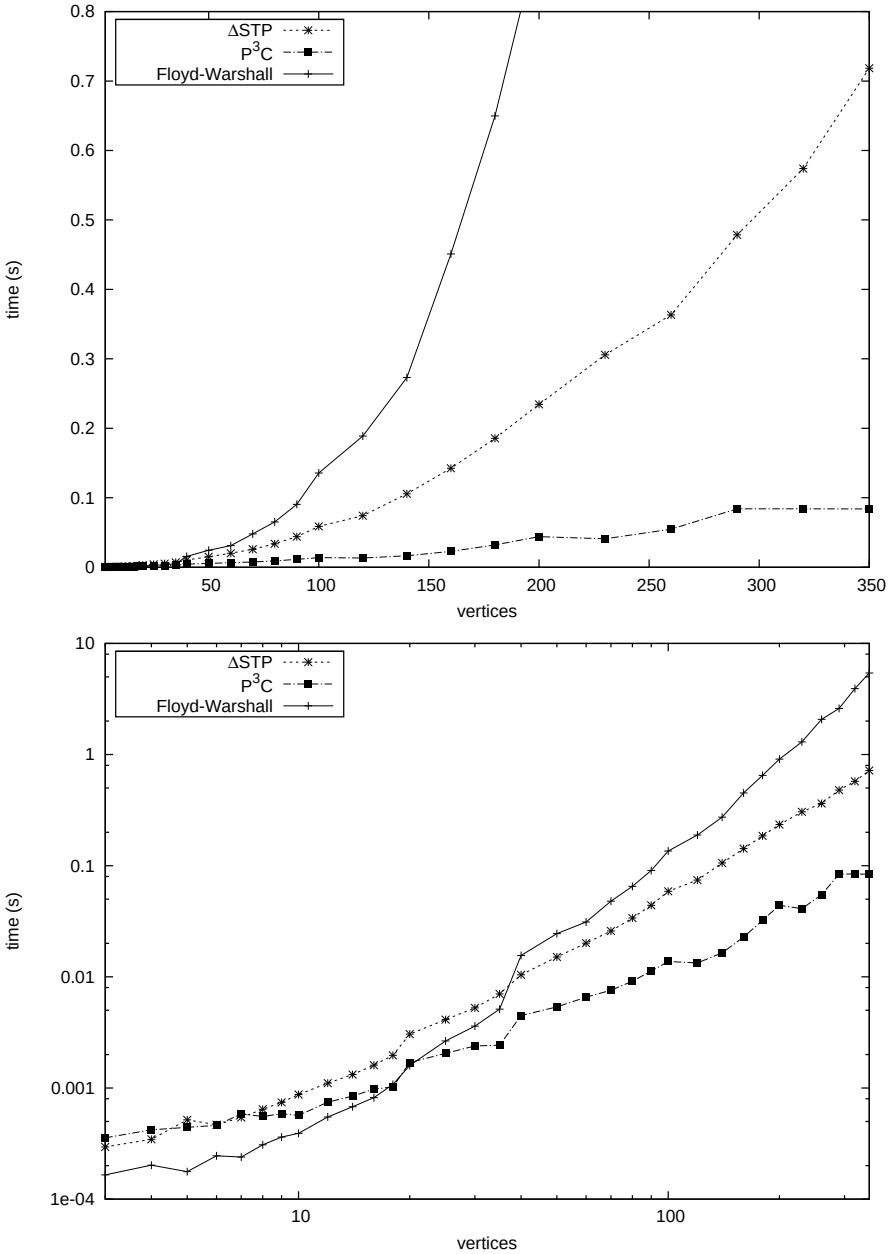


Figure 6.15: Pathological behaviour of △STP
linear scale (top) and double-log scale (bottom)

sured in several ways. Here, we consider two methods: (i) incrementally run the minimum-degree triangulation procedure after the addition of each new constraint edge; and (ii) construct the graph consisting of all constraint edges that will be added and triangulate it once with the minimum-fill triangulation procedure. Clearly, (ii) is only feasible if the structure of the constraint graph is known beforehand. As we mentioned in Section 3.4, incremental triangulation is subject to future research; for now, we opted to naively perform regular (non-incremental) triangulation after addition of each new constraint, using the fast minimum degree heuristic. For the "single-shot" approach, we valued the improved quality of triangulations produced by minimum fill over the better performance of minimum degree.

This resulted in a test of, in total, four incremental algorithms. As input to these algorithms we used the benchmarks discussed before, and scale-free constraint graphs with low values for the m parameter; order in which the constraints were fed to the algorithms was random, but identical across all algorithms to ensure a fair comparison. The results are included in Figures 6.16 through 6.21. The IFPC algorithm, being clearly uncompetitive with regard to the others for large instances, was only run on the DTP and on scale-free graphs consisting of up to 100 vertices.

From the results, it turns out that despite the worst-case analyses given in the previous chapter, both IDPC and IPPC generally perform far better than IFPC; for large enough instances, IPPC also outperforms IDPC. The reader should take note that IDPC does not calculate minimal constraints, which the other algorithms do; for this reason, the comparison of IPPC to IDPC is not entirely fair. An exception can be noted in Figure 6.21, where IFPC performs better than IPPC variant; note, however, that the number of vertices in these instances is fixed at a rather modest value of $n = 35$. A topic for future research is to investigate the induced width w^* of the various problem instances, and to determine its relation with the performance of IDPC and IPPC.

As was to be expected, the single-shot triangulation approach to IPPC performed best; more surprising, however, is the fact that even our naive approach to incremental triangulation outperforms IDPC for large enough instances. In our opinion, this gives great hope for further gains to be achieved by future research on some more intelligent approach to incremental triangulation.

6.5 Summary

In this chapter, we empirically compared the performance of the algorithms proposed in Chapter 5 against that of their precursors from Chapter 3.

For determining consistency of the STP, we concluded that Bellman's and Ford's algorithm remains sovereign. Of the approaches for achieving directed path consistency methods described in this text, the minimum degree heuris-

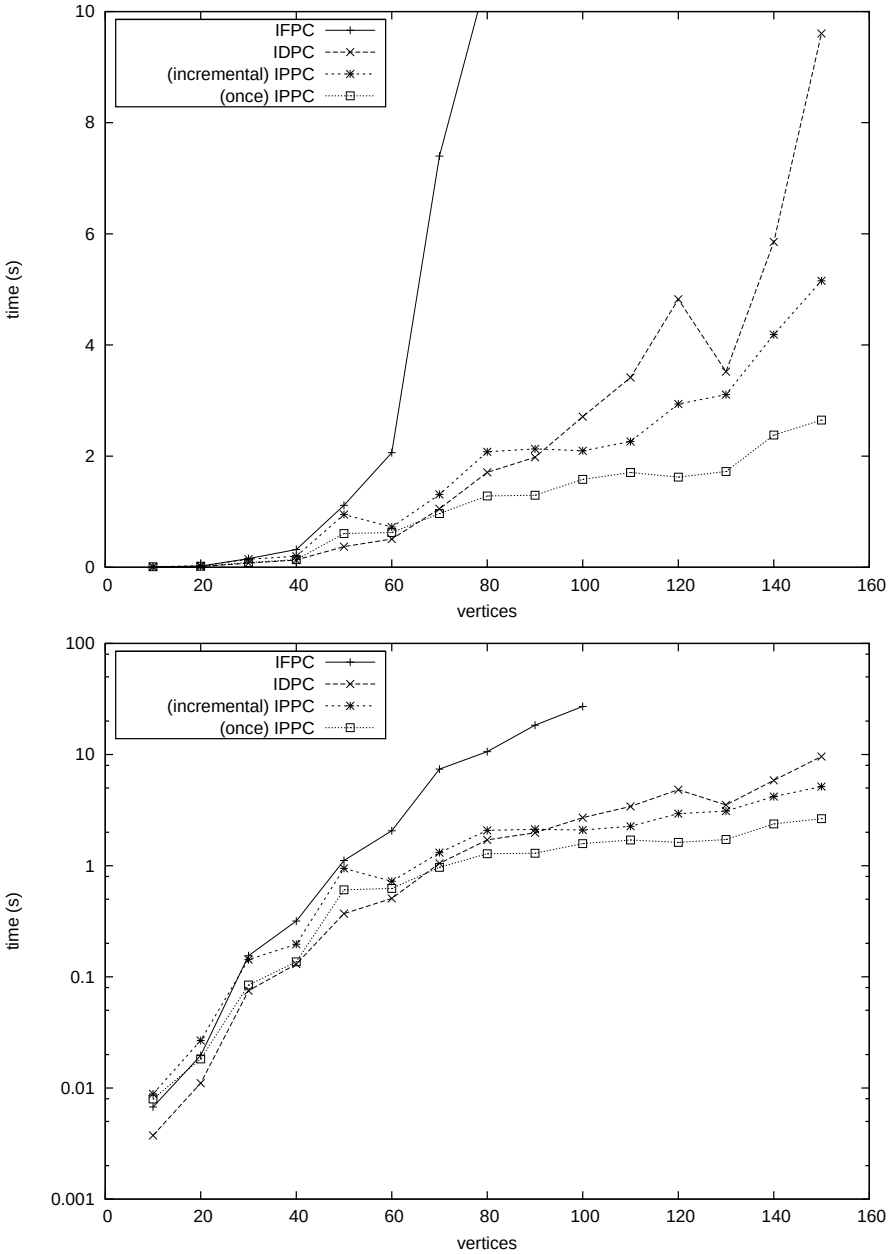


Figure 6.16: Incremental methods on scale-free graphs with $m = 2$ linear scale (top) and log scale (bottom)

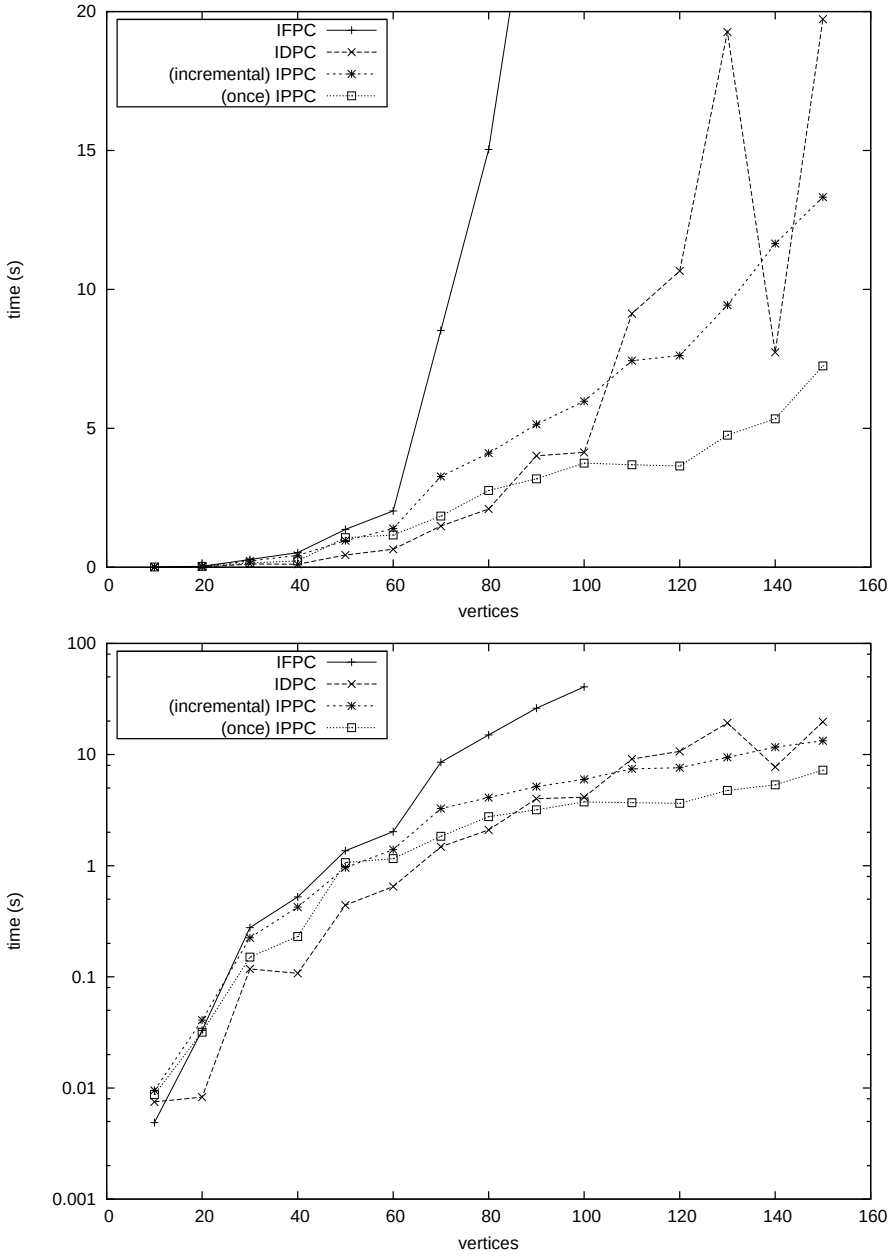


Figure 6.17: Incremental methods on scale-free graphs with $m = 3$ linear scale (top) and log scale (bottom)

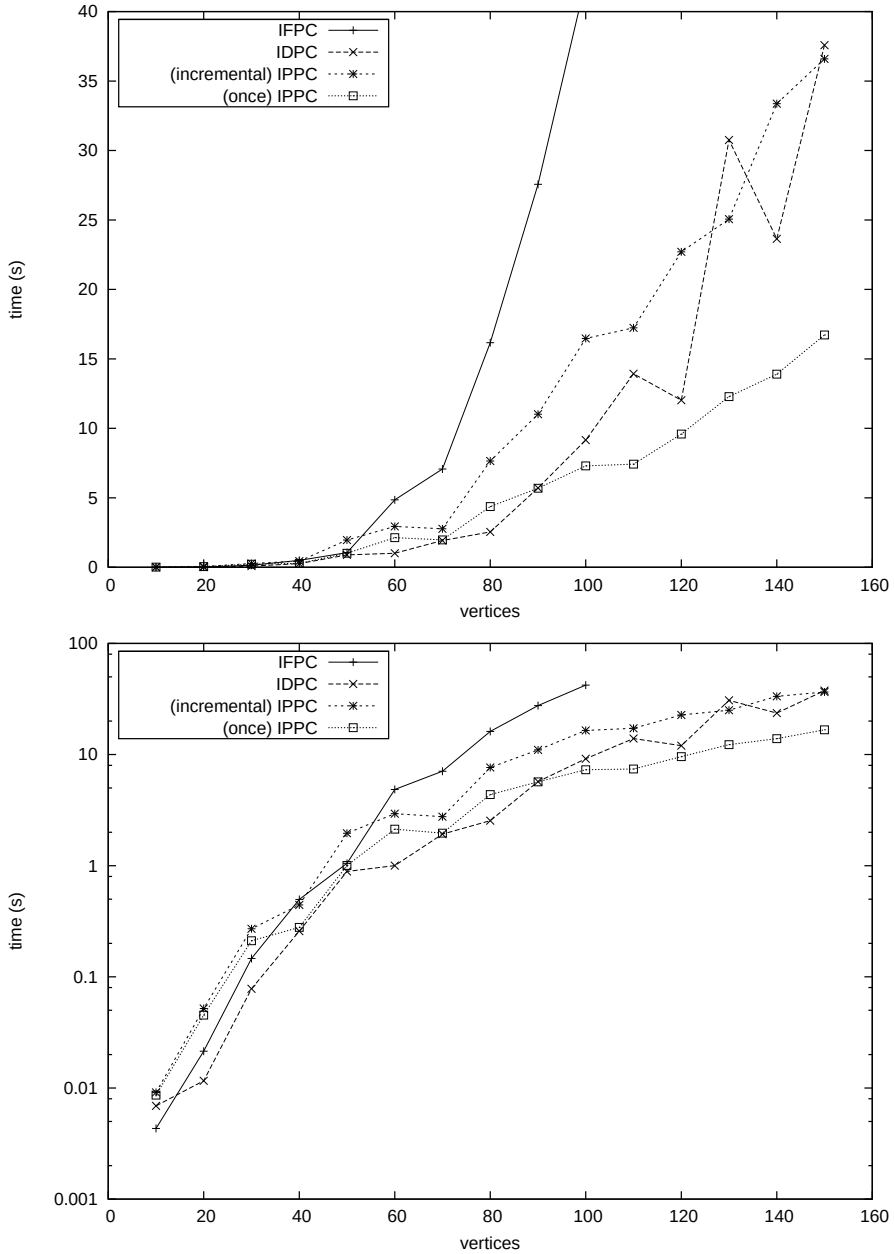


Figure 6.18: Incremental methods on scale-free graphs with $m = 4$ linear scale (top) and log scale (bottom)

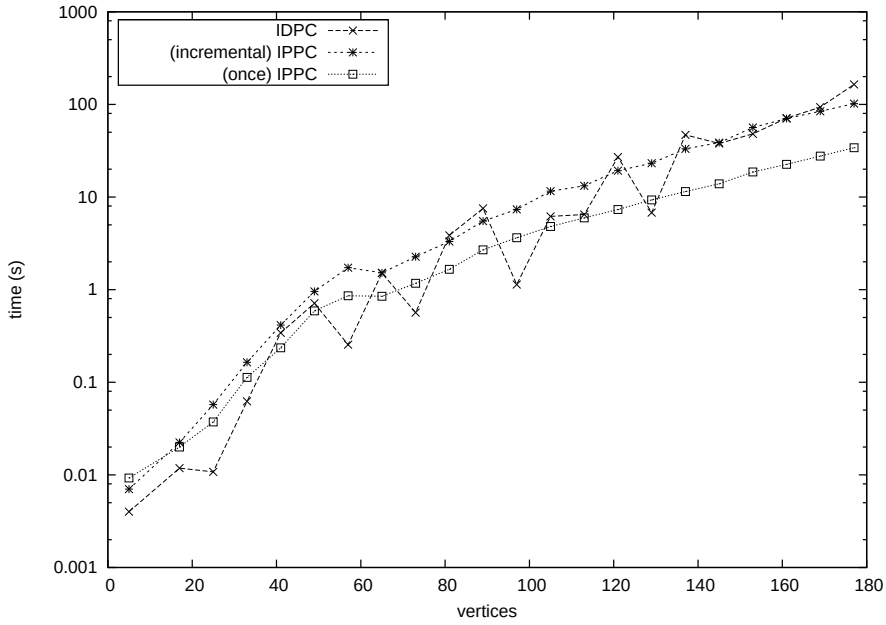


Figure 6.19: Incremental methods on the job shop problem with $n < 180$ linear scale (top) and log scale (bottom)

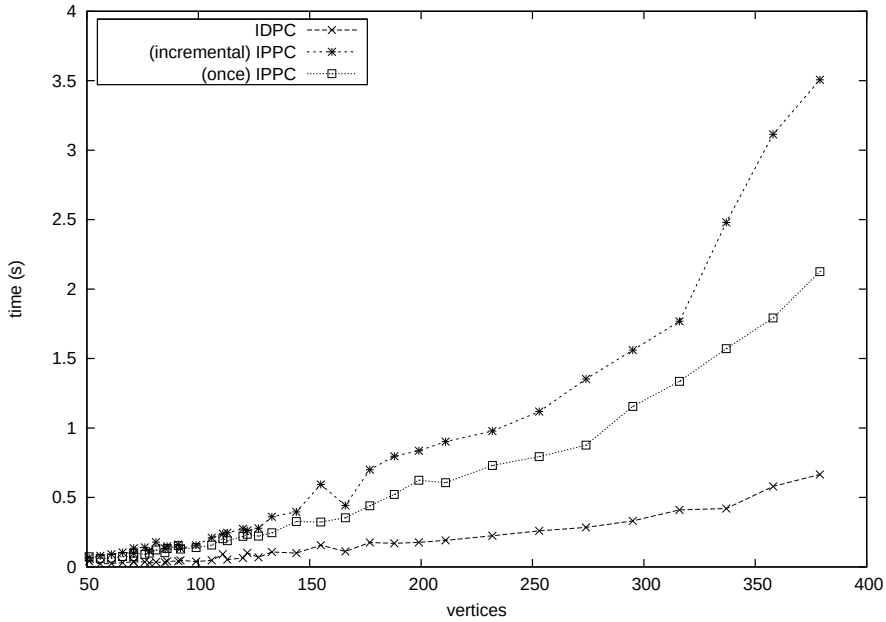


Figure 6.20: Incremental methods on the diamonds problem

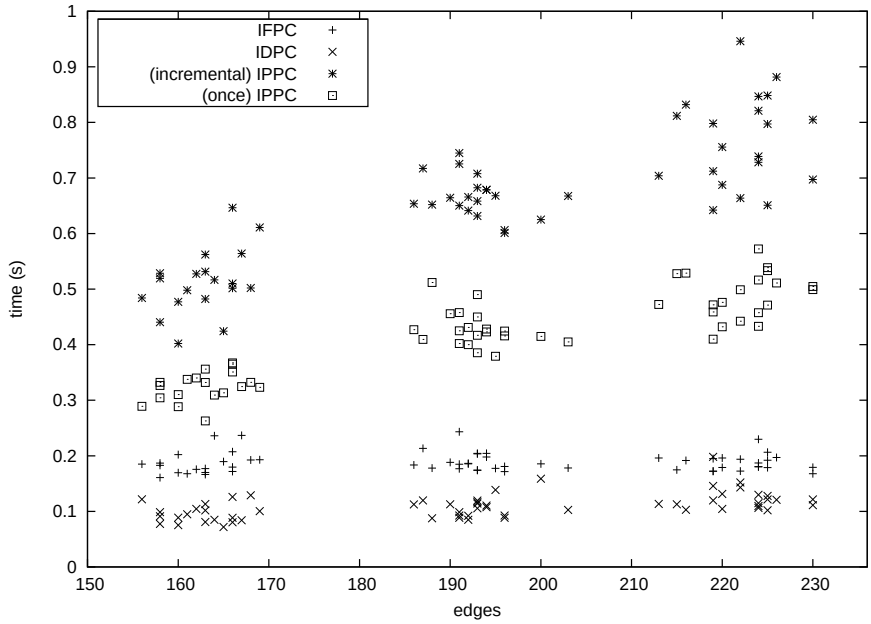


Figure 6.21: Incremental methods on the DTP ($n = 35$)

tic performed best, with little difference between the directed and undirected variants. For the minimum fill heuristic, these differences were more pronounced, with the directed variant performing slightly better than the undirected one.

For problem instances of at least moderate size, our P^{3}C algorithm consistently outperformed the previous state-of-the-art algorithm $\triangle$STP. Again, the minimum degree heuristic performed much better than the minimum fill heuristic. We also tested the pathological problem instance for $\triangle$STP, designed in the previous chapter, and confirmed the theoretical result that the difference in performance with P^{3}C increases linearly with the size of the instances.

Finally, we evaluated the performance of the incremental partial path consistency algorithm (IPPC). This yielded the most unexpectedly positive result of this chapter: it turned out that in practice, despite the worst-case complexity analysis from the previous chapter, the performance of our new incremental algorithm far surpasses that of its competitor, incremental full path consistency (IFPC).

Chapter 7

Discussion

In this final chapter, we first draw conclusions from the matter discussed in preceding chapters, summarising the most important results. Then, we indicate interesting and promising directions for future research.

7.1 Summary and conclusions

In Chapter 2, we gave a formal definition of the STP and formulated a motivating example. We listed three possible definitions of solving an STP instance $\mathcal{S}$, in order of increasing difficulty: (i) determining whether $\mathcal{S}$ is consistent; (ii) finding a single instantiation to all time-point variables; and (iii) calculating all minimal constraints. In this thesis, we concerned ourself only with (i) and (iii), referring to them as STP-CONSISTENCY and STP-MINIMALITY, respectively.

Our main contribution in Chapter 2 consisted of formally establishing the complexity of these problems. We proved NL-hardness and membership in NC^2 for both problems; the latter means that the problem can be solved efficiently (in $\mathcal{O}(\log^2 n)$ time) by a parallel algorithm. For polynomially bounded weights, both problems become NL-complete, which means that they can be solved by nondeterministic algorithms (or randomised algorithms) using only logarithmic space and unbounded time.

Then, in Chapter 3, we turned our attention to known algorithms for dealing with the STP, in three classes:

1. those that decide whether the STP instance is consistent (cf. (i) above);
2. those that calculate minimal constraints (cf. (iii) above); and
3. those that perform either of the preceding tasks in an incremental fashion.

For the second of these classes, we included not only algorithms that solve the all-pairs shortest paths (APSP) problem on the complete constraint graph, but also considered the $\triangle$STP algorithm, which enforces partial path consistency

(PPC). For PPC, the constraint graph is not completed; instead, every constraint edge that is present is guaranteed to have minimal weight. Included in the bargain of calculating minimal constraints are the other two definitions of "solving", numbered (i) and (ii) in the first paragraph of this section. Incremental STP solvers—the third class we mentioned—are important whenever dealing with more complex temporal reasoning problems, such as the Disjunctive Temporal Problem, in which the STP appears as a subproblem and is constructed step by step.

Before proposing a new algorithm for each of these classes, we took a step back and explored some graph theory in Chapter 4. The theory presented there concerned the concept of "chordality", which is pivotal in the state-of-the-art algorithm △STP. For undirected graphs, this concept had already been thoroughly explored in existing literature; however, to the best of our knowledge, we are the first to have extended this concept to the directed case. In addition to these concepts, we described two known heuristics for enforcing chordality—the *minimum-fill* and *minimum-degree* heuristics—and showed how these can be extended to the directed case.

From graph theory, it is known that undirected graphs are chordal if and only if they have a simplicial elimination ordering. We defined the similar concept "transitive elimination ordering" for directed graphs and showed that its existence is sufficient, but not necessary, for chordality.

The main contributions of this thesis were presented in Chapter 5. Here, we drew on the theory from the previous chapter to present our new algorithms, one for each of the tasks listed above:

1. *Directed path consistency (DPC), run on a transitive elimination ordering of the directed graph*

   We already described the DPC approach in Chapter 3; our contribution to this method concerns the order in which the time points are processed. We showed that when run along a simplicial or transitive elimination ordering of a chordal graph, the algorithm introduces no new constraints and considers every triangle at most once, possibly less if inconsistency is concluded. If a graph is not yet chordal, the minimum-fill or minimum-degree heuristic can be used to determine the ordering.

2. P^3C, *a new algorithm for enforcing partial path consistency*

   This algorithm is an improvement over the △STP algorithm described in Chapter 3; it processes each triangle in the constraint graph at most twice (less if inconsistency is concluded), in contrast to its predecessor, which may process triangles many times. Indeed, we designed an STP instance on which △STP may take longer than P^3C to complete by a factor proportional to the number of vertices n.

3. *Incremental partial path consistency (IPPC), a logical successor to incremental full path consistency and incremental directed path consistency*

This method takes an already partially path-consistent constraint graph and a new constraint edge, and processes the triangles in the graph along a cleverly chosen ordering to once more ensure the PPC property. However, the theoretical worst-case performance was unfortunately shown to be worse than that of IFPC, and on par with that of IDPC, both described in Chapter 3.

Finally, in Chapter 6, we performed an extensive empirical evaluation of all new algorithms against existing ones; our test cases consisted of benchmark problems, randomly generated problems and the pathological instance mentioned above. We can draw the following conclusions:

1. For determining *consistency*, Bellman's and Ford's algorithm remains the method of choice. It outperformed the other methods by an ample margin, except on very sparse graphs such as the diamonds benchmark (Figure 6.8); here, DPC showed promising behaviour.

   With regard to the different orderings for DPC we tested, the random ordering (using no heuristic) performs worst. There are also clear differences between the performances of the heuristics. The minimum-degree heuristic consistently outperforms minimum-fill. The differences between the directed and undirected heuristics are very small. For the minimum-fill heuristic, it is safe to state that the directed heuristic performs better; for the minimum-degree heuristic, the results are to close to designate a winner, though the directed heuristic seems to have a slight edge over the undirected one.

2. When calculating *minimal constraints*, our P^3C algorithm never performs significantly worse than $\triangle$STP and often performs much better. For both algorithms, the minimum-degree heuristic is again clearly superior to the minimum-fill heuristic; even though the latter may produce smaller triangulations, the heuristic itself is just too costly for this advantage to pay off.

   The all-pairs shortest paths algorithms are never competitive to either of the PPC algorithms, except for very small problem instances.

3. Our new *incremental* algorithm IPPC shows promising results, outperforming IFPC already for moderate amounts of vertices. These results hold even when performing a naive incremental triangulation, and are diametrically opposite to the theoretical worst-case analysis performed in Chapter 5. Nonetheless, they were to be expected: after all, IFPC has to maintain a constraint graph with $\Theta(n^2)$ edges, which must take its toll.

   The comparison with IDPC is less clear-cut, but as the problem instances grow in both amount of vertices and density, IPPC eventually outperforms IDPC too. It must be noted here, once more, that the task

performed by IDPC—enforcing directed path consistency—is significantly easier than that performed by IPPC, which calculates minimal constraints.

Summarising these findings, we can safely state that with both P^{3}C and IPPC, we make a significant contribution to the STP literature. In the next section, we give some hints as to what further research can be done on this subject matter.

7.2 Future work

In this section, we briefly state some promising topics for future research, roughly ordered according to the subjects addressed in the main text.

- In Chapter 2, we presented an abstract algorithm for the *parallel random access machine* (PRAM) model. The value of a parallel approach on a concrete parallel computer has yet to be demonstrated.
- As mentioned in Chapter 2, the STP is NL-complete when the weights of its constraints are polynomially bounded in the amount of vertices. One may wonder to what extent this limits the expressiveness of the STP; as stated before, our expectation is that for many practical applications, the bounded variant is easily sufficient.
- We mentioned in the conclusions to Chapter 2 that NL-completeness implies that a randomised approach requiring logarithmic space and unbounded time must exist. Future research may be conducted into the viability of such an approach; further, it may be possible to find a subclass $\mathcal{C}$ of the STP for which a time bound on the randomised algorithm can also be guaranteed.
- In Chapter 4, we presented some approaches for enforcing chordality on graphs, i.e. graph triangulation. The IPPC algorithm would greatly benefit from a method that performs incremental triangulation; to the best of our knowledge, no such methods are available in current graph literature.
- Also in Chapter 4, we extended the concept of chordality to directed graphs and stated that the existence of a transitive elimination ordering is a sufficient (but not necessary) condition for it; this means that there is a set of directed graphs that are chordal, but for which as yet no efficient recognition method is known. Further research may address what properties characterise the graphs that fill in this "gap".
- Again in Chapter 4, we mentioned that the directed triangulation operation may separate a graph into multiple biconnected components (see Figure 4.4). A heuristic may exploit this phenomenon to produce a smaller triangulations; as yet, no such heuristic has been described yet.

- In Chapter 5, we mentioned that PPC algorithms (viz. $\triangle$STP and P^{3}C) may also be run on directed chordal graphs, with possibly better efficiency. However, in this case, minimal constraints are not always calculated, though a sufficient condition that guarantees minimality of a constraint can be stated. Future experiments are required to demonstrate whether this trade-off is viable.

- At the root of the $\triangle$STP algorithm lies the partial path consistency (PPC) property for general constraint satisfaction problems (CSPs), described by Bliek and Sam-Haroud [BSH99]. For convex CSPs such as the STP, enforcing PPC is equivalent to enforcing path consistency (PC); however, for general CSPs, Bliek and Sam-Haroud stated that PPC may still be a useful approximation of PC.

  With minor adjustments, the $\triangle$STP algorithm as presented in this text can also be used for general CSPs. It is our expectation that this also holds for our P^{3}C algorithm, which would further extend the scope of its usefulness.

- In nearly all cases, Bellman's and Ford's algorithm outperforms any DPC variant by a large margin; an exception is the diamonds benchmark, where Bellman's and Ford's algorithm still performs best, but the results are much closer. Future research is required to determine exactly which circumstances cause the results to be so close, and if DPC can be further improved to outperform Bellman's and Ford's algorithm in such cases.

- In practice, the incremental partial path consistency algorithm (IPPC) outperforms its counterpart, incremental full path consistency (IFPC), by a larger margin than we expected from the theoretical worst-case analysis performed in Chapter 5. Further experiments are needed to analyse the practical relation between the performance of IPPC and the induced width w^* (defined in Section 3.1.2) of the constraint graph.

- Finally, the most important topic for future research is the application of our new algorithms to more expressive temporal reasoning problems, such as the Temporal Constraint Satisfaction Problem [DMP91] and the Disjunctive Temporal Problem [SK00]. These problems are NP-complete and are usually solved with a backtracking approach; since an STP instance must be solved for each node in the backtracking search tree, we expect that the use of our new P^{3}C and (especially) IPPC algorithms will yield great benefits.

Our hopes for this last topic are especially high. We expect that our new methods can find practical application in any domain that requires efficient solving of temporal problems, such as logistics, planning and scheduling.

Bibliography

[AB02] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47–97, Jan 2002.

[ATMB06] Luca Anselma, Paolo Terenziani, Stefania Montani, and Alessio Bottrighi. Towards a comprehensive treatment of repetitions, periodicity and temporal constraints in clinical guidelines. *Artificial Intelligence in Medicine*, 38(2):171–195, October 2006.

[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.

[BW04] Pieter Buzing and Cees Witteveen. Distributed (re)planning with preference information. In R. Verbrugge, N. Taatgen, and L. Schomaker, editors, *Proceedings of the 16th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2004)*, pages 155–162, 2004.

[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.

[CP89] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35(2):164–176, 1989.

[Dec03] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[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.

[Eve79] Shimon Even. *Graph Algorithms*. W. H. Freeman & Co., New York, NY, USA, 1979.

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

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

[FRCY97] A. Fukunaga, G. Rabideau, S. Chien, and D. Yan. Aspen: A framework for automated planning and scheduling of spacecraft control and operations. In *Proceedings of the International Symposium on AI, Robotics and Automation in Space*, 1997.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, 1988.

[JáJ92] Joseph JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

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

[Kjæ90] Uffe Kjærulff. Triangulation of graphs - algorithms giving small total state space. Technical report, Aalborg University, March 1990.

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

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[Ros72] Donald J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In Ronald C. Read, editor, *Graph theory and computing*, pages 183–217. Academic Press, N.Y., 1972.

[RT03] Silvio Ranise and Cesare Tinelli. The SMT-LIB format: An initial proposal. In *Proceedings of PDPAR'03*, July 2003.

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

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

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

[Sze87] Róbert Szelepcsényi. The method of forcing for nondeterministic automata. *Bulletin of the European Association for Theoretical Computer Science*, 33:96–100, October 1987.

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

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

[Wes96] Douglas B. West. *Introduction to Graph Theory*. Prentice-Hall, 1996.

[XC03] 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.