Algorithms for
Simple Temporal Reasoning

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft;
op gezag van de Rector Magnificus prof. ir. K. Ch. A. M. Luyben,
voorzitter van het College voor Promoties
in het openbaar te verdedigen op
maandag 9 december 2013 om 10.00 uur
door

Léon Robert PLANKEN

ingenieur in de technische informatica
geboren te Alkmaar

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. C. Witteveen

Copromotor: Dr. M. M. de Weerdt

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. C. Witteveen	Technische Universiteit Delft, promotor
Dr. M. M. de Weerdt	Technische Universiteit Delft, copromotor
Prof. dr. ir. J. A. La Poutré	Technische Universiteit Delft / Centrum Wiskunde & Informatica
Prof. dr. ir. C. W. Oosterlee	Technische Universiteit Delft
Prof. dr. S. F. Smith	Carnegie Mellon University, Verenigde Staten van Amerika
Dr. H. L. Bodlaender	Universiteit Utrecht
Dr. A. Oddi	Consiglio Nazionale delle Ricerche, Italië
Prof. dr. ir. H. J. Sips	Technische Universiteit Delft, reservelid

Dissertation series no. 2013–42

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

ISBN 97890-6562-337-9

Published by Delft Academic Press
Cover design: Jurjen de Jong
Illustration by Sir John Tenniel (1820–1914) for *Alice's Adventures in Wonderland*

in memory of my mother
Anneke Planken-Overtoom

Acknowledgements

Obtaining a Ph.D. is not an easy job; nor should it be. I'm happy to have enjoyed support in many guises along the way and I'm glad to be able to express my thanks here to those who stood by me.

First of all, thanks are due to my supervisors Mathijs de Weerdt and Cees Witteveen. Their trust in my abilities seemed nearly boundless at times and it was certainly higher than my own. Moreover, they were virtually always available, and the health and fatigue problems that I ran into in no way impacted their confidence in me. I believe they went above and beyond what a Ph.D. candidate can expect from his supervisors in this respect, and I owe them a debt of gratitude.

My other (former) colleagues in the Algorithmics group — Adriaan, Bob, Chetan, Gleb, Hans, Joris, Marijn, Matthijs, Michel, Peter, Renze, Shruti, Sicco, Tamás, Tomas, and Yingqian — made day-to-day working life at the faculty a very enjoyable experience. With a smile I recall many interesting discussions at lunchtime, and the working atmosphere was always one where everybody's door was open for a question or just to go get a cup of coffee.

For two months in the autumn of 2012, I swapped my work environment in Delft for one in Rome, Italy. Thanks are due to my hosts, Riccardo Rasconi and Angelo Oddi, and to Luca Coraci for helping out with practical issues such as finding lodgings there. I also thank my other temporary colleagues in Rome — Alessandro, Amadeo, AndreA, Gabriella, Giulio, Lorenza, Marco, and Riccardino — for making me feel at home. *Grazie mille a tutti!*

Recurring international conferences form an important part of the charm of academia. I've met many nice and interesting people at such venues and have been able to lay the groundwork both for cooperation with my co-authors and for my visit to Italy. In particular, I thank Neil Yorke-Smith and Jim Boerkoel for working on research together. I also thank Roman van der Krogt, my most frequent co-author after my supervisor Mathijs, for our discussions and for his contributions, mainly in the area of experimentation. I would also like to thank Ot ten Thije and Jan Elffers for choosing me as their (co-)supervisor and picking a research subject in "my" field of Simple Temporal Reasoning. Each of these cooperations has deepened my own knowledge of the field.

Much of the work presented in this dissertation is of a theoretical nature. It is often not bound to a laboratory, a computer, or even pen and paper, but instead takes place in the mind; it can go on outside working hours, on the bus,

while taking a shower, or in bed. It is then important to regularly get one's mind otherwise involved and I thank all my friends who have managed to abduct me over the years to play a game or simply have a chat over dinner.

Music, especially, has proved to be a wonderful distraction for much of my life. I thank my friends at the Krashna Musika student orchestra, the wind quintet "Kwintvlaag", the Dordrechts Philharmonisch Orkest, and my many other friends in music for providing me with diversion of a very high quality in so many ways.

I am grateful to my family for supporting me and believing in me without fail. Although it is sad that my mother saw the beginning of this journey but is no longer here to witness its end, her pride in me has continued to be a strong motivation. And finally, thank you, Diana, for your unconditional patience, support, confidence, and love.

Contents

List of Algorithms

CHAPTER 1

Introduction

Time is an important concept in our daily lives, and we continuously deal with it in our plans and actions. We generally have a pretty clear idea of what we will be doing during the next few hours or so. We're going to prepare a meal when we get home in the evening, and then intend to do some household chores or maybe see a movie after dinner. The rest of the week or month has some landmark events and meetings, often marked in our calendars so we won't forget them; but as we browse further ahead, we find more and more blank pages.

Everyone is also familiar with unexpected events that require our schedules to be reshuffled and reorganised. While we're cooking, we may find ourselves out of milk; when we finally depart to the movie theatre, already a little late, we discover that we have a flat tire. Nor do we always have ourselves to blame: for instance, a friend could call to cancel or reschedule next Friday's game night.

These examples illustrate that maintaining temporal information is an important and frequently performed task. The field of study in artificial intelligence and computer science that deals with this is called *temporal reasoning*. In this dissertation, we focus on a simple yet powerful framework for temporal reasoning called the *Simple Temporal Problem (STP)*. Our main contributions consist in a formal study of the types of queries that can be asked of the STP and the proposal of four new state-of-the-art algorithms that can be used for answering them.

In the remainder of this introduction, we first give some historical context and background of the field of temporal reasoning and show where the STP fits in. We investigate related "simple" problem classes and show that this word in the STP's name certainly does not imply that it is a trivial or uninteresting problem class. Next, Section 1.2 presents an outline of the dissertation. An itemised list highlights the main contributions and the peer-reviewed papers in which they first appeared. The chapter is concluded by Section 1.3, which suggests an efficient route to the most important results in the dissertation.

1.1 Context and motivation

The last two decades of the 20th century saw the proposal of several formalisms for temporal reasoning. These include (i) interval algebra (Allen, 1983), (ii) point algebra (Vilain and Kautz, 1986), (iii) time maps (Dean and McDermott, 1987), and (iv) temporal constraint networks (Dechter, Meiri, and Pearl, 1991). Only the latter two allow for *quantitative* reasoning, i.e. specifying not just the order of events but also the temporal distance between them. Moreover, since temporal constraint networks generalise time maps and allow the powerful techniques of constraint processing to be used, they are the subject area of this dissertation.

The field of constraint networks was pioneered by Montanari (1974) and further expanded by Mackworth (1977). These networks consist of variables and constraints between pairs of them,* and allow for intuitive representation of and reasoning about real-life situations and crafted puzzles alike. Each constraint can then be seen as a limitation of possibilities, or equivalently as an expression of what is allowed. There is a close analogy with relational databases, where relations take the role of constraints and operations over them are described by relational algebra. Indeed, like a database, a constraint network can be used to store knowledge of different types.

Conventionally, the universe of discourse for constraint networks (as well as relational databases) is discrete. Time, however, is continuous. Through their proposal of temporal constraint networks, Dechter et al. (1991) extended the theory of constraint networks to the continuous temporal domain and investigated the applicability of known techniques from constraint processing to this case. This formalism continues to find application in such diverse areas as space exploration (Muscettola et al., 1998), operation of Mars rovers (Bresina et al., 2005), medical informatics (Anselma et al., 2006), and coordination of disaster relief efforts (Barbulescu et al., 2010).

Tractability

Among the four formalisms for temporal reasoning listed above, the two most expressive are interval algebra (i) and our subject area of temporal constraint networks (iv). Hence, these allow for representation of the largest set of real-life problems. At the same time, exactly these two correspond to NP-hard problem classes, and therefore they are presumed to be intractable in general.

In computer science, when facing a problem P from an intractable class $\mathcal{C}$, the first thing to do may seem obvious but is easy to overlook: find out if P is not in fact in a simpler, tractable class $\mathcal{S} \subset \mathcal{C}$, or if a related problem P' from $\mathcal{S}$ suffices to describe the task at hand. If this happy circumstance is not the case, computer scientists have several ways to cope with this. One is to tackle P with a backtracking approach and try to limit the extent of the search tree. This can be

*Thus, strictly speaking, these are *binary* constraint networks: each constraint involves exactly two variables. One can also define constraint networks of higher arities; however, they are outside the scope of this text.

done by employing a *branch-and-bound search* or by formulating *heuristics* (rules of thumb) to guide the algorithm into the parts of the search tree that hold the most promise of containing a solution. If P is an optimisation problem, another common approach is to use an *approximation*: to settle for a solution to P that is not guaranteed to be optimal but is "good enough".

Whatever approach is taken, approaches that solve $\mathcal{S}$ constitute a toolkit that allow the computer scientist to deal with the complete class $\mathcal{C}$. This explains why it often pays to look for and study tractable subclasses of hard problems, and a substantial amount of literature is devoted to this. Thus, although the class $\mathcal{S}$ may be described as "simple", its study is certainly not trivial.

In fact, point algebra — formalism (ii) above — was proposed by Vilain and Kautz (1986) as a tractable variant of the complete interval algebra (i) after they proved that deciding consistency of the latter is NP-hard. The point algebra was further distilled by Van Beek (1991) into a problem class called *simple interval algebra (SIA)*. Nebel and Bürckert (1995) went the other direction and proposed the ORD-Horn problem class, which they proved to be the maximal tractable subclass of interval algebra.

When presenting their temporal constraint networks (iv), Dechter et al. (1991) proposed the general Temporal Constraint Satisfaction Problem (TCSP) along with a tractable subclass, called the *Simple Temporal Problem (STP)*. Dean and McDermott (1987) introduced their time map (iii) as a method for *"shallow temporal reasoning"*, and upon closer inspection, the time map appears to be equivalent to (an informal specification of) the STP.

A further motivation for the focus on tractable problem classes, from a practical point of view, is given by Van Beek (1991, pp. 328–329). He points to five separate applications that claim to make use of interval algebra while in fact only using the SIA subset, and notes that "[with] the exception of [one of them], it does not appear that the authors intentionally restricted the representation language or were aware of [tractability considerations]; rather, the relations used were simply the right ones for the task at hand." Thus, these represent real examples of the situation described above where one faces a problem $P \in \mathcal{C}$ that turns out to be from $\mathcal{S} \subset \mathcal{C}$.

To summarise the discussion so far, Figure 1.1 schematically lays out the temporal reasoning frameworks described above, together with the more general Disjunctive Temporal Problem (DTP). It gives a hierarchy of their expressiveness and thus, to an extent, of their complexity. Table 1.1 gives a glossary of the depicted problem classes. For those familiar with propositional satisfiability and mathematical optimisation, the figure also depicts relations with problem classes from these fields. We also note that some of these are referred to in upcoming chapters: the problem of linear programming (LP) is mentioned a few times in Chapter 2, and several of the benchmark sets used for our experiments are originally from a library of benchmarks for solvers of satisfiability modulo theories (SMT).

All problem classes above the "tractability boundary" in Figure 1.1 are NP-hard, whereas those below it can be solved in polynomial time. From the

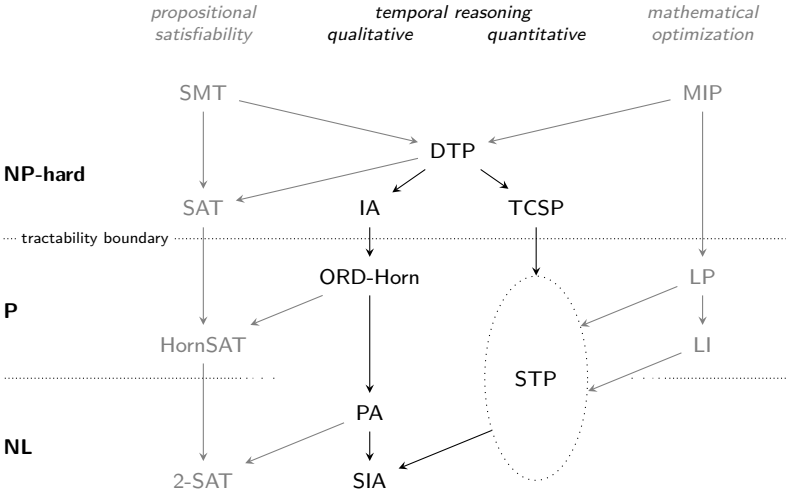


Figure 1.1: Relation between formalisms for temporal reasoning and other problem classes. Refer to Table 1.1 for a glossary. Arrows $P \to Q$ indicate that problem P is more expressive than problem Q, and that a trivial reduction from Q to P exists.

Table 1.1: Glossary of problem classes used in Figure 1.1 and their stated complexity classes in literature. Note that new complexity results can already be derived from Figure 1.1.

Problem class		Complexity
Temporal reasoning		
DTP	Disjunctive Temporal Problem	NP-complete
IA	Interval algebra	NP-complete
TCSP	Temporal Constraint Satisfaction Problem	NP-complete
ORD-Horn	Maximal tractable subclass of IA	in P
PA	Point algebra	in P
STP	Simple Temporal Problem	in P
SIA	Simple interval algebra	in P
Propositional satisfiability		
SMT	Satisfiability modulo theories	NP-hard
SAT	Propositional satisfiability	NP-complete
HornSAT	Horn-satisfiability	P-complete
2-SAT	2-satisfiability	NL-complete
Mathematical optimization		
MIP	Mixed integer programming	NP-complete
LP	Linear programming	P-complete
LI	Linear inequalities	P-complete

figure, we see that the STP is strictly less expressive than solving a set of linear inequalities (LI) or a linear programming problem (LP), both of which are P-complete. In contrast, 2-SAT is known to complete for the subclass NL, so this problem can (also) be solved in nondeterministic logarithmic space. It is important to note that apart from its membership in P, the exact complexity of the STP has not yet been studied in existing literature; hence the dashed ellipse surrounding it in the figure. This fundamental question is investigated in the next chapter. The stage is then set to assemble an algorithmic toolkit for simple temporal reasoning in the remainder of the dissertation. Let us now give a full outline of the dissertation and highlight our main contributions.

1.2 Contributions and outline

In the remainder of this work, our focus is on the STP. Surprisingly, although references to this problem abound in the literature, no single formal definition seems to fit all these appearances. We show that this is because there are in fact several separate problem queries on simple temporal information. After presenting a primer on simple temporal reasoning and including three minor yet previously unstated results, we formally define nine Simple Temporal Queries (STQs) and derive their theoretical complexity: although all queries are tractable, some of them are in strict subclasses of P, while the hardest turns out to be P-complete. Finally, we show that if simple temporal information is in some sort of normal form, many queries become strictly easier to answer.

In Chapter 3, we present background from graph-theoretical and constraint-theoretical literature and explore the links between these fields. Although part of the presented graph theory already formed the basis for extant temporal reasoning approaches, we uncover and prove new relations with constraint networks, presenting it all in a unified way that was sometimes absent in the literature. In particular, we unearth a seemingly forgotten triangulation algorithm and show for the first time how it applies to temporal constraint networks. By the end of the chapter, we show how the presented concepts of directional, partial, and full partial path consistency (respectively DPC, PPC, and FPC) relate to the Simple Temporal Queries from Chapter 2. We show that both PPC and FPC networks can be considered as a normal form for temporal information. We also provide a proof for a fact that was tacitly taken for granted in the literature.

In the remaining chapters, we study algorithms to establish PPC (Chapter 4) and FPC (Chapter 5), as well as algorithms to maintain either of these normal forms when new information becomes available (Chapter 6). Each of these chapters follows the same general structure. New algorithms are presented, and their correctness and complexity are formally established. These are then compared to previous art, both theoretically and empirically. It turns out that each of the new algorithms can be considered to represent the new state of the art in its respective class, at least for sparse temporal networks and sometimes even across the board.

We conclude this section with a concise list of our main contributions. Where applicable, we refer to the peer-reviewed papers (highlighted in boldface) in which a particular contribution was first published.

- An inventory of nine Simple Temporal Queries and complexity analysis placing them in strict subclasses of P and, in particular, proof that STQ 9 is P-complete;
- New results that establish a strict hierarchy between three network-wide path-concistency concepts, and show how they can be applied to Simple Temporal Queries;
- P^{3}C, the current state-of-the-art algorithm for PPC **(Planken, De Weerdt, and Van der Krogt, 2008)**
- Snowball, a very efficient algorithm for full path consistency (FPC), alias all-pairs shortest paths (APSP):
  - First published by **Planken, De Weerdt, and Van der Krogt (2011)** and awarded with the ICAPS honourable mention for best student paper;
  - Extended by **Planken, De Weerdt, and Van der Krogt (2012)** in the JAIR award-winning papers track;
- IFPC, a simple yet efficient algorithm for incremental full path consistency **(Planken, 2008b)**;
- IPPC, a state-of-the-art algorithm for incremental partial path consistency **(Planken, De Weerdt, and Yorke-Smith, 2010b)**; and
- A brief discussion of related work by **Boerkoel, Planken, Wilcox, and Shah (2013)** and **Ten Thije, Planken, and De Weerdt (2011)**.

1.3 Guide to the reader

For the reader who wants to quickly make use of our toolkit for simple temporal reasoning, we suggest the following minimal recipe.

In Chapter 2, start by reading the primer on STNs. It is important to understand the difference between the nine simple temporal queries, but the complexity results can be glossed over. Then, from Chapter 3, try to grasp the ideas behind chordality, triangulation, and the three variations of network-wide path consistency. This should give enough background to read and understand each of the chapters 4, 5, and 6, except for technical details of proofs.

CHAPTER 2

Problem statement and complexity

As discussed in the previous chapter, Dechter et al. (1991) proposed the Temporal Constraint Satisfaction Problem (TCSP), thus extending the literature on constraint programming to the temporal domain. Temporal variables (or time points) can assume values from $\mathbb{R}$; temporal constraints take the form of a union of intervals and constrain exactly two variables. For a constraint to be satisfied, the (time) difference between the two variables involved must lie within one of its intervals.

In this chapter — and in the rest this dissertation — we are concerned with a restriction of the TCSP where each constraint takes the form of exactly one (possibly unbounded) interval instead of a union of intervals. This restriction, called the Simple Temporal Problem (STP), was also introduced by Dechter et al. and they showed it to be tractable: algorithms that compute shortest paths, such as the $\Theta(n^3)$-time Floyd–Warshall algorithm, can be brought to bear on it.

In a nutshell, this dissertation investigates properties of the STP and gives an inventory of efficient algorithms for solving it; it also proposes several new algorithms for this purpose. These new algorithms, as well as extant algorithms readily available from the literature, are of several different types: they do not all perform the same task nor produce the same output. Apparently, they do not solve exactly the same problem, and yet they are all presented as algorithms for the STP. Thus, before we can start thinking about algorithms, it is a good idea to determine precisely what problem we want them to solve, so:

What is the Simple Temporal Problem?

It is our opinion that a clear answer to this fundamental question is not given in the literature. Therefore, we study it in this chapter. For this, we first need a formal definition of "problems" as they are studied in computer science.

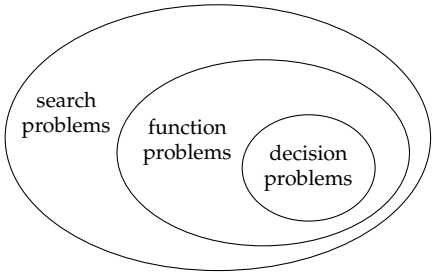


Figure 2.1: Three types of computational problem

Formally, a computational problem is defined as a set of pairs of *instances* and *answers* (Johnson, 1990). We solve a problem if, when presented with any instance, we know how to produce a valid answer for it:

> **How to solve a problem** *In everyday speech we often talk about "solving" particular instances of problems, as in solving a crossword puzzle. Complexity theorists, however, do not consider a problem X solved unless they have a general method that will work for* any *instance (assuming enough time, memory, and other resources are provided). In practice, such methods may be usable only for a small finite set of instances, given physical bounds on the resources available. Because the methods are general, however, they will automatically let us solve larger instances should the amount of available resources ever increase.* —Johnson, 1990, p. 71

All problems that fit the general definition of a problem as a set of instance–answer pairs constitute the set of *search problems*. If every instance is paired with exactly one answer, we are talking about *function problems*. The most restricted type of problem is the *decision problem*, which is a function problem where the answer is either "yes" or "no". The Euler diagram in Figure 2.1 illustrates the inclusion relations between these three problem types.

Temporal constraint satisfaction has a clear interpretation in terms of a formal computational problem. The TCSP's instances were already briefly discussed above, and with respect to the type of answers we might be looking for, we usually consider two closely related versions: the search problem FIND SCHEDULE, which looks for an assignment of a specific time to each temporal variable such that all constraints are satisfied, or the analogous decision problem CONSISTENCY that asks whether any such schedule exists at all. From an algorithmic perspective, one could say that it does not matter very much which variant we choose to solve. The complexity of both problems is NP-complete*, which has two important implications: (i) any approach to solve either of them is presumed to require more than polynomial time; and (ii) the answer for one of these problems can be

*Strictly speaking, this is an abuse of terminology and only decision problems can be called NP-complete. Search and function problems of the same complexity are instead called NP-equivalent. For simplicity, however, we ignore this distinction. See also the footnote on page 21.

translated into an answer for the other with only polynomial extra cost. As the size of problem instances increases, the relative cost of this transformation rapidly becomes cheaper. Finally, it should be noted that since the fundamental task of determining consistency of a TCSP instance is already NP-hard, any interesting query that builds on consistency must also be.

For the specific case of the STP, matters are different. Although its instances, which we discuss in more detail in Section 2.1, are just restricted versions of the TCSP, it is not so clear what shape the answers must take. For one thing, both FIND SCHEDULE and CONSISTENCY are now in P: Dechter et al. (1991) showed how they can be solved in polynomial time. We cannot then ignore a polynomial-time cost for translating an answer between them. Another issue is that in practice, one is often interested in "bigger" STP queries than these two. This is evident from the fact that the venerable $\mathcal{O}(nm)$-time Bellman–Ford algorithm solves both, and yet less time-efficient algorithms are routinely proposed for and applied to the STP. Clearly, they are of interest because they can solve other, harder "simple temporal problems".

In this chapter, we first present the Simple Temporal Network, which forms (part of) the instance for all Simple Temporal Queries (STQs) we subsequently present. We formally define each such STQ in our inventory and determine its theoretical complexity. Thus, the stage is set for the remainder of the dissertation, where we investigate properties of the STP and algorithmic techniques that can be applied to it.

2.1 The Simple Temporal Network

In their seminal 1991 paper, Dechter et al. proposed temporal constraint networks to function as the knowledge base in a temporal reasoning system. Along with this knowledge base, such a system should then also provide methods to reason about the information represented by it. That is, it must be possible to determine whether the temporal information is consistent, to answer queries about the information, and to derive new information. In this section, we define a simple form of such a temporal constraint network and discuss its properties. This gives the basis for the the next section, where we formally state Simple Temporal Queries that can be asked of the knowledge base to form the type of temporal reasoning system proposed by Dechter et al..

Most of the concepts and results presented below are taken from the original article by Dechter et al. (1991). Toward the end of the section, we present some new results, which are clearly marked as propositions.

Temporal variables and constraints A network of temporal constraints is defined by a set $X = \{x_1, \ldots, x_n\}$ of continuous *temporal variables* or *time points* and a set C of m binary *temporal constraints*. A temporal variable represents an event occurring at some point in time; often, a pair of them together represent the start and end of some activity. Each temporal constraint then limits the temporal distance between exactly two time points. In the Simple Temporal Network (STN),

such a constraint $c_{ij} \in C$ takes the form of a simple interval $[lb, ub] \subseteq \mathbb{R}$, restricting the time that may elapse between x_i and x_j by requiring $x_j - x_i \in c_{ij}$. Thus, if x_i and x_j represent the start and end time of some activity, c_{ij} represents a constraint on the duration of this activity.

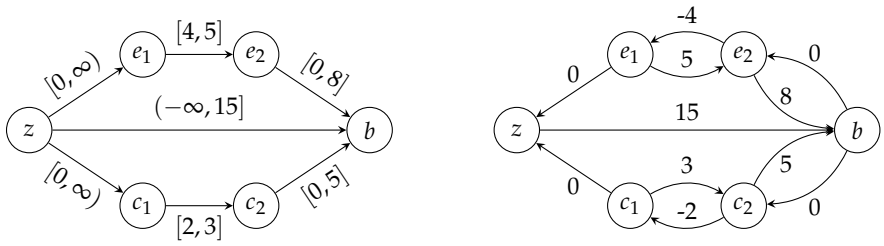
Each constraint $c_{ij} = [lb, ub]$ is equivalent to its reverse $c_{ji} = [-ub, -lb]$. To get rid of this ambiguity, we often instead specify two upper bounds, one in each direction, defining the pair of *constraint weights* $w_{ij} = ub$ and $w_{ji} = -lb$. Instead of a single interval, we then have a pair of linear inequalities $c_{ij} : x_j - x_i \leq w_{ij}$ and $c_{ji} : x_i - x_j \leq w_{ji}$. This illustrates the convenient assumption that each constraint corresponds to a single inequality, defining only an upper bound. When discussing STNs, it is customary to only consider closed intervals and non-strict inequalities, as above, but Dechter et al. note that the same treatment applies to (semi-)open intervals and strict inequalities.* Intervals may extend to (negative) infinity, implying that one or both of the bounds are unconstrained; the interval $(-\infty, \infty)$ represents the *universal constraint* and can be added or removed anywhere without consequence. Constraint weights are taken from the set $\mathbb{R} \cup \{\infty\}$; again, an infinite upper bound can be omitted. Ignoring universal constraints, we always assume the network to be connected. This assumption can be made without loss of generality: if it does not hold, each connected component forms a completely independent subnetwork.

To be able to also refer to absolute time instead of just the temporal distance between events, we identify a *temporal reference point* $z \in X$ that represents an epoch: some agreed-upon moment of synchronization. Together with this choice of z it must then also be decided what unit of time is used. Two real-life examples are "Unix time" with its epoch at midnight UTC on 1 January 1970 and each unit representing a second, and an Anno Mundi chronology which counts years since the creation of the world. In practice, one usually selects the start of the current day or the start of the events under consideration, with some convenient time unit. Whatever time the temporal reference point represents, it is often useful to interpret it as the *origin of time*, explicitly constrained to occur before any other event. Sometimes a *horizon* time point is also specified, which is then likewise constrained to occur after all other events.

Schedules and consistency A *schedule* $\tau \in \mathbb{R}^X$ for an STN is an assignment of a value from $\mathbb{R}$ to each time point in X such that all constraints are satisfied.† Note that any schedule τ can be transformed into a new schedule τ' by performing a translation in time, i.e. adding a value $d \in \mathbb{R}$ to each element in τ. Therefore, we usually only use normalised schedules, where the temporal reference point z is fixed to the value $\tau_z = 0$. An STN is called *consistent* if it admits at least one schedule. Let us now present an example to illustrate the concepts introduced so far.

*Any strict inequality $x - y < c$ can be restated as $x - y \leq c - \epsilon$, for some small number $\epsilon > 0$.

†In the literature, τ is then traditionally called a "solution" for the STP instance. Since our purpose is to distinguish between a variety of solution concepts, as discussed in Section 2.2, we prefer to avoid this term.



(a) Constraint network for $\mathcal{S}$

(b) Distance graph for $\mathcal{S}$

Figure 2.2: Example STN $\mathcal{S}$: preparing breakfast

Example. *We face the problem of having breakfast. Our breakfast requires some preparation: we want to make coffee and boil eggs. Both of these activities are represented by their start and end time points: e_1 and e_2 for the task of boiling the eggs, which takes between 4 and 5 minutes, and c_1 and c_2 for making coffee, which requires 2 to 3 minutes. We start preparing our meal at 9 AM, marked by the temporal reference point z, and b takes the role of horizon, denoting the moment when both coffee and eggs are done and we sit down to enjoy breakfast. We do not want our coffee or eggs to get cold, so we constrain the duration of time from e_2 and c_2 to b to at most 8 and 5 minutes, respectively. Since everybody is a little hungry, we want to be done within 15 minutes.*

The constraint network $\mathcal{S}$ for this example is depicted in Figure 2.2a. The reader can verify that the assignment $\tau = \langle z{=}0,\ e_1{=}1,\ c_1{=}3,\ e_2{=}5,\ c_2{=}6,\ b{=}10\rangle$ is a (normalised) schedule for our example STN, which therefore is consistent.

To avoid clutter in this example, we only specified a few explicit constraints involving the origin and horizon, but the rest of the constraints ensure that all time points occur between these boundaries. In practice, this is often specified explicitly. The constraint between the origin z and the horizon b is an example of the common practice to require that all schedules fit in a specific time window. This approach can be used to search for a schedule with minimal makespan, i.e. earliest completion time. Chapter 6 features an example of this.

Distance graph The direction of constraint edges in the network in Figure 2.2a is arbitrary. For example, instead of the directed edge (e_1, e_2) labelled with $[4, 5]$ we could just as well depict the reversed edge (e_2, e_1) and label it with the interval $[-5, -4]$ to get the same set of constraints, although the direction of edges indicates temporal precedence by convention. An equivalent representation of the STN that we will use extensively in this work is a directed weighted graph called the *distance graph*; see Figure 2.2b. Each vertex $u \in V$ of the distance graph coincides with a variable $x \in X$; we write $u \equiv x$, but often also use them interchangeably. For each constraint $c_{xy} \in C$ we have a pair of directed edges in both directions between vertices $u \equiv x$ and $v \equiv y$. We then have a *weight*

function $w \in \mathbb{R}^{V \times V}$ that, given vertices $u, v \in V$, yields an edge weight $w_{uv} \in \mathbb{R}$ that corresponds to the upper bound on the time difference between the involved variables. For example, note that between vertices e_1 and e_2 we now have two directed edges with labels of 5 and -4. Edges labelled by infinity can be omitted, and often are: an infinite upper bound is the same as no upper bound at all. Conversely, if for a pair of vertices u and v the corresponding temporal variables are unconstrained, we have $w_{uv} = w_{vu} = \infty$.

The distance graph is a convenient tool when reasoning about the STN. If we add the constraints involving the origin and horizon mentioned above, we further ensure the useful property that the distance graph is strongly connected, i.e. that from every vertex, there is a path (with finite total weight) to every other vertex. Since an origin and horizon time point together with a suitably large time window can be added to any STN, we usually assume that the distance graph has this property. Like above, this assumption can be made without loss of generality: if it does not hold, the network decomposes into a set of strongly connected subnetworks which can be topologically ordered and considered independently. A schedule for the full STN can then be merged from any set of schedules for the subnetworks by placing them far enough apart in time.

Constraint composition To derive new information from a constraint network, we can perform a walk and take the *composition* of the constraint edges we traverse. For example, in our example STN as depicted in Figure 2.2a, there are three paths from z to b. The bottom path corresponds to the following expression, where $\otimes$ denotes the composition of constraints.

$$0 \leq c_1 - z \leq \infty \quad \otimes \quad 2 \leq c_2 - c_1 \leq 3 \quad \otimes \quad 0 \leq b - c_2 \leq 5 \tag{2.1}$$

The composition of a set of inequalities along a path can be found conveniently by summing them, which in this case results in a new inequality $2 \leq b - z \leq \infty$. Thus, we find a tighter lower bound for the distance between z and b than the lower bound of $-\infty$ given by the constraint c_{zb}. Conversely, c_{zb} gives a tighter upper bound than the infinite bound implied by Expression 2.1.

The same bounds are found by performing this walk in the distance graph from Figure 2.2b and computing the sum of the edge weights. Here, the infinite upper bound follows from the fact that b is not reachable from z along c_1 and c_2, whereas the summed edge weights along the path (b, c_2, c_1, z) yield the negated inequality $z - b \leq -2$. A useful fact stated by Dechter et al. is that an STN is consistent if and only if its distance graph contains no *negative cycle* (i.e. a cycle whose summed edge weights are negative). We see now why this is true: let x be any time point represented by a vertex in the cycle and let $d < 0$ be its summed weight. We then get $x - x \leq d < 0$, a contradiction that cannot be satisfied by any schedule. Conversely, this implies that if all constraint weights are nonnegative, the STN must be consistent. Indeed, assigning the value 0 to all time points then yields a valid schedule.

Constraint minimality The *minimal constraint* c^*_{xy} between a pair of time points $x, y \in X$ is the tightest constraint on the time difference between x and y implied by the STN. Dechter et al. showed that we can find c^*_{xy} by considering all paths from x to y (including the path formed by the direct constraint edge c_{xy}) and intersecting the compositions of constraints along each of these paths. In our example network $\mathcal{S}$, we already found that $c_{zc_1} \otimes c_{c_1c_2} \otimes c_{c_2b} = [2, \infty)$. For the top path, we find that $c_{ze_1} \otimes c_{e_1e_2} \otimes c_{e_2b} = [4, \infty)$. Thus we find a minimal constraint $c^*_{zb} = (-\infty, 15] \cap [2, \infty) \cap [4, \infty) = [4, 15]$. Naturally, a minimal constraint must still be satisfied in any schedule for the STN. However, the converse is now also true: for any temporal distance $d \in c^*_{xy}$ allowed by a minimal constraint c^*_{xy}, there exists a schedule τ in which $\tau_y - \tau_x = d$, so y is scheduled by τ to occur exactly d time units after x. Put differently, were we to tighten c^*_{xy} any further, we would invalidate some schedules.

Constraint minimality, like composition of constraints, also comes with a natural analogue in the distance graph representation. In this case, the analogy is very useful: for vertices $u \equiv x$ and $v \equiv y$, the upper bound imposed by the minimal constraint c^*_{xy} is exactly equal to the *shortest distance* $\omega_{uv} \in \mathbb{R}$ from u to v in the distance graph. This can be verified in the distance graph depicted in Figure 2.2b, where the shortest distance from z to b is 15, and the shortest distance from b to z is -4. Note that the shortest distance is well-defined for consistent STNs, where negative cycles are absent.

Minimal network The network consisting of minimal constraints between all pairs of variables is called the *minimal network* and forms a complete graph. Dechter et al. showed that each consistent STN admits a unique minimal network. Since each constraint has been tightened as much as possible without invalidating any schedules, any schedule for the original STN is still a schedule for the minimal network. Depicting the constraint network or distance graph of a minimal network usually results in too much clutter, but a clear and equivalent representation is the matrix $\omega \in \mathbb{R}^{V \times V}$ of shortest distances. In Table 2.1a, we give this matrix for our example STN.

As will become clear below, having the shortest distances ω makes many problem queries easier to answer. For example, Dechter et al. already noted that given a temporal reference point $z \in X$, two special schedules can be easily found from ω. They correspond to the row for z and the negation of the column for z, assigning respectively the latest and earliest possible time value to each temporal variable. Table 2.1b lists schedules for the example STN and includes these special ones as τ^1 and τ^2, respectively. For our example STN $\mathcal{S}$ with temporal reference point z, we see that we have $\tau^1_x = \omega_{sx}$ and $\tau^2_x = -\omega_{xs}$ for all $x \in X$. As Dechter et al. stated, since the shortest distances ω are well-defined* for any

*A technicality that was not addressed by Dechter et al. was the possible presence of infinities in ω, in particular in the row and column for the temporal reference point z. However, if z is constrained to occur before any other time point x as mentioned above, the shortest distance ω_{xz} for all such x is guaranteed to be finite and z's column in ω represents a well-defined schedule. Even more effective is to also add a horizon time point and constrain the overall time window to some

	to					
from	z	e_1	c_1	e_2	c_2	b
z	0	11	13	15	15	15
e_1	0	0	11	5	13	13
c_1	0	4	0	8	3	8
e_2	−4	−4	6	0	8	8
c_2	−2	1	−2	5	0	5
b	−4	−4	−2	0	0	0

(a) Shortest distances ω in $\mathcal{S}$

	z	e_1	c_1	e_2	c_2	b
τ^1	0	11	13	15	15	15
τ^2	0	0	0	4	2	4
τ^3	0	4	0	8	3	8
τ^4	−13	−11	0	−6	2	2
τ^5	0	2	13	7	15	15

(b) Some schedules for $\mathcal{S}$

Table 2.1: Schedules and minimality for the example STN $\mathcal{S}$

STN without negative cycles, this proves that absence of such negative cycles guarantees consistency.

What happens if we take a row or column other than those corresponding to the temporal reference point? As it turns out, we still wind up with a valid schedule, although it is not guaranteed to be normalised. In Table 2.1b, τ^3 and τ^4 correspond to the row and negated column for c_1 from the distance matrix in Table 2.1a. The latter is not normalised; its normalised equivalent τ^5 is found by adding 13 minutes to each value in τ^4. In our example, the row schedule τ^3 corresponds to the situation where we start making coffee immediately, and then schedule all other time points as late as possible. For τ^5 (and τ^4) the situation is reversed: we delay making the coffee, but we schedule all other events as early as we can. We can go on like this and find 12 different schedules from Table 2.1a, which gives evidence for the following new result.

Proposition 2.1. *Given an STN with n time points, the rows and columns in its matrix of shortest distances correspond to up to $2n$ different schedules.*

This maximum is not always attained, however; see Table 1 in the original example by Dechter et al. (1991) for a case where only n schedules are actually distinct and every row schedule matches a column schedule.

Convexity Besides the two equivalent interpretations as a constraint network and as a distance graph, recall that the STN is also simply a set of linear inequalities. Given an STN $\mathcal{S} = \langle X, C \rangle$ and substituting the value 0 for all occurrences of the temporal reference variable $z \in X$, these inequalities define a convex polytope $P \in \mathbb{R}^{X \setminus \{z\}}$. This polytope is akin to the feasible region known from linear programming. The following result now holds.

suitably large value. This ensures that the distance matrix ω contains no infinities at all.

Proposition 2.2. *Given an STN $\mathcal{S} = \langle X, C\rangle$ with temporal reference point $z \in X$ and the convex polytope P defined by its inequalities where 0 is substituted for z, P coincides exactly with all normalised schedules for $\mathcal{S}$.*

Another consequence of the convexity of this set of linear inequalities is the following.

Proposition 2.3. *Let $Conv(T)$ denote the convex hull of the set T. Given an STN $\mathcal{S}$ and a set T of schedules for $\mathcal{S}$, any $\tau \in Conv(T)$ is a schedule for $\mathcal{S}$.*

This means that, given a set T of schedules, we can take a convex combination of them (in other words, a weighted average) to obtain a new schedule. As an example, let us combine schedules τ^1 to τ^3 from Table 2.1b and use the index of each schedule as its relative weight, so $(\tau^1 + 2\tau^2 + 3\tau^3)/6$. This yields the assignment $\tilde{\tau} = \langle z{=}0'00'', e_1{=}3'50'', c_1{=}2'10'', e_2{=}7'50'', c_2{=}4'40'', b{=}7'50''\rangle$, which is readily verified as a valid schedule.

This concludes our discussion of the Simple Temporal Network and concepts defined on it. Next, we present an inventory of Simple Temporal Queries, all of which take at least an STN as their input; many of these follow logically from the concepts discussed in this section.

2.2 Simple Temporal Queries

Recall from the introduction to this chapter that a computational problem is formally defined as a set of pairs of instances and answers, and that we have solved the problem when we know how to produce an answer for any given instance. In the previous section, we defined the Simple Temporal Network and concepts relating to it. Now, in this section, we investigate what questions can be asked given an STN. Thus, the main contribution of this chapter is to give an inventory of formal Simple Temporal Queries (STQs), for each of which the "instance" part consists at least of an STN, sometimes with a little extra input.

All STQs that we list can be solved in polynomial time — they are thus members of P — and the $\mathcal{O}(n^3)$-time Floyd–Warshall algorithm that was already mentioned briefly in the introduction can be applied to answer each of them with $o(n^3)$ extra work. This is convenient and leads to the temptation to ignore the finer differences between them, which may be completely fine if a query is asked just once or twice. However, in practice this is often not the case and queries are asked many times. It then pays to look for faster approaches.

Another risk of neglecting differences in complexity between the STQs is that a general, "wide" query is asked when in fact only the answer to a more restricted query is required. This is particularly the case with such STQs that can be answered efficiently by means of some of the algorithms discussed later in this dissertation, which were not yet available when the STN was first proposed. Thus, one can select exactly the queries of interest from the list provided below, and then make an informed choice among the available algorithms.

We also study for the first time* the formal computational complexity of each of the STQs we list. We show that many STQs are contained in strict subclasses of P and also give lower bounds in the form of hardness results. This gives a ranking of the queries' difficulty, but it also tells us something about whether they are amenable to a parallelised approach or can be solved in limited space.

The complexity of computational problems depends on the amount of time or space used, expressed in terms of the size of an efficient encoding $|I|$ of a problem instance I. Thus, before we can determine the complexity of STQs, we must first give a tight bound on the size of an efficient encoding of an STN. Now, given an STN $\mathcal{S} = \langle X, C \rangle$, let w_{max} be the largest absolute, finite constraint weight we need to represent, so:

$$w_{max} = \max \left\{ \left| w_{xy} \right| \mid x, y \in X \wedge w_{xy} < \infty \right\} \tag{2.2}$$

Recalling that m and n respectively denote the number of constraints and time points in $\mathcal{S}$, an efficient encoding then gives $|I| \in \Theta(m \log w_{max})$; since we assume the network to be connected, we have $m \geq n - 1$. Note further that the constraint weights are assumed to be integer-valued; rational weights can easily be recast in this form. Real-valued weights are outside the scope of this discussion; Schwalb and Dechter (1997) note that rational weights always suffice in practice.

If w_{max} is polynomially bounded in the number of constraints and time points, so $w_{max} \in \mathcal{O}(n^c)$ for some constant $c > 0$, the logarithmic factor disappears and we get $|I| \in \Theta(m)$.† We make this assumption throughout the discussion of the complexity of the STQs. In other words, we assume what is called the *uniform cost model* of computation, where arithmetic operations of numbers take constant time regardless of their magnitude. This assumption is certainly justified for polynomially bounded w_{max}; however, we briefly return to the issue of large constarint weights below, on page 24. Let us now give an overview of the complexity classes we consider (see e.g. Sipser, 1996, Chapters 8 and 10).

The class NC contains all problems solvable in parallel, requiring polylogarithmic time and a polynomial number of processors. Thus, these are problems that can be solved much faster in parallel than by any sequential algorithm requiring polynomial time $\Omega(|I|^c)$ for $c > 0$. This class can be parameterised by an integer $k \geq 0$ such that for NC^k, time is bounded to $\mathcal{O}(\log^k |I|)$. In particular, we consider the classes NC^1 and NC^2.

L is the class of problems that can be solved by a Turing machine using only logarithmic working space $\mathcal{O}(\log |I|)$. Note that this limit does not apply to input and output, which is modelled with read-only and write-only memory. The class NL has the same space limits but allows nondeterminism. A known fact that was proved independently by Szelepcsényi (1987) and Immerman (1988) is that $\mathsf{NL} = \mathsf{coNL}$. Within this logarithmic space, one can keep a constant number of

*Some of these results were also included in the author's Master's thesis (Planken, 2008a).

†Throughout this dissertation, we write $x \in \mathcal{O}(f(n))$ instead of the more common $x = \mathcal{O}(f(n))$. Formally, the right-hand side represents the set of all functions that grow no faster than $f(n)$, and the traditional equality in fact only works in one direction (see also Graham et al., 1989, Section 9.2).

Likewise, we use set notation for $\Theta(\cdot)$, $\Omega(\cdot)$, $o(\cdot)$ and $\omega(\cdot)$.

pointers into the input and polynomially-bounded integers in memory, as well as a logarithmic number of single-bit flags.

It is known that the following inclusions hold:

$$\mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{NC}^2 \subseteq \mathsf{NC}^3 \subseteq \cdots \subseteq \mathsf{NC} \subseteq \mathsf{P}$$

Although none of these inclusions is known to be strict, all are presumed to be. It must be noted that even NC^1 has further subclasses; however, for our purposes in this chapter, when we show membership of a problem query in NC^1 we look no further. Complete problems are known for NL and P, and one of each is presented below. Problems that are P-complete are often characterised as being *inherently sequential*. By this, it is meant that the polynomial lower bound on their time complexity (presumably) cannot be escaped through parallelisation. Likewise, they presumably require a polynomial amount of working memory to solve.

We are now ready to present the Simple Temporal Queries (STQs), bearing in mind that one must divorce discussion of computational problems from a presentation of approaches to solve them. We only present algorithms here if they are used to establish membership of a problem in a class, and where we do this, such algorithms are often of a somewhat abstract nature: they exhibit nondeterminism or assume a theoretical model of parallel computing.

In the following, the STQs are split into four categories, each presented in its own section. First, we consider problems relating to the basic problem of determining whether the information in an STN is consistent. Then, assuming consistency, one can look for a schedule: an assignment of values to temporal variables that satisfies all constraints. Next is the orthogonal problem of tightening temporal constraints to be minimal: reducing the constraint interval to its smallest subset without invalidating any schedules. The final category contains just one query, which turns out to be the hardest among the problem variants that we discuss: it asks for the construction of any valid schedule.

Consistency

The decision problem CONSISTENCY asks whether a given STN is consistent, i.e. whether it admits any valid schedules. As was shown in Section 2.1, this problem is equivalent to determining that the distance graph does not contain any negative cycles.

Simple Temporal Query 1: CONSISTENCY

Type: Decision problem.

Instance: An STN $\mathcal{S} = \langle X, C \rangle$.

Question: Is $\mathcal{S}$ consistent? That is, does there exist a schedule $\tau \in \mathbb{R}^X$ having for all $c_{xy} \in C$ that $\tau_y - \tau_x \in c_{xy}$?

Let us now derive the complexity of this problem, which is arguably the most fundamental query with respect to the STN.

Algorithm 2.1: Nondeterministic algorithm for INCONSISTENCY

Input: An STN with distance graph $G = \langle V, E \rangle$
and constraint weights $w \in \mathbb{R}^{V \times V}$
Output: ACCEPT or REJECT

```
1  choose s ∈ V
2  u ← s
3  sum ← 0
4  repeat
5  |  if u has no outgoing edges or |sum| ≥ n w_max then return REJECT
6  |  choose an outgoing constraint edge (u, v) ∈ E
7  |  sum ← sum + w_{u→v}
8  |  u ← v
9  until u = s
10 if sum < 0 then return ACCEPT else return REJECT
```

Theorem 2.4. CONSISTENCY *is in* NL.

Proof. Algorithm 2.1 provides a nondeterministic approach to solve the inverse problem INCONSISTENCY in logarithmic space; the desired result then follows from the fact stated above that NL = coNL.

The algorithm chooses a starting vertex s and repeatedly chooses an edge to walk from the current vertex until it returns to s. It makes these choices nondeterministically, and it rejects if it winds up in a dead end. In this manner it can select any closed walk (with possible vertex repetitions) and in particular any cycle in the STN.

During its walk, the algorithm sums the weights of the constraint edges it traverses. Recall that w_{max}, defined in Equation 2.2, denotes the largest absolute constraint weight encountered. Now, the absolute weight of any cycle in the STN (with finite weight) is at most nw_{max}. If this threshold is exceeded, either an infinite edge has been traversed or at least one constraint edge has been traversed twice, and the algorithm rejects. When the original vertex is reached again, the algorithm checks whether the total weight is negative, accepting if and only if this is the case. The STN must then contain a closed walk with negative total weight, which is the case if and only if it contains a negative cycle. Conversely, if the STN contains a negative cycle, there exists an accepting path of computation for the algorithm.

The algorithm can deal with constraints specifying strict inequalities and (semi-)open intervals in a way similar to the approach described in the footnote on page 10. When the algorithm traverses a strict constraint edge, it sets an "epsilon flag" to record this. Upon reaching the original vertex, it must then also accept if the epsilon flag is set and the sum is equal to zero.

The working space for the algorithm is limited to $\mathcal{O}(\log n)$. The three pointers s, u, v and the epsilon flag satisfy this constraint; we now show that the value of the variable *sum* maintained by the algorithm does as well. The sum has a

threshold of nw_{max} and can thus be represented in $\lceil \log nw_{max} \rceil$ bits of memory space. Recalling our assumption that $w_{max} \leq n^c$ for some $c > 0$, this simplifies to $\lceil \log n^{c+1} \rceil \in \mathcal{O}(\log n)$. We conclude that the space requirements are met.

Hence, if the constraint weights are thus bounded, INCONSISTENCY is a member of NL. This result then transfers to CONSISTENCY since NL = coNL. □

This gives an upper bound on the complexity of STQ 1. Next, we look at the lower bound.

Theorem 2.5. 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?

As in the previous proof, we first show NL-hardness for the inverse problem INCONSISTENCY by providing a reduction from ST-CONNECTIVITY. Note that for this reduction to be valid, it must use logarithmic memory space.

We transform every vertex into a time point and we associate every arc $(u, v) \in A$ with a constraint c_{uv} enforcing the upper bound $w_{uv} = 0$. This leads to a trivially consistent STN, which admits at least the schedule that assigns the value 0 to all time points. Finally, we add the constraint c_{ts} enforcing the upper bound $w_{ts} = -1$; if a constraint between t and s already existed, it is replaced. This transformation requires but a single pointer into the original problem instance and thus clearly satisfies the memory constraints.

Now, it holds that the STN 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, composition in the STN of the constraint edges along this path yields $t - s \leq 0$. The constraint edge between t and s additionally enforces $s - t \leq -1$, yielding inconsistency.

($\Rightarrow$) Conversely, if the STN resulting from the transformation is inconsistent, it must have a negative cycle. This cycle must incorporate the constraint edge from t to s, because this is the only one with negative weight; but then the shortest distance from s to t must also be smaller than 1, which can only be the case if there was a path from s to t in G.

We conclude that INCONSISTENCY is NL-hard, and since NL = coNL, also that CONSISTENCY is NL-hard. □

Theorems 2.4 and 2.5 together give the following result.

Corollary 2.6. CONSISTENCY *is* NL*-complete.*

Therefore, we can decide quite efficiently whether a given collection of simple temporal information is self-contradictory or not. Since $\mathsf{NL} \subseteq \mathsf{NC}^2$, a parallel algorithm requiring at most $\mathcal{O}(\log^2 |I|)$ time must exist; however, we must reserve at least a logarithmic amount of working memory for the task because $\mathsf{L} \subseteq \mathsf{NL}$.

Looking at the STN as a temporal knowledge base, a small rewording of STQ 1, which is often encountered in practice and will prove useful later, is called CONSTRAINT COMPATIBILITY. It asks whether a new piece of information is compatible with the knowledge represented in the STN. In other words, we want to determine the consistency of an STN together with a single new constraint.

Simple Temporal Query 2: CONSTRAINT COMPATIBILITY

Type: Decision problem.

Instance: A consistent STN $\mathcal{S} = \langle X, C \rangle$ and an additional constraint $c \notin C$.

Question: Is $\mathcal{S} = \langle X, C \cup \{c\} \rangle$ consistent?

Given the fact that STQ 1 is in NL, it is easy to see that STQ 2 must have the same upper bound. We now show that it is exactly as hard.

Theorem 2.7. CONSTRAINT COMPATIBILITY *is* NL*-complete.*

Proof. Membership in NL follows from the fact that this problem can be seen as a restriction of CONSISTENCY. For hardness, we can use the same reduction as above, setting the single constraint edge with weight -1 as c' and letting C contain all other constraints. The instance is trivially consistent without c' and the result follows. □

Although it is useful to know that a temporal knowledge base contains no contradictions, the applicability of these STQs remains somewhat limited. Next, we look at problem queries concerning schedules for the STN.

Schedules

Given an STN, one of the original queries listed by Dechter et al., and the natural extension of the CSP search problem to the temporal domain, is to find a schedule or report that no schedules exist. Note that since this is a search problem, any schedule suffices, provided that it is valid.

Simple Temporal Query 3: FIND SCHEDULE

Type: Search problem.

Instance: An STN $\mathcal{S} = \langle X, C \rangle$.

Question: Find a schedule $\tau \in \mathbb{R}^X$ having for all $c_{xy} \in C$ that $\tau_y - \tau_x \in c_{xy}$, or report that no such schedule exists.

This query implicitly answers STQ 1 and is therefore NL-hard. But, since it is in fact a subproblem of STQ 6 below, which we show to be in NL, we conclude the following.

Corollary 2.8. FIND SCHEDULE *is* NL*-complete.*[*]

For the next query, recall from the introduction to this chapter that deciding consistency of the more general Temporal Constraint Satisfaction Problem (TCSP) is NP-complete. As a consequence, its certificates can be checked in polynomial time. These take the form of schedules, and the existence of a tractable method to validate them naturally extends to the easier STN. In fact, this is a separate STQ: is a given schedule consistent with the information represented by the temporal network?

Simple Temporal Query 4: VALIDATE SCHEDULE

Type: Decision problem.

Instance: An STN $\mathcal{S} = \langle X, C \rangle$ and an assignment $\tau \in \mathbb{R}^X$.

Question: Is τ a schedule for $\mathcal{S}$?
That is, does it hold for all $c_{xy} \in C$ that $\tau_y - \tau_x \in c_{xy}$?

This problem can be solved by checking all $\mathcal{O}(m)$ linear inequalities in parallel. Each can be checked with a constant number of additions and comparisons. Since addition and comparison are in NC^1 (Karp and Ramachandran, 1990), we conclude that STQ 4 is in this class as well, which makes it strictly easier than any of the other STQs so far. As an aside, note that this result extends to the TCSP as well: after the additions and comparisons, we can compute the required polynomial number of logical disjunctions within the same bounds.

Proposition 2.9. VALIDATE SCHEDULE *is in* NC^1 *(for any TCSP instance).*

Note that this result gives an upper bound on the complexity of the query, but no lower bound. However, as indicated above, in this chapter we look no further than NC^1; this result simply ranks VALIDATE SCHEDULE among the easiest problems we consider.

Finally, one may wonder: can we also ask for the set of all schedules for a given STN? Of course, this set is generally infinite. But recalling that the linear inequalities that make up the STN define a convex polytope P, we can ask for a finite representation of P instead. Any polytope defines a set of *polytope vertices* or corner points; note that this term must be distinguished from the graph vertices mentioned earlier in this chapter. Each polytope vertex of P describes an extreme point in the space of all schedules for the STN: the value of each temporal variable by itself is either minimal or maximal. As described in Section 2.1, a convex combination or weighted average of these extremes can then be easily computed to find any schedule for the STN, so a query that asks for the full set might seem interesting.

[*]For simplicity, we permit ourselves this abuse of terminology. Formally, the classes NL, NC, and P contain only decision problems and the equivalent classes for function and search problems are FNL, FNC, and FP, respectively. See also the footnote on page 8.

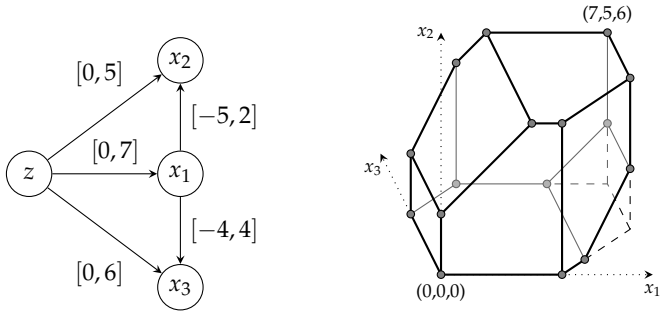


Figure 2.3: An STN with 4 time points and the corresponding 3-dimensional convex polytope of schedules. This polytope has 16 vertices out of a maximum of 20. Omitting the upper bound of 5 on the distance between x_1 and x_2 would add the dashed region.

However, enumerating all polytope vertices is usually not a good idea, since in general, their number is exponential.* We must therefore make do with a more implicit and terse representation of P. Of course, each STP instance already implicitly defines P, but we have access to a canonical representation: the minimal network defined in Section 2.1. Hence, we now move on to STQs that have to do with minimality. In Section 4, we state a problem query that can ask for a specific polytope vertex of P.

Constraint minimality

Two of the queries posed by Dechter et al. were (i) "Given some event x, at what times can it occur?" and (ii) "Given a pair x, y of events, what are the bounds on the amount of time that can pass between them?" These are function problems: they have a unique correct answer. Recalling the temporal reference variable z introduced in Section 2.1, we see that they are in fact two formulations of the same STQ: (i) is a special case of (ii) with x, z as variable pair. We state a variant of the query that asks for only the upper bound on the temporal distance. This suffices if we recall that the lower bound on the difference between x and y is equal to the negation of the upper bound on the difference between y and x.

Simple Temporal Query 5: MINIMAL CONSTRAINT

Type: Function problem.

Instance: A consistent STN $\mathcal{S} = \langle X, C \rangle$ and a pair of time points $(x, y) \in X^2$.

Question: Find $\omega_{xy} = \max\{\tau_y - \tau_x \mid \tau \text{ is a valid schedule for } \mathcal{S}\}$, the maximal temporal difference between x and y entailed by the set C of constraints.

*For the STN, the upper bound is apparently $\binom{2n-2}{n-1}$, which is always higher than 2^{n-1} and grows like a power of 4 in the limit. Already for trivial STNs with a strongly connected distance graph and finite distances, the polytope is a hypercube with 2^{n-1} vertices.

Algorithm 2.2: Nondeterministic algorithm for MINIMAL NETWORK

Input: A consistent STN $\mathcal{S} = \langle X, C \rangle$
Output: Minimal distances $\omega \in \mathbb{R}^{X^2}$

1 **foreach** $x, y \in X$ **do**
2 | **choose** $\omega_{xy} \in [-nw_{max}, w_{xy}]$
3 | **if** $c'_{yx} : x - y < \omega_{xy}$ is compatible with $\mathcal{S}$ **then return** REJECT
4 | **if** $c''_{xy} : y - x = \omega_{xy}$ is not compatible with $\mathcal{S}$ **then return** REJECT
5 **end**
6 **return** ω

Computing the minimal network, defined in Section 2.1, corresponds to answering this STQ for all events and pairs of events at once.

Simple Temporal Query 6: MINIMAL NETWORK

Type: Function problem.

Instance: A consistent STN $\mathcal{S} = \langle X, C \rangle$.

Question: Find the minimal distances $\omega \in \mathbb{R}^{X^2}$ for $\mathcal{S}$.

We now show that both of these STQs still have a complexity equivalent to consistency checking.

Theorem 2.10. MINIMAL CONSTRAINT *and* MINIMAL NETWORK *are* NL*-complete.*

Proof. For the hardness result, note that ST-CONNECTIVITY can easily be reduced to MINIMAL CONSTRAINT by a minor variation of the approach used for proving Theorem 2.5; hardness of MINIMAL NETWORK follows from the fact that it is more general. We show that these problems are also contained in NL by means of Algorithm 2.2. For MINIMAL NETWORK, the algorithm iterates over all pairs of time points $x, y \in X$, whereas for MINIMAL CONSTRAINT, lines 2 to 4 need to be executed only once, for the query pair. We now prove the correctness of the algorithm and show that it runs in logarithmic space.

The algorithm nondeterministically tightens the upper bound on the distance between each pair of time points considered. This tightening operation works by choosing a new bound $\omega_{xy} \in [-nw_{max}, w_{xy}]$. In this way, ω_{xy} is at least as tight as the original bound, and there are no universal constraints; both are requirements for the minimal network.* The minimal value in the interval corresponds to a path with lowest possible total weight. As above, the space required for ω_{xy} is $\mathcal{O}(\log n)$ for polynomially bounded weights.

For the constraint to be minimal, ω_{xy} must satisfy two further properties, checked in lines 3 and 4 of Algorithm 2.2:

*An exception to the latter requirement applies if the STN is not strongly connected; if there is no path from x to y in the distance graph, we must set $\omega_{xy} = \infty$. However, because ST-CONNECTIVITY is in NL, the algorithm can check this as a subroutine.

1. The (strict) inverse constraint must be incompatible with the original STN; if this is not the case, this means that ω_{xy} has been made too tight and invalidates some schedules.

2. The constraint specifying a distance of exactly ω_{xy} between x and y must be compatible; otherwise, the bound is too loose.

If both of these are satisfied, ω_{xy} indeed corresponds to the minimal distance and the algorithm moves on to the next pair of time points. Note that since STQ 2 is in NL, constraint (in)compatibility can be checked as often as desired during the solving process.

During operation, at all times only a single constraint weight and the pointers to x and y have to be kept in memory; therefore, under the assumption that w_{max} is polynomially bounded, the space requirement is satisfied. We conclude that MINIMAL CONSTRAINT and MINIMAL NETWORK are NL-complete. □

Here, we briefly address again the matter of dealing with large constraint weights mentioned before on page 16. In the author's Master's thesis (Planken, 2008a) it has been shown that if it cannot be assumed that constraint weights are polynomially bounded in the size of the network, a parallel algorithm is available that returns the minimal network or determines inconsistency. Thus, these queries are then still contained in a class just one step wider than NL.

Proposition 2.11. *For unbounded constraint weights,* CONSISTENCY *(STQ 1) and* MINIMAL NETWORK *(STQ 6) are in* NC^2*, while retaining their* NL*-hardness as a lower bound.*

Proof. Both of these queries can be solved by a parallel algorithm requiring $\mathcal{O}(\log^2 n)$ time and $\mathcal{O}(n^3 \log n w_{max})$ processors (Planken, 2008a). For the hardness result, the reduction used to prove Theorem 2.5 of course still holds. □

This fact naturally extends to STQs 2, 3, and 5, as well as STQ 8 which we come to below. In contrast, the unboundedness of constraint weights has no impact on the simple arithmetic operations required by STQ 4, and STQ 7 below, both of which remain in NC^1, nor on the harder STQ 9 from Section 4.

Consequences of minimality Apart from being a unique, canonical representation of simple temporal knowledge, the minimal network is desirable because some of the queries specified above become strictly easier when their input is restricted to be minimal.

Although STQ 1 becomes trivial since any minimal network is consistent, its restatement as STQ 2 is still very useful. Given an STN $\mathcal{S} = \langle X, C \rangle$ with minimal distances ω and a new constraint $c'_{xy} : [-w'_{yx}, w'_{xy}]$, checking its compatibility with $\mathcal{S}$ is now a simple matter of checking two entries in ω. To see this, recall that for any temporal distance $d \in [-\omega_{yx}, \omega_{xy}]$, there exists a schedule τ for $\mathcal{S}$ with $\tau_y - \tau_x = d$. Thus, if the intersection $[-w'_{yx}, w'_{xy}] \cap [-\omega_{yx}, \omega_{xy}]$ is non-empty, c'_{xy} is compatible with $\mathcal{S}$. For another way to derive this result, note that for $u \equiv x$ and $v \equiv y$, setting the weights of the arcs (u, v) and (v, u) respectively to

w'_{xy} and w'_{yx} must not result in a negative cycle. With the minimal distances ω available, this boils down to verifying that $w'_{xy} + \omega_{yx} \geq 0$ and $w'_{yx} + \omega_{xy} \geq 0$.

It is easy to see that STQ 3 becomes easier with ω available for an STN $\mathcal{S}$: we already showed in Section 2.1 that some specific schedules for $\mathcal{S}$ can be assembled directly from its distance matrix ω. STQ 4 already was very easy, and the others are now in the same class. Recall that once we establish membership in NC^1, we look no further.

Proposition 2.12. *The problems* CONSTRAINT COMPATIBILITY *(STQ 2) and* FIND SCHEDULE *(STQ 3) are in* NC^1 *if minimal distances ω are provided with the STN.*

This result illustrates once more why minimality is a desirable property. Solving the NL problem of MINIMAL NETWORK just once allows for checking constraint compatibility and generating schedules in NC^1 and as often as desired. Note also that this result still holds partially if not the entire distance matrix ω is available. Checking CONSTRAINT COMPATIBILITY is easy for all constraints c'_{xy} for which shortest distances ω_{xy}, ω_{yx} are available. Likewise, FIND SCHEDULE is still in NC^1 if there exists a timepoint $z \in X$ such that shortest distances ω_{yz}, ω_{zy} are known for all $y \in X \setminus \{z\}$.

In upcoming chapters, we will return to the problem of finding this partial distance matrix. Below, we first propose queries that ask for maintaining simple temporal information in the face of changes.

Maintaining minimality As stated above, temporal constraint networks, and the STN in particular, can be seen as a temporal knowledge base; the knowledge represented in an STN may change over time. When we find that a new piece of information is compatible with the current set, we can add it. Conversely, we can retract information from the knowledge base. These problems are trivial in general: we just update the set C of constraints. They become interesting if we want to maintain the STN in its canonical minimal form. We now define two STQs for adding information to and retracting information from a minimal STN.

Simple Temporal Query 7: INCREMENTAL MINIMALITY

Type: Function problem.

Instance: A consistent STN $\mathcal{S} = \langle X, C \rangle$ with minimal distances $\omega \in \mathbb{R}^{X^2}$ and a compatible new constraint $c'_{ab} : b - a \leq w'_{ab}$.

Question: Find the minimal distances $\omega' \in \mathbb{R}^{X^2}$ (as in STQ 6) for $\mathcal{S}' = \langle X, C \cup \{c'_{ab}\} \rangle$.

Simple Temporal Query 8: DECREMENTAL MINIMALITY

Type: Function problem.

Instance: A consistent STN $\mathcal{S} = \langle X, C \rangle$ with minimal distances $\omega \in \mathbb{R}^{X^2}$ and a constraint $c \in C$.

Algorithm 2.3: Parallel algorithm for INCREMENTAL MINIMALITY

Input: An STN $\mathcal{S} = \langle X, C \rangle$ with minimal distances $\omega \in \mathbb{R}^{X^2}$ and a compatible new constraint $c'_{ab} : b - a \leq w'_{ab}$
Output: Minimal distances ω' for $\mathcal{S}' = \langle X, C \cup \{c'_{ab}\} \rangle$

```
1 foreach x, y ∈ X do in parallel
2 |  ω'_xy ← min{ω'_xy, ω'_xa + w'_ab + ω'_by}
3 end
4 return ω'
```

Question: Find the minimal distances $\omega' \in \mathbb{R}^{X^2}$ (as in STQ 6) for $\mathcal{S}' = \langle X, C \setminus \{c\} \rangle$.

The following theorem states that once we have a minimal network, adding a piece of new information to it is easier than computing minimality from scratch.

Theorem 2.13. INCREMENTAL MINIMALITY *is in* NC^1.

Proof. Algorithm 2.3 solves this problem. It performs a constant number of additions and comparisons in parallel for each of the n^2 pairs of time points and can thus be implemented to run in logarithmic time on a parallel computer with a polynomial number of processors. □

Conversely, the problem of maintaining minimality under constraint retraction remains as hard as the problem of establishing minimality in the first place.

Theorem 2.14. DECREMENTAL MINIMALITY *is* NL*-complete.*

Proof. Membership in NL is easily seen, since after removing c from C, Algorithm 2.2 can be directly applied. We sketch a variation of the reduction from ST-CONNECTIVITY$(G = \langle V, A \rangle, s, t)$ used earlier to demonstrate NL-hardness.

Set $X = V \cup \{y_1, y_2\}$ and add a constraint $c_{uv} : v - u \leq 0$ to C for each $(u, v) \in A$, as before. Then, for all $v \in V$, add two constraints $c_{vy_1} : y_1 - v \leq 0$ and $c_{y_2 v} : v - y_2 \leq 0$ to C. Finally, add the constraint $c_{y_1 y_2} : y_2 - y_1 \leq 0$. Now, it holds that the shortest distance $\omega_{x_1 x_2}$ between all pairs of variables in X is zero, since it is easily seen that there is a zero-weight path running through y_1 and y_2 for each such pair. Hence, the STN is consistent. After retracting the constraint $c_{y_1 y_2}$ from the network and re-establishing minimality, we find whether t is reachable from s by checking whether ω'_{st} is still equal to 0. □

Next, we return to schedules as object of interest, instead of minimal constraints. We define a final query that is harder than any of the STQs introduced so far.

Dispatching

It is important to note that the STQs presented above allow one to find only a subset of the schedules for a given STN. Recall from Proposition 2.1 that up to $2n$ different schedules can be retrieved directly from the minimal network; these include the earliest and latest possible schedules. By Proposition 2.3, we can take any convex combination of them, yielding a continuum of schedules in $n-1$ dimensions. However, recall from Proposition 2.2 and the footnote on page 22 that in general, the space of all schedules for a given STN defines a convex polytope with an exponential number of vertices. Thus, the schedules that can be "read off" the minimal network constitute only a small minority of the total number of these corner points. As will become clear in this section, the other polytope vertices represent schedules that cannot be obtained through the STQs we already presented.

However, there is a known method that allows any schedule for the STN to be found. Dechter et al. presented an approach that maintains a *time window* for each time point. It iterates over the time points, assigning to each of them a value from its window and shrinking the windows of all other, as-yet-unassigned time points. Dechter et al. showed that when run from the minimal network, this approach is backtrack-free: none of the time windows ever becomes empty, and thus a schedule is constructed in a "hands-on" manner. This allows one to *dispatch* an STN: as time progresses, we execute time-points one by one, assigning the current time to them. Hence, it is clear that this approach provides the answer to yet another STQ. In this section, we formally define the DISPATCH query and investigate its complexity.

The query we are going to define must be a function problem that asks to find a specific schedule. Apart from the STN, the problem instance must therefore also contain an unambiguous specification of the schedule that should be returned by the query. Now, note that any vertex of the convex polytope containing all schedules can be identified in the following way, which is inspired by the dispatching approach outlined above. First, we assign the value 0 to the temporal reference point z. Let π be a permutation of the remaining time-points $X' = X \setminus \{z\}$, and let $f : X' \to \{0,1\}$ be a function mapping each $x \in X'$ to the choice of minimizing or maximizing τ_x (represented by the values 0 and 1 for $f(x)$, respectively). Dispatching the temporal variables in the order defined by π, and assigning to each x in turn either the minimal or the maximal still available value from its time window, we wind up in a vertex of the polytope.

The reader can verify that all polytope vertices can be identified in this way. For example, consider again the STN and convex polytope in Figure 2.3 on page 22. Let $f(x_1) = 0$, $f(x_2) = f(x_3) = 1$ and consider the permutations $\pi_1 = (x_1, x_2, x_3)$ and $\pi_2 = (x_3, x_2, x_1)$. We see now that f and π_1 yield the schedule $\tau_1 = \langle z = 0,\ x_1 = 0,\ x_2 = 2,\ x_3 = 4\rangle$, whereas combining f with π_2 results in $\tau_2 = \langle z = 0,\ x_1 = 3,\ x_2 = 5,\ x_3 = 6\rangle$.

If we want to be able to find any interior point of the polytope as well as its vertices, we have to extend the definition of f a little. The function signature

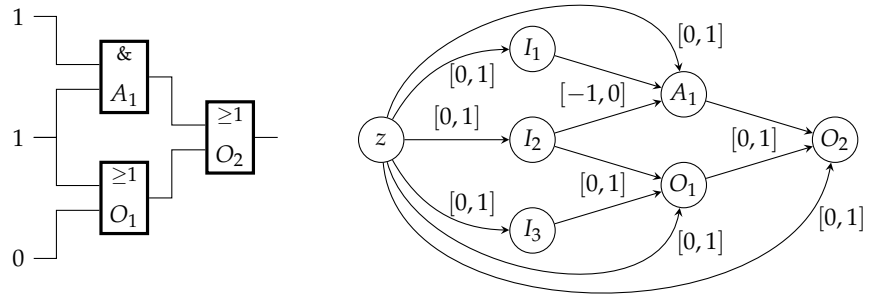


Figure 2.4: A monotone circuit and its corresponding STN. We set $f(I_1) = f(I_2) = f(A_1) = 1$, $f(I_3) = f(O_1) = f(O_2) = 0$, and $\pi = (I_1, I_2, I_3, A_1, O_1, O_2)$.

$f : X \to [0, 1]$ allows for linear interpolation between the minimum and maximum. We can now define our final STQ.

Simple Temporal Query 9: DISPATCH

Type: Function problem.

Instance: An STN S, together with a permutation π and a function f as defined above.

Question: Find the schedule τ identified by (S, π, f).

Next, we show that this problem query is P-complete. The CIRCUIT VALUE problem is a canonical problem in this class. It asks whether a given Boolean circuit has output 1 (or "true"), thus simulating an arbitrary Boolean function. This problem remains P-complete under a variety of restrictions (Greenlaw et al., 1995). We give a reduction from the version restricted to compute monotone Boolean functions $M : \{0, 1\}^n \to \{0, 1\}$. These are increasing functions; put differently, they have the properties that (i) $M(0, \ldots, 0) = 0$; (ii) $M(1, \ldots, 1) = 1$; and (iii) when changing any input from 0 to 1, the output must either remain unchanged or become 1. The circuits representing these functions can be built entirely out of logical AND and OR-gates. We give formal definitions of this type of circuit and the decision problem defined on it, and then present our proof.

Definition 2.1. *A* monotone circuit $M = \langle G, t \rangle$ *is a directed acyclic graph* $G = \langle V, A \rangle$ *with a labelling* $t : V \to \{\text{IN}, \text{AND}, \text{OR}\}$*, such that* G *has exactly one sink, each vertex* v *with* $t(v) = \text{IN}$ *is a source, and all other vertices have at least 2 incoming edges. As a notational convenience, let* $V_{\text{IN}} = \{v \in V \mid t(v) = \text{IN}\}$ *and let* V_{AND} *and* V_{OR} *be defined analogously.*

Definition 2.2. *Given a monotone circuit* M *as in Definition 2.1 together with an input* $\iota : V_{\text{IN}} \to \{0, 1\}$*, the problem* MONOTONE CIRCUIT VALUE *asks whether the value* $M(\iota)$ *of the monotone Boolean function represented by the circuit, i.e. the value at the sink of* M*, is equal to 1.*

Theorem 2.15. DISPATCH *is* P*-complete (under* NC^1 *reductions).*

Proof. Membership in P is obvious: the approach by Dechter et al. (1991), outlined above, takes $\mathcal{O}(n^3)$ time to compute the minimal network and then $\mathcal{O}(n^2)$ time to assemble the schedule. To prove P-hardness, we give a reduction from MONOTONE CIRCUIT VALUE. We first describe how to transform instances of this problem into DISPATCH instances, and then show the correctness of the reduction.

Given a monotone circuit $M = \langle G = \langle V, A\rangle, t\rangle$ and input ι as defined above, we define an STN $\mathcal{S} = \langle X, C\rangle$, a permutation π, and a function f as follows. We first add a temporal reference variable $z \notin V$, so the set of variables becomes $X = V \cup \{z\}$. We constrain all variables $v \in V$ by $c_{zv} : v - z \in [0,1]$; thus, any normalised schedule τ for $\mathcal{S}$ must lie in $[0,1]^X$.

For each AND-gate $v \in V_{\text{AND}}$ and each of its inputs u such that $(u,v) \in A$, we define a constraint $c_{uv} : v - u \in [-1,0]$. For an OR-gate $v \in V_{\text{OR}}$ together with input u we set $c_{uv} : v - u \in [0,1]$. Finally, let the permutation π of $X \setminus \{z\} = V$ correspond to an arbitrary topological (sources-to-sink) ordering of the circuit, and define $f(v) = 1$ if $v \in V_{\text{IN}} \wedge \iota(v) = 1$ or $v \in V_{\text{AND}}$, and $f(v) = 0$ otherwise. Figure 2.4 depicts an example circuit and the STN that is the result of this transformation.

This completes the description of the transformation; we now show that the transformation itself is in NC^1. This is trivial for all steps above except determining the topological ordering π. Computing a topological ordering is in NC^2 (Cook, 1985); however, we may also assume that the representation of the circuit is already in a topological order (Greenlaw et al., 1995). Therefore, the transformation can indeed be computed in NC^1.

Let τ be the schedule for the DISPATCH instance defined by $(\mathcal{S}, \pi, f)$. Note that, since the distance graph representation contains no constraint edges with negative weights, $\mathcal{S}$ must be consistent and therefore such a schedule τ exists. To complete the proof, we now show by structural induction that the value τ_v for any $v \in V$ is equal to the value of the corresponding element in the circuit M, and in particular, if $s \in V$ is the sink of M, that $\tau_s = 1$ if and only if the value computed by M equals 1.

As a base case, consider each input $v \in V_{\text{IN}}$. Here, the value for $f(v)$ ensures that $\tau_v = \iota(v)$. For the induction step, consider a gate $v \in V \setminus V_{\text{IN}}$ with its inputs $U_v = \{u \in V \mid (u,v) \in A\}$; v is either an AND-gate or an OR-gate. Note that, because π corresponds to a topological ordering of M, a value τ_u has been assigned to each $u \in U_v$ before τ_v is determined.

If v is an AND-gate and there exists an $u \in U_v$ with $\tau_u = 0$, the constraint $c_{uv} : v - u \in [-1,0]$ correctly ensures that $\tau_v = 0$. Otherwise, $\tau_u = 1$ for all $u \in U_v$ and the constraints require that $\tau_v \in [0,1]$. The value assigned to τ_v then depends on our function f, and since $f(v) = 1$ for AND-gates, we get $\tau_v = 1$.

If v is an OR-gate, analogous reasoning with the constraint $c_{uv} : v - u \in [0,1]$ shows that $\tau_v = 1$ if there is at least one input u with $\tau_u = 1$, and otherwise $f(v) = 0$ ensures that $\tau_v = 0$.

We therefore conclude by induction that the values τ_v for every $v \in V$ are identical to the corresponding outputs of the elements in M. It follows in particular that for the sink $s \in V$, τ_s is equal to the value $M(\iota)$ computed by the circuit. □

As mentioned above, the P-completeness of DISPATCH implies that this problem is probably inherently sequential, since it is widely suspected that $\mathsf{P} \neq \mathsf{NC}$. In other words, it is unlikely that there exists a parallel algorithm for it that requires less than polynomial time on a polynomial number of processors. Another consequence is that it is unlikely that the problem can be solved in less than polynomial space.

One might argue that the definition of DISPATCH itself is inherently sequential and accuse us of committing the fallacy of begging the question of P-completeness. However, note that our reduction works for *any* problem query that allows one to ask for a specific schedule for an STN (and in particular one of the polytope vertices), and therefore any such query must be P-complete. As an example, consider a formulation akin to linear programming, where given an STN and a linear objective function $g : \mathbb{R}^X \to \mathbb{R}$, the problem is to find the assignment $\tau \in \mathbb{R}^X$ such that $g(\tau)$ is maximised. This problem formulation allows any vertex of the convex polytope to be found and is therefore also P-complete.

The proof of Theorem 2.15 shows that we do not need the full representational power of DISPATCH for the problem to be P-hard. In particular, we can restrict the set of allowed permutations π of the time points X. We can even restrict π to a fixed ordering of the time points, e.g. the order in which they appear in the problem representation, without impacting P-completeness, thus removing it altogether as a parameter in the problem specification. In the next chapter, we show that all schedules can indeed still be found even if π is fixed or restricted.

2.3 Discussion

In this chapter, we undertook to define the Simple Temporal Problem (STP) in a formal way. In Section 2.1 we discussed the Simple Temporal Network (STN), along with many of its properties, some of which had not been explicitly presented before. Then, in Section 2.2, we investigated the Simple Temporal Queries (STQs) that can be asked given an STN, formally defined them, and derived their theoretical complexity. By means of this inventory, we have answered the question "what is the STP?" asked in the introduction to this chapter. The complexity results give an idea as to which approaches may or may not be fruitful. For example, for the DISPATCH query we have shown that a parallel approach is unlikely to be much more efficient than a sequential algorithm. In contrast, we proved that each of the other queries must admit an efficient parallel algorithm that requires at most polylogarithmic time $\mathcal{O}(\log^2 n)$, given a polynomial number of processors. We futher showed that a parallel approach requiring only $\mathcal{O}(\log n)$ time must exist for STQ 4, and given a minimal network also for STQs 2, 3, and 7.

We interpreted this result as a classification of these queries into the easiest class, and the question of a lower bound on their complexity was deliberately left open.

We found that the concept of minimality plays a central role in the theory of simple temporal reasoning. In fact, one can consider minimal constraints, and the minimal network, to constitute a normal form for simple temporal knowledge, which allows some STQs to be answered more easily. We draw a parallel here to normal forms in other fields, such as the conjunctive normal form in propositional satisfiability, which is an equivalent representation of an arbitrary set of Boolean formulae that allows one to efficiently check whether a given valuation of Boolean variables satisfies all constraints. Moreover, note that Theorem 2.13 shows that it is easier to add a new piece of information to a temporal knowledge base in normal form than to compute the minimal STN from scratch. We thus focus on STQs 5, 6, and 7, which establish and maintain this normal form; for the most part, we do not address STQ 8, as Theorem 2.14 showed that it is as hard as STQ 6.

Most of the STQs presented in this chapter are implicitly familiar from the STP literature. However, to the best of our knowledge, the full inventory as proposed in this chapter was not yet available, and neither was their complexity known. The only prior art we have found was by Brusoni et al. (1995). Extending work by Van Beek (1991) on the interval algebra, their work proposes queries on a simple temporal knowledge base; however, it is partly orthogonal to ours. The authors include our queries of CONSTRAINT COMPATIBILITY (STQ 2) and MINIMAL CONSTRAINT (STQ 5), where the former is phrased as asking whether a constraint (or a set of constraints together) is *possibly true*. In addition to our inventory, they distinguish a query asking whether a set of constraints is *necessarily true*, i.e. whether it is entailed by the currently represented temporal information. In effect, this type of query is therefore a simple extension of MINIMAL CONSTRAINT and likewise becomes trivial when asked of a minimal network. Brusoni et al. prove polynomial time bounds on approaches to answering these queries and thereby establish their membership in P. However, they do not consider lower bounds or subclasses of P, nor do they mention any of our other queries.

The theoretical analysis performed in this chapter does not yet provide an explicit method for answering the STQs we identified. Therefore, we consider this issue in the remainder of this dissertation. First, in Chapter 3, we investigate the available methods for bringing temporal information into normal form. Recall that Theorem 2.10 showed NL-completeness for both STQ 5, asking for a single minimal constraint, and STQ 6, asking for all minimal distances; hence, they are of equivalent complexity. However, when designing algorithms to answer these STQs, we show that there is a significant difference between them, both in time and in space. We draw on graph theory to define an intermediate problem query that offers a desirable trade-off. Then, in Chapters 4 and 5, we study extant as well as new methods to answer respectively this "partial" query and the "full" STQ 6. Finally, in Chapter 6, we investigate how to maintain either of these types of normal form when adding new information, thus solving STQ 7.

CHAPTER 3

Path Consistency and Triangulation

In the previous chapter, we introduced the Simple Temporal Network (STN) as a framework for maintaining simple temporal knowledge. We investigated its properties and gave an inventory of Simple Temporal Queries (STQs) that can be asked given such a collection of temporal information. Finally, we formally determined the computational complexity of each of these STQs.

An important discovery was that from a purely theoretical viewpoint, the computational complexity of asking for a single minimal constraint or the complete minimal network is identical. Moreover, we showed that once the minimal network is available, some STQs become strictly easier to solve. Therefore, we suggested that the minimal network can be seen as a "normal form" for maintaining simple temporal information.

In this chapter, we investigate the question how to attain this normal form. Being a special case of the Temporal Constraint Satisfaction Problem, the STN has strong roots in constraint theory. It is therefore not surprising that already in the seminal paper introducing the STN, Dechter et al. (1991) investigated the application of the constraint-theoretical notion of *path consistency* to temporal constraint networks.

The property of path consistency pertains to a single constraint, but it can be extended in several ways to a network-encompassing property. The three network-wide variations we discuss in this chapter are directional, partial, and full path consistency, respectively abbreviated as DPC, PPC, and FPC. Ever since the original 1991 paper by Dechter et al., it has been common knowledge in the literature that enforcing full path consistency on an STN is tantamount to "solving" it — that is, answering STQ 6 to determine the minimal network. Towards the end of this chapter, we again derive this fact, but our interest is wider: we also investigate the applicability of each of the three network-wide

path consistency concepts for other STQs. Specifically, we evaluate to what extent each of them might function as a normal form for temporal information.

As will become clear, for a constraint network to be DPC or PPC, the underlying graph must be *chordal*. Informally, a graph is chordal if any cycle containing more than three vertices has a *chord*: an edge that connects a pair of vertices not adjacent in the cycle. Since many properties of DPC and PPC depend on this chordality, we devote significant attention to the theory of chordal graphs, as well as to the process of *triangulation*, i.e. enforcing chordality.

The theoretical nature of the previous chapter also became apparent in the nature of algorithms presented there. These were only used to determine upper bounds on the complexity of some of the STQs. They all assumed either a nondeterministic or a parallel model of computation and, as such, they were not directly suited for implementation. In contrast, the algorithms presented in the remainder of this dissertation, starting from this chapter, all assume the conventional random-access-machine model of computation, which is much closer to practice.* Moreover, we present asymptotic bounds on the time and space complexity of these algorithms. For this reason, it is important to devote some attention to the practical representation of an STN in a data structure, since the choice of representation may have a significant effect on these bounds.

All in all, this chapter lays the groundwork for subsequent chapters, investigates the canonical normal form for temporal information, and suggests a new type of normal form. In the chapters that follow, we use the results included here as an underpinning for new algorithms to compute or maintain these normal forms.

This chapter is structured as follows, in bottom-up fashion. In Section 3.1, we investigate the options for representing the STN in a computer and the consequence of each choice for time and space complexity. Next, we present theory of chordal graphs in Section 3.2, followed by Section 3.3 that discusses how a general graph can be made chordal. The stage is then set to discuss the constraint-theoretical property of path-consistency, and its generalisation to the network-wide concepts of DPC, PPC, and FPC, in Section 3.4. In Section 3.5 we finally return to our area of interest, the Simple Temporal Network, and investigate the consequences of what was discussed before. Section 3.6 concludes the chapter with a summary and a discussion.

3.1 Data structures

The previous chapter formally defined the STN and its theoretical properties. In this section, we discuss how it can be represented in practice. Of the equivalent perspectives as a set of linear inequalities, a constraint network, and a distance graph, we feel that the latter is simplest and most natural to keep in mind when reading this section; however, both other perspectives fit these data structures

*The Turing machine model does not suffice because for the asymptotic bounds we give, we assume constant-time access of arbitrary memory locations, e.g. through pointers or array indexing.

just as well. For our discussion below, recall that n is the number of time points or vertices in the STN and that m is the number of constraint edges. We also use δ to denote the *degree* of the graph: the greatest number of edges incident on a single vertex. A final parameter of the STN is w_{max}, the largest absolute (finite) constraint weight. In the discussion below, and for the experiments described in later chapters, we assume that constraint weights can be stored and handled in constant time and space, which always suffices for our purposes.

It will come as no surprise that STNs are most naturally stored in graph data structures. Roughly, these come in two flavours: the *adjacency matrix* and the *adjacency list*. The former is an n-by-n array from which edge weights can be accessed in constant time; the latter maintains for each vertex the list of its neighbours and the corresponding edge weights, thus allowing for efficient iteration over the neighbours of any vertex. The matrix naturally requires $\Theta(n^2)$ space, whereas the list fits in linear space $\mathcal{O}(m)$. Often, one chooses to either save space by maintaining only a list, or to use a hybrid approach and maintain both list and matrix in $\Theta(n^2)$ total space. In the latter case, iteration over neighbours as well as access to an individual edge can be performed efficiently.

For the adjacency list, several implementations are available, all of which provide efficient iteration over vertex neighbours. A *linked list* allows constant-time addition of a new edge that is known not to exist yet, but requires linear time $\mathcal{O}(\delta) \subseteq \mathcal{O}(n)$ to look up an edge weight. In a *sorted array*, edge weights can be looked up in $\mathcal{O}(\log \delta)$ time, but addition of a single new edge requires linear time $\mathcal{O}(\delta)$. If a relatively large number of edges is to be added at once, an advantage is that the entire data structure can be re-sorted in $\mathcal{O}(m)$ time. With a *binary search tree*, both addition and lookup can be performed in $\mathcal{O}(\log \delta)$ time, although the constant factor hidden in this bound is somewhat higher. The choice of which data structure to use thus depends on the relative amounts of operations one expects to perform.

A space-efficient alternative for the n-by-n adjacency matrix which still allows constant-time access may be offered by the *band matrix*. Assume that we have a bijection $\# : V \leftrightarrow \{1, \ldots, n\}$ so that all vertices in an STN correspond to a unique number. Now, $b = \max_{\{u,v\} \in E} |\#(u) - \#(v)|$, the largest difference between the numbers assigned to the endpoints of any edge, is called the *bandwidth* of the STN under this mapping. The STN can then be represented in a band matrix taking $\Theta(nb)$ space. Of course, one wants to obtain a bijection corresponding to a small bandwidth; however, the problem of finding a bijection that minimises bandwidth is NP-hard in general (Papadimitriou, 1976). In Chapter 6, we discuss a setting where the minimal bandwidth is easily found. Note that an ancillary adjacency list always fits within the space required by the band matrix, regardless of the value of b.

A final alternative is provided by maintaining *hash tables* of edges. These take linear space and allow addition, deletion, and access all in expected constant time; moreover, it is simple to also provide for efficient iteration over vertex neighbours. In practice, these data structures are often very efficient, but the trouble here lies in the word "expected": the constant-time bounds cannot be guaranteed and

there always exist pathological cases where run time deteriorates to at least linear time. In contrast, when considering the time complexity of the algorithms we discuss in the following chapters, we always derive guaranteed bounds rather than expected bounds. Within the context of this work, therefore, we never use hash tables in our implementations.

Regardless of the type of data structure one uses, one must choose which distance values to store for each constraint edge $(u, v) \in E$: the constraint weight w_{uv}, the shortest distance ω_{uv}, or both? For our purposes in the following chapters, it turns out that we can very well make do with storing just the shortest distances ω. Most pseudocode explicitly "overwrites" the original constraints with the currently tightest known bounds during operation. However, it is of course a simple matter to store the pair (w_{uv}, ω_{uv}) of both weights for each constraint edge $(u, v) \in E$. It often proves useful to also store pointers from (u, v) to (v, u) and vice versa. Edges can be annotated with more information still; as long as we store a constant amount of information per edge, this has no negative impact on the asymptotic time and space bounds. We touch upon this matter again in Chapter 6, which concerns dynamic algorithms for the STN.

Next, we briefly depart from the subject of Simple Temporal Networks to discuss required graph and constraint theory. We return to the subject of STNs in Section 3.5

3.2 Chordal graphs

In this section, we define chordal graphs and discuss their properties. The definitions and results presented in this section are readily available from graph-theoretical literature (e.g. West, 1996) and will prove to be useful when discussing normal forms for temporal information in upcoming sections.

Recall from the previous chapter that the STN has an equivalent perspective as a distance graph. This is a directed graph in which an upper bound on the temporal difference between two time points is represented by a weighted arc connecting the corresponding vertices. However, we are about to introduce the concept of chordal graphs, which are defined only for *undirected* graphs. Therefore, in this section and the next, we ignore the direction of arcs, assuming that two vertices are connected by an edge if there exists at least one arc between them (in either direction). Likewise, the weights of the arcs are irrelevant for these concepts. Thus, we simply obtain an undirected graph $G = \langle V, E \rangle$ underlying a given STN, allowing us to define chordal graphs. Informally, a graph is chordal if any cycle containing more than three vertices has a *chord*: an edge that connects a pair of vertices that are not adjacent in the cycle. Let us now give a formal definition. As we did for distance graphs, we assume all graphs to be connected throughout this chapter.

Definition 3.1. *Let $G = \langle V, E \rangle$ be an undirected graph. We define the following:*

- *If $C = (v_1, v_2, \ldots, v_k, v_{k+1} = v_1)$ is a cycle in G with length $k > 3$, then any edge on two nonadjacent vertices $\{v_i, v_j\}$ with $1 < j - i < k - 1$ is a* chord *of C.*

- *If a cycle in G with length more than 3 has no chords, it is called a* hole *or* unchorded cycle.
- *If G has no holes, it is* chordal *(also ambiguously called* triangulated*).*[*] *In other words, G is chordal if every cycle with length more than 3 has a chord.*

Next, we define some concepts that lead up to a useful property of chordal graphs. It offers a natural way of either adding or removing a vertex while maintaining chordality. Let us introduce some notation.

Definition 3.2. *Given an undirected graph $G = \langle V, E \rangle$, the* neighbours $\mathcal{N}_G(v)$ *in G of a vertex $v \in V$ are those vertices connected to v by an edge, so $\mathcal{N}_G(v) = \{u \mid \{u, v\} \in E\}$. We drop the subscript if the graph referred to is clear from context.*

We define simplicial vertices and show how they relate to chordal graphs.

Definition 3.3. *Given an undirected graph $G = \langle V, E \rangle$, a vertex $v \in V$ is* simplicial *if all of its neighbours $\mathcal{N}(v)$ are pairwise connected. Thus, a simplicial vertex v together with its neighbours induces a clique (or simplex) in G.*

Proposition 3.1. *Any chordal graph contains a simplicial vertex.*

Now, a simplicial vertex v together with its incident edges can be removed from a chordal graph without creating a hole, because all of v's neighbours are connected. This prompts us to define the following pair of concepts.

Definition 3.4. *Let $G = \langle V, E \rangle$ again be undirected graph and let $d = (v_n, \ldots, v_1)$ be an ordering of its vertices V. Denote by G_i the subgraph of G induced by $V_i = \{v_1, \ldots, v_i\}$, so $G_n = G$. Now, d is a* simplicial elimination ordering *of G if every vertex v_i is a simplicial vertex of the graph G_i. The reverse ordering $d' = (v_1, \ldots, v_n)$ is then called a* simplicial construction ordering.

The following proposition states that existence of a simplicial elimination ordering is not just necessary but also sufficient for chordality.

Proposition 3.2. *An undirected graph $G = \langle V, E \rangle$ is chordal if and only if it has a simplicial elimination ordering.*

Figure 3.1b depicts a chordal graph with one of its simplicial elimination orderings. It can be seen that the vertex numbered 8 is simplicial for the entire graph $G = G_8$ because vertices 1, 2, 7, and 8 together form a clique. After removing vertex 8 together with its incident edges, the vertex numbered 7 is simplicial for the graph G_7, and so on for the rest of the ordering down to 1.

To find a simplicial construction ordering given a chordal graph $G = \langle V, E \rangle$, Tarjan and Yannakakis (1984) proposed the maximum cardinality search algorithm (MCS), included here as Algorithm 3.1. This algorithm tags vertices one by one in a greedy manner, choosing for each iteration a vertex adjacent to the largest

[*]The term "triangulated graph" can also refer to maximal planar graphs, which do not concern us here.

Algorithm 3.1: Maximum Cardinality Search (MCS)

Input: A chordal graph $G = \langle V, E \rangle$
Output: Simplicial construction ordering d of G

forall $v \in V$ **do**
  $\textsc{count}[v] \leftarrow 0$
  $\textsc{tagged}[v] \leftarrow \textsc{false}$
end
$i \leftarrow 0$
while $\exists u \in V : \neg \textsc{tagged}[u] \wedge \textsc{count}[u] = \max\{\textsc{count}[v] \mid v \in V\}$ **do**
  $i \leftarrow i + 1$
  $d(i) \leftarrow u$
  $\textsc{tagged}[u] \leftarrow \textsc{true}$
  forall $v \in \mathcal{N}(u)$ **do**
    if $\neg \textsc{tagged}[v]$ **then** $\textsc{count}[v] \leftarrow \textsc{count}[v] + 1$
  end
end
return d

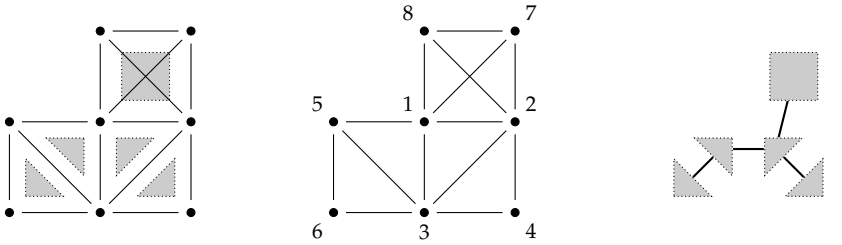
number of vertices tagged before. For each vertex $v \in V$, $\textsc{tagged}[v]$ simply indicates whether a vertex has been tagged, whereas $\textsc{count}[v]$ is equal to the number of neigbours of v that have already been tagged before v. Since MCS starts from an arbitrary vertex and any other ties are also broken arbitrarily, it follows that given a chordal graph, many simplicial elimination orderings exist. Tarjan and Yannakakis show that MCS runs in linear time $\mathcal{O}(n + m)$, which simplifies to $\mathcal{O}(m)$ under our assumption of connectedness, because then $m \geq n - 1$.

Clique trees

Chordal graphs can be seen as a generalisation of *trees* (i.e. connected graphs without cycles). Since a tree by definition has no cycles, it certainly has no holes. Therefore, any tree is trivially a chordal graph, and we say that it has *treewidth* $w^* = 1$, a concept that we come back to below. For any chordal graph G, whether it contains cycles or not, there is a corresponding *clique tree* T. Next, we define this clique tree and concepts relating to it, as always assuming graphs to be connected. For a more extensive discussion, see the survey by Heggernes (2006, Section 3.2).

Definition 3.5. *Let $G = \langle V, E \rangle$ be a chordal graph. Then, the tree $T = \langle C, S \rangle$ is a* clique tree *for G if it satisfies the following properties.*

1. Vertex coverage. *Each clique tree node $c \in C$ is associated with a subset of vertices $V_c \subseteq V$. Conversely, each of G's vertices is associated with at least one node in T, so $\bigcup_{c \in C} V_c = V$.*
2. Edge coverage. *For each edge $\{u, v\} \in E$, there is a node $c \in C$ such that $u, v \in V_c$.*



(a) Chordal graph (b) A simplicial elimination ordering (c) Clique tree

Figure 3.1: A chordal graph and its clique tree. Each shaded shape represents a maximal clique of the graph, containing the vertices at its corners. (Figure courtesy of Ot ten Thije)

3. Coherence. *If for two clique tree nodes $c_1, c_2 \in C$ there exists a vertex $v \in V$ such that $v \in V_{c_1}$ and $v \in V_{c_2}$, then for any node c' on the path in T between c_1 and c_2, it must hold that $v \in V_{c'}$. In other words, the nodes associated with any vertex $v \in V$ induce a subtree of T.*

4. Clique maximality. *Each such subset V_c induces a maximal clique in G; conversely, every maximal clique in G has an associated clique tree node $c \in C$.*

Proposition 3.3. *A graph $G = \langle V, E \rangle$ is chordal if and only if there is a clique tree $T = \langle C, S \rangle$ corresponding to G. T can be constructed in linear time $\mathcal{O}(m)$.*

So, any chordal graph has a corresponding clique tree which can be found efficiently.* Our choice for the symbol S to denote the set of edges in a clique tree is because each member of this set corresponds to a *graph separator*.

Definition 3.6. *Given a connected graph $G = \langle V, E \rangle$, a* separator *is a set $V' \subseteq V$ such that the induced subgraph of G on $V \setminus V'$ is no longer connected. V' is a* minimal separator *if no proper subset of V' is a separator.*

We then have the following result.

Proposition 3.4. *Let $G = \langle V, E \rangle$ be a connected chordal graph and let $T = \langle C, S \rangle$ be its clique tree. If two clique tree nodes $c_i, c_j \in C$ are connected by an edge $\{c_i, c_j\} \in S$, then $V_{c_i} \cap V_{c_j}$ is a minimal separator in G. Conversely, for each minimal separator V' in G, there is a clique tree edge $\{c_i, c_j\} \in S$ such that $V' = V_{c_i} \cap V_{c_j}$.*

See Figure 3.1 for an example that illustrates these concepts. This example graph has the following separators, each corresponding to one of the four edges in the clique tree: $\{1,2\}$, $\{1,3\}$, $\{2,3\}$, and $\{5,3\}$. We remark that the clique tree for a chordal graph need not be unique, although in the example it is. Given a clique tree, it is simple to define the treewidth of the corresponding chordal graph; in the next section, we extend this concept to the general case.

*Note that any general graph $G = \langle V, E \rangle$ can be associated with a *tree decomposition* $T = \langle C, S \rangle$ which lacks only property 4 of Definition 3.5.

Definition 3.7. *Let G be a chordal graph and let $T = \langle C, S \rangle$ be a corresponding clique tree. Then, the* treewidth *w^* of G is equal to the size of its largest clique minus one:*

$$w^* = \max \{|V_c| \mid c \in C\} - 1.$$

It can thus be seen that the example graph in Figure 3.1 has $w^* = 3$. The treewidth can also be derived from the simplicial elimination ordering of a graph.

Proposition 3.5. *Given a chordal graph $G = \langle V, E \rangle$ with simplicial elimination ordering $d = (v_n, \ldots, v_1)$, the treewidth of G is equal to the largest number of lower-numbered neighbours of any vertex, so $w^* = \max_{1 \leq k \leq n} \left|\{v_j \in \mathcal{N}(v_k) \mid j < k\}\right|$.*

From both of these characterisations, it follows that a chordal graph is a tree if and only if it has $w^* = 1$. To see this, note that each maximal clique here corresponds to an edge and has size 2, and each simplicial vertex corresponds to a leaf: a vertex with exactly one neighbour. Each internal vertex of the tree is a minimal separator of size 1 and corresponds to an edge in the clique tree. A simplicial construction ordering can be found by designating an arbitrary vertex as root and then traversing the tree downwards from this root in any order.

Note that a complete graph is trivially chordal, since all cycles with length greater than 3 must have a chord. It then follows from the above characterisations that their treewidth is at the other end of the scale: a chordal graph is complete if and only if it has $w^* = n - 1$.

If a graph is not chordal, it must contain at least one hole. The next section describes how these holes can be filled to enforce chordality for any graph.

3.3 Triangulation

Two of the three normal forms for the STN, introduced in Section 3.4 below, are defined for chordal graphs. When the graph underlying an STN is not chordal, it can be made so by *triangulating* the graph. In this section, we introduce concepts having to do with this procedure and present two approaches for triangulation. Let us first define what a triangulation is. This formulation is based on the work by Kjærulff (1990).

Definition 3.8. *Given an undirected graph $G = \langle V, E \rangle$, a set of edges F are called* fill edges *if $F \cap E = \emptyset$ and $G' = \langle V, E \cup F \rangle$ is chordal. The graph G' is then called a* triangulation *of G.*

The basic technique to triangulate a graph G is to add the fill edges by iterating over the vertices of G in some order; after visiting a vertex, it is removed from the graph, or simply ignored for the remainder of the process. This process is therefore called *elimination* (also: the *elimination game*), and the order d in which this is done is called the elimination ordering. When eliminating a vertex v, fill edges are added among all non-adjacent pairs of those neighbours of v that have not yet been eliminated. In this manner, a clique is formed and v becomes simplicial among these neighbours. Note that if G is chordal, following its

Algorithm 3.2: Naive triangulation

Input: An undirected graph $G = \langle V, E \rangle$ and an ordering $d = (v_n, v_{n-1}, \ldots, v_1)$
Output: Chordal graph $G' = \langle V, E' \rangle$

```
1 E' ← E
2 for k ← n to 1 do
3 |   forall v_i, v_j ∈ N_{G'}(v_k) such that i, j < k do
4 |   |   if {v_i, v_j} ∉ E then E' ← E' ∪ {{v_i, v_j}}
5 |   end
6 end
7 return G' = ⟨V, E'⟩
```

simplicial elimination ordering means that no fill edges are added. Contrariwise, if G was not chordal, d becomes the simplicial elimination ordering of the resulting triangulation G'. Algorithm 3.2 corresponds exactly to this description. We determine the time complexity of this algorithm below, after having introduced the required concepts.

A useful parameter that appears in the time complexity of many algorithms that require a chordal graph as input is m_c, the total number of edges including fill edges. Note that $m_c \geq m$, with equality only if the graph was already chordal. Another parameter of a triangulation is the *induced width*, denoted w_d. When iterating over vertices v along the elimination ordering d and ensuring that all pairs of non-eliminated neighbours of v are connected, the induced width is exactly the highest number of such neighbours encountered for any v. It can be formally defined as follows.

Definition 3.9. *Let $G = \langle V, E \rangle$ be a graph and let $G' = \langle V, E \cup F \rangle$ be a triangulation of G obtained through the elimination ordering $d = (v_n, v_{n-1}, \ldots, v_1)$. The* induced width *$w_d$ along d of the original graph G is the following measure: $w_d = \max_{1 \leq k \leq n} \left| \{ v_j \in \mathcal{N}_{G'}(v_k) \mid j < k \} \right|$.*

From this definition, it follows immediately that $m_c \leq n w_d$. Referring to Proposition 3.5, we see that the induced width w_d of the original graph G along d is exactly equal to the treewidth w^* of the triangulation G'. We can now extend the definition of treewidth to the general case, where graphs are not necessarily chordal.

Definition 3.10. *Let G be an undirected graph. Its* treewidth *w^* is equal to its minimum induced width along any ordering of its vertices.*

It is important to note that the induced width w_d is not a property of a graph per se; rather, it is dependent on both the graph and the vertex ordering d used. The treewidth, in contrast, depends only on the graph itself.

There are many possible triangulations of a graph, each corresponding to a different elimination ordering. They can be ranked by their induced with and the

number of fill edges. In general, it is desirable to strive for *minimality*: achieving chordality with as few fill edges as possible. The following definition is again based on the work by Kjærulff (1990).

Definition 3.11. *A triangulation $G' = \langle V, E \cup F \rangle$ of a graph $G = \langle V, E \rangle$ is a* minimal triangulation *if there exists no proper subset $F' \subset F$ of fill edges for G. It is a* minimum triangulation *if there exists no set F'' of fill edges for G having $|F''| < |F|$.*

A minimal triangulation can be found in $\mathcal{O}(nm)$ time (Kjærulff, 1990). In contrast, the problem of finding a minimum triangulation or a triangulation with lowest induced width is NP-complete; in fact, just determining the value of the minimum induced width (i.e. the treewidth) already is (Arnborg et al., 1987). Since finding an optimal triangulation is so hard, several heuristics have been proposed for this problem; we discuss a selection in the next section. However, the following remarkable fact must be noted. It is due to Bodlaender (1996).

Proposition 3.6. *Given a graph $G = \langle V, E \rangle$ and some constant κ, it can be determined in $\mathcal{O}(n)$ time whether there exists an elimination ordering d for G with $w_d \leq \kappa$, and if so, d itself can be constructed within the same bound.*

This result can be reconciled with the NP-completeness of the general problem by the observation that, although the asymptotic run time is linear in n, the hidden constant grows superpolynomially with κ.

We now return to the naive triangulation approach of Algorithm 3.2. For each iteration of the outer loop, the algorithm considers all pairs of lower-numbered vertices. It then follows immediately from the definition of induced width that line 4 is executed no more than nw_d^2 times. Recalling that $m_c \leq nw_d$, a tighter bound that is not much harder to derive is $m_c w_d$. This bound holds because each edge $\{v_k, v_j\}$ in the chordal graph is considered exactly once by the algorithm, and is supplemented by at most w_d vertices v_i with $i < k$.

The time complexity of line 4 itself depends on the data structure used. In an adjacency list representation, these operations incur a cost factor of at least $\mathcal{O}(\log n)$, which yields a total run-time complexity of $\mathcal{O}(m_c w_d \log n) \subseteq \mathcal{O}(nw_d^2 \log n)$. This extra cost could be avoided by (also) maintaining an adjacency matrix, trading off an $\Omega(n^2)$ space requirement for an $\mathcal{O}(m_c w_d) \subseteq \mathcal{O}(nw_d^2)$ time complexity. Moreover, recall that setting up this data structure additionally requires $\Theta(n^2)$ time.

A more efficient approach was proposed by Tarjan and Yannakakis (1984), and we dub it "TY triangulation". Like the naive method, their algorithm iterates over vertices along the elimination ordering, but whereas the naive approach starts from the original graph and adds fill edges between pairs of lower-numbered vertices, the TY method starts from scratch, adding edges between the current vertex and all of its higher-numbered neighbours. Pivotal in this approach is the concept of *vertex followers*, maintained by the algorithm in an array FOLLOW. Given a triangulation $G' = \langle V, E' \rangle$ of a graph G along an ordering $d = (v_n, v_{n-1}, \ldots, v_1)$, the authors define the follower of a vertex v_j as its neighbour (in G') with highest index that comes after it in d. For example, in Figure 3.1b, the vertex numbered 3

Algorithm 3.3: TY triangulation

Input: An undirected graph $G = \langle V, E \rangle$ and an ordering $d = (v_n, v_{n-1}, \ldots, v_1)$
Output: Chordal graph $G' = \langle V, E' \rangle$

```
1  E' ← ∅
2  for k ← n to 1 do
3  |   SEEN[v_k] ← k
4  |   FOLLOW[v_k] ← v_k
5  |   forall v_j ∈ N_G(v_k) such that j > k do
6  |   |   v_i ← v_j
7  |   |   while SEEN[v_i] > k do
8  |   |   |   E' ← E' ∪ {{v_i, v_k}}
9  |   |   |   SEEN[v_i] ← k
10 |   |   |   v_i ← FOLLOW[v_i]
11 |   |   end
12 |   |   if FOLLOW[v_i] = v_i then FOLLOW[v_i] ← v_k
13 |   end
14 end
15 return G' = ⟨V, E'⟩
```

is the follower of the vertices numbered 5 and 4, whereas vertex 2 follows vertices 7 and 3. Note that a vertex need not have a follower, and that the transitive closure of the follower relation imposes a partial order on the vertices in a graph. Tarjan and Yannakakis then prove the following.

Proposition 3.7. *Let $G' = \langle V, E' \rangle$ be a triangulation of a graph $G = \langle V, E \rangle$, obtained along an ordering $d = (v_n, v_{n-1}, \ldots, v_1)$. Then, E' contains an edge $\{v_i, v_k\}$ with $i > k$ if and only if there is a vertex v_j with $j > k$ such that $\{v_j, v_k\}$ was already in the set E of original edges and either $v_j = v_i$, or there is a chain of followers from v_j to v_i.*

Thus, given some graph G, an elimination ordering d, and a vertex v_k, the "upward" edges incident on v_k in the triangulation G' of G along d can be found by walking the chain of followers from each $v_j \in \mathcal{N}_G(v_k)$ with $j > k$. Since only the followers appearing in d before v_k are required, Algorithm 3.3 can take an efficient iterative approach. Throughout the algorithm, it is guaranteed that $\textsc{follow}[v_i] = v_j$ is correctly set for $i > j > k$. If v_i has no follower appearing in d before v_k, $\textsc{follow}[v_i] = v_i$. When such a dead end is encountered while adding edges incident on v_k, $\textsc{follow}[v_i]$ is set to v_k in line 12.

The array entry $\textsc{seen}[v_j]$ is set to k if either $v_j = v_k$ (line 3) or in line 9 when the edge $\{v_j, v_k\}$ has been added to E'. It is used to ensure that overlapping parts of follower chains are walked only once for each iteration of the outer loop. It can now be seen that the algorithm has an $\mathcal{O}(m_c)$ time complexity: it performs no more than a constant number of operations for each edge in E'. Since the size of the output is $\Theta(m_c)$, this is optimal.

Although the TY approach is very efficient, it has an important disadvantage: the elimination ordering must be determined from the original graph. In the naive method, fill edges are added immediately for each vertex that is eliminated. When selecting the next vertex, the new situation can thus be taken into account. The next section discusses some vertex ordering heuristics that can be employed when aiming to produce small triangulations. We present heuristics of a static nature that can be used with TY, as well as dynamic approaches that reconsider the state of the graph after each elimination.

Ordering heuristics

As became clear above, the quality of triangulation depends solely on the vertex ordering followed during the process, and finding an optimal ordering is an intractable task in general. Instead, heuristics of varying quality and efficiency can be followed. These can be classified into static and dynamic versions. A *dynamic* heuristic requires the vertex elimination to be fully completed before it chooses the next step; a *static* heuristic, in contrast, only uses the original (non-chordal) graph.

Well-known dynamic heuristics, investigated by Kjærulff (1990), include minimum degree (DMD) and minimum fill (DMF). These choose at each step the vertex with smallest number of remaining neighbours and the smallest number of resulting fill edges added, respectively. Computing a minimum-degree vertex ordering requires $\mathcal{O}(nm_c) \subseteq \mathcal{O}(n^2 w_d)$ time in total (Heggernes et al., 2001); the (naive) triangulation itself fits comfortably within this bound. Minimum fill is harder to compute. It can be made to run in $\mathcal{O}(n^2 + nw_d^2\delta_c) \subseteq \mathcal{O}(n^2 w_d^2)$ time, where δ_c is the degree of the resulting chordal graph. Note that these bounds do not just depend on the size of the input graph but also on the output, i.e. on the quality of the triangulation they find: if they produce a small triangulation, w_d and m_c are lower so the heuristics run faster.

We already introduced one static approach above. The maximum cardinality search algorithm (MCS) produces a simplicial elimination ordering for graphs that are already chordal, but can also be used as a heuristic for triangulating general graphs. It requires linear time $\mathcal{O}(m)$ in the size of the original graph, which is dominated by the triangulation. Both the minimum degree and minimum fill heuristics also give rise to static variants (abbreviated SMD and SMF), by considering only the degree or potential fill for each vertex in the original graph. Their respective time complexities are $\mathcal{O}(n)$ and $\mathcal{O}(m\delta) \subseteq \mathcal{O}(n\delta^2)$; the latter bound is not easily comparable to the time bound on triangulation. Finally, as a baseline comparison, we often compare with a random permutation of the vertices (RND), which can be produced in linear time. As is to be expected, the time bounds on these static heuristics depend only on the input graph.

Finally, we discuss approximate minimum degree (AMD; Amestoy et al., 1996), an approach that improves on the time complexity of the dynamic minimum degree heuristic while being more accurate than static minimum degree. It is much used in practice, e.g. by the MATLAB and CPLEX software packages

to factorise matrices.* Instead of working on the graph directly, AMD uses a data structure called a *quotient graph*. It allows the algorithm to simulate vertex elimination without actually needing to add fill edges, so that no more than the original $\mathcal{O}(m)$ space is required. This is essential when working with matrices whose numbers of rows and columns run into the millions, and AMD takes care not to overstep this space bound.

Any heuristic that keeps track of the exact degree of vertices, while at the same time using only $\mathcal{O}(m)$ space, must pay for this in its computational complexity: Heggernes et al. prove that at best, such heuristics have a tight bound of $\mathcal{O}(n^2m)$ on their run time. Therefore, the idea behind AMD is to inexpensively maintain an upper bound $\Delta(v)$ on the actual degree $\delta(v)$ of each vertex $v \in V$, with $\Delta(v) = \delta(v)$ initially. By thus sacrificing some accuracy, efficiency is improved over conventional minimum degree: the heuristic produces an ordering in $\mathcal{O}(nm)$ time, after which TY triangulation can be employed for an overall time complexity of $\mathcal{O}(nm + m_c)$. However, none of the published versions of AMD we found, including the source code used in MATLAB (Davis et al., 2012), fully specifies an algorithm that guarantees the $\mathcal{O}(nm)$ bound. This analysis of AMD is outside the scope of this chapter, but we include it in Appendix A. There, we also show how the problematic parts of the heuristic can be implemented so that the claimed time bound is met.

In summary, the following heuristics are available.

RND Simply eliminate vertices in a random order, produced in $\mathcal{O}(n)$ time.

MCS This ordering is produced in $\mathcal{O}(m)$ time by Tarjan and Yannakakis's (1984) maximum cardinality search algorithm. For chordal graphs, this produces a perfect elimination ordering resulting in no fill edges.

SMD Look only at the original graph and repeatedly eliminate the vertex with the lowest degree. The vertices can thus simply be sorted, in $\mathcal{O}(n)$ time, according to their degree in the original graph.

SMF At each step, eliminate the vertex causing the fewest fill edges to be added, without taking addition of those fill edges into account. This takes $\mathcal{O}(m\delta) \subseteq \mathcal{O}(n\delta^2)$ time.

DMD Repeatedly eliminate the vertex with the lowest degree in the chordal graph that is being constructed. This dynamic variant takes $\mathcal{O}(nm_c) \subseteq \mathcal{O}(n^2w_d)$ time.

DMF In each iteration, eliminate the vertex causing the fewest fill edges to be added, in $\mathcal{O}(n^2 + nw_d^2\delta_c) \subseteq \mathcal{O}(n^2w_d^2)$ total time. Note that this heuristic

*The problem of finding a minimum triangulation is almost identical to finding a permutation of matrix pivots to produce an LU decomposition or Cholesky factorisation with smallest fill-in. The only difference lies in the fact that two nonzero elements in a matrix may sometimes combine to form a zero, whereas during graph triangulation, fill edges are always added between lower-numbered pairs of neighbours.

is (trivially) guaranteed to find a perfect elimination ordering for a chordal graph.

AMD This is the approximate minimum degree heuristic originally proposed by Amestoy et al. (1996) and discussed above. It runs in $\mathcal{O}(nm)$ time.

This concludes our discussion of chordal graphs and how to triangulate graphs that are not chordal. Next, we return to the domain of (temporal) constraint networks and show how these concepts can be put to use.

3.4 Path consistency

In the previous chapter, we showed that constraints along a path in the STN can be composed to find implied constraints between the endpoints of such a path. In general binary constraint networks, one can do the same. In this section, we describe the property of *path consistency*, known from literature on constraint networks (Montanari, 1974; Dechter, 2003). This property tells us something about how explicit constraints are.

Definition 3.12. *Let $\mathcal{C} = \langle X, C \rangle$ be a constraint network. Given a constraint $c_{xy} \in C$ and a path π connecting x to y in the network, c_{xy} is* path-consistent with π *if any pair τ_x, τ_y of assignments consistent with c_{xy} can be extended to all variables in π. Equivalently, c_{xy} is path-consistent with π if it forbids assignments τ_x, τ_y for which not all constraints along π can be satisfied. If c_{xy} is path-consistent with all paths from x to y, it is simply called* path-consistent.

So, informally, a constraint edge is path-consistent with a path connecting its endpoints if it only allows values which also satisfy all constraints along that path, taken together. Note that universal constraints (which do not constrain the variables at all) normally need not be associated with an explicit edge in the constraint graph. However, as a convention, for a constraint to be path-consistent, it must be represented explicitly by a constraint edge if there is at least one path π connecting its endpoints even if the constraint is universal.

It follows from the above definition that if a constraint is not path-consistent, it can be tightened until it is, without impacting the set of solutions for the constraint network (i.e. the set of schedules for the STN). This recalls the concept of a minimal constraint, introduced in the previous chapter; indeed, a minimal constraint is necessarily path-consistent. Although the converse does not hold in general, it does hold for the STN, as we show in Section 3.5.

If multiple constraints in the network are path-consistent, either with all paths or with a selection, path-consistency changes from a local property into a network-wide property. In the remainder of this section, we present three categories: full, directional, and partial path consistency.

Algorithm 3.4: PC-1

Input: A binary constraint network $\mathcal{C} = \langle X, C \rangle$
Output: The FPC network of $\mathcal{C}$

```
repeat
    for k ← 1 to n do
        for i, j ← 1 to n do
            c_ij ← c_ij ∩ (c_ik ⊗ c_kj)
        end
    end
until no constraint is changed
return C
```

Full path consistency

The property of full path consistency (FPC) was proposed in the original 1974 publication by Montanari, although he did not yet use this terminology. It was the first type of network-wide consistency to be proposed. Our definition is based on the formulation by Dechter (2003, Definition 3.3).

Definition 3.13. *A constraint network $\mathcal{C} = \langle X, C \rangle$ is* fully path-consistent *if for each pair of variables $x, y \in X$, the (possibly universal) constraint c_{xy} is path-consistent.*

Assuming that the original constraint network was connected, it follows that a fully path-consistent constraint network corresponds to a complete graph. The following useful result was proved by Montanari (1974); it is sometimes used in lieu of the above definition.

Proposition 3.8. *A constraint network $\mathcal{C} = \langle X, C \rangle$ is fully path-consistent if and only if for each pair of variables $x, y \in X$, the (possibly universal) constraint c_{xy} is path-consistent with every path of length 2; so, for each path (x, z, y) in $\mathcal{C}$, it must hold that $c_{xy} \subseteq c_{xz} \otimes c_{zy}$.*[*]

Full path consistency can be enforced on a constraint network by tightening its constraints. As stated before, making a single constraint path-consistent does not invalidate any solutions for the constraint network, and this observation logically extends to the complete set of constraints. The seminal method for enforcing FPC is due to Montanari (1974) and makes use of his result from Proposition 3.8. It is called PC-1 in the literature and we include it as Algorithm 3.4. It repeatedly tightens each constraint to be path-consistent with each path of length 2, until a fixed point is reached.

On general constraint networks, PC-1 may require many iterations before reaching a fixed point. Moreover, even if the original constraint network was sparse, FPC produces a complete constraint graph, with edges between all pairs of variables. For these reasons, later publications introduced several weaker

[*]Recall from the previous chapter that the operator $\otimes$ denotes constraint composition.

variations of path consistency that can be enforced more efficiently. Next, we present two of these variants which are of interest to the STN.

Directional path consistency

In 1987, Dechter and Pearl proposed the concept of directional path consistency (DPC), which requires constraints to be path-consistent with only a subset of the paths in the network. The "direction" in DPC's name is reflected in the fact that besides the constraint network, it requires an ordering of variables as input. We now provide our own definitions of directionally path-consistent constraints and networks, as parallels to Definitions 3.12 and 3.13. Proposition 3.10 below then links them to the conventional characterisation of DPC available in the literature.

Definition 3.14. *Let $\mathcal{C} = \langle X, C\rangle$ be a constraint network; let $d = (x_n, x_{n-1}, \ldots, x_1)$ be an ordering of its variables. A constraint $c_{x_i x_j} \in C$ is* directionally path-consistent along *d if it is path-consistent with all paths π that connect x_i to x_j in the network and which, except for their endpoints x_i and x_j, involve only variables $x_k \in X$ with $k > i, j$.*

Definition 3.15. *A constraint network $\mathcal{C} = \langle X, C\rangle$ is* directionally path-consistent along an ordering *$d = (x_n, x_{n-1}, \ldots, x_1)$ if for each pair of variables $x_i, x_j \in X$, the (possibly universal) constraint $c_{x_i x_j}$ is directionally path-consistent along d.*

*For brevity, if $\mathcal{C}$ satisfies this property, we also say that it is d-*DPC*, or simply DPC if the ordering is clear from context.*

An important point in DPC's favour is that unlike FPC, it can be enforced without producing a complete constraint graph. This is because for DPC, we only require an explicit constraint edge when the pair of variables involved is connected by a path as in the above definition. If no such paths exist between some pair x_i, x_j of variables, then the implicit universal constraint $c_{x_i x_j}$ is trivially DPC and is not represented by a constraint edge. We can now link this newly introduced concept to the graph theory discussed before.

Proposition 3.9. *If a constraint network $\mathcal{C} = \langle X, C\rangle$ is directionally path-consistent along an ordering $d = (x_n, x_{n-1}, \ldots, x_1)$, the graph corresponding to $\mathcal{C}$ is chordal with simplicial elimination ordering d.*

Proof. Let G_k denote the graph corresponding to the constraint network containing only variables x_1 through x_k, so G_n corresponds to the full constraint network. To prove our claim, it suffices to show that each variable (vertex) x_k is simplicial for G_k. For a vertex x_k to be simplicial, there must exist an explicit constraint edge $\{x_i, x_j\}$ for each pair $x_i, x_j \in \mathcal{N}(x_k)$ with $i, j < k$. Since the path $\pi = (x_i, x_k, x_j)$ is of the type mentioned in Definition 3.14 and the network is d-DPC, this edge does in fact exist. Therefore, each x_k is simplicial for G_k, d is a simplicial elimination ordering, and by Proposition 3.2, the constraint network corresponds to a chordal graph. □

Algorithm 3.5: DPC

Input: A binary constraint network $\mathcal{C} = \langle X, C \rangle$ with underlying graph $G = \langle V, E \rangle$, and a vertex ordering $d = (v_n, v_{n-1}, \ldots, v_1)$
Output: The DPC network of $\mathcal{C}$ along d

```
1 for k ← n to 1 do
2 |   forall i, j < k such that {v_i, v_k}, {v_j, v_k} ∈ E do
3 |   |   c_{x_i x_j} ← c_{x_i x_j} ∩ (c_{x_i x_k} ⊗ c_{x_k x_j})
4 |   |   E ← E ∪ {v_i, v_j}      // omit if G is triangulated along d
5 |   end
6 end
7 return S
```

As a consequence of this result, enforcing DPC on a graph along some ordering d corresponds to triangulation of the underlying graph along this ordering. When enforcing DPC, in general the goal is to keep the constraint graph as sparse as possible by employing one of the ordering heuristics discussed towards the end of Section 3.3.

From Definition 3.15, the following analogue to Proposition 3.8 follows immediately. In fact, rather than through our definition, DPC is usually defined using this weaker characterisation. They can be shown to be equivalent by an inductive argument.

Proposition 3.10. *A constraint network $\mathcal{C} = \langle X, C \rangle$ is directionally path-consistent along an ordering $d = (x_n, x_{n-1}, \ldots, x_1)$ if and only if for each pair of variables $x_i, x_j \in X$, the (possibly universal) constraint $c_{x_i x_j}$ is d-DPC with every path of length 2; so, for each path (x_i, x_k, x_j) in $\mathcal{C}$ where $k > i, j$, it must hold that $c_{x_i x_j} \subseteq c_{x_i x_k} \otimes c_{x_k x_j}$.*

Based on this characterisation, Dechter et al. (1991) proposed Algorithm 3.5, simply called DPC, to enforce directional path consistency. Note the similarity to Algorithm 3.4 that enforces full path consistency, but also to Algorithm 3.2 for naive graph triangulation. Aside from (potentially) maintaining sparsity of the constraint graph, DPC has another advantage: it does not require iteration until a fixed point is reached, whereas PC-1 does.

Since the DPC algorithm is effectively just a small extension of naive triangulation, it is not hard to see that it has the same worst-case time bounds. it runs within $\mathcal{O}(m_c w_d \log n) \subseteq \mathcal{O}(n w_d^2 \log n)$ time on an adjacency list data structure; using an adjacency matrix causes the logarithmic factor to be dropped, since line 4 then requires only constant time. However, even on an adjacency list, the logarithmic factor in DPC's run time can be avoided by first triangulating the input constraint network along d; line 4 can then be ignored. Recall that this can be accomplished with TY-triangulation in $\mathcal{O}(m_c)$ time. Thus, by splitting the algorithm into a two-step approach, it can enforce DPC on any constraint network in $\mathcal{O}(m_c w_d) \subseteq \mathcal{O}(n w_d^2)$ time and $\mathcal{O}(m_c)$ space.

Partial path consistency

In 1999, independently of DPC's authors, Bliek and Sam-Haroud proposed a third network-wide path consistency concept called *partial path consistency* (PPC). Noting that enforcing full path consistency necessarily results in a complete graph, the authors studied the question whether this property could be approximated in a way that preserves sparsity of the constraint network to some extent, and whether this might also yield improvements in computational efficiency. To this end, they introduced the concept of PPC, defined as follows.

Definition 3.16. *A constraint network $\mathcal{C} = \langle X, C\rangle$ with underlying graph $G = \langle V, E\rangle$ is* partially path-consistent *if for each pair of variables $x, y \in X$ whose corresponding vertices $u, v \in V$ are connected by an edge $\{u, v\} \in E$, the (possibly universal) constraint c_{xy} is path-consistent.*

The reader may again wish to compare this definition to Definitions 3.13 and 3.15 for FPC and DPC. Clearly, unlike FPC, sparsity of the constraint network is preserved by PPC, while at the same time the individual constraints are more thoroughly tightened than through DPC. Bliek and Sam-Haroud then proved the following analogue to Propositions 3.8 and 3.10 that requires one to only consider paths of length 2.

Proposition 3.11. *A constraint network $\mathcal{C} = \langle X, C\rangle$ with underlying chordal graph $G = \langle V, E\rangle$ is partially path-consistent if and only if for each pair of variables $x, y \in X$ represented by an edge in E, the (possibly universal) constraint c_{xy} is path-consistent with every path of length 2; so, for each path (x, z, y) in $\mathcal{C}$ with $\{x, y\} \in E$, it must hold that $c_{xy} \subseteq c_{xz} \otimes c_{zy}$.*

It must be stressed that this proposition requires the underlying graph G to be chordal. The same holds for the algorithm proposed by Bliek and Sam-Haroud to enforce partial path consistency, which depends on this result. For this reason, it is usually assumed that PPC, like DPC, implies chordality; in the remainder of this dissertation, we do the same.

We include the original PPC algorithm as Algorithm 3.6. The algorithm simply maintains a queue initialised to contain all edges in the chordal graph and iterates until this queue is empty. For each edge removed from the queue, the algorithm considers all triangles in the graph in which that edge participates, and enforces path consistency on the three 2-paths in each such triangle, enqueueing constraint edges that are changed by this operation. Thus, like PC-1 (Algorithm 3.4), the algorithm continues until a fixed point is reached.*

Having introduced these three network-wide path-consistency concepts, we conclude this section by studying the relations between them.

*In fact, long before the advent of PPC, Mackworth (1977) already proposed the PC-2 algorithm for FPC, which essentially takes an identical approach on a complete graph. Thus, in PC-2, every edge has $n - 2$ related triplets.

Algorithm 3.6: PPC

Input: A binary constraint network $\mathcal{C} = \langle X, C \rangle$ with underlying (chordal) graph $G = \langle V, E \rangle$
Output: The PPC network of $\mathcal{C}$

```
1  Queue ← E
2  while Queue ≠ ∅ do
3  |  q ← Dequeue(Q)
4  |  forall (x, y, z) ∈ Related-Triplets(q) do
5  |  |  c_xz ← c_xz ∩ (c_xy ⊗ c_yz)
6  |  |  if c_xz was changed then
7  |  |  |  Enqueue({x, z}, Queue)
8  |  |  end
9  |  end
10 end
11 return C
```

Relation between network-wide path consistency concepts

We now discuss the relative strength of the three concepts of DPC, PPC, and FPC. First, we present the following result that follows immediately from Definitions 3.13 and 3.15.

Proposition 3.12. *Given a constraint network $\mathcal{C} = \langle X, C \rangle$ that is DPC along an ordering $d = (x_n, x_{n-1}, \ldots, x_1)$, the constraint $c_{x_1 x_2}$ is path-consistent.*

Although PPC was first presented as a concept separate from DPC, the two are intimately related, as the following (new) result shows.

Theorem 3.13. *A constraint network $\mathcal{C} = \langle X, C \rangle$ with underlying (chordal) graph G is PPC if and only if it is DPC along any simplicial elimination ordering of G.*

Proof. ($\Rightarrow$) Immediate from Propositions 3.9, 3.10, and 3.11.

($\Leftarrow$) Consider a triplet of variables in $\mathcal{C}$. If they do not correspond to a triangle (i.e. a 3-clique) in G, the condition in Proposition 3.11 is trivially satisfied: either $\{x, z\} \notin E$ or at least one of c_{xz} and c_{zy} is a universal constraint. Assume therefore that they correspond to a triangle in G. Now, for any permutation π of this triplet, there exists a simplicial elimination ordering that has π as a suffix. To see this, note that maximum cardinality search (Algorithm 3.1) can produce a simplicial construction ordering that starts out with any permutation of vertices in a clique. It then follows from Proposition 3.10 that that the condition in Proposition 3.11 is fulfilled for each triangle. □

The following relation between DPC and FPC follows immediately from their definitions.

Proposition 3.14. *A constraint network $\mathcal{C} = \langle X, C \rangle$ is FPC if and only if it is DPC along any ordering of the variables X.*

With regard to the relation between PPC and FPC, note that FPC induces a complete constraint graph. Since a complete graph is trivially chordal (and any vertex ordering is simplicial), any fully path-consistent constraint network is also partially path-consistent. This also follows from the definitions of PPC and FPC (Definitions 3.13 and 3.16, respectively). Thus, we can identify a strict hierarchy.

Corollary 3.15. *The class of DPC constraint networks strictly contains the class of PPC networks, which in turn strictly contains the class of FPC networks.*

Next, we finally study the consequences of these concepts on our area of interest: the Simple Temporal Network.

3.5 Answering Simple Temporal Queries

In this section, we discuss the Simple Temporal Queries (STQs) introduced in the previous chapter in the context of fully, directionally, and partially path-consistent networks.

Consistency

As discussed in the previous chapter, the most basic query — albeit not the easiest one — is STQ 1: determining CONSISTENCY for a given STN. It is known that the DPC algorithm can be used to this end: Dechter et al. (1991) proved that a constraint interval will become empty in line 3 of Algorithm 3.5 if and only if the network is inconsistent.In the STN's distance graph perspective, this translates to the formation of a negative 2-cycle with $\omega_{xy} + \omega_{yx} < 0$. The algorithm can terminate immediately when it detects this; alternatively, a final sweep over the constraints can be made to check for such negative 2-cycles.

As it turns out, this result is not limited to Algorithm 3.5 specifically: the proof by Dechter et al. actually extends to all directionally path-consistent networks, so that we can state the following result.

Proposition 3.16. *Let $\mathcal{S} = \langle X, C \rangle$ be a Simple Temporal Network that is d-DPC for an arbitrary ordering d. Then, $\mathcal{S}$ is consistent if and only if $\forall x, y \in X$: $\omega_{xy} + \omega_{yx} \geq 0$.*

By Corollary 3.15, this result extends to PPC and FPC, so all three algorithms presented in the previous section can be used to determine an STN's CONSISTENCY and answer STQ 1.

Constraint minimality

Recall from the previous chapter that a minimal constraint in a Simple Temporal Network captures exactly the allowed time differences between a pair of time points. STQ 5 asks for a single MINIMAL CONSTRAINT, whereas the complete MINIMAL NETWORK must be given to answer STQ 6. Moreover, STQ 2, which asks to determine CONSTRAINT COMPATIBILITY — whether a given piece of temporal

information is compatible with the information already represented by the STN — becomes much easier when the corresponding minimal constraint is available.

The following result, which to the best of our knowledge has never been explicitly presented before, explains why path consistency is a very useful concept for the STN.

Proposition 3.17. *Given an STN $\mathcal{S} = \langle X, C \rangle$, a simple temporal constraint $c_{xy} \in C$ is minimal if and only if it is path-consistent.*

Proof. As discussed in the previous chapter, in the distance graph perspective, a simple temporal constraint is minimal if both of its arcs are labelled by the shortest path between their endpoints. Recall also that the composition of constraints along a path corresponds to the length of that path in the distance graph (in both directions), and note that the intersection of two simple temporal constraints corresponds to taking the minimum of the involved path lengths. This argument, together with Definition 3.12, yields the claimed result. □

Together with Proposition 3.12, this provides the following insight:

Corollary 3.18. *Given an STN $\mathcal{S} = \langle X, C \rangle$ that is DPC along an ordering $d = (x_n, x_{n-1}, \ldots, x_1)$, the constraint $c_{x_1 x_2}$ is minimal.*

So, directional path consistency can be used to answer STQ 5 asking for a single MINIMAL CONSTRAINT $c_{xy} \in C$ by means of setting x and y as the last two variables in the ordering d. Once the network is d-DPC, CONSTRAINT COMPATIBILITY of any other c'_{xy} can be checked in constant time.

As above, we have of course that this property extends to the concepts of PPC and FPC, but Proposition 3.17 allows us to show something stronger. By Definition 3.16, we immediately obtain the following result for partial path consistency.* It was already proved as an independent result by Bliek and Sam-Haroud (1999).

Corollary 3.19. *Given an STN $\mathcal{S} = \langle X, C \rangle$ that is PPC, all constraints in C are minimal.*

Thus, partial path consistency is significantly stronger than directional path consistency and yields a full complement of MINIMAL CONSTRAINTS (STQ 5), although it does not produce the full MINIMAL NETWORK (STQ 6). Moreover, for all constraint edges in the network, CONSTRAINT COMPATIBILITY of a new constraint can be checked in constant time. This result is useful enough to suggest formulating an intermediate query between these STQs, and we return to this issue at the end of this chapter.

This leaves us to discuss the strongest property, namely full path consistency. The following will come as no surprise to the reader; it follows from our Definition 3.13 and Proposition 3.17 but was already shown as an independent result in the seminal work by Dechter et al. (1991).

*Note that another way to derive Corollary 3.19 is through Theorem 3.13 and Corollary 3.18 and by making use of the fact that for each edge $\{u, v\}$ in a chordal graph, there exists a simplicial elimination ordering d in which u and v appear as a suffix.

Algorithm 3.7: Floyd–Warshall

Input: A STN $\mathcal{S} = \langle V, E \rangle$
Output: Minimal network of $\mathcal{S}$

```
for k ← 1 to n do
    for i, j ← 1 to n do
        ω_ij ← min{ω_ij, ω_ik + ω_kj}
    end
end
return S
```

Corollary 3.20. *Given an STN $\mathcal{S} = \langle X, C \rangle$ that is FPC, there is a minimal constraint $c_{xy} \in C$ for each pair of variables $x, y \in X$.*

Hence, enforcing FPC on an STN produces a MINIMAL NETWORK and answers STQ 6. Since the complete distance matrix ω is available, CONSTRAINT COMPATIBILITY can be checked in constant time for any new piece of temporal information at all, as long as it can be represented as a single new constraint.

We conclude this section with an important fact about FPC: enforcing this property on the STN is more efficient than on general constraint networks. Recall that PC-1 (Algorithm 3.4) may require many iterations of its outer loop before reaching a fixed point. However, for the STN, the following very useful result comes into play. It was proved by Montanari in 1974.

Proposition 3.21. *Let $\mathcal{C}$ be a distributive* constraint network. Then, with $\mathcal{C}$ as input, PC-1 requires only one iteration of its outer* repeat *loop to reach a fixed point, which is the minimal network of $\mathcal{C}$.*

When introducing the STN, Dechter et al. (1991) showed that the STN is distributive and that this result can thus be applied. Consider again the distance graph perspective on the STN. The single iteration of PC-1 (Algorithm 3.4) is then exactly equivalent to the Floyd–Warshall algorithm for computing all-pairs shortest paths, included as Algorithm 3.7. It is not hard to see that with an adjacency matrix, this approach requires $\mathcal{O}(n^3)$ time. The close similarity between Floyd–Warshall and PC-1 was already noted by Montanari.

Schedules

Dechter et al. (1991) showed that, given a fully path-consistent STN, any schedule can be constructed without the need for backtracking, thus answering STQ 9 of DISPATCHing an STN. Xu and Choueiry (2003) later stated without proof that the same holds for partially path-consistent networks, and their claim has been repeated in other publications (e.g. Bui et al., 2007; Planken et al., 2008;

*In a distributive constraint network, the composition operator must distribute over the intersection operator, so $(c_{ij} \cap c'_{ij}) \otimes c_{jk} = (c_{ij} \otimes c_{jk}) \cap (c'_{ij} \otimes c_{jk})$.

Algorithm 3.8: DPC–dispatch

Input: An STN $\mathcal{S} = \langle V, E \rangle$ that is DPC along $d = (v_n, v_{n-1}, \ldots, v_1)$
Output: A schedule τ for $\mathcal{S}$

1 **forall** $v_i \in V$ **do**
2 | $TW(v_i) \leftarrow \mathbb{R}$
3 **end**
4 **for** $k \leftarrow 1$ **to** n **do**
5 | pick a value τ_k from $TW(v_k)$
6 | **forall** $v_j \in \mathcal{N}(v_k)$ **such that** $j > k$ **do**
7 | | $TW(v_j) \leftarrow TW(v_j) \cap \left\{ \tau_k + x \mid x \in c_{v_k v_j} \right\}$
8 | **end**
9 **end**
10 **return** τ

Bui and Yorke-Smith, 2010). We now show that in fact, this result already holds for directionally path-consistent STNs.

Our approach is outlined as Algorithm 3.8 and operates as follows. For each time point $v \in V$, an initially unconstrained time window interval $TW(v)$ is kept. Then, for each time point iterated over, any value from its time window can be assigned as part of the schedule being assembled. When this is done, the time windows of neighbouring, unvisited vertices are updated (shrunk) by removing those values not consistent with the newly assigned value. This process continues until a full schedule is constructed.

Now, we prove some properties of this approach. Let us first show that it is indeed backtrack-free, as claimed.

Theorem 3.22. *DPC–dispatch is well-defined.*

Proof. Only line 5 is potentially problematic: the operation is undefined if the time window for the current time point is empty, so the algorithm reaches a dead end. We show that, when run with a consistent STN as input, this will not occur.

To reach a contradiction, we assume the contrary: $TW(u) = \varnothing$ when trying to pick a value τ_u for some $u \in V$. For this to happen, it must hold that two neighbouring time points $v_1, v_2 \in V$, both tagged before u, enforce mutually non-overlapping time windows for τ_u in line 7:

$$\{\tau_{v_1} + x \mid x \in c_{v_1 u}\} \cap \{\tau_{v_2} + x \mid x \in c_{v_2 u}\} = \varnothing$$

Recall that d is a simplicial elimination ordering of the constraint graph and its reverse, i.e. the order in which time points are visited by the algorithm, is a simplicial construction ordering. Thus, u must be simplicial with respect to v_1 and v_2 because it is visited after both of them, and therefore, there must exist a constraint $c_{v_1 v_2}$. Further, because $\mathcal{S}$ is DPC along d, this constraint $c_{v_1 v_2}$ is path-consistent with all constraints involving time points visited after both v_1

and v_2, and specifically with c_{v_1u} and c_{uv_2}. Now, there exists no schedule for the pair of assignments (τ_{v_1}, τ_{v_2}) because it cannot be extended to u; hence, it must follow that this pair violates $c_{v_1v_2}$. Assuming w.l.o.g. that v_1 was visited before v_2, the value τ_{v_2} was removed from $TW(v_2)$ when iterating over v_1's unvisited neighbours after picking the value τ_{v_1}. Therefore, this assignment could not have been made and a contradiction is reached. □

The time complexity and correctness of the algorithm are easier to prove.

Theorem 3.23. *DPC–dispatch runs in $\Theta(m_c)$ time.*

Proof. The algorithm first iterates over all vertices, then over all edges; the operations during each iteration require $\mathcal{O}(1)$ time. Recall that there are m_c edges in the STN and that we assume the constraint graph to be connected, so $n \in \mathcal{O}(m_c)$.* Using an adjacency list data structure, the claimed bound then follows. □

Theorem 3.24. *An assignment τ is a schedule for an STN $\mathcal{S}$ that is DPC along d if and only if τ can be produced when running DPC–dispatch on $(\mathcal{S}, d)$.*

Proof. ($\Leftarrow$) Assume that DPC–dispatch produces an assignment τ that is not a schedule for $\mathcal{S}$. Then, there must exist a constraint c_{uv} that is violated, i.e. $\tau_v - \tau_u \notin c_{uv}$. Assume w.l.o.g. that the value τ_u is assigned before τ_v. Then, line 7 ensures that $\tau_v \notin TW(v)$, so the algorithm cannot assign τ_v to time point v: a contradiction.

($\Rightarrow$) Let τ be an arbitrary schedule for $\mathcal{S}$. Then, in the vertex order followed by the algorithm, each value from τ can be assigned to its respective time point. Since τ is by definition consistent with all constraints, the operation in line 7 will never remove a value to be assigned to any time point $v \in V$ from $TW(v)$. □

We can therefore summarise our findings in the following way.

Corollary 3.25. *Algorithm 3.8 can be used to* DISPATCH *a d-DPC STN $\mathcal{S} = \langle X, C \rangle$ and thus answer STQ 9 in $\Theta(m_c)$ time, assigning values to time points in reverse order of d.*

Recall that by Theorem 3.13, a PPC network is DPC along any simplicial elimination ordering. Hence, we extend this result to PPC as follows.

Corollary 3.26. *Algorithm 3.8 can be used to* DISPATCH *a PPC STN $\mathcal{S} = \langle X, C \rangle$ and thus answer STQ 9 in $\Theta(m_c)$ time, assigning values to time points in any simplicial construction ordering of $\mathcal{S}$.*

We can further extend this result to FPC by Proposition 3.14, which states that a FPC network is DPC along any variable ordering whatsoever. Note that a complete constraint graph, as required for FPC, has $m_c \in \Theta(n^2)$, and that all vertices $v \in V$ then have $n - 1$ neighbours. Thus, given an FPC STN, Algorithm 3.8 can be run with an arbitrary order of time points to construct a schedule, which yields the $\mathcal{O}(n^2)$-time method mentioned by Dechter et al. (1991).

*For an explanation of our use of the notation $x \in \mathcal{O}(f(n))$, see the footnote on page 16.

Corollary 3.27. *Algorithm 3.8 can be used to* DISPATCH *an FPC STN* $\mathcal{S} = \langle X, C \rangle$ *and thus answer STQ 9 in* $\Theta(n^2)$ *time, assigning values to time points in any ordering of the time points* X*.*

3.6 Summary and discussion

In this chapter, we investigated how constraint theory might be applied to the Simple Temporal Network (STN) to produce normal forms for simple temporal information. Our motivation was the fact, already known since the STN was first proposed by Dechter et al. in 1991, that enforcing full path consistency on the STN produces its minimal netwok, i.e. the canonical normal form for temporal information. Before presenting a discussion of this chapter and identifying topics for future work, let us first sum up its contents and highlight our contributions.

Summary

We first discussed in Section 3.1 how a STN might be represented in practice on a computer. In broad lines, the choice is between an adjacency list and an adjacency matrix. These require linear and quadratic space, respectively; however, the matrix representation offers constant-time access to an arbitrary edge and its weight, whereas for the adjacency list, this takes at least logarithmic time. An adjacency matrix also requires quadratic time to set up, which may be significant in some applications. However, once one has made this investment, an ancillary adjacency list can be maintained for no extra (asymptotic) cost. Finally, the band matrix is a data structure that may be useful in special cases, where the STN is known to have a certain structure.

In Sections 3.2 and 3.3, we respectively introduced the class of chordal constraint graphs and the process of triangulation, i.e. making graphs chordal. We introduced the concepts of treewidth and induced width, which appear in the time and space bounds of many of the algorithms introduced in this chapter as well as the remainder of this dissertation. We also presented the approach dubbed "TY triangulation" (Algorithm 3.3) due to Tarjan and Yannakakis (1984) which allows a triangulation to be found in linear time, and which has so far received virtually no attention at all in the STN literature.

Then, in Section 3.4, we reviewed the constraint-theoretical concepts of full, partial, and directional path consistency (abbreviated respectively FPC, PPC, and DPC). For the first time, we defined these three concepts together, from the ground up, and emphasised the close parallels between them. Where this approach led to a definition different from the characterisations customary in the literature, we proved that they were equivalent (cf. Definition 3.15 and Proposition 3.10 for DPC). We proved in Proposition 3.9 that enforcing DPC results in a chordal constraint graph. Through Corollary 3.15, we identified a strict hierarchy between these three concepts, in that every FPC constraint network must be PPC, and every PPC constraint network must be DPC along many variable orderings.

Section 3.5 finally saw a return to the domain of the STN and investigated the consequences of DPC, PPC, and FPC on this special type of constraint network. We derived again the well-known fact that enforcing FPC on the STN produces the minimal network, answering STQ 6, and showed that the canonical PC-1 algorithm now directly corresponds to the Floyd–Warshall algorithm for computing all-pairs shortest paths. We showed that DPC can be used to answer STQ 1, as well as STQs 2 and 5 for a single given constraint edge. PPC was shown to additionally solve the latter two STQs for any constraint edge present in the STN, whereas FPC of course answers them for any pair of time points. We also proved for the first time (in Corollary 3.25) the much-repeated claim that in $\mathcal{O}(m_c)$ time, Algorithm 3.8 can construct an arbitrary schedule for PPC networks in a backtrack-free manner, thus answering STQ 9; in fact, this result already holds for DPC networks. For FPC networks, this corresponds exactly to the $\Theta(n^2)$-time approach originally outlined by Dechter et al. (1991).

Discussion

The results presented in Section 3.5 show that partial path consistency is a powerful concept in the realm of Simple Temporal Networks. After establishing PPC, almost all Simple Temporal Queries identified in the previous chapter can be answered as efficiently as on a fully path-consistent STN. Moreover, the PPC network requires only $\mathcal{O}(m_c)$ space and thus has the potential to maintain sparsity of the STN, whereas FPC always produces a complete graph taking up $\mathcal{O}(n^2)$ space. The next chapter investigates whether this advantage extends to the time complexity of establishing PPC on the STN.

There are only two special cases of queries for which an answer is available from a (complete) minimal STN, but not from the chordal PPC network: (i) asking for a MINIMAL CONSTRAINT between an arbitrary pair of time points, not necessarily connected by a constraint; and (ii) assembling a schedule, i.e. DISPATCHing the STN, along an arbitrary ordering of timepoints.

For case (i), if the set of such query pairs is known up front, they can be added to the STN as universal constraints (i.e. with infinite edge weights in the distance graph) before enforcing PPC. Thus, if the set is not too big, enforcing PPC may still maintain an edge over enforcing FPC. Regarding case (ii), a similar work-around is available if this ordering is known beforehand, by ensuring that the ordering constitutes a simplicial construction ordering for the STN, i.e. triangulating the network along the reverse ordering. Within reason, it is even possible to cater to multiple such orderings. Again, one pays for this in the number of constraint edges to maintain and in the space required by the network.

In contrast to full and partial path consistency, establishing directional path consistency does not hold promise as a way to attain a normal form for the STN; the number of queries that can be answered is very limited. Although Dechter et al. explicitly proposed to use the DPC algorithm to determine CONSISTENCY of STNs, even for this limited use there are better approaches. Since this problem is equivalent to negative cycle detection, algorithms such as the $\mathcal{O}(nm)$-time

Bellman–Ford can be used as well and usually perform better in practice.

We can identify several topics for future research from the results presented in this chapter. To begin with, we showed in Section 3.5 that each of the network-wide path-consistency concepts becomes much stronger when applied to the STN. However, recall from Proposition 3.21, due to Montanari (1974), that for FPC this holds for any distributive constraint network. It is therefore reasonable to ask the question whether all of the STN-specific results presented in Section 3.5 in fact hold for this wider class of problems. This includes for instance the questions whether DPC detects inconsistency for any distributive CSP instance, and a solution can then be assembled in $\mathcal{O}(m_c)$ time, through an analogue of Algorithm 3.8; or whether all constraints in a PPC distributive CSP instance are minimal.

Another topic for future research is inspired by the work-around for DISPATCHing an STN along an arbitrary ordering of time points, as proposed above. Recall that a DPC network can be DISPATCHed along only a single ordering, whereas for an FPC network, any ordering at all can be used. Obviously, the PPC network is somewhere in between. Can the set of allowed orderings be characterised in some intuitive way, other than as simplicial construction orderings? For example, given some partial order $\hat{d}$ of time points, can we produce a PPC network that can be DISPATCHed along any total order d compatible with $\hat{d}$, preferably maintaining sparsity of the network as much as possible?

We conclude this section with a glance forward at the remainder of this work. Now that we have established that the two most useful normal forms for simple temporal information are attained through establishing PPC and FPC, we discuss in the following chapters how these normal forms can be found and maintained.

First, we discuss how to establish PPC. Since the presentation of the general algorithm by Bliek and Sam-Haroud in 1999, three more algorithms aimed at the special case of the STN have been proposed. We present them, and compare their performance, in Chapter 4. Then, in Chapter 5, we look at the more general task of establishing FPC on the STN. Although it is equivalent to computing all-pairs shortest paths on a graph with real edge weights, for which the state of the art is several decades old, we will present a new, competitive approach based on insights from this chapter and the next. Finally, Chapter 6 studies the problem of maintaining either PPC or FPC when new temporal information becomes available.

CHAPTER 4

Partial Path Consistency*

In the previous chapter, we described the concepts of full, directional, and partial path consistency (respectively FPC, DPC and PPC) and showed that an STN on which PPC has been enforced has two desirable properties: (i) the constraints in this STN are labelled by their minimal weights; and (ii) this STN can be DISPATCHed in linear time. These properties are shared by the minimal network produced by enforcing full path consistency, but the PPC network has the advantage of requiring less space. Therefore, algorithms enforcing PPC have the potential to be more time-efficient as well.

For these reasons, the focus problem of this chapter is to enforce *partial path consistency* on a given STN. We first investigate the current state of the art of algorithms that tackle either exactly this problem or two related problems: the more general problem of enforcing FPC (equivalently, computing all-pairs shortest paths), described in the previous chapter; and a more specific problem, which is to compute multiple-pairs shortest paths (MPSP). The main reason for also including a discussion of algorithms for solving these problems is that they can be used for enforcing partial path consistency as well. Computing full path consistency inherently also determines partial path consistency; the resulting graph is complete and thus trivially chordal. Any approach to the more specific problem of multiple-pairs shortest paths can be set up to compute exactly the shortest paths between all edges in the triangulated graph. Moreover, if it is not required to find a satisfying assignment to the variables in the STN, finding MPSP still has property (i) above; conversely, any algorithm for PPC (or FPC) can be used to compute MPSP.

We discuss extant solution approaches, some of which have not been applied to the STN before, to the best of our knowledge; we also present new theoretical insights into their efficiency. Then, we introduce P^3C, a new algorithm for

*This chapter includes work from the author's Master thesis (Planken, 2008a) and a conference paper (Planken et al., 2008).

enforcing PPC. We show that P^{3}C's theoretical time complexity is best among the class of PPC algorithms and is optimal for the class of STNs with constant treewidth. We verify that this advantage holds up in practice by empirically comparing our new algorithm against its competitors, while at the same time investigating the trade-off between quality and efficiency of the ordering heuristics presented in the previous chapter. Thus, P^{3}C represents the new state of the art for enforcing partial path consistency.

4.1 Known algorithms

In this section, we investigate extant solution approaches for PPC and two related problems. We discuss them in decreasing order of generality, starting with the problem of enforcing FPC, since this is perhaps the most straightforward of the three problem variants. After that, we discuss algorithms for enforcing PPC and finally those that compute MPSP.

All problems are defined on a weighted directed (distance) graph $G = \langle V, E \rangle$. The possibly negative weight on an arc from i to j is represented as $\omega_{ij} \in \mathbb{R} \cup \{\infty\}$. We use n and m to denote the number of vertices in V and edges in E, respectively.

Full path consistency

As mentioned in the previous chapter, since the STP is convex, enforcing FPC on it produces the minimal network and is equivalent to computing all-pairs shortest paths (APSP) on its STN. Here, we briefly discuss the two algorithms that represent the state of the art in finding APSP, and which we also use to benchmark our ideas: Floyd–Warshall and Johnson's algorithm (simply referred to as Johnson). The topic of computing APSP returns in the next chapter, where we propose a new algorithm for this problem.

The Floyd–Warshall algorithm applies dynamic programming to compute all-pairs shortest paths in $\Theta(n^3)$ time and $\Theta(n^2)$ memory through three nested loops over all vertices (Floyd, 1962). The algorithm is based on Warshall's (1962) formulation of efficiently computing the transitive closure of Boolean matrices. As mentioned in Chapter 3, this algorithm is very similar to the algorithm PC-1 for enforcing full path consistency (FPC), known from constraint satisfaction theory (Montanari, 1974). All that lies hidden inside the three loops are three array read operations, an addition and a comparison, and an optional array write operation. Therefore, the cubic upper bound hides only a small constant factor. The algorithm is also very simple to implement, making it one of the most popular methods for computing all pairs-shortest-paths.

However, the state of the art for computing APSP on sparse graphs is an algorithm based on the technique originally proposed by Johnson (1977), which does some preprocessing to allow n runs of Dijkstra's (1959) algorithm. Using a Fibonacci heap* (Fredman and Tarjan, 1987), the algorithm runs in $\mathcal{O}(n^2 \log n +$

*For the Fibonacci heap, note that the widely used pseudocode listed by Cormen et al. (2001) contains mistakes; Fiedler (2008) proposes corrections.

Algorithm 4.1: △STP

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

```
1  Queue ← all triangles in S
2  while Queue ≠ ∅ do
3  |  T ← some triangle from Queue
4  |  foreach permutation (v_i, v_j, v_k) of T do
5  |  |  ω_ik ← min{ω_ik, ω_ij + ω_jk}
6  |  |  if ω_ik has changed then
7  |  |  |  if ω_ik + ω_ki < 0 then return INCONSISTENT
8  |  |  |  Queue ← Queue ∪ {triangle T' in S | v_i, v_k ∈ T'}
9  |  |  end
10 |  end
11 |  Queue ← Queue \ T
12 end
```

nm) time; the simpler binary heap gives a bound of $\mathcal{O}(nm \log n)$. For more details on the difference between these data structures in practice, refer to Section 5.3 in the next chapter.

Partial path consistency

We described in the previous chapter that the property of partial path consistency was proposed by Bliek and Sam-Haroud (1999). In this section, we describe the two algorithms available from literature that enforce PPC on the STP, giving a first comparison of their strengths and weaknesses from a theoretical perspective.

△STP

We mentioned in the previous chapter that Xu and Choueiry (2003) were the first to realise that the STP is a convex problem and that PPC therefore computes minimal constraints for it. They published a version of PPC for the STP, called △STP, which we include as Algorithm 4.1. The algorithm maintains a queue of triangles, initially containing all triangles in the chordal STN, from which it dequeues triangles to process, one by one. When a constraint edge in a triangle is changed, all other triangles containing that edge are enqueued (if they were not already). This process continues until the queue is empty.

The authors of △STP discussed three variations of the algorithm that have to do with adding a triangle to the queue in line 8. They showed that appending the triangle at the back of the queue always resulted in fewer constraint checks than adding the triangle at the front of the queue (i.e. treating it as a stack) or inserting it in a random spot. We follow their lead and consider only the variant where the triangle is appended at the back.

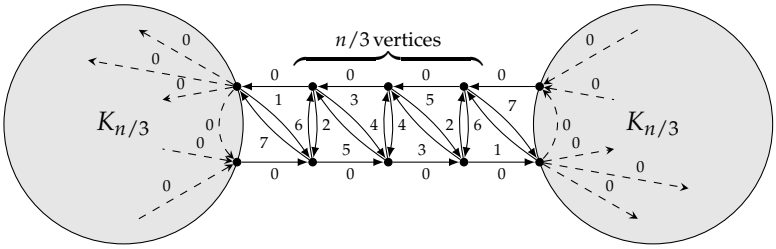


Figure 4.1: Structure of the new pathological instances $\mathcal{P}^*$

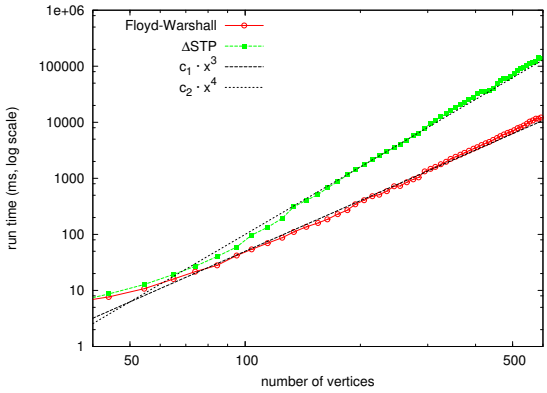


Figure 4.2: Performance of △STP and Floyd–Warshall on the new pathological instances $\mathcal{P}^*$

△STP is attractive because its pseudocode is straightforward and intuitive. Its authors did not give a formal analysis, but empirically showed that it outperformed both Floyd–Warshall and the original PPC algorithm by Bliek and Sam-Haroud. However, we can identify two weaknesses. First, since there is a tight upper bound of $\mathcal{O}(nw_d^2)$ on the number of triangles in an STN, all of which have to be kept in memory at least initially, the memory requirements of the algorithm may be large. For (almost) complete graphs, where $w_d \in \Omega(n)$, △STP requires $\Omega(n^3)$ memory.* Secondly, the queueing process followed by the algorithm may cause triangles to be processed many times. We now show that there exist instances for which △STP requires $\Omega(n^4)$ time.

Our approach is an extension of the pathological instances we presented before (Planken et al., 2008): the class $\mathcal{P}$ of STNs consisting of a chain of triangles with edge weights so selected that between the pairs of vertices at the ends there exist paths of progressively lower total weight as their length increases. For the new class $\mathcal{P}^*$, we tack a clique onto each end of this truss-like structure; see Figure 4.1. In these networks, the original edge weights within the cliques are infinite. In the example figure, taking a single step into the truss from either direction yields a path with total weight 7. For each additional step the weight decreases, until eventually the shortest path of length 0 is found.

*For an explanation of our use of the notation $x \in \Omega(f(n))$, see the footnote on page 16.

△STP maintains a queue of triangles to process, initially containing all triangles in the network, and the algorithm effectively needs to process its entire queue to propagate each step through the truss. Therefore, when solving these pathological instances, all triangles are processed $\Omega(n)$ times. Whereas the networks from $\mathcal{P}$ had $\Theta(n)$ triangles and treewidth $w^* \in \mathcal{O}(1)$, $\mathcal{P}^*$ contains STNs with $\Omega(n^3)$ triangles, i.e. $w^* \in \Omega(n)$. For $\mathcal{P}$ this resulted into an $\Omega(n^2)$ run-time performance of △STP, but for $\mathcal{P}^*$ we find a complexity of $\Omega(n^4)$, i.e. strictly worse than Floyd–Warshall. In Figure 4.2, we include empirical evidence of this claim, plotted on a log-log scale. Floyd–Warshall's $\Theta(n^3)$-time performance is included for reference. Based on both sets of pathological instances, we can claim that the worst-case time complexity for △STP cannot be better than $\mathcal{O}(n^2 w_d^2)$.

This analysis suggests that △STP's queue of triangles, irrespective of the triangle insertion policy, is not the optimal way of dealing with STNs. This was recognised by Choueiry and Wilson (2006), who proposed to take the simplicial elimination ordering into account when dealing with chordal STNs. Next, we describe an algorithm which does just that.

Prop-STP

Prop-STP (Bui et al., 2007; Yorke-Smith and Bui, 2010) was proposed as a solver for semi-structured STP instances; that is, STNs for which some structural information on a higher level than the graph representation is available. Although its authors do not explicitly state that Prop-STP enforces PPC, this follows implicitly from the fact that it produces a chordal STN in which all constraints are minimal. The problem domain addressed by the authors is the class of STNs originating from hierarchical task networks (HTNs). The HTNs at the root of these STNs have a tree structure which is also apparent in the constraint graphs.

Prop-STP exploits this structural information by computing a tree decomposition of the STN. It adapts the approach of cluster tree elimination (CTE; Dechter, 2003, pp. 258–261), which was proposed for standard CSPs with discrete variables. Because variables in the STP have continuous domains, using this method is not trivial. The authors of Prop-STP present a CSP-theoretical proof that the messages required by the CTE approach can be represented efficiently as sub-STNs, and moreover, that it is not required to pass these messages explicitly; they are contained in the STN, so a reference suffices.

The CTE approach requires two passes of messages, from the leaves of the tree decomposition up to its root and back down again. From the STP perspective, the consequence is that on both passes, a sub-solver is run locally for each of the tree nodes in turn; the constraint graph is triangulated as a preprocessing step. Each sub-STN considered is a complete graph of at most $w_d + 1$ vertices (since each tree node represents a clique no larger than that) and the sensible choice for the sub-solver is therefore Floyd–Warshall, resulting in $\mathcal{O}(w_d^3)$ work per clique and, for K cliques, a $\mathcal{O}(K w_d^3)$ time complexity overall (assuming no inconsistency is discovered). Now, since it holds that $K \leq n - w_d$, we can also state a looser bound of $\mathcal{O}(n w_d^3)$. While conducting the experiments presented in

Algorithm 4.2: Prop-STP

Input: A chordal STN $\mathcal{S} = \langle V, E \rangle$ with a simplicial elimination ordering $d = (v_n, v_{n-1}, \ldots, v_1)$
Output: The PPC network of $\mathcal{S}$ or INCONSISTENT

```
1  Stack ← ∅
2  for k ← n to 1 do
3  |  C ← {v_k} ∪ {v_j | j < k ∧ {v_j, v_k} ∈ E}
4  |  if C is a maximal clique in S then
5  |  |  push v_k onto Stack
6  |  |  run Floyd–Warshall on S_C = ⟨C, E_C⟩
7  |  |  return INCONSISTENT if Floyd–Warshall did
8  |  end
9  end
10 discard the top entry from Stack      // we just processed that clique
11 while Stack ≠ ∅ do
12 |  pop v_k from Stack
13 |  C ← {v_k} ∪ {v_j | j < k ∧ {v_j, v_k} ∈ E}
14 |  run Floyd–Warshall on S_C = ⟨C, E_C⟩
15 end
16 return S
```

Section 4.3, we found that for our benchmarks, K was in most cases very close to the upper bound of $n - w_d$ given above. Note also that as the cliques grow larger, their number decreases: as w_d approaches its maximum value of $n - 1$, K approaches 1.

Prop-STP was originally presented in terms of the tree decomposition. For simplicity and to avoid introducing extra notation, we include as Algorithm 4.2 an equivalent formulation of the algorithm in terms of just the STN and the simplicial elimination ordering d. Note that, given a chordal graph and a simplicial elimination ordering, the check for clique maximality in line 4 can be done in constant time. The order in which the cliques are processed in our statement of the algorithm is a specific instance of the leaves-to-root type of orderings required for the algorithm. Many other valid orderings exist, but each results in the same amount of work performed by the algorithm.

Yorke-Smith and Bui (2010) performed experiments where Prop-STP was pitted against △STP and Floyd–Warshall. Their algorithm might have had a slight edge, but the difference was not big enough for the authors to declare a clear winner, though both algorithms outperformed Floyd–Warshall by a large margin in most cases. A reason why △STP was not outperformed by a larger margin may lie in the fact that Prop-STP, also, may perform some unnecessary work. This is because the cliques it processes often overlap: each triangle of constraints may be part of many cliques, represented by connected nodes in the tree decomposition.

However, in Section 4.3, we show that in the experiments on our benchmarks,

Prop-STP usually does outperform △STP by a wide margin; further, we present empirical evidence for the problem of overlapping cliques. Whereas one is usually looking for a triangulation in which the size of the largest clique (i.e. w_d) is kept small, Prop-STP was sometimes found to benefit from a coarser triangulation, where several small, overlapping cliques are merged into a single large clique.

Multiple-pairs shortest paths

Enforcing full or partial path consistency has the advantage that one can assemble a consistent assignment of values to time-point variables in the STP efficiently. However, if one is only interested in finding some set of minimal constraints, it suffices to compute shortest paths for each such pair of time points we are interested in (e.g. those originally connected by a constraint). In graph terms, this means that we want to compute shortest paths not for all pairs of vertices or from a single source to all other vertices, but for a set $Q = \{(s_i, t_i) \mid 1 \leq i \leq q\}$ of source–destination *query pairs*, thus solving the problem of computing multiple-pairs shortest paths (MPSP).

For our purposes, we construct the set Q from the set of edges E in the STN. Every edge $\{u, v\} \in E$ gives rise to two query pairs $(u, v), (v, u) \in Q$; thus, $q = 2m$. Note that this is a very specific use case for MPSP algorithms, which may well perform better when Q is smaller, less symmetric, or less evenly distributed over the graph.

In this section, we first introduce a "true" MPSP algorithm. To provide it with some competition in its category, we then describe how the Bellman–Ford single-source shortest paths algorithm and Johnson's APSP algorithm can be adapted to compute MPSP.

DLU

The algorithm DLU was proposed in 2005 by Wang et al. as the first specific algorithm for MPSP, to the best of our knowledge. Instead of computing all-pairs shortest paths and discarding the surplus information, or computing single-source shortest paths from the involved vertices, it performs some preprocessing and then computes the shortest path for each query pair. It is inspired by matrix techniques such as LU decomposition and maintains its information in an n-by-n matrix. It therefore requires $\Theta(n^2)$ space. The time complexity is $\mathcal{O}(n^3)$, with no asymptotic edge over Floyd–Warshall; however, the authors experimentally show that on a benchmark constructed from real-world flight networks, DLU is the most efficient choice among the methods they compared.

We include DLU as Algorithm 4.3. The algorithm begins with a preprocessing step called `A_LU` (lines 3 to 10) which upon closer inspection turns out to be equivalent to DPC and was independently rediscovered by DLU's authors. Next, it iterates over the query pairs Q. For each such pair (s_i, t_i), the subroutine `Get_D_L`(t_i) computes the shortest path tree to t_i in the directed acyclic subgraph of the STN with only arcs pointing "downward" in terms of the vertex order d;

Algorithm 4.3: DLU

Input: An STN instance $\mathcal{S} = \langle V, E \rangle$, a set of query pairs $Q = \{(s_i, t_i) \mid 1 \leq i \leq q\}$, and an ordering $d = (v_n, v_{n-1}, \ldots, v_1)$

Output: Distance matrix D containing shortest distances between all query pairs, or INCONSISTENT

```
1  initialise distance matrix D from S
2  forall v ∈ V do DoneDL[v] ← FALSE ; DoneDU[v] ← FALSE
3  for k ← 1 to n − 1 do                                              // A_LU
4  |  for s ← k + 1 to n do
5  |  |  for t ← k + 1 to n do
6  |  |  |  D[s][t] ← min{D[s][t], D[s][k] + D[k][t]}
7  |  |  end
8  |  |  if D[s][s] < 0 then return INCONSISTENT
9  |  end
10 end
11 for i ← 1 to q do
12 |  if not DoneDL[t_i] then
13 |  |  DoneDL[t_i] ← TRUE
14 |  |  for s ← t_i + 2 to n do                                      // Get_D_L(t_i)
15 |  |  |  for k ← t_i + 1 to s − 1 do
16 |  |  |  |  D[s][t_i] ← min{D[s][t_i], D[s][k] + D[k][t_i]}
17 |  |  |  end
18 |  |  end
19 |  end
20 |  if not DoneDU[s_i] then
21 |  |  DoneDU[s_i] ← TRUE
22 |  |  for t ← s_i + 2 to n do                                      // Get_D_U(s_i)
23 |  |  |  for k ← s_i + 1 to t − 1 do
24 |  |  |  |  D[s_i][t] ← min{D[s_i][t], D[s_i][k] + D[k][t]}
25 |  |  |  end
26 |  |  end
27 |  end
28 |  for k ← max{s_i, t_i} + 1 to n do                              // Min_Add(s_i, t_i)
29 |  |  D[s_i][t_i] ← min{D[s_i][t_i], D[s_i][k] + D[k][t_i]}
30 |  end
31 end
32 return D
```

conversely, `Get_D_U`(s_i) computes the shortest path tree from s_i in the upward-pointing subgraph. Finally, `Min_Add` merges these shortest path trees to correctly compute the shortest distance from s_i to t_i in the entire STN, as proven by the authors, and stores it in $D[s_i][t_i]$.

Since DLU starts out by effectively computing DPC, it requires a vertex or-

dering as input. The authors remark that the specific ordering used may have a large influence on the algorithm's run time in practice, and that for optimal efficiency the ordering should take both the structure of the graph and the set Q into account. However, for further details, they refer to an unpublished technical report. The first author's PhD thesis (Wang, 2003) does consider the effect of several ordering heuristics on DLU's run time, although the time spent computing the ordering itself is not taken into account. It was found there that DLU performed best with a minimum degree vertex ordering. In Section 4.3, we perform experiments with several vertex ordering heuristics, reporting on the combined run times of the heuristic and DLU itself for the case where Q coincides with the graph structure.

Repeated Bellman–Ford

Instead of using a dedicated MPSP algorithm, another possibility is to use a single-source shortest paths (SSSP) algorithm and iterate over the vertices in the graph, running it once for each vertex as source or destination. Wang et al. (2005) compared their DLU algorithm against multiple SSSP algorithms. From those that fit within an $\mathcal{O}(n^3)$-time bound, the algorithm they refer to as BFP always performed best. This is Bellman–Ford, supplemented with the *parent checking* heuristic proposed by Cherkassky et al. (1996). In short, with this simple adaptation, a vertex v is ignored if the queue still contains its parent, i.e. a vertex u which is known from a previous iteration to precede v on the shortest path from the source. The asymptotic time complexity is unaffected and remains $\mathcal{O}(nm)$, which yields $\mathcal{O}(n^2m)$ when repeating it for all vertices.

It is not hard to modify Bellman–Ford so that given some vertex v, it computes at the same time shortest paths originating and ending in v. With this modification, it could be said to be a two-way approach, where the original is one-way, and instead of running it for all vertices in the graph, it only requires a vertex cover. One can find a 2-approximation of an optimal vertex cover in linear time using the approach by Gavril (1974; see Garey and Johnson, 1979). The parent checking heuristic can be applied in the two-way approach, but the filtering effect is weaker: a vertex can only be ignored if it has a parent in the queue for both directions of search. Thus, an open question is whether it is of benefit in this case.

Altogether, then, we considered four variants of BF–MPSP, using either the one-way or the two-way implementation, and with or without the parent checking heuristic. They are abbreviated 1, 1-pc, 2, and 2-pc. We performed experiments with all of them and describe our results in Section 4.3.

Modified Johnson's algorithm

A third possibility is to modify Johnson's algorithm to compute shortest paths between only a subset of pairs of vertices, instead of all pairs. This approach is similar to the one-way BF–MPSP described above; the chief differences are that the graph is reweighted to get rid of negative edge weights and that Dijkstra's algorithm is used instead of Bellman–Ford. When computing shortest paths from

some vertex u, Dijkstra's algorithm normally runs until its priority queue is empty and distances to all vertices have been computed. In this variant, we stop Dijkstra's algorithm as soon as the distances to all of u's neighbours are known.

This modification is not guaranteed to actually improve over the original algorithm's run time, since the shortest path may run through the entire graph. Therefore, the time bound of $\mathcal{O}(nm + n^2 \log n)$ on the original algorithm still holds, although we expect to see some improvement in practice. However, before we investigate this, we propose our new algorithm for PPC in the next section.

4.2 A new algorithm for partial path consistency

In the previous section, we described the best known methods for enforcing partial path consistency (PPC): △STP and Prop-STP; Recall that the worst-case time complexity of Prop-STP is $\mathcal{O}(nw_d^3)$. △STP's is not better than $\mathcal{O}(n^2 w_d^2)$ and even requires $\Omega(n^4)$ time on some instances as demonstrated above, performing strictly worse than Floyd–Warshall. We described how Prop-STP takes more care of the order in which it processes the STN's constraints, traversing a tree decomposition of the constraint graph. Both these algorithms have the shortcoming that they may process some parts of the constraint network multiple times unnecessarily.

In this section, we present a new algorithm that processes constraints in a principled order like Prop-STP does. Our algorithm, called P^{3}C, has an $\mathcal{O}(nw_d^2)$ time complexity, which is equal to the bound on DPC . In fact, P^{3}C can be seen as something of a complement to DPC: instead of performing a sweep along the simplicial elimination ordering, it iterates over vertices in the opposite direction. Its pseudocode bears great similarities to DPC's and is very concise. It is included as Algorithm 4.4. As expected, the outer loop over k is exactly the reverse of DPC's; the inner loop is identical. Within the inner loop, the difference is that only the constraints among v_i, v_j, v_k involving v_k are updated instead of those *not* involving it. Finally, no check for inconsistency is required anymore, since this is detected by DPC . Note that, unlike Prop-STP and △STP, P^{3}C is guaranteed to process each triple of time-point vertices at most twice, thus potentially avoiding much unnecessary work.

We now prove the correctness of the algorithm.

Theorem 4.1. *Algorithm P^{3}C achieves PPC on consistent chordal STNs.*

Proof. Recall that for the STN, enforcing PPC corresponds to computing minimal constraints for every pair of vertices connected by an edge in the chordal graph.

The algorithm first enforces DPC along d. As shown in Chapter 3, after this step, every edge $\{i, j\}$ is labelled by the shortest path in the subgraph induced by $\{v_k \in V \mid k > \max(i, j)\} \cup \{v_i, v_j\}$. This means in particular that c_{12} is already minimal when P^{3}C proper starts its loop.

We now show by induction that after iteration k of the loop, all constraint edges in the subgraph G_k induced by $\{v_i \in V \mid i \leq k\}$ are minimal. The base case for $k \leq 2$ has already been shown to hold. Assuming that the proposition holds

Algorithm 4.4: P^{3}C

Input: A chordal STN $\mathcal{S} = \langle V, E \rangle$ with a simplicial elimination ordering $d = (v_n, v_{n-1}, \ldots, v_1)$
Output: The PPC network of $\mathcal{S}$ or INCONSISTENT

```
1 call DPC(S, d)
2 return INCONSISTENT if DPC did
3 for k ← 1 to n do
4 |   forall i, j < k such that {v_i, v_k}, {v_j, v_k} ∈ E do
5 |   |   ω_ik ← min{ω_ik, ω_ij + ω_jk}
6 |   |   ω_kj ← min{ω_kj, ω_ki + ω_ij}
7 |   end
8 end
9 return S
```

for $k-1$, let us show that it also holds for k. Consider any constraint c_{ik} with $i < k$; by the induction hypothesis, we know that all constraints c_{ij} with $i, j < k$ are already minimal. To arrive at a contradiction, assume that c_{ik} is not minimal after the kth iteration; i.e. after the iteration completes, there still exists some path $\pi = (v_i = v_{j_0}, v_{j_1}, \cdots, v_{j_{\ell-1}}, v_{j_\ell} = v_k)$ with length $\ell \geq 3$ and total weight $\omega_\pi < \omega_{ik}$. We will show that this cannot occur.

Let v_{j_h} be the vertex in π appearing first in d. If $j_h > k$, we have $0 < h < \ell$, and by the DPC property (Proposition 3.10) we can replace (j_{h-1}, j_h, j_{h+1}) by the shortcut (j_{h-1}, j_{h+1}). This line of reasoning can be repeated so that we may assume that π lies entirely within G_k.

Now, v_k is simplicial with respect to its neighbours $v_{j_{\ell-1}}$ and v_i since d orders it before both, so there must exist a constraint edge $\{v_i, v_{j_{\ell-1}}\} \in E$. By the fact that $i, j_{\ell-1} < k$ together with the induction hypothesis, $c_{ij_{\ell-1}}$ is minimal; the shortest path can thus be reduced to $\pi' = (v_i, v_{j_{\ell-1}}, v_k)$. But then, the operation performed in line 5 ensures that $\omega_{ik} \leq \omega_{ij_{\ell-1}} + \omega_{j_{\ell-1}k} = \omega_{\pi'} = \omega_\pi$, contradicting our assumption that $\omega_\pi < \omega_{ik}$.

This reasoning extends to c_{ki} by symmetry, proving the theorem. □

Next, we derive P^{3}C's time complexity. We first prove a partial result.

Lemma 4.2. *Lines 5 and 6 of P^{3}C can be implemented to run in constant time.*

Proof. If the algorithm maintains edge weights in a matrix, this result follows immediately. For an adjacency list, naive implementation would result in a $\mathcal{O}(\log n)$ penalty, but we can get around this. Assume that we have access to a monotone adjacency function $MAdj : V \to 2^E$, which can be (pre)computed in $\mathcal{O}(m_c)$ time. This function maps each vertex to the set of edges connecting it to lower-numbered neighbours.

Now, for each k and j iterated over, the algorithm initialises two lookup tables with edges from $MAdj(k)$ and $MAdj(j)$, respectively, in $\mathcal{O}(w_d)$ time. The edge

weights $\omega_{ki}, \omega_{ik}, \omega_{kj}$, and ω_{jk} can then be accessed in constant time through the first table, whereas ω_{ij} and ω_{ji} are available in the second. □

We now extend this to a worst-case bound on the full algorithm.

Theorem 4.3. *P³C runs in $\mathcal{O}(m_c w_d) \subseteq \mathcal{O}(n w_d^2)$ time.*

Proof. The algorithm starts off by calling DPC, which also honours the $\mathcal{O}(m_c w_d)$ bound (see page 49). Each of the m_c edges in the chordal graph appears exactly once as a pair $\{v_j, v_k\}$. For each such pair we fill the lookup tables mentioned in the proof of Lemma 4.2, using $\mathcal{O}(m_c w_d)$ time in total. Recall from Definition 3.9 that for a chordal graph with simplicial elimination ordering d, the induced width is defined as $w_d = \max_k \{\{v_i, v_k\} \in E \mid i < k\}$ which means that each pair $\{v_j, v_k\}$ can be extended to at most w_d triples $\{v_i, v_j, v_k\}$. Therefore, lines 5 and 6 are executed at most $m_c w_d$ times in total. Since they require constant time by Lemma 4.2, and $m_c \in n w_d$ by Definition 3.9, the claimed bounds follow. □

Recall from the previous chapter that chordal graphs can be recognised in $\mathcal{O}(m)$ time, yielding an elimination ordering d with $w_d = w^*$. Since these do not need to be triangulated, we also have $m_c = m$, so P³C's time complexity becomes $\mathcal{O}(mw^*)$ for this class. We also stated in Proposition 3.6 that for a given constant κ, it can be determined in $\mathcal{O}(n)$ time whether a graph has treewidth $w^* \leq \kappa$. If so, a vertex ordering d with $w_d = w^*$ can also be obtained in linear time. The factor w_d^2 in P³C's bound is then also constant, which means that the worst-case time complexity of our algorithm is $\mathcal{O}(n)$ time. This also follows from the algorithm's pseudocode: every vertex v_k then has a constant number (at most w^*) of neighbours v_j with $j < k$. Note that, because the input is of size $\Omega(n)$, this bound is optimal. We thus have the following.

Corollary 4.4. *On chordal graphs, P³C runs in $\mathcal{O}(mw^*)$ time. On graphs of constant treewidth, P³C runs in $\mathcal{O}(n)$ time, which is optimal.*

Having introduced our new algorithm and demonstrated its theoretical complexity, we are ready to evaluate its practical performance in the next section.

4.3 Empirical evaluation

The theoretical upper bounds on the algorithms we investigate give an indication of their performance on large instances, and sometimes also of their dependency on the induced graph width w_d. However, they cannot give conclusive arguments for their relative performance on real problem instances. The relative performance in practice could give a significantly different picture, for example due to the unknown constant factors in these bounds. Moreover, the bounds are mostly only *upper* bounds. Consequently, they may describe the run time exactly for one algorithm, whereas for others the practical performance over a specific set of instances (e.g. where the underlying graph has a specific structure) may be much better. For this reason, we perform an empirical evaluation.

Table 4.1: Discussed algorithms and bounds on their space and time complexities

		complexity	
algorithm	type	space	time
△STP	PPC	$\mathcal{O}(nw_d^2)$	$\mathcal{O}(n^2w_d^2)^*$
Prop-STP	PPC	$\Theta(m_c)$	$\mathcal{O}(Kw_d^3) \subseteq \mathcal{O}(nw_d^3)$
P^{3}C (list)	PPC	$\Theta(m_c)$	$\mathcal{O}(m_cw_d) \subseteq \mathcal{O}(nw_d^2)$
P^{3}C (matrix)	PPC	$\Theta(n^2)$	$\mathcal{O}(m_cw_d) \subseteq \mathcal{O}(nw_d^2)$
BF–MPSP	MPSP	$\Theta(m)$	$\mathcal{O}(n^2m)$
DLU	MPSP	$\Theta(n^2)$	$\mathcal{O}(n^3)$
Johnson–sparse	MPSP	$\Theta(m)$	$\mathcal{O}(nm + n^2 \log n)$
Johnson (Fibonacci heap)	FPC	$\Theta(n^2)^\dagger$	$\mathcal{O}(nm + n^2 \log n)$
Johnson (binary heap)	FPC	$\Theta(n^2)^\dagger$	$\mathcal{O}(nm \log n)$
Floyd–Warshall	FPC	$\Theta(n^2)$	$\Theta(n^3)$

* At least (see Section 4.1)
† Although its output is of size $\Theta(n^2)$, Johnson can be implemented to require only $\Theta(m)$ *working* space. In our implementation, however, we choose to take advantage of the extra efficiency provided by using an adjacency matrix.

In this section, we evaluate the performance of the algorithms we presented on several benchmarks. We first summarise the algorithms and heuristics we consider and discuss our experimental platform. Next, we formulate predictions based on the theoretical discussion above. We then introduce the benchmarks we used to test whether (and where) our measurements contradict them. Finally, we present the results of our experiments to see whether our predictions hold up in practice.

What to compare?

The properties of the algorithms we compare are summarised in Table 4.1. The algorithms can be divided into three classes. Apart from the algorithms computing PPC, we include the weaker class of algorithms computing MPSP and the stronger class computing FPC (or APSP). We also ran P^{3}C on a (faster) adjacency matrix data structure to afford it an equal footing with the other algorithms using this data structure. When empirically comparing the algorithms, we always make a fair comparison with respect to space usage; moreover, in each plot we always include both variants to allow for a better comparison.

All three PPC algorithms, as well as DLU, require a vertex ordering; the PPC algorithms moreover require the input STN to be triangulated. We used the vertex ordering heuristics discussed on pages 44–46 in Chapter 3. Given a problem instance from one of the benchmark sets described below, our procedure is then as follows.

1. If a vertex ordering is required, i.e. if we use DLU or a PPC algorithm:
   a) Determine a vertex ordering using one of the heuristics;
   b) Given the STN and the ordering, triangulate the network.

   When we use a dynamic heuristic, i.e. DMD or DMF, these steps are taken together.
2. Next, run the algorithm on the (triangulated) STN.

The run times described in the remainder of this section comprise exactly these steps; thus, parsing the instance, reporting results, and other bookkeeping tasks of the experimental environment are excluded from the measurements. For all algorithms, we report the total time taken to complete both steps. All measurements are averaged over 10 runs. When a random number generator was used during the computation (e.g. for the random ordering heuristic), we used the same seed for different runs on the same instance, so as to ensure comparability of the results. The experiments were run using Java 1.6 in server mode, on Intel Xeon E5430 CPUs. Each process had a maximum of 10 minutes to complete its 10 runs, and was allotted 2 GiB of memory.

Predictions and benchmarks

Our experiments are partially of an exploratory nature, investigating how the algorithms perform in practice. However, the theoretical properties also lead us to formulate some predictions of their relative efficiency. In this section, we first state six predictions. Then, we discuss the benchmarks that we use to test whether (and where) our empirical results contradict them.

Predictions

The main new algorithm presented in this chapter is $\mathsf{P^3C}$, which requires a chordal graph as input. In general, the input graphs thus first need to be triangulated. The following two predictions state our expectations regarding the available triangulation methods and ordering heuristics. Recall from Chapter 3 that naive triangulation requires at least $\mathcal{O}(nw_d^2)$ time, whereas TY-triangulation runs in $\mathcal{O}(m_c)$ time. We therefore expect a significant improvement by using the TY method. Note that both use the same amount of memory and produce identical output.

Prediction 4.1. TY-triangulation significantly improves the run-time efficiency over naive triangulation.

Approximate minimum degree (AMD) is the most sophisticated of the ordering heuristics we discussed, and moreover it can be combined with TY-triangulation. This leads us to expect that AMD offers the best trade-off between computational efficiency and triangulation quality for the algorithms requiring a vertex ordering, i.e. the PPC algorithms and DLU .

Prediction 4.2. The AMD heuristic gives the best overall run-time performance for the algorithms that require a vertex ordering.

However, our first important prediction regarding the performance of our new algorithm itself is for graphs that are already triangulated where we expect P^{3}C to outperform all competitors. This is because it has the best theoretical worst-case time bound on this benchmark: it runs in $\mathcal{O}(mw^*) \subseteq \mathcal{O}(nm)$ time.

Prediction 4.3. P^{3}C is the most efficient algorithm on chordal graphs.

P^{3}C can also be used to determine PPC for general graphs, but these then need to be triangulated first. The following predictions now state the expected performance of P^{3}C, our new algorithm for partial path consistency, on general graphs. The upper bound on P^{3}C of $\mathcal{O}(m_c w_d) \subseteq \mathcal{O}(nw_d^2)$ is clearly better than the bound on $\triangle$STP (which is at least $\mathcal{O}(n^2 w_d^2)$ in the worst case) and the bound of $\mathcal{O}(nw_d^3)$ on Prop-STP. Thus, our next prediction naturally follows.

Prediction 4.4. P^{3}C is the most efficient PPC algorithm.

In the introduction to this chapter, we mentioned that all constraints in a PPC network are labelled by their minimal weights, and that any solution to a PPC network can be constructed backtrack-free. MPSP is equivalent to PPC if we triangulate the graph and take all edges in the chordal graph as query pairs. However, if we don't triangulate and just take the edges in the original STN as query pairs, all minimal weights of the original constraints can still be found by computing MPSP, and this is the approach we took in our experiments. In Section 4.1, we introduced DLU and the approaches dubbed BF–MPSP and Johnson–sparse for MPSP, and we presented Floyd–Warshall and Johnson's algorithm for FPC.

Our next prediction concerns the performance of P^{3}C with respect to these approaches. Comparing P^{3}C's run-time bound with those for the MPSP and FPC algorithms, we predict that if the induced width is small enough in a relative sense, it is more efficient to triangulate the graph and then run P^{3}C, both within $\mathcal{O}(m_c w_d) \subseteq \mathcal{O}(nw_d^2)$ time.

Prediction 4.5. If $w_d \in \mathcal{O}(\sqrt{n})$, P^{3}C is more efficient than the algorithms for MPSP and FPC.

Note that the induced width w_d does not appear as a parameter in the time complexity of the algorithms for MPSP and FPC, but recall that $w_d < n$. For $w_d \in \Omega(n)$, P^{3}C's time complexity is $\mathcal{O}(n^3)$, but if w_d is constant, this becomes $\mathcal{O}(n)$. This suggests our final prediction.

Prediction 4.6. The relative performance of P^{3}C with respect to the algorithms for MPSP and FPC decreases as the relative induced width w_d/n grows.

Having stated our predictions, we now present the benchmark sets that we used to put them to the test.

Table 4.2: Properties of the benchmark sets

Type	#cases	n	m	δ	w_d
Chordal					
– variable	280	56–1,024	1,525–49,925	55–1,011	50
– fixed	300	250	1,964–31,115	187–249	8–245
Diamonds	520	51–5,251	50–5,251	4	2
HTN	121	500–625	748–1,599	8–13	2–128
Job shop	370	17–993	32–62,496	16–992	3–249
Scale-free					
– variable	160	250–1,000	2,176–3,330	61–125	147–200
– fixed	350	500	996–29,988	36–318	41–405

Benchmark sets

We evaluated the algorithms across 2101 problem instances from five types of benchmarks, three of which were taken from the SMT-LIB (Ranise and Tinelli, 2003). We describe them in this section; their properties are also listed in Table 4.2. This table lists the number of instances for each benchmark, along with the range of the number of vertices n and edges m and the range of the graph degree δ and the induced width w_d. Of course, w_d depends by definition on the vertex ordering d. The numbers listed in the table are for the AMD heuristic, except for the chordal and HTN benchmarks. For the former, we report the exact treewidth w^*, found with MCS, whereas for the latter we determine a vertex ordering from structural metadata (see below).

diamonds This benchmark is based on problem instances in difference logic proposed by Strichman et al. (2002) and appears in the SMT-LIB (Ranise and Tinelli, 2003), where the constraint graph for each instance takes the form of a circular chain of "diamonds"; see Figure 4.3. Each such diamond consists of two parallel paths of equal length starting from a single vertex and ending in another single vertex. From the latter vertex, two paths start again, to converge on a third vertex. This pattern is repeated for each diamond in the chain; the final vertex is then connected to the very first one. Within a network, all diamonds have the same size; however, the total number of diamonds and their size are varied between benchmarks.

Problems in this class are actually instances of the NP-complete Disjunctive Temporal Problem (DTP): constraints take the form of a disjunction of inequalities. From each DTP instance, we obtain an STP instance by randomly selecting one inequality from each such disjunction. This STP is most probably inconsistent, so its constraint graph contains a negative cycle; we remedy this by modifying the weights on the constraint edges. The idea behind this procedure is that the *structure* of the graph still conforms to the type of networks that one would encounter when solving the corresponding DTP instance, and that the run time of the algorithms mostly depends

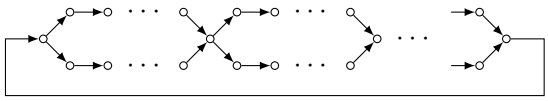


Figure 4.3: The general shape of diamonds instances

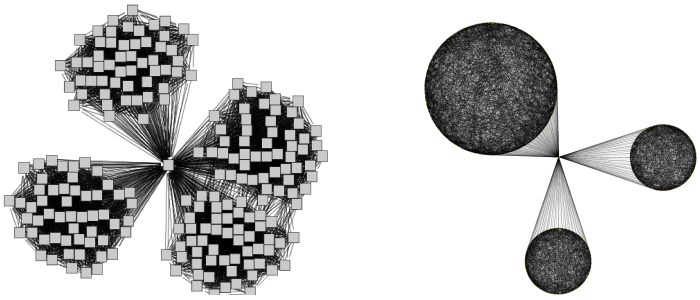


Figure 4.4: The general shape of job shop instances (courtesy of Lars Tijhuis and Nick de Jong)

on this structure. Moreover, to reduce the influence of the randomized extraction procedure, we repeat it for ten different seeds.

For our benchmark set, we considered problem instances which had the size of the diamonds fixed at 5 and their number varying. The most interesting property of this set is that the STNs generated from it are very sparse. We ran experiments on 520 networks, ranging in size from 51 to 5251 vertices, all with a degree of 4.

All instances in the diamonds benchmark have a treewidth of exactly 2, and an ordering with this width is always found. From the theoretical bound on $\mathsf{P}^3\mathsf{C}$, it follows that it must thus be linear in the number of vertices on this benchmark. Since the condition in Prediction 4.5 is certainly satisfied, $\mathsf{P}^3\mathsf{C}$ must be faster than the MPSP and FPC algorithms. Moreover, because the relative induced width decreases as the instances grow, Prediction 4.6 can also be put to the test.

job shop Each of the 370 graphs in the "job shop" set is generated from an instance randomly drawn from job-shop problems of the type available in the SMT-LIB (Ranise and Tinelli, 2003), but of a larger range. To obtain an STN from a job-shop instances, we used the extraction procedure described above for the diamonds benchmark.

With the job-shop benchmark, we expect to find results at the other end of the spectrum. As illustrated by Figure 4.4, each instance consists of a few large and strongly-connected subgraphs, mutually reachable only through a central vertex. Consequently, the induced width of these networks is asymptotically maximal: it is proportional to $n/4$. Prediction 4.6 now stipulates that the performance of the PPC algorithms be similar to that of the FPC algorithms.

Operational Prediction 4.6a. On the job-shop benchmark, the run times of P^3C and the FPC algorithms are within a constant factor of each other.

scale-free Scale-free graphs can be used to model real-world networks in which many vertices have a relatively small degree (number of neighbours) and a few vertices have a 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. Transport networks are another example of the occurrence of a small number of hubs and a much larger number of weakly-connected nodes. Formally, the degrees of the vertices in a scale-free graph satisfy a *power law*; that is, for some constant exponent γ, the probability $P(\delta)$ of a vertex having degree δ is proportional to $\delta^{-\gamma}$.

Albert and Barabási (2002) proposed a method to randomly generate scale-free graphs. Their 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 a constant number of $k \leq n_0$ 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. The evolution thus follows the "rich get richer" scheme: vertices that have a high degree are likely to get even more neighbours, whereas those with a low degree have a high probability to remain "poor".*

We used this method to generate 510 scale-free STNs in two sets. In the "fixed" set, the number of vertices is kept constant at $n = 500$ while varying k from 2 to 98. The "variable" set has n ranging from 250 to 1000 while we keep the range of the induced width limited, selecting only those instances where $w_d \in [147, 200]$. For these instances, k varied between 3 and 15. Note that especially the first set may have an overly large range for k, which is supposed to be constant. However, the variation of the relative induced width in both sets of STNs allows us to test Prediction 4.6.

chordal We argued above that chordal graphs are an important special case on which PPC algorithms are expected to do well — after all, no triangulation is required and a perfect elimination ordering can be found in linear time, after which solving can start immediately. Therefore, we generated a set of graphs of this type.

The generator was written by ourselves and randomly builds graphs with a desired number of vertices n and treewidth $w^* < n$. This is done by starting with a clique of size w^*, after which vertices are added one by one and each is connected to w^* pairwise adjacent vertices already in the graph, thus forming a new clique of size $w^* + 1$. In fact, the order in

* There are some problems with this method, such as the question of how to choose a seed graph. For this reason, Bollobás and Riordan (2004) propose a slightly different generator that starts from an empty graph; they provide a rigorous proof of its correctness. We performed preliminary experiments with both methods and indeed found that the latter approach produces graphs where the distribution of vertex degrees more closely follows a power law. The scale-free graphs used for the experiments in this dissertation were run on networks generated using the approach by Albert and Barabási. In future work, we plan to repeat them for networks obtained through the improved generator.

which vertices are added is a *simplicial construction ordering*: the reverse of the perfect (simplicial) elimination ordering mentioned above.

We generated two benchmark sets as we did for the scale-free instances. In one set, we kept the number of vertices constant at 250 while varying the treewidth from 8 to 245. In the other, the treewidth was kept constant at 50 while the number of vertices ranged from 56 to 1024. The range of the ratio of the induced width to the number of vertices for this benchmark set again allows us to test Prediction 4.6. We also use this benchmark to test Prediction 4.3, which concerns chordal graphs specifically.

HTN Finally, we consider a benchmark set whose instances imitate so-called sibling-restricted STNs originating from Hierarchical Task Networks. This set is therefore particularly interesting from a planning point of view. In these graphs, constraints may occur only between parent tasks and their children, and between sibling tasks (Bui and Yorke-Smith, 2010). We consider an extension that includes *landmark variables* (Castillo et al., 2002) that mimic synchronisation between tasks in different parts of the network, and thereby cause some deviation from the tree-like HTN structure.

We generate HTNs using the following parameters:

1. the number of tasks in the initial HTN tree (fixed at 250; note that tasks have a start and end point);
2. the branching factor, determining the number of children for each task (between 4 and 6);
3. the depth of the HTN tree (between 3 and 7);
4. the ratio of landmark time points to the number of tasks in the HTN; varying from 0 to 0.5 with a step size of 0.05; and
5. the probability of constraints between siblings, varying from 0 to 0.5 with a step size of 0.05.

These settings result in graphs of between 500 and 625 vertices. In contrast to the other benchmarks requiring triangulation, we have some structural information about these networks that suggests an elimination ordering of the vertices. The vertices representing the tasks can be eliminated in a leaf-to-root ordering of the HTN tree. The landmarks, if any, are left till last; thus they are potentially a member of every clique in the resulting chordal graph. This results in the following bound for the induced width: $w_d \leq 2 \times \textit{branching factor} + \#\textit{landmarks}$. Filling in the maximal values of 6 and 125 respectively, we obtain an upper bound of 137 for w_d. Using the elimination ordering outlined above, this bound is easily met, giving $w_d \in [2, 128]$. In fact, none of the ordering heuristics can match the triangulation quality offered by this approach.

Since the relative induced width of these instances depends on the landmark ratio, we have yet another opportunity to test Prediction 4.6.

Next, we present the results of our empirical evaluation on these benchmarks and use them to find out to what extent they correspond to our predictions.

Table 4.3: Normalised run-time results for four variants of an MPSP algorithm based on Bellman–Ford. The smallest numbers are in boldface.

variant		chordal fixed	chordal var	HTN	scale-free fixed	scale-free var	job-shop	dia-monds
1	min	0.696	0.650	0.704	0.703	0.699	0.606	0.569
	mean	0.873	0.882	0.927	0.831	0.987	0.794	**0.864**
	max	1.109	1.134	1.050	1.185	1.178	**1.075**	**1.370**
1-pc	min	**0.537**	**0.568**	**0.612**	**0.499**	**0.556**	**0.447**	**0.552**
	mean	**0.674**	**0.658**	**0.793**	**0.606**	**0.791**	**0.614**	0.941
	max	**0.899**	**0.819**	**0.909**	**1.064**	**1.026**	1.125	1.460
2	min	1.008	1.059	0.866	0.853	0.861	0.931	0.688
	mean	1.229	1.258	1.004	1.303	1.040	1.444	1.092
	max	1.506	1.717	1.258	1.787	1.494	2.245	1.755
2-pc	min	1.096	1.157	1.126	0.893	1.018	0.767	0.791
	mean	1.381	1.369	1.356	1.523	1.232	1.422	1.126
	max	1.901	1.592	1.559	1.950	1.525	1.985	1.453

Results

The stage is now set for presenting our experimental results proper. We first give the results of a preliminary experiment on the four variants of BF–MPSP, to see if we can reduce the number of algorithms we need to consider in the main experiments. Then, we investigate each of the predictions from the previous section in order.

Variants of BF–MPSP

Table 4.3 includes normalised results of the four variants of the MPSP algorithm based on Bellman–Ford, presented in Section 16. For each problem instance, the time required by each variant is divided by the mean time required by all variants, thus obtaining a relative measure for the variant's efficiency. The table reports the minimum, mean and maximum of these ratios for each benchmark. Since we use the geometric mean, the product of the means equals 1 in every column.

The first conclusion we draw from this table is that on our benchmarks, the four variants are very close in terms of their run time: the fastest and slowest run times are at most roughly a factor 2 away from the mean in both directions. On almost all benchmarks, the one-way variant with the parent checking heuristic (1-pc) gives the best results. The only exception is the diamonds benchmark, where it gives better performance to switch off the heuristic. In the results we present below, we therefore exclude the two-way variants (those that compute a vertex cover), and use the heuristic everywhere but for the diamonds benchmark.

Table 4.4: The run times of naive versus TY-triangulation, for various benchmarks and the AMD and SMD heuristics. We also report data on the range of their relative performance. The "#cases" column reports how many instances required more than 1 ms for both triangulation methods, since only these were taken into account.

			T_{naive} / T_{TY}			average (ms)	
		#cases	min	mean	max	naive	TY
Diamonds	AMD	520	0.52	0.97	1.56	21.0	21.7
	SMD	444	0.47	0.89	2.00	5.8	6.8
Job shop	AMD	370	0.62	1.24	2.98	53.3	31.8
	SMD	320	0.67	1.82	9.09	52.0	14.2
Scale-free (variable)	AMD	160	0.95	1.45	2.09	56.5	38.9
	SMD	160	1.00	2.16	3.15	38.4	17.7
Scale-free (fixed)	AMD	350	0.70	2.71	5.00	175	59.3
	SMD	350	0.75	5.59	9.91	150	23.0

TY-triangulation versus naive triangulation (Prediction 4.1)

In our experiments, we observed that triangulation sometimes contributes significantly to the overall run time of DLU and the PPC algorithms. For example, for P^{3}C the time for triangulation makes up between 1.3% and as much as 80% of the total. Although these figures depend on the combination of benchmark and triangulation, it must be noted that for every ordering heuristic percentages over 45% were found; the same is true for each triangulation method and benchmark. It is therefore worthwhile to devote some attention to the efficiency of the triangulation procedure.

Table 4.4 exhibits our findings on the four benchmarks that require ordering heuristics. We include results for AMD and SMD, both of which are static heuristics. The table includes the minimum, mean, and maximum relative run time of naive triangulation with respect to TY-triangulation, as well as the absolute averages. For both heuristics and on all benchmarks but diamonds, TY-triangulation gives a lower average run time than naive triangulation; the relative results show the same. Pictorial results for the relative performance are included in Figure 4.5. We plot the relative time against the induced width and against the number of fill edges. When either of these passes a threshold, the relative performance of naive triangulation deteriorates. In the left-hand plot, we see that the the penalty eventually becomes proportional to the induced width w_d. This agrees with the theoretical bounds, since TY-triangulation runs in $\mathcal{O}(m_c)$ time whereas naive triangulation requires at least $\mathcal{O}(m_c w_d)$ time.

We conclude that the TY approach is more efficient than the naive method if the triangulation is demanding enough. For the diamonds benchmark, there is simply not enough work involved in triangulation for the improvement to become apparent. The remaining question is whether the difference is significant. Our answer is affirmative: we observed an (eventual) order-of-magnitude separation in our experiments and found empirical evidence to support the better theoretical

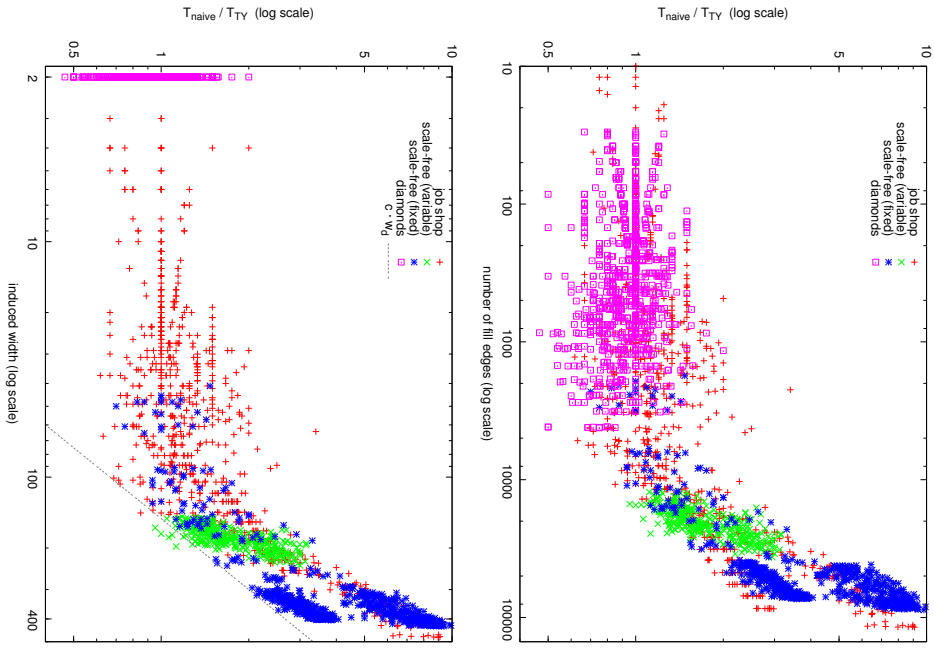


Figure 4.5: As induced width increases or more fill edges are added, naive triangulation becomes significantly more costly than TY-triangulation.

complexity of TY-triangulation. We therefore confirm Prediction 4.1.

AMD versus other ordering heuristics (Prediction 4.2)

Prediction 4.2 states that using the AMD heuristic yields the best overall run-time for DLU and the PPC algorithms. To investigate this prediction, we include Table 4.5, listing the number N of instances solved within the time-out of 1 minute and the average run time for each combination of benchmark, heuristic, and algorithm. If we were to base our verdict solely on N, we would reject the prediction outright for Prop-STP and $\triangle$STP; other heuristics allow these algorithms to solve more instances within a minute. Looking at the average times, we still reject the prediction for $\triangle$STP. Although it is hard to identify a winning heuristic over all benchmarks, DMF comes close and solves the most instances in sum. For Prop-STP, AMD is the best-performing heuristic on the scale-free instances, yet the poorest choice on the job-shop benchmark; identifying an overall winner is again difficult. DLU's results show SMD as a strong contender for the spot of overall winner, although AMD is not far away on the scale-free instances and is the fastest heuristic on the job-shop instances. So far, we must reject the prediction, but the situation changes when we look at P^3C. Ignoring the diamonds benchmark for the moment, AMD is in fact the best heuristic overall. Although for the adjacency-list implementation, on the job-shop instances, AMD is outperformed

Table 4.5: Average run times (in milliseconds) and numbers of instances N solved by DLU and the PPC algorithms for various ordering heuristics. Bold numbers for N are maximal for their category; bold average times are minimal, provided that N is maximal. Large numbers have been rounded to 3 significant digits.

		P^{3}C (matrix)		DLU		P^{3}C (list)		Prop-STP		△STP		
		average	N	average	N	average	N	average	N	average	N	
diamonds	AMD	37	**520**	2,200	**510**	43	**520**	46	**520**	40	**520**	AMD
	SMD	**18**	**520**	1,620	**510**	25	**520**	29	**520**	23	**520**	SMD
	DMD	19	**520**	1,780	**510**	**16**	**520**	**22**	**520**	**14**	**520**	DMD
	SMF	30	**520**	**1,610**	**510**	26	**520**	29	**520**	23	**520**	SMF
	DMF	36	**520**	1,670	**510**	32	**520**	37	**520**	30	**520**	DMF
	MCS	30	**520**	2,930	500	25	**520**	29	**520**	24	**520**	MCS
	RND	36	**520**	1,740	**510**	33	**520**	36	**520**	32	**520**	RND
jobshop	AMD	**156**	**370**	**217**	**370**	267	**370**	2,460	350	1,580	336	AMD
	SMD	186	**370**	280	**370**	332	**370**	**1,240**	**370**	1,920	325	SMD
	DMD	183	**370**	238	**370**	500	**370**	3,460	360	2,170	339	DMD
	SMF	242	**370**	344	**370**	358	**370**	1,270	**370**	1,910	325	SMF
	DMF	390	**370**	448	**370**	460	**370**	3,500	360	1,920	339	DMF
	MCS	181	**370**	237	**370**	**265**	**370**	3,590	360	**1,730**	**340**	MCS
	RND	514	**370**	569	**370**	849	**370**	1,590	**370**	1,030	277	RND
scale-free (variable)	AMD	**153**	**160**	446	**160**	**284**	**160**	**832**	**160**	3,830	**160**	AMD
	SMD	157	**160**	**417**	**160**	297	**160**	1,850	**160**	4,840	**160**	SMD
	DMD	170	**160**	444	**160**	399	**160**	875	**160**	3,810	**160**	DMD
	SMF	226	**160**	483	**160**	297	**160**	1,940	**160**	4,960	**160**	SMF
	DMF	323	**160**	583	**160**	374	**160**	975	**160**	**3,390**	**160**	DMF
	MCS	467	**160**	912	**160**	673	**160**	15,400	**160**	11,500	121	MCS
	RND	1,570	**160**	1,600	**160**	2,480	**160**	19,900	76	9,850	40	RND
scale-free (fixed)	AMD	**584**	**350**	578	**350**	**1,010**	**350**	**10,200**	**350**	5,490	67	AMD
	SMD	617	**350**	**573**	**350**	1,050	**350**	13,500	**350**	5,100	58	SMD
	DMD	812	**350**	819	**350**	2,770	**350**	11,500	**350**	6,870	67	DMD
	SMF	754	**350**	708	**350**	1,100	**350**	14,000	**350**	4,240	54	SMF
	DMF	1,760	**350**	1,750	**350**	2,050	**350**	12,000	**350**	**6,280**	**71**	DMF
	MCS	836	**350**	761	**350**	1,250	**350**	24,000	346	5,960	45	MCS
	RND	1,230	**350**	1,020	**350**	1,960	**350**	34,100	349	8,970	23	RND

by MCS, the difference is small (less than 1%) and MCS is significantly slower in all other cases.

This leaves the diamonds benchmark, where matters are rather different, as was also the case for the previous prediction. We see that here, AMD is in fact consistently the worst heuristic, with the sole exception of MCS used for DLU . For extremely sparse instances like these, DMD seems to be the best choice. Looking at the theoretical bounds of $\mathcal{O}(nm)$ for AMD and $\mathcal{O}(nm_c) \subseteq \mathcal{O}(n^2 w_d)$ for DMD, this is perhaps not surprising. Recall that we have $w_d = 2$ on the diamonds benchmark, which yields $m_c \leq 2m$ and asymptotically equivalent bounds. Thus, the lower constant factor for DMD comes into action and plays a deciding role.

To reach a final overall verdict for P^{3}C, we can average the results for each benchmark. Whether we weight the averages by the number of instances in the benchmark or not, AMD is then the best overall heuristic, with SMD and DMD taking second and third place, for both P^{3}C implementations.*

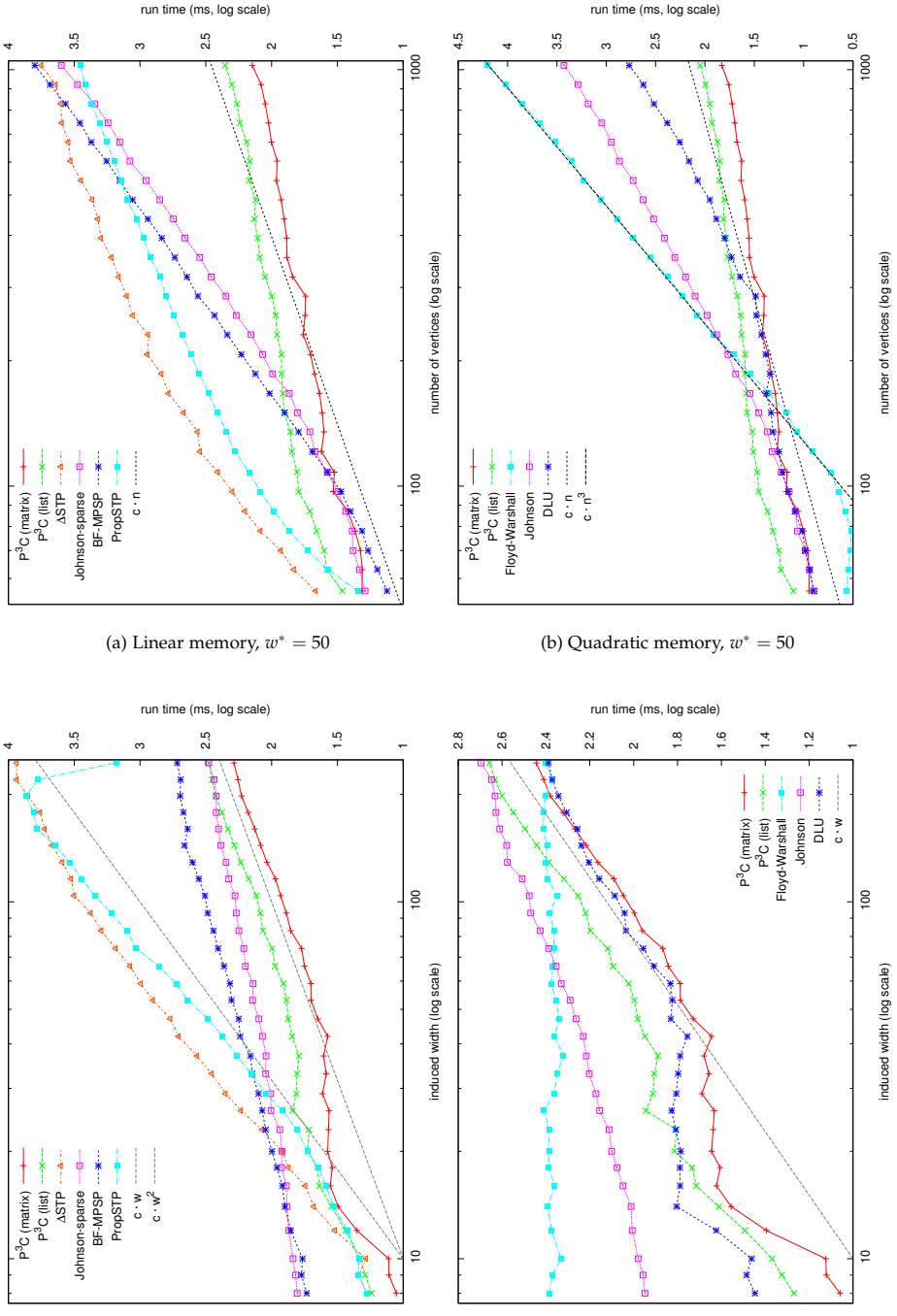
We conclude that the prediction does not hold for DLU, Prop-STP or $\triangle$STP. The prediction does hold for P^{3}C with the provision that we compare the total run time summed over all benchmarks, or we ignore the diamonds benchmark. Below, we investigate how these algorithms compare amongst each other and with others. In these comparisons, we will always use the heuristic with which the algorithm performs best for each benchmark (i.e. we use the bold entries in Table 4.5). Thus, we allow each algorithm to reach its full potential while at the same time considerably reducing the amount of experimental results to compare.

Chordal STNs (Prediction 4.3)

To be able to discuss Prediction 4.3, which states that P^{3}C is the best among all algorithms on chordal STNs, we include Figure 4.6, which shows experimental results on this benchmark; average numbers are also listed in Tables 4.6 and 4.7. The plots and numbers speak for themselves in confirming the prediction. Only on the smallest instances some algorithms can compete with P^{3}C. For example, Floyd–Warshall profits from the low constant factor hidden in its asymptotic $\Theta(n^3)$ bound. However, as instances get larger, its cubic trend sets in soon enough. Meanwhile, the empirical results support P^{3}C's $\mathcal{O}(n)$ time bound when w^* is constant; on the other STNs with a constant number of vertices, they give evidence for a trend proportional to w^*. Given the worst-case complexity of $\mathcal{O}(nw_d^2)$, this better than expected. For the other PPC algorithms, the empirical results point

*Of course, this approach biases large run times; this can be mitigated by computing the geometric mean, which is based on relative performance. The top three is then the same, with AMD taking the third spot in both cases.

A question one may raise is whether the average times we report in Table 4.5 are too biased by large instances in general. For this reason, we also compared geometric means. In half of the algorithm-benchmark combinations, these gave an identical ranking of heuristics, and some others only yielded a slightly different ranking. Where the geometric mean resulted in a different winner, further investigation showed that this was mostly due to noise in smaller instances; as soon as instances became large enough for a clear trend to emerge, this trend was always better for the heuristic with the best arithmetic mean. For these reasons, we only report arithmetic means, i.e. what is commonly understood as the average.



(a) Linear memory, $w^* = 50$

(b) Quadratic memory, $w^* = 50$

(c) Linear memory, $n = 250$

(d) Quadratic memory, $n = 250$

Figure 4.6: Results on the chordal benchmarks

Table 4.6: Average times in milliseconds for DLU and the PPC algorithms using the best available ordering heuristic identified in Table 4.5. The leftmost two algorithms require quadratic space, whereas the others run in linear space. Bold entries are minimal on that benchmark for their space requirements. Large numbers have been rounded to 3 significant digits. For entries marked by an asterisk (*), not all instances were solved; refer to Table 4.5.

	adjacency matrix		adjacency list		
	P^{3}C	DLU	P^{3}C	Prop-STP	△STP
HTN	**59**	183	**80**	120	340
diamonds	**18**	1,610*	16	22	**14**
chordal					
– variable	**61**	139	**92**	522	927
– fixed	**79**	87	**119**	1,580	2,080
job shop	**156**	217	**265**	1,240	1,730*
scale-free					
– variable	**153**	417	**284**	832	3,390
– fixed	584	**573**	**1,010**	10,200	6,280*

to an $\Omega(w^{*2})$ lower bound, which is nevertheless in accordance with their known worst-case complexities.

For the instances with highest treewidth, we see a sudden drop in Prop-STP's run time. At first glance, one might suspect a measurement error, but in fact this behaviour is predicted by theory. We already noted in Section 4.1 that as w^* approaches n, the number of cliques K in Prop-STP's $\mathcal{O}(Kw_d^3)$ time bound approaches 1, which explains the decrease. Setting $K = n - w_d$, the run-time trend actually follows the curve for Kw_d^3 closely.

PPC algorithms (Prediction 4.4)

Referring to the rightmost three columns in Table 4.6, we see that of the PPC algorithms, P^{3}C outperforms its competitors almost across the board, sometimes by over an order of magnitude. Only on the diamonds benchmark, △STP is (narrowly) faster. Also, recall from Table 4.5 that P^{3}C is the only algorithm that managed to solve all instances for all ordering heuristics within the time-out of 1 minute. Let us take a closer look at the diamonds and HTN benchmarks, where the competition is most fierce.

The plots for the diamonds benchmark (Figures 4.7a–b) confirm that △STP is here narrowly but consistently more efficient than P^{3}C. The trendline we plot in Figure 4.7a gives evidence for a linear run time of the PPC algorithms; this evidence is strengthened by Figure 4.7b, where run times are divided by n. This behaviour follows from the theoretical bounds of $\mathcal{O}(nw_d^2)$ and $\mathcal{O}(nw_d^3)$, respectively, on P^{3}C's and Prop-STP's time complexities, and also indicates that the $\mathcal{O}(n^2w_d^2)$ bound on △STP is probably not tight on this benchmark. The exception is P^{3}C run on an adjacency matrix, whose results start to curve up for

Table 4.7: Average times in milliseconds for the algorithms for FPC and MPSP (except DLU). The leftmost three algorithms compute MPSP and run in linear space, whereas the others compute FPC and require quadratic space. Bold entries are minimal on that benchmark for their category. Large numbers have been rounded to 3 significant digits. All instances were solved, except for Floyd–Warshall not solving 9 diamonds benchmarks, marked by an asterisk (*).

	BF–MPSP		Johnson			
	1-pc	1	sparse	Fib.	binary	F-W
HTN	**120**	140	203	209	**199**	1,620
diamonds	214	**195**	761	775	**328**	1,680*
chordal						
– variable	587	891	**389**	**434**	3,350	1,420
– fixed	234	305	**151**	**162**	1,240	168
job shop	465	719	**292**	**323**	2,340	972
scale-free						
– variable	**242**	300	310	**319**	498	3,730
– fixed	627	852	**391**	**414**	2,650	1,170
	MPSP (adj. list)			FPC (adj. matrix)		

the bigger instances. This, too, is to be expected, since the data structure requires $\Theta(n^2)$ time to initialise.

Results for the HTN benchmark are included in Figures 4.7c–d. They show increasing separation between P^{3}C and the other PPC algorithms as the induced width grows, although the latter start off faster than P^{3}C using an adjacency list. In Figure 4.7d we examine the algorithms' run-time behaviour relative to nw_d^2 and find that Prop-STP and $\triangle$STP cannot match P^{3}C in meeting the latter's theoretical worst-case bound, explaining the divergence between their results.

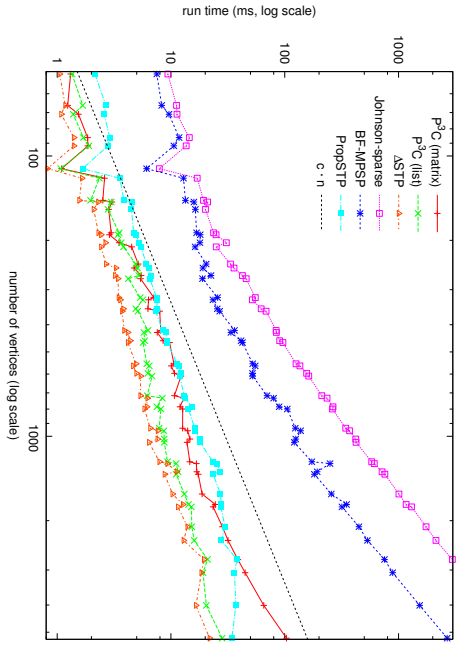
Note further that BF–MPSP and Johnson–sparse, the two MPSP algorithms running within linear memory, also meet P^{3}C's bound on this benchmark comfortably. Next, therefore, we compare P^{3}C against the non-PPC algorithms.

P^{3}C versus algoritms for MPSP and FPC (Predictions 4.5 and 4.6)

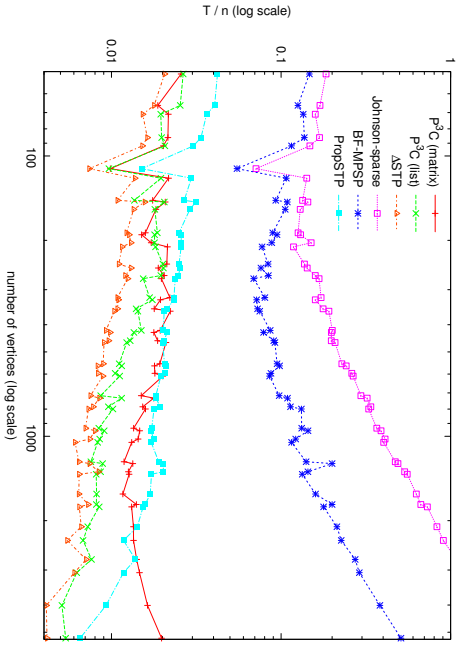
To investigate Prediction 4.5, which states that P^{3}C outperforms the algorithms for MPSP and FPC if $w_d \subseteq \mathcal{O}(\sqrt{n})$, let us first consider the benchmarks where w_d is constant or restricted to a limited range, independent of n. These are the diamonds benchmark and the "variable" chordal and scale-free benchmarks.

For the diamonds benchmark, having $w_d = 2$, Tables 4.6 and 4.7 show that P^{3}C is indeed fastest by between one and two orders of magnitude. We already saw that it is also fastest on the "variable" chordal instances with treewidth $w^* = 50$. DLU's relatively good performance is to be expected: it, too, profits from the fact that a perfect elimination ordering is determined efficiently.

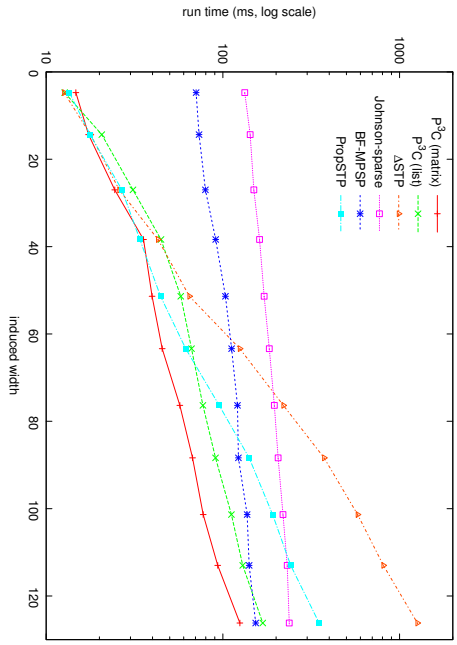
On the "variable" scale-free results, w_d is still higher: between 147 and 200.



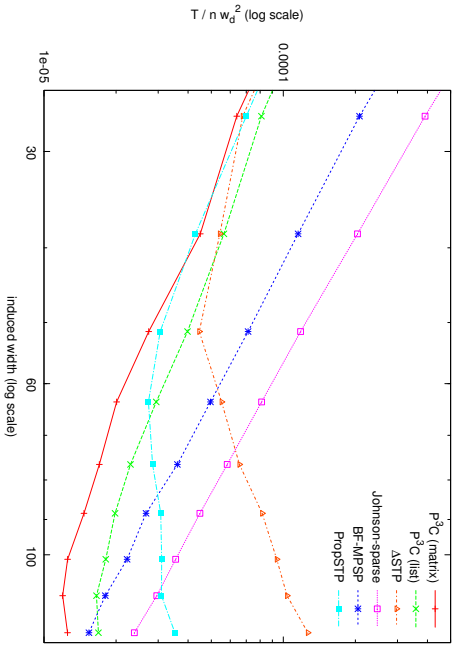
(a) Run times on diamonds



(b) Run times on diamonds, relative to n

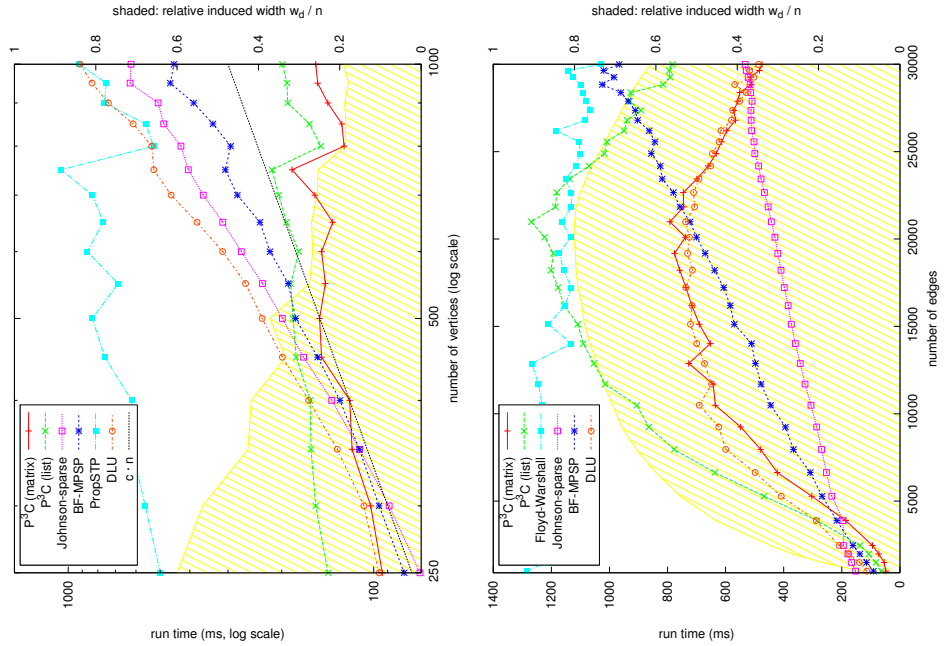


(c) Run times on HTN



(d) Run times on HTN, relative to nw_d^2

Figure 4.7: Results on the diamonds and HTN benchmarks



(a) Variable: $n \in [250, 1000], w_d \in [147, 200]$ (b) Fixed: $n = 500, w_d \in [41, 405]$

Figure 4.8: Results on the scale-free benchmarks

We see that this time, BF–MPSP exhibits better average performance than P^{3}C. But as Figure 4.8a shows, there is more to this story, and BF–MPSP only gains the upper hand through better performance on the instances where w_d is high with respect to the number of vertices. In fact, the empirical results give evidence for a $\mathcal{O}(n)$-time performance of both P^{3}C implementations, whereas the others appear to perform superlinearly.

We can confirm the prediction based on these three benchmarks, since the others are outside its scope. HTN and the "fixed" variants of the chordal and scale-free benchmarks have the roles of n and w_d reversed: the former is now constant or restricted to a limited range, whereas w_d ranges across a much wider interval. The last remaining benchmark, job shop, has $w_d \approx n/4$ which is likewise outside $\mathcal{O}(\sqrt{n})$. An interesting topic for future research could be to conduct experiments on instances that have $w_d \in \Theta(\sqrt{n})$. A candidate benchmark would be the class of STNs on a k-by-k grid, which have $n = k^2$ and $w^* = k$.

This leaves Prediction 4.6 and, as a special case, Operational Prediction 4.6a. The former, more general prediction states that the relative performance of P^{3}C relative to the algorithms for MPSP and FPC deteriorates as the relative induced width w_d/n increases. In other words, it claims that we can use a monotonically decreasing function to describe the relation between $x = w_d/n$ and $y = T_A/T_{\mathsf{P^3C}}$, where A is an algorithm for MPSP or FPC.

Table 4.8: The rank correlation between w_d/n and run time relative to P^{3}C, according to Kendall's τ.

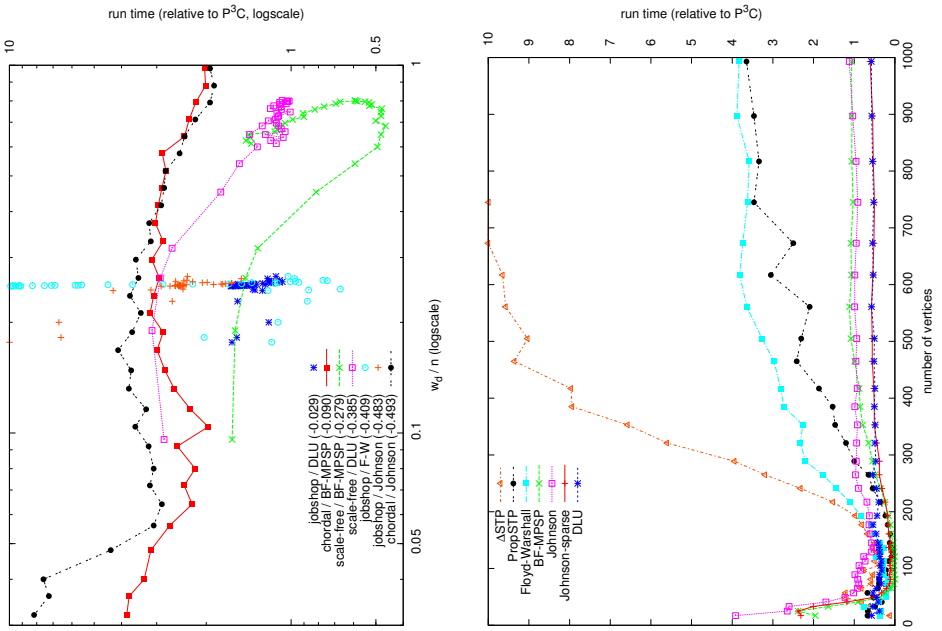
	DLU	BF–MPSP	Johnson		Floyd–Warshall
			MPSP	FPC	
job shop	-0.029	-0.573	-0.519	-0.483	-0.409
chordal (fixed)	-0.569	-0.090	-0.631	-0.493	-0.858
scale-free (fixed)	-0.385	-0.279	-0.534	-0.544	-0.734
HTN	-0.600	-0.781	-0.839	-0.834	-0.771
diamonds	-0.885	-0.687	-0.876	-0.743	-0.866
chordal (variable)	-0.673	-0.932	-0.887	-0.905	-0.908
scale-free (variable)	-0.826	-0.839	-0.861	-0.849	-0.862

The extent to which this true can be found by computing Kendall's tau coefficient for rank correlation (Kendall, 1938). Given two random variables X and Y and a set of observations $O = \{(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\}$ of joint realisations of X and Y, Kendall's method considers all pairs of observations $\{(x_i, y_i), (x_j, y_j)\} \subseteq O$. Each pair is then classified as concordant, discordant, or a tie. In a *concordant* pair, the ranks for x and y agree: either $x_i < x_j$ and $y_i < y_j$ or $x_i > x_j$ and $y_i > y_j$. If the ranks disagree, e.g. $x_i < x_j$ and $y_i > y_j$, the pair is *discordant*; if $x_i = x_j$ or $y_i = y_j$ the pair is a *tie*. Letting C and D respectively denote the numbers of concordant and discordant pairs, Kendall then defines $\tau = (C - D)/(C + D)$.* We see that if all pairs are concordant (so the function is strictly increasing) we get $\tau = 1$, whereas a value of $\tau = -1$ corresponds to a strictly decreasing function.

We computed Kendall's tau coefficient for the FPC and MPSP algorithms on all benchmarks; results are included in Table 4.8. First, note that all correlations are negative, which is a first confirmation of our prediction. However, in a few cases, the value of τ is very close to zero, indicating that the dependence is quite weak. Overall, the job-shop instances give the weakest correlation, which is not surprising given the fact that $w_d/n \approx 1/4$: there is little room to establish a trend. This is why we further investigate this benchmark below when discussing Prediction 4.6a. To interpret the values of τ, note that relatively small absolute values already indicate a significant correlation: for $\tau \leq -0.334$ we have $D \geq 2C$ and for $\tau \leq -0.500$ we get $D \geq 3C$.

We plot the weakest correlations in Figure 4.9a to take a closer look at them. Three of these are on the job-shop benchmark and do not seem to exhibit any correlation at all in this plot; however, on a larger scale, the data for Floyd–Warshall and Johnson does show a decreasing trend. Note further that Kendall's procedure is probably stymied by the scale-free data curving back on itself, an artifact of the decreasing induced width for the largest instances which can be seen in Figure 4.8b. Splitting these data sets in two at the rightmost point would yield

*This is if there are no ties, so $C + D = k(k-1)/2$. If there are ties, the denominator is adjusted with a small correction outside the scope of this text.



(a) The weakest correlations from Table 4.8

(b) The job-shop benchmark

Figure 4.9: Run times relative to P^{3}C

two sets with much greater correlation.* The data for BF–MPSP on the chordal benchmark with constant n seems to lack a clear overall trend, confirming the low value of τ. Consulting Figure 4.6c, we in fact discover a very clean trend, which may be confusing at first, until one discovers that the "bumps" in P^{3}C's run time on this benchmark cause the lack of correlation. Finally, the data points for Johnson on the chordal benchmark, which has the highest τ-value among these seven, indeed follows a nice general downward trend.

Finally, we come to Prediction 4.6a, concerning the job-shop instances. In Figure 4.9b, we plot the number of vertices against the run time of the respective algorithms divided by the run time of P^{3}C (with the same memory footprint). As can be seen, the trends for the MPSP and FPC algorithms start to flatten out between $n = 300$ and $n = 600$, thus confirming this prediction.

Other observations

Finally, we include some other observations based on our experimental results.

First, we note that in Tables 4.6 and 4.7, P^{3}C almost always ranks first in its memory-usage category, with only three exceptions. Two of these were already

*We found that this problem does not occur with scale-free networks obtained using the improved generator by Bollobás and Riordan (2004), described in the footnote on page 78. In these graphs, w_d only increases with m.

discussed above: the diamonds benchmark for algorithms using linear space, where △STP narrowly outperforms P^{3}C, and the "variable" scale-free benchmark in the same category, where BF–MPSP is fastest although P^{3}C's trend is best. The third is the "fixed" scale-free benchmark (Figure 4.8b), which has a high relative induced width: $w_d \geq n/2$ for most of the instances. Here, Johnson with a Fibonacci heap performs very well, and all MPSP algorithms outperform P^{3}C; but the latter is still the best PPC algorithm and is also faster than Floyd–Warshall throughout the benchmark, even where $w_d/n > 0.8$.

Next, we come back to the statement in Section 4.1 that Prop-STP sometimes benefits from a coarser triangulation, thus keeping the number of cliques low at the cost of a higher induced with. Evidence for this behaviour can be found in the results on the job-shop benchmark, included in Table 4.5. For all algorithms except Prop-STP, the random ordering yields worst efficiency, and performance with SMF is mediocre; for Prop-STP, both these heuristics are in the top three. Contrariwise, AMD and MCS yield good performance for all algorithms except Prop-STP, where these two rank last. We verified that this behaviour is due to the fact that with RND and SMD, the number of cliques is kept small.

We mentioned in Section 16 that Wang (2003), DLU's originator, identified DMD as the heuristic for which his algorithm performed best. In Table 4.5, we find that DMD is not clearly the best heuristic for DLU; AMD and SMD are also strong contenders. However, Wang's judgement did not account for the time computing the ordering; if we do the same and only consider the time spent executing DLU proper, Wang's results are reproduced by our experiments and we find that the top spots are always taken by the dynamic ordering heuristics DMD and DMF, the latter of which was not considered by Wang. Since these two heuristics make the largest investment in terms of their time complexity to produce a good vertex ordering, this is not surprising. As Wang already noted, this method of comparison may be justified when the same network is used many times for different constraint weights or sets of query pairs. However, it is our opinion that the full picture must be considered in our case, where we are interested in enforcing PPC once on a specific STN.

A final observation is that although for all benchmarks, Johnson–sparse performs just a little faster than Johnson on average, the results are very close. In fact, when ranking all algorithms in Tables 4.6 and 4.7 for each benchmark, Johnson–sparse comes immediately before Johnson with a Fibonacci heap. On average, the sparse variant requires about 6% less time.

4.4 Discussion

We end this chapter by listing our conclusions and by identifying some topics for future research.

The main contribution of this chapter is P^{3}C, an algorithm for partial path consistency (PPC) with a theoretical worst-case time complexity of $\mathcal{O}(m_c w_d) \subseteq \mathcal{O}(n w_d^2)$, the best among the algorithms in its class. For chordal graphs, its complexity is $\mathcal{O}(m w^*)$, and it even drops to an optimal $\mathcal{O}(n)$ for graphs of

constant treewidth $w^* \in \mathcal{O}(1)$. Our experiments on a wide range of benchmarks confirmed P^{3}C's linear run time where applicable and showed that it is the best-performing algorithm for PPC also in practice.

We argued why the problems of computing full path consistency (FPC) and multiple-pairs shortest paths (MPSP) are closely related to PPC and therefore we also compared P^{3}C with algorithms for these problems. DLU is in the latter category and to the best of our knowledge, it has never yet been applied to the STP. MPSP can also be computed by repeatedly running the Bellman–Ford algorithm; we compared four implementation variants and found that it is usually most efficient to simply run this algorithm n times as opposed to first computing a vertex cover.

The worst-case bounds on the run times of MPSP and FPC algorithms indicate that also among this wider set of competitors, P^{3}C has the best theoretical complexity if the induced width w_d is small enough, viz. $w_d \in \mathcal{O}(\sqrt{n})$; moreover, its relative performance depends on the relative induced width w_d/n. These claims, too, were confirmed empirically, although their support could be strengthened by examining the class of STNs on a grid graph, where $w^* = \sqrt{n}$.

Another contribution was our empirical evaluation of seven ordering heuristics as required by the PPC algorithms as well as DLU . These experiments were aimed at determining which of these heuristics offers the best trade-off between quality and computational efficiency in practice. Based on a comparison of time complexities and sophistication, we predicted that AMD would offer the best such trade-off. For P^{3}C, this prediction was supported by the experimental results. For the other algorithms, the results were inconclusive or even resulted in a different overall winner. Since these algorithms themselves are less efficient, it turns out to pay off to make a bigger computational investment into a good ordering. Also, Prop-STP's behaviour showed that that an ordering which works well for one algorithm may in fact be a poor choice for another, and vice versa.

This deviating behaviour by Prop-STP was explained in our theoretical discussion. Here, we also included a proof that $\triangle$STP may in some cases require $\Omega(n^4)$ time, performing strictly worse than Floyd–Warshall, which was confirmed empirically.

When identifying topics for future work, two issues stand out. First, in this chapter we evaluated MPSP algorithms when used to solve a "light" version of PPC, where the set of query pairs coincides exactly with the constraint edges. This solution concept suffices to answer queries for minimal constraint weights. Future research could take the opposite approach and investigate for what number or distribution of query pairs PPC is an efficient replacement for a dedicated MPSP algorithm.

The other issue is to go back to the roots of PPC. Recall from the previous chapter that PPC was originally proposed by Bliek and Sam-Haroud (1999) as an efficient approximation for path consistency on general constraint networks. Although we proved that P^{3}C can be used to enforce PPC on the simple temporal network, it is an open question whether its correctness extends to this wider problem domain.

However, in the next chapter, we stay within the domain of the STN and generalise the ideas presented in this chapter in another direction: to compute full path consistency, or all-pairs shortest paths.

CHAPTER 5

Full Path Consistency*

The previous chapter looked into methods for enforcing partial path consistency (PPC) on the STN. In this chapter we shift our focus to full path consistency (FPC). In Chapter 3, we showed that the PPC and FPC STNs can be considered as normal forms for temporal information. The FPC network embodies an answer to Simple Temporal Query 6 asking for a MINIMAL NETWORK and thus contains the MINIMAL CONSTRAINT (STQ 5) between each pair of time points. Moreover, the FPC network can be DISPATCHed along any ordering of its temporal variables.

Recall from Chapter 2 that establishing FPC on an STN is exactly equivalent to computing all-pairs shortest paths (APSP) on a weighted directed graph, and that for networks with n vertices and m edges, this can be done in $\mathcal{O}(n^3)$ time with the Floyd–Warshall algorithm (Floyd, 1962), based on Warshall's (1962) formulation of efficiently computing the transitive closure of Boolean matrices. However, the state of the art for computing APSP on sparse graphs is an algorithm based on the technique originally proposed by (Johnson, 1977), which does some preprocessing followed by n runs of Dijkstra's 1959 algorithm. Using a Fibonacci heap (Fredman and Tarjan, 1987), the algorithm runs in $\mathcal{O}(n^2 \log n + nm)$ time. In the remainder of this chapter, we simply refer to this algorithm as "Johnson".

In Section 5.1 of this chapter, we present two new algorithms for computing APSP on graphs with real edge weights. One algorithm, dubbed Chleq–APSP, is based on a point-to-point shortest path algorithm by Chleq (1995); the other, named Snowball, is similar to the P^{3}C algorithm presented in the previous chapter. These new algorithms advance the state of the art in computing APSP. In graphs of constant treewidth, such as sibling-restricted STNs based on HTNs with a constant branching factor, the run time of both algorithms is bounded by $\mathcal{O}(n^2)$, which is optimal since the size of the output is $\Theta(n^2)$. Other examples of graphs with constant treewidth are outerplanar graphs and series-parallel graphs, both

*This chapter includes work from a conference paper (Planken et al., 2011) and an extended journal paper (Planken et al., 2012).

of which have treewidth at most two. Moreover, there are many related notions of graph width such as pathwidth, branchwidth, bandwidth, and cutwidth; if any of these properties is bounded by a constant, then so is the treewidth (Bodlaender, 1998).

When Chleq–APSP and Snowball are applied to chordal graphs, they have a run time of $\mathcal{O}(nm)$, which is a strict improvement over the state of the art (Chaudhuri and Zaroliagis, 2000, with a run time of $\mathcal{O}(nmw_d^2)$). Chordal graphs are an important subset of general sparse graphs: interval graphs, trees, k-trees and split graphs are all special cases of chordal graphs (Golumbic, 1980). Moreover, any graph can be made chordal using a so-called triangulation algorithm, as discussed in Chapter 3. Recall that such an algorithm operates by eliminating vertices one by one, connecting the neighbours of each eliminated vertex and thereby inducing cliques in the graph.

The *induced width* w_d of the vertex ordering d was defined in that chapter to be equal to the cardinality of the largest such set of neighbours encountered. The upper bound of the run time of both proposed algorithms on these general graphs, $\mathcal{O}(n^2w_d)$, depends on this induced width. However, recall that finding a vertex ordering of minimum induced width is an NP-hard problem (Arnborg et al., 1987). This minimum induced width is the tree-likeness property of the graph mentioned above, known as the *treewidth* and denoted w^*. The induced width, in contrast, is not a direct measure of the input (graph), so the bound of $\mathcal{O}(n^2w_d)$ is not quite proper. Still, it is better than the bound on Johnson if $w_d \in o(\log n)$.To see this, note that the bound on Johnson is never better than $\mathcal{O}(n^2 \log n)$, regardless of the value of m.

In this chapter, we also present a variant of Snowball that exploits graph separators and attains an upper bound on the run time of $\mathcal{O}(nw_d^2 + n^2s_d)$. This upper bound is even better than the one for the two other new algorithms, since $s_d \leq w_d$ is the size of the largest minimal separator induced by the vertex ordering d. While theoretical bounds on the run time usually give a good indication of the performance of algorithms, we see especially for this last variant that they do not always predict which algorithm is best in which settings. In Section 5.2, therefore, we experimentally establish the computational efficiency of the proposed algorithms on a wide range of graphs, varying from random scale-free networks and parts of the road network of New York City, to STNs generated from HTNs and job-shop scheduling problems. However, we first present the new algorithms and their analysis.

5.1 Algorithms

Recall from Chapter 3 that DPC performs a single iteration over the vertices in a temporal network, tightening some of the constraints along the way. In the previous chapter, we saw that it forms the basis of the P^{3}C algorithm that computes shortest paths between all pairs of vertices connected by an edge in a chordal graph. Even though, to the best of our knowledge, no DPC-based APSP algorithms have yet been proposed, algorithms for computing single-source

Algorithm 5.1: Min–path (Chleq, 1995)

Input: Weighted directed DPC graph $G = \langle V, E \rangle$;
(arbitrary) source vertex s and destination vertex t
Output: Distance from s to t

$\forall i \in V : D[i] \leftarrow \infty$
$D[s] \leftarrow 0$
for $k \leftarrow s$ **to** 1 **do**
  forall $j < k$ **such that** $\{j, k\} \in E$ **do**
    $D[j] \leftarrow \min\{D[j], D[k] + \omega_{kj}\}$
  end
end
for $k \leftarrow 1$ **to** t **do**
  forall $j > k$ **such that** $\{j, k\} \in E$ **do**
    $D[j] \leftarrow \min\{D[j], D[k] + \omega_{kj}\}$
  end
end
return $D[t]$

Algorithm 5.2: Chleq–APSP

Input: Weighted directed graph $G = \langle V, E \rangle$; vertex ordering
$d : V \to \{1, \ldots, n\}$
Output: Distance matrix D, or INCONSISTENT if G contains a negative cycle

$G \leftarrow \mathsf{DPC}(\mathrm{G},\mathrm{d})$
return INCONSISTENT **if** DPC did
for $i \leftarrow 1$ **to** n **do**
  $D[i][*] \leftarrow \mathsf{Min\text{–}paths}(G, i)$
end
return D

shortest paths (SSSP) based on DPC can be obtained from known results in a relatively straightforward manner. Chleq (1995) proposed a point-to-point shortest path algorithm that with a trivial adaptation computes SSSP; the IPPC algorithm, presented in the next chapter, also implicitly computes SSSP. Recalling that m_c denotes the number of edges in a chordal graph, these algorithms run in linear time $\mathcal{O}(m_c)$ and thus can simply be run once for each vertex to yield an APSP algorithm with $\mathcal{O}(nm_c) \subseteq \mathcal{O}(n^2 w_d)$ time complexity. Below, we first show how to adapt Chleq's algorithm to compute APSP; then, we present a new, efficient algorithm named Snowball that relates to $\mathsf{P}^3\mathsf{C}$.

Algorithm 5.3: Snowball

Input: Weighted directed graph $G = \langle V, E \rangle$; vertex ordering $d : V \to \{1, \ldots, n\}$
Output: Distance matrix D, or INCONSISTENT if G contains a negative cycle

```
1  G ← DPC(G,d)
2  return INCONSISTENT if DPC did
3  ∀i, j ∈ V : D[i][j] ← ∞
4  ∀i ∈ V : D[i][i] ← 0
5  for k ← 1 to n do
6  |  forall j < k such that {j,k} ∈ E  do
7  |  |  forall i ∈ {1,...,k−1} do
8  |  |  |  D[i][k] ← min{D[i][k], D[i][j] + ω_jk}
9  |  |  |  D[k][i] ← min{D[k][i], ω_kj + D[j][i]}
10 |  |  end
11 |  end
12 end
13 return D
```

Chleq's approach

Chleq's (1995) point-to-point shortest path algorithm was simply called Min–path and computes the shortest path between two arbitrary vertices $s, t \in V$ in a directionally path-consistent graph G, i.e. a graph on which the DPC algorithm from Chapter 3 has been run. It is reproduced here as Algorithm 5.1 and can be seen to run in $\mathcal{O}(m_c)$ time because each edge is considered at most twice. The shortest distance from the source vertex s is maintained in an array D; the algorithm iterates downward from s to 1 and then upward from 1 to t, updating the distance array when a shorter path is found.

Since the sink vertex t is only used as a bound for the second loop, it is clear that D actually contains shortest distances between all pairs (s, t') with $t' \leq t$. Therefore, we can easily adapt this algorithm to compute SSSP within the same $\mathcal{O}(m_c)$ time bound by setting $t = n$ and returning the entire array D instead of just $D[t]$. We call the result Chleq–APSP, which is included as Algorithm 5.2 and calls this SSSP algorithm (referred to as Min–paths) n times to compute all-pairs shortest paths in $\mathcal{O}(nm_c)$ time. Since the call to DPC in line 1 also fits within this bound, Chleq–APSP has a run-time complexity of $\mathcal{O}(nm_c) \subseteq \mathcal{O}(nw_d^2)$.

The Snowball algorithm

In this section, we present an algorithm that computes APSP, dubbed Snowball and included as Algorithm 5.3, that has the same asymptotic worst-case time bounds as Chleq–APSP but requires strictly less computational work.

Like Chleq–APSP, this algorithm first ensures that the input graph is direc-

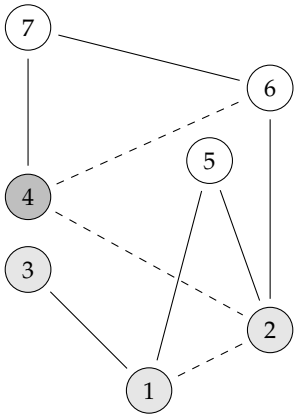


Figure 5.1: Snapshot ($k = 4$) of a graph during the operation of Snowball.

tionally path-consistent. The idea behind the algorithm is then that, during the execution of the outermost loop, we grow a clique $\{1, \ldots, k\}$ of computed (shortest) distances, one vertex at a time, starting with the trivial clique consisting of just vertex 1; after DPC performs a backward sweep along d, Snowball iterates in the other direction. When adding vertex k to the clique, the two inner loops ensure that we compute the distances between k and all vertices $i < k$. This works because we know by DPC that for any such pair (i, k), there must exist a shortest path from i to k of the form $i \to \cdots \to j \to k$ (and vice versa), such that $\{j, k\} \in E$ with $j < k$ is an edge of the chordal graph. This means that the algorithm only needs to "look down" at vertices $i, j < k$, and it follows inductively that $D[i][j]$ and $D[j][i]$ are guaranteed by an earlier iteration to contain shortest distances.

The name of our algorithm derives from its "snowball effect": the clique of computed distances grows quadratically during the course of its operation. A small example of the operation of Snowball is given in Figure 5.1. Originally, the graph contained a shortest path 4–7–6–2–5–1–3. Dashed edges have been added by DPC, and the path 4–2–1–3 is now (also) a shortest path; in particular, ω_{42} holds the shortest distance between vertices 4 and 2. This snapshot is taken for $k = 4$; the shaded vertices 1–3 have already been visited and shortest distances $D[i][j]$ have been computed for all $i, j \leq 3$. Then, during the iteration $k = 4$, for $j = 2$ and $i = 3$, the algorithm sets the correct weight of $D[4][3]$ by taking the sum $\omega_{42} + D[2][3]$.

Theorem 5.1. *Algorithm 5.3 (Snowball) computes all-pairs shortest paths in $\mathcal{O}(nm_c) \subseteq \mathcal{O}(n^2 w_d)$ time.*

Proof. The proof is by induction. After enforcing DPC, ω_{12} and ω_{21} are labelled by the shortest distances between vertices 1 and 2. For $k = 2$ and $i = j = 1$, the algorithm then sets $D[1][2]$ and $D[2][1]$ to the correct values.

Now, assume that $D[i][j]$ is set correctly for all vertices $i, j < k$. Let $\pi : i = v_0 \to v_1 \to \cdots \to v_{\ell-1} \to v_\ell = k$ be a shortest path from i to k, and further let

$h_{\max} = \arg\max_{h \in \{0,1,\ldots,\ell\}} \{v_h \in \pi\}$. By DPC, if $0 < h_{\max} < \ell$, there exists a path of the same weight where a shortcut $v_{h_{\max}-1} \to v_{h_{\max}+1}$ is taken.

Recall from Definition 3.4 that G_i denotes the subgraph of G induced by $V_i = \{v_1, \ldots, v_i\}$ (so $G_n = G$). The above argument can then be repeated to conclude that there must exist a shortest path π' from i to k that lies completely in G_k and, except for the last arc, in G_{k-1}. Thus, by the induction hypothesis and the observation that the algorithm considers all arcs from the subgraph G_{k-1} to k, $D[i][k]$ is set to the correct value. An analogous argument holds for $D[k][i]$.

With regard to the algorithm's time complexity, note that the two outermost loops together result in each of the m_c edges in the chordal graph being visited exactly once. The inner loop always has fewer than n iterations, yielding a run time of $\mathcal{O}(nm_c)$ time. From the observation that $m_c \leq nw_d$, we can also state a looser time bound of $\mathcal{O}(n^2 w_d)$. □

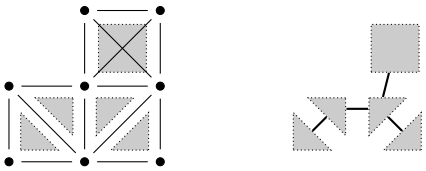
We now briefly discuss the consequences for two special cases: graphs of constant treewidth and chordal graphs. For chordal graphs, which can be recognised in $\mathcal{O}(m)$ time as discussed in Chapter 3, we can just substitute m for m_c in the run-time complexity; further, recall that in that case an elimination ordering d with $w_d = w^*$ exists and can be found in $\mathcal{O}(m)$ time. This gives the total run-time complexity of $\mathcal{O}(nm)$. Likewise, we stated in that chapter that for a given constant κ, it can be determined in $\mathcal{O}(n)$ time whether a graph has treewidth $w^* \leq \kappa$, and if so, a vertex ordering d with $w_d = w^*$ can be found within the same time bound. Then, omitting the constant factor w_d, the algorithm runs in $\mathcal{O}(n^2)$ time. This also follows from the algorithm's pseudocode by noting that every vertex k has a constant number (at most w^*) of neighbours $j < k$.

We note here the similarity between Snowball and the P^{3}C algorithm, presented as Algorithm 4.4 in Chapter 4. Like Snowball, P^{3}C operates by enforcing DPC, followed by a single backward sweep along the vertex ordering. P^{3}C then computes, in $\mathcal{O}(nw_d^2)$ time, shortest paths only for the arcs present in the chordal graph. This similarity and a property of chordal graphs in fact prompt us to present a version of Snowball with improved time complexity, where we take advantage of *graph separators*.

Improving run-time complexity using separators

In this section, we present an improvement of Snowball for an $\mathcal{O}(nw_d^2 + n^2 s_d)$ run time, where s_d is the size of the largest minimal separator (see Definition 3.6 on page 39) in the chordal graph obtained by triangulation along d. This bound is better because, as seen below, it always holds that $s_d \leq w_d$. The improvement hinges on the property called *partial path consistency* (PPC), formally defined in Chapter 3. As described there, in a partially path-consistent graph, each arc is labelled by the length of the shortest path between its endpoints.* P^{3}C, presented in Chapter 4 as Algorithm 4.4, depends on DPC and computes PPC in $\mathcal{O}(nw_d^2)$ time, which is the current state of the art. We use a clique tree of the PPC graph

*Full path-consistency (FPC) is achieved if such an arc exists for all pairs of vertices $u, v \in V$.



(a) Chordal graph (b) Clique tree

Figure 5.2: A chordal graph and its clique tree. Each shaded shape represents a maximal clique of the graph, containing the vertices at its corners.

to compute the shortest path between all vertices. Figure 5.2 shows an example of a chordal graph and its associated clique tree. Such a clique tree has the following useful properties (Heggernes, 2006, Section 3.2).

Definition 5.1. *Given a connected graph $G = \langle V, E \rangle$, a* separator *is a set $V' \subseteq V$ such that the subgraph of G induced by $V \setminus V'$ is no longer connected. A separator V' is* minimal *if no proper subset of V' is a separator.*

Property 5.1. Every chordal graph $G = \langle V, E \rangle$ has an associated clique tree $T = \langle C, S \rangle$, which can be constructed in linear time $\mathcal{O}(m_c)$.

Property 5.2. Each clique tree node $c \in C$ is associated with a subset $V_c \subseteq V$ and induces a maximal clique in G. Conversely, every maximal clique in G has an associated clique tree node $c \in C$.

Property 5.3. T is *coherent*: for each vertex $v \in V$, the clique tree nodes whose associated cliques contain v induce a subtree of T.

Property 5.4. If two clique tree nodes $c_i, c_j \in C$ are connected by an edge $\{c_i, c_j\} \in S$, $V_{c_i} \cap V_{c_j}$ is a minimal separator in G. Conversely, for each minimal separator V' in G, there is a clique tree edge $\{c_i, c_j\} \in S$ such that $V' = V_{c_i} \cap V_{c_j}$.

Property 5.5. All vertices appear in at least one clique associated with a node in T, so: $\bigcup_{c \in C} V_c = V$.

Since we have by Proposition 3.7 on page 40 that the size of the largest clique in a chordal graph is exactly $w_d + 1$, it follows from Properties 5.2 and 5.4 that $s_d \leq w_d$.

Now, the idea behind Snowball–separators is to first compute PPC in $\mathcal{O}(nw_d^2)$ time using P^{3}C, and then traverse the clique tree. PPC ensures that shortest paths within each clique have been computed. Then, when traversing the clique tree from an arbitrary root node out, we grow a set V_{visited} of vertices in cliques whose nodes have already been traversed. For each clique node $c \in C$ visited during the traversal, shortest paths between vertices in the clique V_c and vertices in V_{visited} must run through the separator V_{sep} between c and c's parent. If s_d is the size of the largest minimal separator in G, for each pair of vertices it suffices to consider at most s_d alternative routes for a total of $\mathcal{O}(n^2 s_d)$ routes, yielding the stated

Algorithm 5.4: Snowball–separators

Input: Weighted directed graph $G = \langle V, E \rangle$; vertex ordering $d : V \to \{1, \ldots, n\}$
Output: Distance matrix D, or INCONSISTENT if G contains a negative cycle

```
1  G ← P³C(G,d)
2  return INCONSISTENT if P³C did
3  ∀i, j ∈ V : D[i][j] ← ∞
4  ∀i ∈ V : D[i][i] ← 0
5  ∀ {i, j} ∈ E : D[i][j] ← ω_ij
6  ∀ {i, j} ∈ E : D[j][i] ← ω_ji
7  build a clique tree T = ⟨C, S⟩ of G
8  select an arbitrary root node c_root ∈ C of T
9  (D, V_visited) ← Process–clique–tree–node(c_root, nil, D, ∅)
10 return D
```

procedure Process–clique–tree–node($c, c_{\text{parent}}, D, V_{\text{visited}}$)

Input: Current clique tree node c, c's parent c_{parent}, distance matrix D, set of visited vertices V_{visited}
Output: Updated matrix D and set V_{visited}

```
1  if c_parent ≠ nil then
2  |  V_new ← V_c \ V_{c_parent}
3  |  V_sep ← V_c ∩ V_{c_parent}
4  |  V_other ← V_visited \ V_c
5  |  forall (i, j, k) ∈ V_new × V_sep × V_other do
6  |  |  D[i][k] ← min{D[i][k], D[i][j] + D[j][k]}
7  |  |  D[k][i] ← min{D[k][i], D[k][j] + D[j][i]}
8  |  end
9  end
10 V_visited ← V_visited ∪ V_c
11 forall children c' of c do
12 |  (D, V_visited) ← Process–clique–tree–node(c', c, D, V_visited)   // recursion
13 end
14 return (D, V_visited)
```

overall time complexity of $\mathcal{O}(nw_d^2 + n^2 s_d)$. We formally present the algorithm based on this idea as Algorithm 5.4 with its associated recursive procedure Process–clique–tree–node (above).

Note that because we visit a node's parent before visiting the node itself, it always holds that $V_{c_{\text{parent}}} \subseteq V_{\text{visited}}$. Further note that, for simplicity of presentation, we assume the graph to be connected. If not, we can simply find all connected components in linear time and construct a clique tree for each of them.

The improved algorithm has an edge over the original algorithm when separators are small while the treewidth is not. HTN-based sibling-restricted STNs (which are described as part of our experimental validation in Section 5.2), for instance, have many separators of size 2. If every task has as many as $\mathcal{O}(\sqrt{n})$ subtasks and every task with its subtasks induces a clique, we have $w_d \in \mathcal{O}(\sqrt{n})$ and $s_d = 2$, implying that Snowball–separators still has an optimal $\mathcal{O}(n^2)$ time complexity for these instances.*

Before we proceed to prove that the algorithm is correct and meets the stated run-time bounds, we introduce the following definition.

Definition 5.2. *We define a distance matrix D as* valid *for a set U of vertices, and (D, U) as a* valid pair*, if for all pairs of vertices $(i, j) \in U \times U$, $D[i][j]$ holds the shortest distance in G from i to j.*

We split the correctness proof of the algorithm into three parts: Lemmas 5.2 and 5.3 culminate in Theorem 5.4. The first step is to show that if Process–clique–tree–node is called with a valid pair (D, U) and some clique node c, the procedure extends the validity to $U \cup V_c$.

Lemma 5.2. *Consider a call to procedure* Process–clique–tree–node *with, as arguments, a clique node c, c's parent c_{parent}, a distance matrix D, and the set of visited vertices V_{visited}. If D is valid for V_{visited} upon calling, then D becomes valid for $V_c \cup V_{\text{visited}}$ after running lines 1–8 of* Process–clique–tree–node.

Proof. First, note that by Property 5.2, V_c induces a clique in G. Therefore, edges exist between each pair (i, k) of vertices in V_c, and since the graph is PPC, ω_{ik} is labelled with the shortest distance between i and k. Due to lines 5 and 6 of the main algorithm, D also contains these shortest distances, so D is valid for V_c.

Now, it remains to be shown that for each pair of vertices $(i, k) \in V_c \times V_{\text{visited}}$ the shortest distances $D[i][k]$ and $D[k][i]$ are set correctly. We show here the case for $D[i][k]$; the other case is analogous.

The desired result follows trivially if $c_{\text{parent}} = \text{nil}$, since the procedure is then called with $V_{\text{visited}} = \varnothing$. Otherwise, let $V_{\text{new}} = V_c \setminus V_{c_{\text{parent}}}$, $V_{\text{sep}} = V_c \cap V_{c_{\text{parent}}}$ and $V_{\text{other}} = V_{\text{visited}} \setminus V_c$ as set by the procedure in lines 2–4. If either i or k lies in V_{sep}, the correctness of $D[i][k]$'s value was already proven, so we only need to consider pairs of vertices $(i, k) \in V_{\text{new}} \times V_{\text{other}}$.

For any such pair (i, k), V_{sep} is a separator between i and k by Property 5.4, so any shortest path from i to k is necessarily a concatenation of shortest paths from i to j^* and from j^* to k, for some $j^* \in V_{\text{sep}}$. Since it follows from the definitions of V_{new}, V_{sep} and V_{other} that for all $(i, j) \in V_{\text{new}} \times V_{\text{sep}}$ and $(j, k) \in V_{\text{sep}} \times V_{\text{other}}$, $D[i][j]$ and $D[j][k]$ are correctly set (by the validity of D for V_c and V_{visited}, respectively), the loop on lines 5–8 yields the desired result. □

Our next step is to prove that through the recursive calls, validity is in fact extended to the entire subtree rooted at c.

*However, since in general not every task forms a clique with its subtasks, this low value of s_d will usually not be attained in practice.

Lemma 5.3. *Consider again a call to procedure* Process–clique–tree–node *with, as arguments, a clique node c, c's parent $V_{c_{\text{parent}}}$, a distance matrix D, and the set of visited vertices V_{visited}. If D is valid for V_{visited} upon calling, then the returned, updated pair $(D', V'_{\text{visited}})$ is also valid.*

Proof. First, note that by Lemma 5.2, D is valid for V_{visited} after the update in line 10.

Assume that the clique tree has a depth of d; the proof is by reverse induction over the depth of the clique tree node. If c is a clique tree node at depth d (i.e. a leaf), the loop in lines 11–13 is a no-op, so we immediately obtain the desired result.

Now assume that the lemma holds for all nodes at depth k and let c be a clique tree node at depth $k-1$. For the first call (if any) made for a child node c' during the loop in lines 11–13, this lemma can then be applied. As a consequence, the returned and updated pair is again valid. This argument can be repeated until the loop ends and the procedure returns a valid pair. □

With this result at our disposal, we can state and prove the main theorem of this section.

Theorem 5.4. *Algorithm 5.4 (Snowball–separators) correctly computes all-pairs shortest paths in $\mathcal{O}(nw_d^2 + n^2 s_d)$ time.*

Proof. Note that $V_{\text{visited}} = \varnothing$ for the call to Process–clique–tree–node in line 9 of Snowball–separators and therefore the pair (D, V_{visited}) is trivially valid. By Lemma 5.3, this call thus returns a valid updated pair (D, V_{visited}). Since Process–clique–tree–node has recursively traversed the entire clique tree, V_{visited} contains the union $\bigcup_{c \in C} V_c$ of all cliques in the clique tree $T = \langle C, S \rangle$, which by Property 5.5 equals the set of all vertices in G. Therefore, D contains the correct shortest paths between all pairs of vertices in the graph.

As for the time complexity, note that the initialisations in lines 3 and 4 can be carried out in $\mathcal{O}(n^2)$ time, whereas those in lines 5 and 6 require $\mathcal{O}(m_c)$ time. By Property 5.1, the clique tree can also be built in linear time. Since the clique tree contains at most n nodes, Process–clique–tree–node is called $\mathcal{O}(n)$ times. Line 1 requires $\mathcal{O}(w_d^2)$ time. To implement lines 2–4 and 10 of Process–clique–tree–node, we represent the characteristic function for V_{visited} as an array of size n; using V_{visited} instead of $V_{c_{\text{parent}}}$ everywhere, we then we simply iterate over all $\mathcal{O}(w_d)$ members of V_c to perform the required computations.

Now, only the complexity of the loop in lines 5–8 remains to be shown. Note that $|V_{\text{sep}}| \leq s_d$ by definition, and $|V_{\text{other}}| < n$ always. Further using the observation that each of the n vertices in the graph appears in V_{new} for exactly one invocation of Process–clique–tree–node (after which it becomes a staunch member of V_{visited}), we obtain a total time bound of $\mathcal{O}(n^2 s_d)$ for the loop over all invocations. □

While the recursive description above is perhaps easier to grasp and satisfies the claimed time bounds, we found that efficiency benefited in practice from

Table 5.1: Properties of the benchmark sets

type	#cases	n	m	w_d	s_d
Chordal					
– Figure 5.3	250	1,000	75,840–499,490	79–995	79–995
– Figure 5.4	130	214–3,125	22,788–637,009	211	211
Scale-free					
– Figure 5.5	130	1,000	1,996–67,360	88–864	80–854
– Figure 5.6	160	250–1,000	2,176–3,330	150–200	138–190
New York	170	108–3,906	113–6,422	2–51	2–40
Diamonds	130	111–2,751	111–2,751	2	2
Job-shop	400	17–1,321	32–110,220	3–331	3–311
HTN	121	500–625	748–1,599	2–128	2–127

an iterative implementation. It also turns out that a good heuristic is to first visit child nodes connected to the already visited subtree by a large separator, postponing the processing of children connected by a small separator, because the set of visited vertices is then still small. In this way, the sum of terms $\left|V_{\text{sep}} \times V_{\text{visited}}\right|$ is kept low. In our implementation, we therefore used a priority queue of clique nodes ordered by their separator sizes. Future research must point out whether it is feasible to determine an optimal traversal of the clique tree within the given time bounds.

Having presented our new algorithms and proven their correctness and formal complexity, we now move on to an empirical evaluation of their performance.

5.2 Experiments

We evaluate the two algorithms together with efficient implementations of Floyd–Warshall and Johnson with a Fibonacci heap* across six different benchmark sets.†

The properties of the experimental classes are summarised in Table 5.1. This table lists the number of benchmarks per class, the range of the number of vertices n, edges m, the induced width w_d produced by the minimum degree heuristic, as well as the size of the largest minimal separators s_d in the graphs. More details on the different sets can be found below, but one thing that stands out immediately is that s_d is often equal to or only marginally smaller than w_d. However, the median size of the minimal separator is less than 10 for all instances except the constructed chordal graphs.

All algorithms were implemented in Java and went through an intensive

*For Johnson we used the corrected Fibonacci heap implementation by Fiedler (2008), since the widely used pseudocode of Cormen et al. (2001) contains mistakes.

†Available at
`http://dx.doi.org/10.4121/uuid:49388c35-c7fb-464f-9293-cca1406edccf`

Table 5.2: The summed induced width, triangulation, and total run time of Snowball over all experiments on general (non-chordal) graphs show that the minimum degree heuristic is the best choice.

heuristic	$\sum w_d$	triangulation (s)	Snowball (s)	total (s)
min-fill	**321,492**	1,204,384	**2,047**	1,206,431
min-degree	326,222	**498**	3,166	**3,664**
MCS	365,662	1,520	3,348	4,868
static min-fill	388,569	1,387	2,746	4,133
static min-degree	388,707	1,317	2,748	4,064
random	505,844	2,436	5,179	7,615

profiling phase.* The experiments were run using Java 1.6 (OpenJDK-1.6.0.b09) in server mode, on Intel Xeon E5430 CPUs running 64-bit Linux. The Java processes were allowed a maximum heap size of 4 GiB, and used the default stack size. We report the measured CPU times, including the time that was spent running the ordering heuristic for Chleq–APSP and Snowball. The reported run times are averaged over 10 runs for each unique problem instance. Moreover, we generated 10 unique instances for each parameter setting, obtained by using different random seeds. Thus, each reported statistic represents an average over 100 runs, unless otherwise indicated. Finally, each graph instance was ensured to contain no cycles of negative weight.

Triangulation

As discussed in Chapter 3, finding an optimal vertex ordering (with minimum induced width) is NP-hard, but several efficient ordering heuristics for this problem exist. We ran our experiments with six different heuristics discussed in that chapter: the minimum fill and minimum degree heuristics, static variants of both (taking into account only the original graph), an ordering produced by running maximum cardinality search (MCS) on the original graph, and a random ordering. Recall that all of these, except minimum fill, have time complexities within the bound on the run time of Chleq–APSP and Snowball.

We found that, on average, the minimum degree heuristic gave induced widths less than 1.5% higher than those found by minimum fill, but with drastically lower run time. The exorbitant time consumption of the minimum fill heuristic can be partially explained by the fact that we used the implementation from the LibTW package† to compute this ordering, which can probably be improved. However, it is also known from the literature that the theoretical bound on the minimum fill heuristic is worse than that of minimum degree (Kjærulff, 1990).

*Our implementations are available in binary form at
`http://dx.doi.org/10.4121/uuid:776a266e-81c6-41ee-9d23-8c89d90b6992`

†Available from `http://treewidth.com/`.

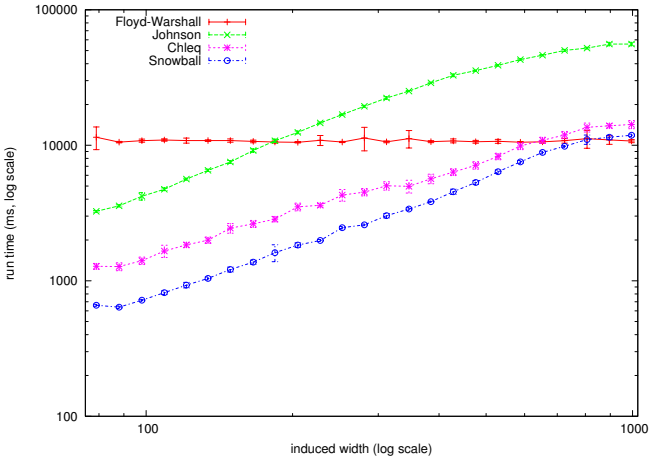


Figure 5.3: Run times on generated chordal graphs with a fixed number of 1000 vertices and varying treewidth.

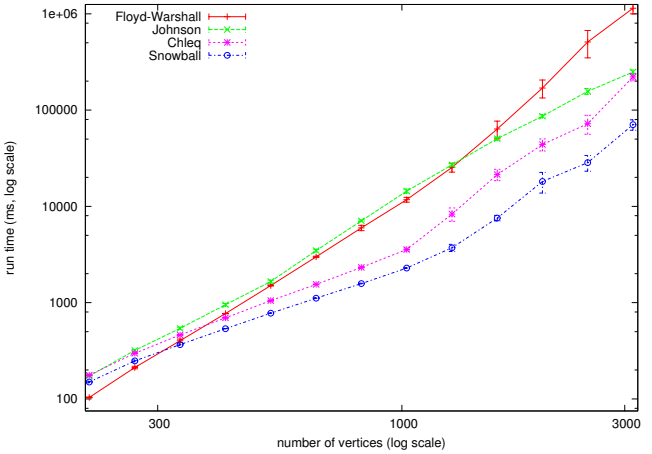


Figure 5.4: Run times on generated chordal graphs of a fixed treewidth of 211.

All other heuristics are not only slower than minimum degree, but also yield an average induced width at least 12% higher, resulting in a longer total triangulation time and a longer total run time of Snowball (see the summary of the results over all benchmarks given in Table 5.2). Again, this confirms Kjærulff's earlier work. In the experimental results included below we therefore only show the results based on the minimum degree heuristic.

Chordal graphs

To evaluate the performance of the new algorithms on chordal graphs, we construct chordal graphs of a fixed size of 1,000 vertices with a treewidth ranging

from 79 up to just less than the number of vertices, thus yielding a nearly complete graph at the high end. Since these graphs are already chordal, we do not need to run all ordering heuristics mentioned above. As shown in Chapter 3, the MCS algorithm is guaranteed to produce an elimination ordering d with $w_d = w^*$.

The results of this experiment are depicted in Figure 5.3. In this figure and others, the error bars represent the standard deviations in the measured run time for the instances of that size. For graphs up to an induced width of about three quarters of the number of vertices, Snowball significantly outperforms Floyd–Warshall (which yields the expected horizontal line), and overall the run time of both new algorithms is well below that of Johnson across the entire range. Figure 5.4 shows the run times on chordal graphs of a constant treewidth and with increasing number of vertices. Here, the two new algorithms outperform Johnson by nearly an order of magnitude (a factor 9.3 for Snowball around $n = 1300$), and even more so regarding Floyd–Warshall, confirming the expectations based on the theoretical upper bounds.

General graphs

For general, non-chordal graphs, we expect from the theoretical analysis that the $\mathcal{O}(nw_d^2)$-time Chleq–APSP and Snowball algorithms are faster than Johnson with its $\mathcal{O}(nm + n^2 \log n)$ time bound when w_d is low, and that Johnson is faster on sparse graphs (where m is low) of a large induced width w_d. The main question is at which induced width this changeover occurs. Regarding Floyd–Warshall with its $\mathcal{O}(n^3)$ bound, we expect that for larger n it is always outperformed by the other algorithms.

Scale-free graphs

Scale-free networks are networks whose degree distribution follows a power law. That is, for large values of k, the fraction $P(k)$ of vertices in the network having k connections to other vertices tends to $P(k) \sim ck^{-\gamma}$, for some constant c and parameter γ. In other words, few vertices have many connections while many vertices have only a few connections. Such a property can be found in many real-world graphs, such as in social networks and in the Internet. Our instances were randomly generated with Albert and Barabási's (2002) preferential attachment method, where in each iteration a new vertex is added to the graph, which is then attached to a number of existing vertices; the higher the degree of an existing vertex, the more likely it is that it will be connected to the newly added vertex.* To see at which induced width Johnson is faster, we compare the run times on such generated graphs with 1,000 vertices. By varying the number of attachments for each new vertex from 2 to $n/2$, we obtain graphs with an induced width ranging from 88 to 866. In these graphs, the induced width is

*See the footnote on page 78 for a remark on the limitations of this approach and an improved generator proposed by Bollobás and Riordan (2004).

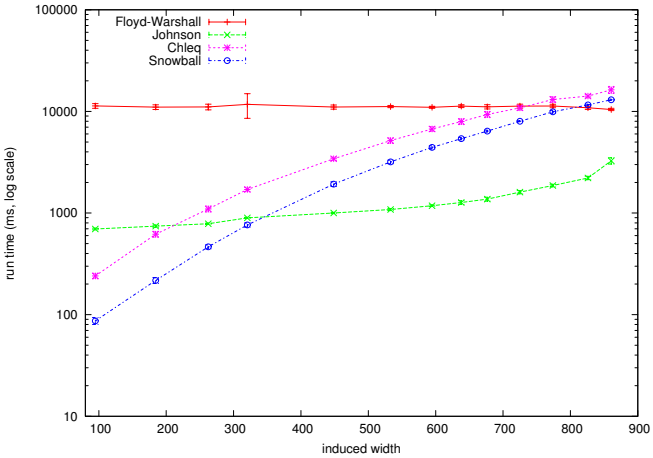


Figure 5.5: Run times on the scale-free benchmarks for graphs of 1,000 vertices and varying induced width.

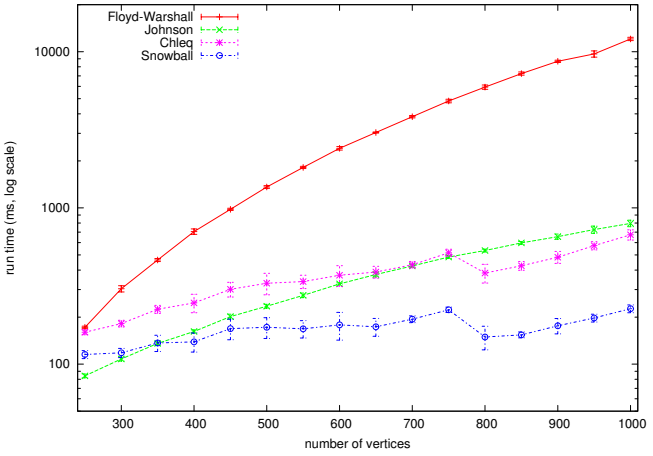


Figure 5.6: Run times on the scale-free benchmarks for graphs of induced widths 150 to 200 and varying vertex count.

already quite large for small attachment values: for example, for a value of 11, the induced width is already over 500.

The results of this experiment can be found in Figure 5.5. Here we see that up to an induced width of about 350 (attachment value 5), Snowball is the most efficient. For higher induced widths, Johnson becomes the most efficient; for w_d around 800, even Floyd–Warshall becomes faster than Snowball. A consistent observation but from a different angle can be made from Figure 5.6, where we keep the induced width between 150 and 200 while the number of vertices is varied from 250 to 1,000. For these cases, the number of edges ranges from 2,176 to 3,330. Here, we see that for small graphs up to 350 vertices, Johnson is the fastest; then Snowball overtakes it, and around 750 vertices Chleq–APSP is also faster than Johnson (this holds for all results up to a sparse graph of 1,000 vertices).

There is a conspicuous decrease in the run time for both Snowball and Chleq–APSP around the mark of 750 vertices. This is an artifact of the (preferential attachment) benchmark generator. Since we cannot generate scale-free graphs with a specific induced width, we modify the attachment value instead. As it turns out, for graphs of this size only one attachment value yields an induced width within the desired range; for the graph of size 750, this width is at the high end of the interval, whereas for the graph of size 800 it is near the low end. This explains the reduced run time for the larger graph.

For these scale-free networks, we conclude that Snowball is the fastest of the four algorithms only when the induced width is not too large (at most one third of the number of vertices in our benchmark set). However, we also observe that the structure of scale-free networks is such that they have a particularly high induced width for relatively sparse graphs, exactly because a few vertices have most of the connections. Therefore, it is only for relatively small attachment values that Snowball becomes competitive.

Selections from New York road network

Perhaps more interesting than the artificially constructed graphs are graphs based on real networks, for which shortest path calculations are relevant. The first of this series is based on the road network of New York City, which we obtained from the DIMACS challenge website.* This network is very large (with 264,346 vertices and 733,846 edges), and since the size of the output of APSP algorithms scales with the number of vertices squared, we opted to compute shortest paths for induced *subgraphs* of varying sizes. These were obtained by running a simple breadth-first search from a random starting location until the desired number of vertices had been visited. The geographical extent of the subnetworks thus obtained is illustrated for three different sizes in Figure 5.7.

The results of all algorithms on these subgraphs can be found in Figure 5.8. Here we observe the same ranking of the algorithms as on the chordal graphs of a fixed treewidth: Floyd–Warshall is slowest with its $\Theta(n^3)$ run time, then each

*http://www.dis.uniroma1.it/~challenge9/

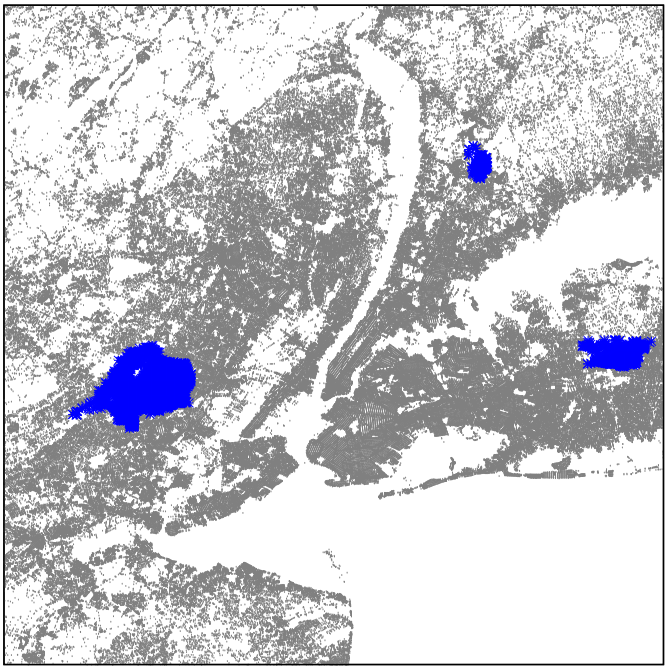


Figure 5.7: Coordinates for the vertices in the New York City input graph, with three highlighted examples, representing subgraphs with respectively 250, 1000, and 5000 vertices.

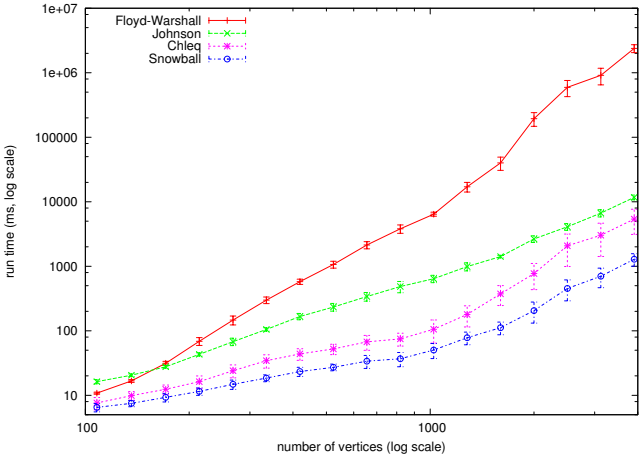


Figure 5.8: Run times on the New York benchmarks for subgraphs of varying vertex count.

of Johnson, Chleq–APSP, and Snowball is significantly faster than its predecessor. This can be explained by considering the induced width of these graphs. Even for the largest graphs, the induced width is around 30, which is considerably smaller than the number of vertices.

STNs from diamonds

This benchmark set is based on problem instances in difference logic proposed by Strichman et al. (2002) and also appearing in SMT-LIB (Ranise and Tinelli, 2003), where the constraint graph for each instance takes the form of a circular chain of "diamonds". Each such diamond consists of two parallel paths of equal length starting from a single vertex and ending in another single vertex. From the latter vertex, two paths start again, to converge on a third vertex. This pattern is repeated for each diamond in the chain; the final vertex is then connected to the very first one. The sizes of each diamond and the total number of diamonds are varied between benchmarks.

Problems in this class are actually instances of the NP-complete Disjunctive Temporal Problem (DTP): constraints take the form of a disjunction of inequalities. From each DTP instance, we obtain an STP instance (i.e. a graph) by randomly selecting one inequality from each such disjunction. This STP is most probably inconsistent, so its constraint graph contains a negative cycle; we remedy this by modifying the weights on the constraint edges. The idea behind this procedure is that the *structure* of the graph still conforms to the type of networks that one might encounter when solving the corresponding DTP instance, and that the run time of the algorithms mostly depends on this structure. Moreover, to reduce the influence of the randomized extraction procedure, we repeat it for 10 different seeds.

For our benchmark set, we considered problem instances which had the size of the diamonds fixed at 5 and their number varying. What stands out about this set is that the graphs generated from it are all very sparse. We ran experiments on 130 graphs, ranging in size from 111 to 2751 vertices, all with an induced width of 2. This induced width is clearly extremely small, which translates into Chleq–APSP and Snowball being considerably faster than Johnson and Floyd–Warshall, as evidenced by Figure 5.9.

STNs from job-shop scheduling

We generated each of the 400 graphs in our "job-shop" set from an instance of a real job-shop problem. These instances were of the type available in SMT-LIB (Ranise and Tinelli, 2003), but of a larger range than included in that benchmark collection. To obtain the graphs from these job-shop instances, we used the extraction procedure described in the previous section.

The most striking observation that can be taken from Figure 5.10 is that the difference between Johnson and the two new algorithms is not quite as pronounced, though Snowball is consistently the fastest of the three by a small margin. The fact that this margin is so small is most likely due to the structure

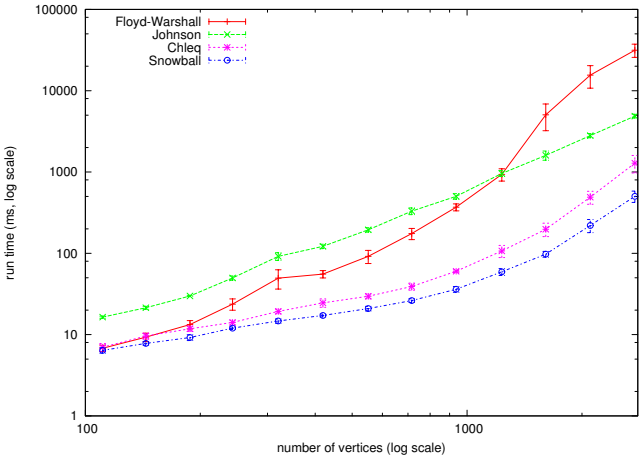


Figure 5.9: Run times on the diamonds benchmarks for graphs of varying vertex count.

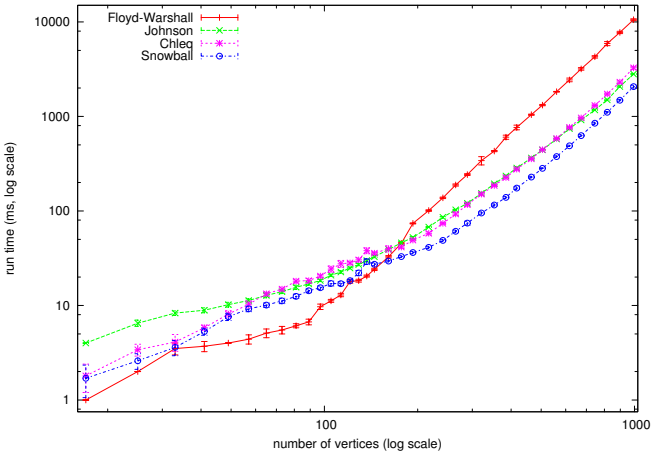


Figure 5.10: Run times on the job-shop benchmarks for graphs of varying vertex count.

of these graphs, which is also reflected in their relatively high induced width. Note also that the run times for Floyd–Warshall are better for graphs of up to 160 vertices, while for larger graphs the other algorithms are significantly faster.

STNs from HTNs

The instances in the last benchmark set under consideration imitate so-called sibling-restricted STNs originating from Hierarchical Task Networks. This set is therefore particularly interesting from a planning point of view. In these graphs, constraints normally occur only between parent tasks and their children, and between sibling tasks (Bui and Yorke-Smith, 2010). We consider an extension that includes *landmark variables* (Castillo et al., 2002) mimicking synchronisation between tasks in different parts of the network, and thereby causing some deviation from the tree-like HTN structure.

We generate HTNs using the following parameters: (i) the number of tasks in the initial HTN tree (fixed at 250; note that tasks have a start and end point), (ii) the branching factor, determining the number of children for each task (between 4 and 7), (iii) the depth of the HTN tree (between 3 and 7), (iv) the ratio of landmark time points to the number of tasks in the HTN, varying from 0 to 0.5 with a step size of 0.05, and (v) the probability of constraints between siblings, varying from 0 to 0.5 with a step size of 0.05.

These settings result in graphs of between 500 and 625 vertices, with induced widths varying between 2 and 128. Though this induced width seems high in light of our statement above that it is constant, by construction it is bounded by $2 \times \textit{branching factor} + \#\textit{landmarks} + 1$ for all instances. Filling in the respective maximal values of 7 and 125, we find an upper bound $w_d \leq 140$, well above the actual maximum encountered.

Figure 5.11 shows the run-time results of these experiments as a function of the induced widths of the graphs. It can be seen that only for the larger induced widths, Johnson's performance comes close to that of Chleq–APSP . These large induced widths are only found for high landmark ratios of 0.5. The results suggest that for STNs stemming from HTNs, Snowball is significantly more efficient than Johnson and is the algorithm of choice.

Snowball–Separators

In Section 13 we presented a version of Snowball that has an improved worst-case run time over vanilla Snowball by taking advantage of the separators in the graph. In this section, we discuss the results of our experiments comparing these two variants. First, we turn our attention to the benchmark problems on regular graphs. Our results are summarised in Figure 5.12. As one can see, Snowball–separators in fact performs strictly worse on these sets in terms of run-time performance when compared to the original Snowball.

However, as can be seen in Table 5.1, the largest minimal separator is often equal to or only marginally smaller than the induced width. Even though there may be only few separators this large, and many may be substantially smaller

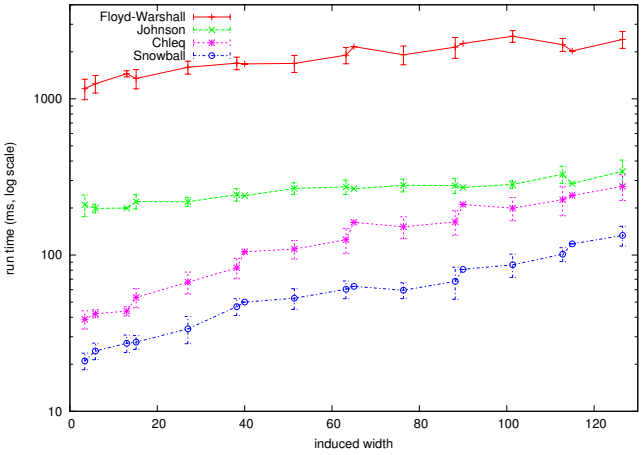


Figure 5.11: Run times on the HTN benchmarks for graphs from 500 to 625 vertices and varying induced width. Each point is the average of instances with an induced width within a range $[5k, 5k + 4]$, for some k. This results in between 5 and 11 instances per data point.

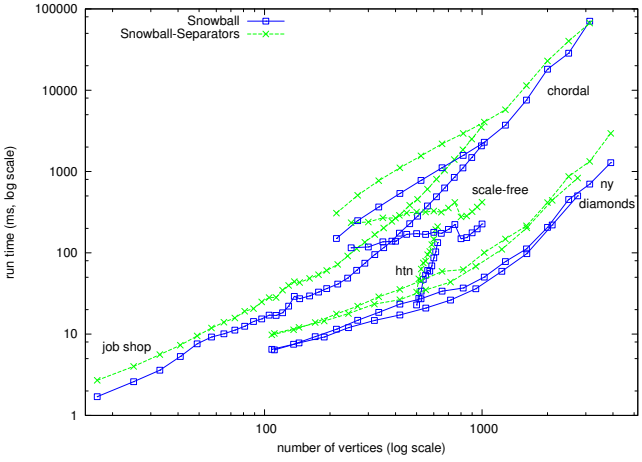


Figure 5.12: Run times of the Snowball algorithms on the benchmark problem sets listed in Table 5.1.

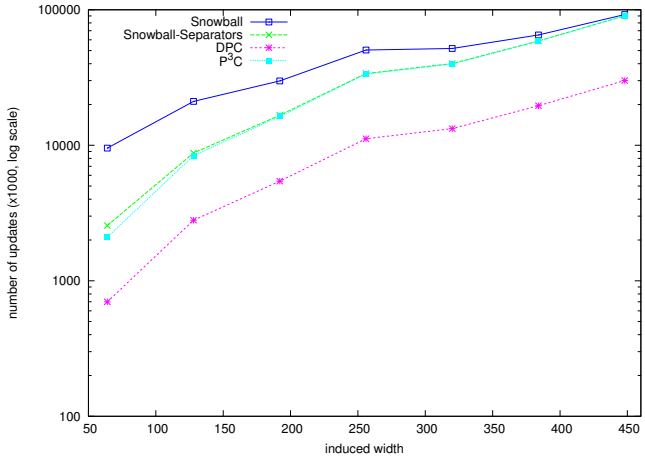


Figure 5.13: Number of distance matrix updates on chordal instances with 512 vertices, largest minimal separator size 2 and varying treewidth. Each point represents between 5 and 10 instances.

(as noted above, for most instances the median separator size is below 10), this prompts us to run experiments on instances where separator sizes are kept small artificially. Indeed, we found that there are cases where Snowball–separators shows an improvement over vanilla Snowball when comparing the number of update operations performed — i.e. lines 8 and 9 of Snowball and lines 6 and 7 in Process–clique–tree–node, along with similar operations in DPC and P^{3}C.

One such case is presented in Figure 5.13. This describes the results on a collection of chordal graphs of 512 vertices, in which the largest minimal separator is fixed at size 2, and the treewidth is varied between 16 and 448. The figure also includes the results of DPC and P^{3}C, as these are the respective subroutines of Snowball and Snowball–separators. For these graphs, Snowball–separators performs strictly fewer update operations than Snowball on all instances, although the difference becomes smaller as the induced width increases. While the number of updates shows a distinct improvement over Snowball, the same is not apparent in the run times. Instead, as can be seen from Figure 5.14, the run times of Snowball are strictly better than those of Snowball–separators on all instances. Snowball can now even be seen to outperform P^{3}C which has a better theoretical bound. The explanation for this behaviour is that the adjacency matrix data structure as used by Snowball is very fast, while the adjacency list used by P^{3}C, though staying within the theoretical bound, inflicts a larger constant factor on the run time. See also the experiments in the previous chapter, where we also ran P^{3}C on an adjacency matrix.

From these experiments, we thus conclude that on graphs of these sizes, the additional bookkeeping required by Snowball–separators outweighs the potential improvement in the number of distance matrix updates.

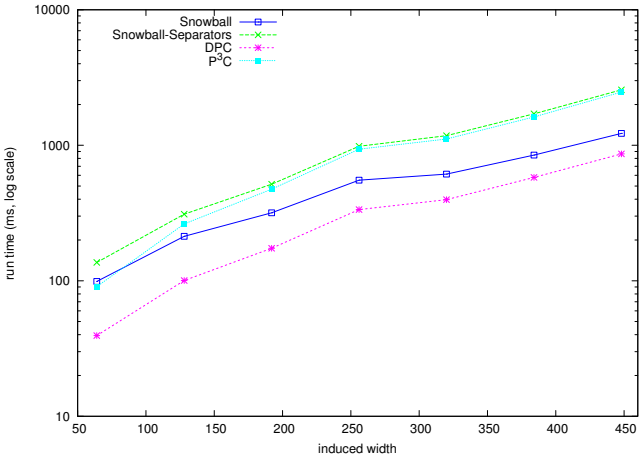


Figure 5.14: Run times on chordal instances with 512 vertices, largest minimal separator size 2 and varying treewidth. Each point represents between 5 and 10 instances.

A proper upper bound on the run time

On general graphs, the run time of the proposed algorithms depends on the induced width w_d of the ordering produced by the ordering heuristic. This induced width is not a direct measure of the input (graph), so the given upper bound on the run time is not quite proper. In this section, we aim to relate the run time to the treewidth (denoted w^*) which is a property of the input and thus to arrive at a proper bound.

However, determining the treewidth is NP-hard, which is intractable for the benchmark problems we used. We therefore compare the measured induced width $w_d \geq w^*$, an upper bound on the treewidth, to a lower bound* $x \leq w^*$.

We are unaware of any guarantee on the quality relative to the treewidth of either the minimum degree ordering heuristic or the lower bound we used. However, we can calculate the ratio w_d/x to get an upper bound on the ratio w_d/w^*. From this measure we then obtain an upper bound on the run time expressed in the treewidth, at least for the benchmark problems in these experiments.

The results of these computations can be found in Figure 5.15, where we plot these ratios for the New York, HTN, scale-free and job-shop benchmarks as a function of the lower bound x. Using a least-squares approach, we then fitted functions $w_d(x) = cx^k$ (showing up as a straight line in this log-log plot) to the plotted data points. For functions found by fitting, we get $k = 4.6$ for New York, $k = 2.3$ for HTN, and $k = 0.98$ for job-shop, all with small multiplicative constants $0.012 < c < 1.62$. As one can see from the plotted data points for the scale-free instances, they are not amenable to such a fit and we therefore omit it from the figure.

*The lower bound was computed with the LibTW package; see `http://treewidth.com/`. We used the MMD + Least-c heuristic.

The decreasing trend for the job-shop data indicates that the quality of the triangulation — and thus of the upper bound represented by the induced width — gradually increases: the lower and upper bound are always less than a factor 2 apart. Indeed, if we plot a line representing a function $w'_d(x) = 2x$, yielding a horizontal line in this figure, we find that it describes a comfortable upper bound on the data points for this benchmark set.

The HTN data prompts us to plot a function $w''_d(x) = \frac{2}{5}x^{2.5}$, with an exponent slightly higher than the one we found from the least-squares fit, and further tweaked slightly by a multiplicative coefficient to bring it into view. This function as plotted represents an ample upper bound for the HTN benchmarks (as well as the job-shop ones).

The fit for the data points for the New York benchmark is not good and the trend of the points themselves is not very clear, because the lower bound only spans an interval from 1 to 4. Therefore, we cannot give an upper bound for this set of benchmarks with any acceptable level of confidence.

However, the scale-free data points we plotted, which could not be fitted with a function yielding a straight line, do mostly follow a clear curving trend. A hypothesis for this behaviour is that the quality of the upper and lower bound deteriorates mostly for the middle sizes of the benchmarks; smaller and larger scale-free graphs are easier to triangulate well.* To give an upper bound, we could plot any line on the outer hull of these data points; e.g. the horizontal line represented by $w_d(x) = 8x$ would work. The most pessimistic assumption would be to choose a function with the highest slope, and we find that the upper bound $w''_d(x) = \frac{2}{5}x^{2.5}$, found for the HTN benchmarks, also works here.

From this discussion, we may conclude that for all benchmarks we ran except for New York, $w_d(x)$ is $\mathcal{O}(x^{2.5})$ which in turn is $\mathcal{O}(w^{*2.5})$; the run time of the algorithms Snowball and Chleq–APSP on these instances can therefore be bounded by $\mathcal{O}(n^2 w^{*2.5})$.

To conclude this section, we remark that as an alternative to ordering heuristics, one may use approximation algorithms with a bound on the induced width that can be theoretically determined. For example, Bouchitté et al. (2004) give an $\mathcal{O}(\log w^*)$ approximation of the treewidth w^*. Using such an approximation would give an upper bound on the run time of Snowball of $\mathcal{O}(n^2 w^* \log w^*)$. However, the worst-case time complexity of this method for obtaining the approximate induced width is $\mathcal{O}(n^3 \log^4 n w^{*5} \log w^*)$ and it has a high constant as well, so it is — for now — mainly of theoretical value.

5.3 Johnson's heap

For the experiments in this chapter, we presented the results for Johnson using a Fibonacci heap, because only then the theoretical bound of $\mathcal{O}(nm + n^2 \log n)$ time is attained. In practice, using a binary heap for a theoretical bound of $\mathcal{O}(nm \log n)$

*This mirrors earlier observations by the authors.

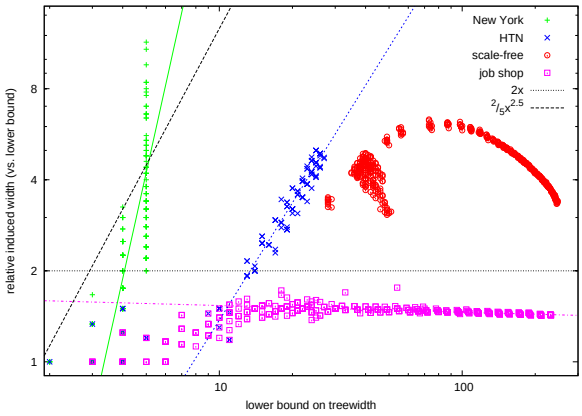


Figure 5.15: An upper bound on the induced width relative to the treewidth can be determined experimentally by comparing it to a lower bound on the treewidth.

time turns out to be more efficient on some occasions, as we show by the results in this section.

Figure 5.16 shows the run times of Johnson with a binary heap and with a Fibonacci heap on all of the benchmark sets listed in Table 5.1. On the diamonds, HTN, and New York benchmarks the binary heap is a few percent faster than the Fibonacci heap, but the slope of the lines in this log-log scale is the same, so we can conclude that the average-case run time has similar asymptotic behavior. However, for larger job-shop problems, a binary heap is a factor 2 slower than a Fibonacci heap, and on our chordal graph benchmark problems even a factor 10. Our benchmark problems on scale-free graphs with a fixed number of vertices help to explain this difference.

In Figure 5.17, the run time of both variants of Johnson can be found for scale-free graphs with 1,000 vertices, with the number of edges varying from about 2,000 to over 67,000. We see that only for the sparsest scale-free graphs with about $2,000$ edges, the binary heap is slightly faster, but when more edges are considered, using the Fibonacci heap significantly outperforms using the binary heap. In particular, the run time of the Fibonacci heap implementation increases only slowly with the number of edges, while the run time of the binary heap increases much more significantly. This can be explained by the fact that when running Dijkstra's algorithm as a subroutine in Johnson, each update of a (candidate) shortest path can be done in amortized constant time with a Fibonacci heap, while in a binary heap this has a worst-case cost of $\mathcal{O}(\log n)$ time per update. The number of updates is bounded by m for each run of Dijkstra's algorithm, yielding a bound of $\mathcal{O}(nm)$ updates for Johnson. For the binary heap this $\mathcal{O}(nm \log n)$ bound accounts for a significant part of the run time, while with a Fibonacci heap other operations (such as extracting the minimum element from

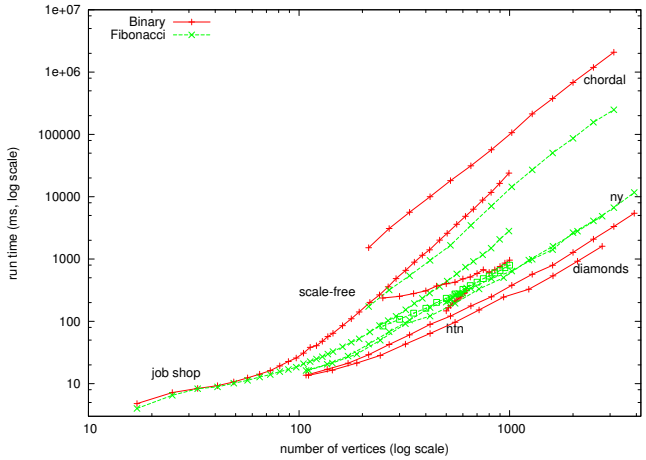


Figure 5.16: Run times of Johnson with a binary heap and with a Fibonacci heap on the benchmark problem sets listed in Table 5.1.

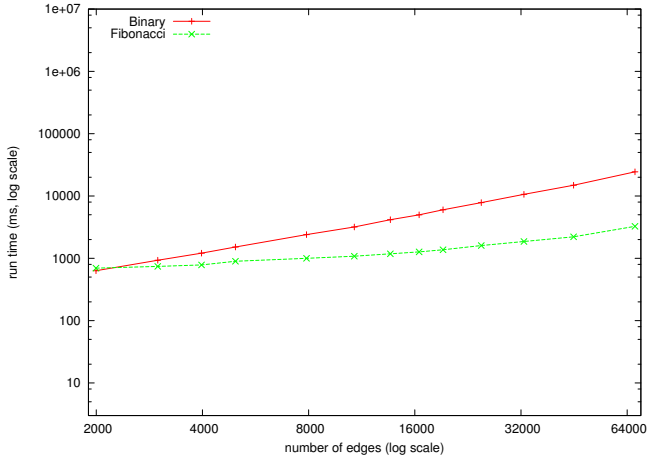


Figure 5.17: Run times of Johnson with a binary heap and with a Fibonacci heap on scale-free graphs with 1,000 vertices and increasing number of edges.

the heap) have a bigger relative contribution to the run time.

Based on the results over all benchmark sets, we conclude that although Johnson with a binary heap can help reducing the actual run time in sparse graphs, Johnson with a Fibonacci heap is overall the better choice if m can be large.

5.4 Related work

For dense, directed graphs with real weights, the state-of-the-art APSP algorithms run in $\mathcal{O}(n^3/\log n)$ time (Chan, 2005; Han, 2008). These represent a serious

improvement over the $\mathcal{O}(n^3)$ bound on Floyd–Warshall but do not profit from the fact that in most graphs that occur in practice, the number of edges m is significantly lower than n^2.

This profit is exactly what algorithms for sparse graphs aim to achieve. Recently, an improvement was published over the $\mathcal{O}(nm + n^2 \log n)$-time algorithm based on Johnson's (1977) and Fredman and Tarjan's (1987) work: an algorithm for sparse directed graphs running in $\mathcal{O}(nm + n^2 \log\log n)$ time (Pettie, 2004). In theory, this algorithm is thus faster than Johnson — in worst cases, for large graphs — when $m \in o(n \log n)$.* However, currently no implementation of this intricate algorithm exists, as confirmed through personal communication with Pettie in June, 2011. The upper bound of $\mathcal{O}(n^2 w_d)$ on the run time of Snowball is smaller than this established upper bound when the induced width is small (i.e. when $w_d \in o(\log\log n)$), and, of course, for chordal graphs and graphs of constant treewidth.

We are familiar with one earlier work to compute shortest paths by leveraging low treewidth. Chaudhuri and Zaroliagis (2000) present an algorithm for answering point-to-point shortest path queries with $\mathcal{O}(w_d^3 n \log n)$ preprocessing time and query time $\mathcal{O}(w_d^3)$. A direct extension of their results to APSP would imply a run time of $\mathcal{O}(n^2 w_d^3)$ on general graphs and $\mathcal{O}(nmw_d^2)$ on chordal graphs. Our result of computing APSP on general graphs in time $\mathcal{O}(n^2 w_d)$, and in $\mathcal{O}(nm)$ time on chordal graphs, is thus a strict improvement.

A large part of the state of the art in point-to-point shortest paths is focused on road networks, with positive arc weights. These studies have a strong focus on heuristics, ranging from goal-directed search and bi-directional search to using or creating some hierarchical structure (see for example Geisberger et al., 2008; Bauer et al., 2008). One of these hierarchical heuristics has some similarities to the idea of using chordal graphs. This heuristic is called *contraction*. The idea here is to distinguish core vertices, which may appear as end points of a path whose length is queried, from vertices that will not be used as such. The latter are then removed (bypassed) one by one, connecting their neighbours directly.

There are other restrictions on the input graphs that can be assumed, sometimes leading to algorithms with tighter bounds. For example, for *unweighted* chordal graphs, APSP lengths can be determined in $\mathcal{O}(n^2)$ time (Balachandhran and Rangan, 1996; Han et al., 1997) if all pairs at distance 2 are known. See (Dragan, 2005) for an overview and unification of such approaches. Considering only planar graphs, recent work shows that APSP can be found in time $\mathcal{O}(n^2 \log^2 n)$ (Klein et al., 2010), which is an improvement over Johnson in cases where $m \in \omega(n \log^2 n)$.

In the context of planning and scheduling, there is often a need for sequentially solving a number of very similar APSP problems, potentially allowing for a more efficient approach using dynamic algorithms, which form the topic of the next chapter. Even and Gazit (1985) provide a method where addition of a single edge can require $\mathcal{O}(n^2)$ time, and deletion $\mathcal{O}(n^4/m)$ *on average*. Thorup (2004)

*For an explanation of our use of the notation $x \in o(f(n))$, see the footnote on page 16.

and Demetrescu and Italiano (2006) later gave an alternative approach with an amortized run time of $\mathcal{O}(n^2(\log n + \log^2 \frac{n+m}{n}))$. Especially in the context of planning and scheduling, it is not essential that the shortest paths between *all* pairs of time points be maintained; the shortest paths between a selection of pairs often suffice. Above, we already mentioned the P^{3}C algorithm presented in the previous chapter for the single-shot case; in the next chapter, we describe an algorithm that incrementally maintains the property of partial path consistency on chordal graphs in time linear in the number of edges.

5.5 Conclusions and future work

In this chapter, we presented three algorithms for computing all-pairs shortest paths, with a run time bounded by (i) $\mathcal{O}(n^2)$ for graphs of constant treewidth, matching earlier results that also required $\mathcal{O}(n^2)$ (Chaudhuri and Zaroliagis, 2000); (ii) $\mathcal{O}(nm)$ on chordal graphs, improving over the earlier $\mathcal{O}(nmw_d^2)$; and (iii) $\mathcal{O}(n^2w_d)$ on general graphs, showing again an improvement over the previously tightest known bound of $\mathcal{O}(n^2w_d^3)$. In these bounds, w_d is the induced width of the ordering used; we have experimentally determined this to be bounded by the treewidth to the power 2.5 for most of our benchmarks.

These contributions are obtained by applying directed path consistency combined with known graph-theoretic techniques — such as vertex elimination and tree decomposition — to computing shortest paths. This supports the general idea that such techniques may help in solving graphically-representable combinatorial problems, but the main contribution of this chapter is more narrow, focusing on improving the state of the art for this single, but important problem of computing APSP.

From the results of our extensive experiments we can make recommendations as to which algorithm is best suited for which type of problems. Only for very small instances, Floyd–Warshall should be used; this is probably mostly thanks to its simplicity, yielding a very straightforward implementation with low overhead. Snowball can exploit the fact that an optimal elimination ordering can be efficiently found for chordal graphs, which makes it the most efficient algorithm for this class of input. From all our experiments on different types of general graphs, we conclude that Snowball consistently outperforms Johnson (and Floyd–Warshall), except when the induced width is very high. Our experiments also show that Snowball always outperforms both Chleq–APSP and Snowball–separators. Although the latter has a better bound on its run time, surprisingly its actual performance is worse than Snowball on all instances of our benchmark sets. This holds even for those instances for which Snowball–separators performs significantly fewer updates. Thus, we conclude that the additional bookkeeping required by Snowball–separators does not pay off.

Regarding these experiments, it must be noted that, although we did the utmost to obtain a fair comparison, a constant factor in the measurements depends in a significant way on the exact implementation details (e.g. whether a lookup-table or a heap is used), as is also put forward in earlier work on experimentally

comparing shortest path algorithms (Mondou et al., 1991; Cherkassky et al., 1996). In our own implementation a higher constant factor for the Snowball algorithms may be caused by adhering to the object-oriented paradigm, i.e. inheriting from the DPC and P^{3}C superclasses, and choosing to reuse code rather than inlining method calls. Nonetheless, we are confident that the general trends we identified hold independently of such details.

Note that strictly speaking, the algorithms introduced in this chapter compute all-pairs shortest *distances*. If one wants to actually trace shortest paths, the algorithms can be extended to keep track of the midpoint whenever the distance matrix is updated, just like one does for Floyd–Warshall. Then, for any pair of vertices, the actual shortest path in the graph can be traced in $\mathcal{O}(n)$ time.

In our current implementation of Snowball–separators, we used a priority queue to decide heuristically which clique tree node to visit next, giving precedence to nodes connected by a large separator to the part of the clique tree already visited. As noted before, we defer answering the question whether an optimal ordering can be found efficiently to future work. We remark however that using the minimum-degree heuristic for triangulation provides Snowball with a natural edge, delaying the processing of vertices where the number of iterations of the middle loop is small until k grows large.

Cherkassky and Goldberg (1999) compared several innovative algorithms for *single-source* shortest paths that give better efficiency than the standard Bellman–Ford algorithm in practice, while having the same worst-case bound of $\mathcal{O}(nm)$ on the run time. In future work, we will investigate if any of these clever improvements can also be exploited in Snowball.

Snowball–separators can be improved further in a way that does not influence the theoretical complexity but may yield better performance in practice. Iterating over V_{other} can be seen as a reverse traversal of the part of the clique tree visited before, starting at c's parent. Then, instead of always using the separator between the current clique node (containing k) and its parent for all previously visited vertices in V_{other}, we can keep track of the smallest separator encountered during this backward traversal for no extra asymptotic cost. Since it was shown in Table 5.1 that the *largest* minimal separator is often hardly smaller than the induced width, it might well pay off to search for smaller separators. However, preliminary experiments with this idea were discouraging.

Another possible improvement is suggested by the following observation on DPC. A variant of DPC can be proposed where edge directionality is taken into account: during iteration k, only those neighbours $i, j < k$ are considered for which there is a directed path $i \to k \to j$, resulting in the addition of the arc $i \to j$. This set of added arcs would often be much smaller than twice the number of edges added by the standard DPC algorithm, and while the graph produced by the directed variant would not be chordal, there is no impact on the correctness of Snowball.

Furthermore, we would also like to experimentally compare our algorithms to the recent algorithm by Pettie (2004) and the algorithms for graphs of constant treewidth by Chaudhuri and Zaroliagis (2000). In addition, we are interested in

more efficient ordering heuristics, or ordering heuristics with a guaranteed quality, to be able to give a guaranteed theoretical bound on general graphs. Another direction, especially interesting in the context of planning and scheduling, is to use the ideas presented here to design a faster algorithm for dynamic all-pairs shortest paths: maintaining shortest paths under edge deletions (or relaxations) and additions (or tightenings), which is among the topics discussed in the next chapter.

CHAPTER 6

Incrementally maintaining simple temporal information*

The previous two chapters described new, efficient methods to compute respectively the PPC and FPC normal forms for the STN, defined in Chapter 3. In this chapter, the Simple Temporal Query under scrutiny is INCREMENTAL MINIMALITY (STQ 7): how to maintain either of these normal forms when new temporal information becomes available. An answer to this query allows one to build up temporal knowledge from scratch, gradually refining it as new information becomes available, while ensuring that most of the queries listed in Chapter 2 can be answered very efficiently throughout the process.

In this chapter, we introduce two algorithms to maintain either normal form under addition of new constraints and/or tightening of existing constraints. This is called *incremental* temporal reasoning, since information is added to the temporal network; the algorithms are therefore named IFPC and IPPC . In contrast, decremental reasoning refers to re-establishing the normal form of an STN when previously postulated information is retracted; it is outside the scope of this chapter, but we briefly touch upon the matter again in the final section.

This chapter is structured as follows. The first two sections describe the new algorithms, respectively named IFPC and IPPC, for incrementally maintaining full and partial path consistency when a new constraint becomes available or an existing constraint is tightened. Whereas IFPC is simple to describe and its correctness is easily grasped, we will require several lemmas and theorems for IPPC, especially to attain a competitive run-time bound.

Next, we survey some extant approaches for incremental full path consistency (alias all-pairs shortest paths) in Section 6.3. To the best of our knowledge, IPPC is the first algorithm of its class and no prior art exists in this area.

*This chapter includes work from a workshop paper (Planken, 2008b) and a conference paper (Planken et al., 2010b).

Algorithm 6.1: Incremental full path consistency (IFPC)

Input: An FPC STN $\mathcal{S} = \langle V, E \rangle$ and a new constraint with weight w_{ab}.
Output: CONSISTENT if w_{ab} has been added to $\mathcal{S}$, which is again FPC; INCONSISTENT otherwise.

```
1  if w_ab + ω_ba < 0 then return INCONSISTENT
2  if w_ab ≥ ω_ab then return CONSISTENT
3  V_out ← ∅; V_in ← ∅
4  forall v ∈ V such that v ≠ a, b do
5      if ω_vb > ω_va + w_ab then
6          ω_vb ← ω_va + w_ab
7          V_out ← V_out ∪ {v}
8      end
9      if ω_av > w_ab + ω_bv then
10         ω_av ← w_ab + ω_bv
11         V_in ← V_in ∪ {v}
12     end
13 end
14 forall (u, v) ∈ V_out × V_in such that u ≠ v do
15     if ω_uv > ω_ub + ω_bv then          // note:  ω_ub = ω_ua + w_ab
16         ω_uv ← ω_ub + ω_bv
17     end
18 end
19 ω_ab ← w_ab
20 return CONSISTENT
```

Then, in Section 6.4, as in the previous chapters, we put our new algorithms to the test and empirically compare their performance against their competitors. However, in contrast to the experiments in the previous chapters, this empirical comparison also includes integration of IFPC and IPPC in a deployed solver for job-shop scheduling problems, which allows us to gauge their effect on the performance of said solver.

Finally, Section 6.5 describes extensions of our work in two directions, and Section 6.6 concludes the chapter with a discussion.

6.1 Incremental full path consistency

In Chapter 3, it was seen that for the STN, the property of full path consistency (FPC) is equivalent to minimality, and that it corresponds to computing all-pairs shortest paths on its distance graph. Thus, maintain the minimal network of an STN in the face of new information becoming available, one is looking for a method to enforce FPC, or maintain shortest paths, incrementally.

The first reference in the literature to a method for incremental full path consistency was by Tsamardinos and Pollack (2003), who cited work by Mohr and Henderson (1986). However, the latter only presented a single-shot path

consistency algorithm for general (discrete) constraint satisfaction problems. Nevertheless, there is a straightforward approach that satisfies the $\mathcal{O}(n^2)$-time bound stated by Tsamardinos and Pollack: one can simply check, for all pairs of vertices, whether the current constraint edge connecting them may be tightened by following a path through the tightened edge (a, b) with weight w_{ab}. This approach can be stated concisely in a single line:

$$\forall u, v \in V : \omega_{uv} \leftarrow \min\{\omega_{uv},\ \omega_{ua} + w_{ab} + \omega_{bv}\}$$

In effect, this is the sequential equivalent of Algorithm 2.3 from page 26, which was used to prove that INCREMENTAL MINIMALITY is in NC^1. Given the fact that all edges were already labelled by the shortest paths, it clearly does the job. This approach produces a complete constraint graph taking up $\Theta(n^2)$ space, so the constraint weights can be stored in a two-dimensional array. Every iteration of the loop then requires constant time, from which the desired bound follows. Since there exist cases where $\Omega(n^2)$ constraint edges must updated, one may be tempted to take the quadratic bound as given and look no further.

However, we now present a different approach, simply called IFPC, that is only slightly more involved but yields significant efficiency improvements when only part of the network needs to be updated. To the best of our knowledge, we were the first to propose and analyse it (Planken, 2008b,a). It is included as Algorithm 6.1 and operates as follows. First, the algorithm performs a constant-time check for two trivial cases. Then, it determines for each vertex $v \in V \setminus \{a, b\}$ whether either ω_{vb} or ω_{av} needs to be updated, in $\Theta(n)$ time; if so, v is inserted into the set V_{out} or V_{in}, respectively (possibly in both). Finally, the algorithm iterates over each pair $(u, v) \in V_{out} \times V_{in}$ and checks for a new shortest path from u to v.

To see that this approach is sound, consider two time points u and v and assume there exists a new shortest path (u, a, b, v) after adding w_{ab}. But then, (u, a, b) and (a, b, v) must also be new shortest path from u to b and from a to v, respectively. In the first loop, ω_{ub} and ω_{av} are updated to reflect this, and u and v are added to V_{out} and V_{in}, respectively. Then, ω_{uv} will be set correctly in the second loop.

We can express the efficiency savings imparted by this approach in a theoretical bound on IFPC's time complexity. Let $V^* = V_{out} \cup V_{in}$ be the set of vertices participating in any updated edge weight, and let $n^* = |V^*|$. This yields the following result.

Theorem 6.1. *IFPC incrementally enforces full path consistency in $\mathcal{O}(n + (n^*)^2)$ time.*

In Section 6.3, we discuss other approaches for this problem that yield different theoretical bounds. However, we first introduce an algorithm for a different, but related problem.

Algorithm 6.2: Incremental partial path consistency (IPPC)

Input: A PPC STN $\mathcal{S} = \langle V, E \rangle$ and a new constraint with weight w_{ab}, for some $\{a, b\} \in E$
Output: CONSISTENT if w_{ab} has been added to $\mathcal{S}$, which is again PPC; INCONSISTENT otherwise.

```
1  if w_ab + ω_ba < 0 then return INCONSISTENT                          *
2  if w_ab ≥ ω_ab then return CONSISTENT                                *
3  (re-)initialise D_{a←}[·] and D_{b→}[·] to ∞                         *
4  (re-)initialise COUNT[·] to 0 and TAGGED[·] to FALSE
5  D_{a←}[a] ← 0; D_{b→}[b] ← 0                                         *
6  Tag(a); Tag(b)
7  while ∃u ∈ V : COUNT[u] = max{COUNT[v] | v ∈ V} ∧ ¬TAGGED[u] do
8  |  Tag(u)
9  end
10 return CONSISTENT
```

6.2 Incremental partial path consistency

In this section, we present an algorithm, called IPPC, that maintains the PPC normal form for an STN under constraint tightenings, i.e. when the weight of a constraint edge is reduced. Its basic version runs in $\mathcal{O}(m_c)$ time; later, we present a relatively small modification that additionally guarantees a bound of $\mathcal{O}(n_c^* \delta_c)$. The latter bound is important if the affected part of the network is small. Recall that m_c and δ_c denote the number of edges in the chordal graph and the chordal graph's degree, respectively; n_c^* is the number of endpoints of updated edges in the chordal graph. To compare this to the parameter n^* introduced above, note that for every update in the PPC normal form there must also be an update in the FPC STN, while the converse is not true; so $n_c^* \leq n^*$. The algorithm we present runs within the $\mathcal{O}(m_c)$ space occupied by the STN.

The basis of IPPC is formed by maximum cardinality search (MCS, included as Algorithm 3.1 on page 38). Recall from Chapter 3 that MCS can be used to visit the vertices in a chordal graph in a principled order, called a simplicial construction ordering. The fact that IPPC also visits the vertices in this order is pivotal in when we prove its correctness and time complexity below. Recall from Chapters 4 and 5 that P^{3}C and Snowball also use such an ordering.

Pseudo-code for our algorithm IPPC is given as Algorithm 6.2; if the lines marked with an asterisk at the right-hand margin are left out, we obtain the MCS algorithm. The algorithm takes as input an STN $\mathcal{S}$ that is already PPC, together with a new constraint weight w_{ab} for some edge $\{a, b\} \in E$. Note that this edge must already be present in the graph; the correctness of the algorithm depends on this requirement. Moreover, a new edge may destroy the property of chordality by introducing a hole. Since this is the first algorithm for incremental PPC, we accept this shortcoming, delegating the problem of incremental triangulation to

procedure Tag($v \in V$)

```
1  TAGGED[v] ← TRUE
2  forall u ∈ 𝒩(v) do
3  |  if ¬ TAGGED[u] then
4  |  |  COUNT[u] ← COUNT[u] + 1
5  |  |  D_{a←}[u] ← min{D_{a←}[u], ω_uv + D_{a←}[v]}                    *
6  |  |  D_{b→}[u] ← min{D_{b→}[u], D_{b→}[v] + ω_vu}                    *
7  |  else
8  |  |  ω_uv ← min{ω_uv, D_{a←}[u] + w_ab + D_{b→}[v]}                  *
9  |  |  ω_vu ← min{ω_vu, D_{a←}[v] + w_ab + D_{b→}[u]}                  *
10 |  end
11 end
12
```

future research.

As was the case for IFPC, if the new weight w_{ab} is smaller than the previously minimal weight ω_{ab}, some other constraints may need to be tightened as well. In the course of the algorithm, we therefore compute for every vertex v the length of the shortest path to a, as well as the length of the shortest path from b, which are maintained in arrays $D_{a\leftarrow}[v]$ and $D_{b\rightarrow}[v]$, respectively. Note that exactly the operations involving edge weights and distance arrays are additions to the MCS algorithm.

The procedure Tag(v) is also borrowed from MCS, although it is now presented separately from the main algorithm. Our addition here is to update ω_{uv} for neighbours u of v, denoted by $\mathcal{N}(v)$, that have already been visited if there is a shorter path from u via a and b to v (and similarly update ω_{vu} if applicable). As in MCS, every vertex v is tagged exactly once; in particular, ω_{ab} itself is updated when calling Tag(b). The arrays COUNT[$\cdot$] and TAGGED[$\cdot$] are used exactly as in MCS, ensuring that the vertices are visited in a simplicial construction ordering (Definition 3.4 on page 37).

We are now ready to prove the theoretical properties of our algorithm. We first state two supporting lemmas.

Lemma 6.2. *The order in which Algorithm 6.2 calls* Tag(v) *on vertices $v \in V$ is a simplicial construction ordering of $\mathcal{S}$ (and its reverse is a simplicial elimination ordering).*

Proof. The order is solely determined by the arrays COUNT[$\cdot$] and TAGGED[$\cdot$]; the algorithm uses them in exactly the same way as MCS does. □

Lemma 6.3. *Upon entering* Tag(v), *for $v \neq a$, $D_{a\leftarrow}[v]$ and $D_{b\rightarrow}[v]$ are equal to the lengths of the shortest paths from v to a and from b to v, respectively.*

Proof. $D_{a\leftarrow}[v]$ and $D_{b\rightarrow}[v]$ can never be lower than the length of a shortest path, because they are both initialised to ∞ for every vertex in line 3 of IPPC and are

only reduced (in lines 5–6 of Tag(v)) when there is a path from v to a or from b to v. To show that they cannot be higher, we use a proof by induction.

After a, the first vertex to be tagged is b. The reader can verify that the assignment $D_{a\leftarrow}[b] \leftarrow \omega_{ba}$ has been done during the call to Tag(a). $D_{b\rightarrow}[b] = 0$ by initialisation, so the lemma holds for this base case. Further, during the call to Tag(b), the assignment $D_{b\rightarrow}[a] \leftarrow \omega_{ba}$ is made, and again by initialisation, $D_{a\leftarrow}[a] = 0$. Thus we have verified the base case of the induction hypothesis that for every tagged vertex $v \in \{a, b\}$, $D_{a\leftarrow}[v]$ and $D_{b\rightarrow}[v]$ are set correctly.

Our induction hypothesis is that all vertices already tagged have their distances correctly set. Now consider a call to Tag(v) with $v \neq a, b$ and assume, for the purpose of reaching a contradiction, that $D_{a\leftarrow}[v]$ is not correctly set upon entering the procedure. This implies that there must exist some shortest path $\pi = (v, u_1, \ldots, u_k = a)$ with total weight $\omega_\pi < D_{a\leftarrow}[v]$. Let k' be the index of the first vertex on π which has already been tagged; in particular, by the induction hypothesis, $D_{a\leftarrow}[u_{k'}]$ is correctly set. Since we know that at least a has already been tagged upon entering Tag(v), we have $k' \leq k$.

Consider the path $\pi' = (v, u_1, \ldots, u_{k'})$ which must have weight $\omega_{\pi'} = \omega_\pi - D_{a\leftarrow}[u'_k]$. We consider two cases: $k' = 1$ and $k' > 1$. For $k' = 1$, π' becomes just the path (v, u_1) and $\omega_{\pi'} = \omega_{vu_1}$. Then, by line 5 of the call to Tag(u_1), we have that $D_{a\leftarrow}[v] \leq \omega_{vu_1} + D_{a\leftarrow}[u_1] = \omega_\pi$, which contradicts $\omega_\pi < D_{a\leftarrow}[v]$.

If $k' > 1$, we know that for $1 \leq i < k'$, the vertices u_i have not yet been tagged; see Figure 6.1a. By Lemma 6.2, all of these u_i appear before both v and $u_{k'}$ in a simplicial elimination ordering (see Definition 3.4 on page 37). But this implies that there must exist an arc $(v, u_{k'})$ by the fact that each of the intermediate vertices can be eliminated in turn, connecting their neighbours. We can thus further shorten π' to $\pi'' = (v, u_{k'})$, and the reasoning above, for $k' = 1$, applies.

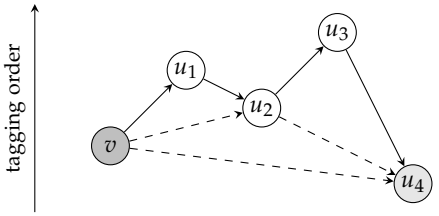
This proves that $D_{a\leftarrow}[v]$ is correctly set. The reasoning for $D_{b\rightarrow}[v]$ is analogous, with the direction of all arcs reversed. □

We can now prove the correctness of IPPC, as well as time and space bounds of $\mathcal{O}(m_c)$.

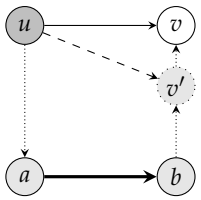
Theorem 6.4. *Algorithm 6.2 correctly re-enforces PPC (or decides inconsistency) in $\mathcal{O}(m_c)$ time and space.*

Proof. If some arc (u, v) must be updated due to tightening (a, b), its new weight is the length of some shortest path $(u, \ldots, a, b, \ldots, v)$. Without loss of generality, assume that u is tagged before v. Then, when calling Tag(v), by Lemma 6.3, $D_{a\leftarrow}[u]$ and $D_{b\rightarrow}[v]$ are correctly set, and in line 8 of Tag(v), ω_{uv} is correctly updated.

Regarding the run time, note that for all $v \in V$, Tag(v) is called at most once, exactly as MCS does. This procedure only accesses edge weights when the algorithm iterates over v's neighbours in the adjacency list. Therefore, all operations in Tag(v) require amortised constant time per edge. All operations and the loop in the main algorithm can also be performed in (at most) linear time; therefore, the run time can be bounded by $\mathcal{O}(m_c)$.



(a) Situation during $\mathsf{Tag}(v)$ used in Lemma 6.3. Depicted is the path $\pi' = (v, u_1, u_2, u_3, u_4)$, so $k' = 4$. Since $d = (u_3, u_1, u_2, v, u_4)$ is a simplicial elimination ordering of the path, the dashed arcs must exist.



(b) Situation during $\mathsf{Tag}(u)$ used in Lemma 6.5. If ω_{uv} must change, then so must $\omega_{uv'}$.

Figure 6.1: Visual support for the proofs of Lemmas 6.3 and 6.5

The algorithm maintains four arrays of length $\mathcal{O}(n)$, as well as an adjacency-list-based graph data structure of size $\mathcal{O}(m_c)$, containing all edge weights ω_{uv}. Space can therefore be bounded by $\mathcal{O}(m_c)$. □

Observations

The bound of $\mathcal{O}(m_c)$ on IPPC's run time is good if we consider that it is tight in general: in the worst case, all m_c edges need to be updated. However, the algorithm as stated always processes the entire graph, even if only a small part of it is affected by the update. If it could be changed to only process the affected part, like IFPC does, its efficiency might be improved further.

Consider the induced subnetwork $\mathcal{S}_{V_c^*}$ of $\mathcal{S}$, where $V_c^* \subseteq V$ consists of all endpoints of edges whose weight is to be updated after tightening $\{a, b\} \in E$. Note that V_c^* is a subset of V^*, the set of edges to change when re-enforcing *full* path consistency. In the following, we show that the algorithm only needs to run on $\mathcal{S}_{V_c^*}$ and give theoretical grounding that suggests a small modification to IPPC for determining V_c^* "on the fly". First, we show that if some edge far away from the tightening must be updated, there is also an update one step closer.

Lemma 6.5. *Consider the processing of the tightening of an edge $\{a, b\}$ and let $\{u, v\} \neq \{a, b\}$ be an edge such that ω_{uv} (or ω_{vu}) must be updated, so u and v are the endpoints of a new shortest path π through a and b.*

If $\mathsf{Tag}(u)$ is called before $\mathsf{Tag}(v)$, there must exist an edge $\{u, v'\}$ such that $\omega_{uv'}$ (or $\omega_{v'u}$) must also be updated, because of the existence of a new shortest path π' from u to v' (or from u' to v) through a and b. Moreover, π' is just π with one edge removed from its end (or start).

Proof. Without loss of generality, assume the new shortest path is in the direction $u \to v$. First, if there are multiple such shortest paths, let π have minimum length (number of vertices involved). Together with the edge $\{u, v\}$ itself, π forms a

cycle. If the cycle is of size 3, then $u = a$ and our claim follows by choosing $v' = b$. (We cannot have $v = b$, because u was tagged before v, yielding $u = a$ and contradicting $\{u, v\} \neq \{a, b\}$.) Otherwise, the cycle $(u, \ldots, a, b, \ldots, v, u)$ is at least of size 4 and must have a chord. See Figure 6.1b.

We know that the algorithm tags vertices a, b, and u before v. By Lemma 6.2, vertices are tagged in a simplicial construction ordering. Vertex v must in fact be the last tagged in the cycle and thus be simplicial with respect to the cycle: if some other vertex in the cycle were last, it must lie on either of the two subpaths $(u, \ldots, a)$ and $(b, \ldots, v)$ of π, yielding a chord on these subpaths and contradicting the minimality of π.

Therefore, v itself is simplicial and there is a chord $\{u, v'\}$ to the penultimate vertex v' on the subpath $(b, \ldots, v)$ of π; note that v' may equal b. We thus obtain $\pi' = (u, \ldots, a, b, \ldots, v')$ as a new shortest path from u to v' through a and b. □

We now show that because the network is chordal and already PPC except for the new edge, all shortest paths are contained within $\mathcal{S}_{V_c^*}$. Therefore, the remainder of the network can safely be ignored.

Proposition 6.6. *For any edge that must be updated due to an edge tightening, the new shortest path can be traced in the subnetwork $\mathcal{S}_{V_c^*}$ induced by V_c^*.*

Proof. Repeated application of Lemma 6.5. □

Another consequence of Lemma 6.5 is that membership of a vertex is already implicitly determined in the IPPC algorithm.

Lemma 6.7. *If some vertex $u \neq a$ is in V_c^*, at least one edge weight is updated when* Tag(u) *is called (lines 8 and 9).*

Proof. For $u = b$, this result is trivial. Assume that $u \neq a, b$; since $u \in V_c^*$, there exists a weight ω_{uv} (or ω_{vu}) that must be updated. If v was tagged before u, this update takes place during Tag(u).

Otherwise, we can repeatedly apply Lemma 6.5 until we obtain an edge $\{u, v'\}$ so that v' was tagged before u and $\omega_{uv'}$ (or $\omega_{v'u}$) is tightened during the call to to Tag(u). □

Using these results, we propose a change to the algorithm to ensure that exactly all vertices in V_c^* are completely visited, and as little effort as possible is wasted on vertices outside V_c^*.

Improving efficiency

The membership of v in V_c^* is implicitly determined in the call to the procedure Tag(v). In order to exploit this knowledge, we replace it by a new procedure Tag–improved(v). The array COUNT$[\cdot]$ is replaced by an array of vertex lists LIST$[\cdot]$, initialised to contain an empty list for each vertex. These lists contain for each vertex v the neighbouring vertices in V_c^* that have already been tagged. This

procedure Tag–improved($v \in V$)

1 $\textsc{tagged}[v] \leftarrow \textsc{true}$
2 **forall** $u \in \textsc{list}[v]$ **do**
3 $\quad \omega_{uv} \leftarrow \min\{\omega_{uv}, D_{a\leftarrow}[u] + w_{ab} + D_{b\rightarrow}[v]\}$
4 $\quad \omega_{vu} \leftarrow \min\{\omega_{vu}, D_{a\leftarrow}[v] + w_{ab} + D_{b\rightarrow}[u]\}$
5 **end**
6 **if** any updates took place, or $v = a$ **then** // $v \in V_c^*$ guaranteed
7 $\quad$ **forall** $u \in \mathcal{N}(v)$ **such that** $\neg\,\textsc{tagged}[u]$ **do**
8 $\quad\quad \textsc{list}[u] \leftarrow \textsc{list}[u] \cup \{v\}$
9 $\quad\quad D_{a\leftarrow}[u] \leftarrow \min\{D_{a\leftarrow}[u], \omega_{uv} + D_{a\leftarrow}[v]\}$
10 $\quad\quad D_{b\rightarrow}[u] \leftarrow \min\{D_{b\rightarrow}[u], D_{b\rightarrow}[v] + \omega_{vu}\}$
11 $\quad$ **end**
12 **end**

reduced set of neighbours is enough to determine for v itself whether edges connecting it to those neighbours must be updated.

Only if this is the case — or if $v = a$, which is trivially in V_c^* — the algorithm iterates over each neighbour w of v, inserting v into $\textsc{list}[w]$. In the main algorithm, the length of these lists assumes the role of $\textsc{count}[\cdot]$.

A final improvement to the algorithm is to only call Tag–improved(v) when the length of $\textsc{list}[v]$ is at least two. It is easily verified that otherwise the vertex does not appear in any cycle with the updated edge, so $v \notin V_c^*$ and no checking is necessary. This implies that as soon as $\max\{|\textsc{list}[v]| \mid v \in V\} < 2$, the main algorithm can terminate its while loop.

Recalling that n_c^* and δ_c are the number of endpoints of updated edges in the chordal graph and its degree, respectively, let us now prove the correctness and run time of the modified IPPC algorithm.

Theorem 6.8. *With the modifications just discussed, Algorithm 6.2 correctly re-enforces PPC or decides inconsistency in $\mathcal{O}(m_c)$ and $\mathcal{O}(n_c^*\delta_c)$ time.*

Proof. By Proposition 6.6, if the original IPPC is run on $\mathcal{S}_{V_c^*}$, it still re-enforces PPC on the entire STN $\mathcal{S}$. We show that with respect to computation of the edge weights, running the modified algorithm on $\mathcal{S}$ is equivalent to running the original algorithm on $\mathcal{S}_{V_c^*}$. It is easily verified that if $V_c^* = V$, the changes to the algorithm have no impact on the edge weights ω_{uv} computed. Otherwise, let v be the first vertex from $V \setminus V_c^*$ to be tagged. Up to this point, the operation of the original and the modified algorithm were equivalent with respect to edge updates. Since by Lemma 6.7, Tag–improved(v) correctly determines $v \notin V_c^*$, its only effect is to set $\textsc{tagged}[v] \leftarrow \textsc{true}$, causing it to be ignored for the remainder of the algorithm, and thus maintaining the equivalence with the original algorithm.

Regarding the run time, the bound of $\mathcal{O}(m_c)$ already proved in Theorem 6.4 still applies, because all operations in Tag–improved(v) take constant amortised time per edge. The bound of $\mathcal{O}(n_c^*\delta_c)$ claimed here can be derived as follows.

Only for the n_c^* vertices in V_c^*, all neighbours are visited. Each vertex has at most δ_c neighbours in the chordal graph, and each visit takes only constant time.

These visits can also be used to amortise the cost of calling Tag–improved(u) when $u \notin V_c^*$, as follows. Whenever u is encountered as an untagged neighbour of any $v \in V_c^*$ and v is added to LIST$[u]$ in constant time, one can already account for the time required for one iteration step over LIST$[u]$ during Tag–improved(u); recall also that each vertex is tagged at most once.

Now, we only have a detail in the main algorithm left to deal with. Naive re-initialisation of the arrays in lines 3 and 4 requires $\Theta(n)$ time, which does not fit within the run-time bound we are trying to derive. However, our approach is to initialise these arrays only once, when first reading the STN. Then, whenever the entry for some vertex v in one of these arrays is set for the first time during an edge update, v is added to a list. Before the algorithm returns, for all vertices in this list, the corresponding array entries are reset, thus cleaning the slate for the next invocation of the algorithm. The cost of this cleaning operation can be amortised over the original array accesses and does not impact the asymptotic bound. Thus, we conclude that the run time is also bounded by $\mathcal{O}(n_c^* \delta_c)$. □

In Section 6.4, we put IFPC and IPPC to the test in an empirical evaluation. First, however, we describe approaches already available in the literature.

6.3 Extant approaches

To the best of our knowledge, we were the first to propose an incremental algorithm for enforcing PPC on STN instances (Planken et al., 2010b). The case of tightening edges and re-computing full path consistency (or all-pairs shortest paths), however, has been much studied. In this section, we first discuss EGR, the seminal algorithm of this type, independently presented in two publications from 1985.* Then, we investigate some of the later work that yields improvements for special cases.

The seminal algorithm

Among the first to present algorithms for this problem were Even and Gazit (1985) and Rohnert (1985). Both first note the unavoidable general $\mathcal{O}(n^2)$-time bound, and mention the simple one-line approach from page 127. Both of them then present essentially the same algorithm, which we dub EGR, by the authors' initials. Here, we discuss EGR and give an analysis of its time complexity. The pseudocode we present as Algorithm 6.3 is based on the formulation by Even and Gazit.

EGR first constructs, in lines 7–19, a *shortest path tree* T_a rooted at b, the endpoint of the tightened arc (a, b). In this tree, the parent u of each vertex v is set so that $(a, b, \ldots, u, v)$ is a new shortest path from a to v, using the tightened

*It must be noted that our IFPC algorithm was proposed independently of EGR, even though it was published quite some time later.

Algorithm 6.3: The EGR algorithm (Even and Gazit, 1985; Rohnert, 1985)

Input: An FPC STN $\mathcal{S} = \langle V, E \rangle$ and a new constraint with weight w_{ab}.
Output: CONSISTENT if w_{ab} has been added to $\mathcal{S}$, which is again FPC; INCONSISTENT otherwise.

```
1  if w_ab + ω_ba < 0 then return INCONSISTENT
2  if w_ab ≥ ω_ab then return CONSISTENT
3  ω_ab ← w_ab
4  Q ← {b}
5  unmark all vertices
6  T_a ← root vertex b
7  while Q ≠ ∅ do
8  |  dequeue u from Q
9  |  foreach v ∈ V such that {u,v} ∈ E ∧ v is unmarked do
10 |  |  if ω_bv = ω_bu + w_uv then
   |  |  |  // v is on a shortest path from b
11 |  |  |  mark vertex v
12 |  |  |  if ω_av > ω_au + w_uv then
   |  |  |  |  // u is on a new shortest path from a
13 |  |  |  |  ω_av ← ω_au + w_uv
14 |  |  |  |  add {u,v} to T_a
15 |  |  |  |  enqueue v in Q
16 |  |  |  end
17 |  |  end
18 |  end
19 end
20 unmark all vertices; mark vertex a
21 foreach u ∈ V such that {u,a} ∈ E ∧ u is unmarked do
22 |  if ω_ua = w_ua then                    // {u,a} is a shortest path to a
23 |  |  TWS(a, b, u, T_a)
24 |  end
25 end
26 return CONSISTENT
```

procedure TWS(a, b, u, T_a)

```
1  mark vertex u
2  if ω_ub > ω_ua + w_ab then          // u is on a new shortest path to b
3      ω_ub ← ω_ua + w_ab
4      Q ← {b}
5      Pruned ← ∅
6      while Q ≠ ∅ do
7          dequeue x from Q
8          foreach {x, y} ∈ T_a do
9              if ω_uy > ω_ux + w_xy then
10                 ω_uy ← ω_ux + w_xy
11                 enqueue y in Q
12             else
13                 remove {x, y} from T_a
14                 add {x, y} to Pruned
15             end
16         end
17     end
18     foreach x ∈ V such that {x, u} ∈ E ∧ x is unmarked do
19         if ω_xa = w_xu + ω_ua then      // x is on a shortest path to a
20             TWS(a, b, x, T_a)
21         end
22     end
23     T_a ← T_a ∪ Pruned
24 end
```

constraint and considering the *original* constraint weights (denoted w_{uv}). Note that many such trees may exist. Thus, the set of non-root vertices in T_a is exactly V_{out} from our IFPC algorithm. The tree contains $\mathcal{O}(n^*)$ vertices and arcs and is constructed in $\mathcal{O}(n^*\delta)$ time, where δ is the degree of the graph. During its construction, shortest distances ω_{av} are updated for all $v \in V_{out}$, in line 13.

Next, the algorithm performs a recursive two-way search (TWS) in T_a and the implicit tree T_b consisting of all new shortest paths *ending* in b. Again, for each vertex u encountered, it is first checked in constant time whether u is on a new shortest path $(u, \ldots, a, b)$ to b. If so, in lines 6–17 of the TWS procedure, T_a is recursively traversed in $\mathcal{O}(n^*)$ time to check whether there are also new shortest paths to descendants of b, after which, in lines 18–22, recursion continues with the u's children in T_b.

As in the construction of T_a, the algorithm encounters at most $\mathcal{O}(n^*\delta)$ dead ends when traversing the implicit tree T_b. Traversal of T_a is only done for $u \in V_{in}$, where V_{in} is again as we defined it for IFPC . Consequently, we can already derive a time complexity of $\mathcal{O}(n^*\delta + (n^*)^2)$ for EGR . However, the algorithm also exploits the fact that T_a does not necessarily have to be traversed completely for all $u \in V_{in}$. If for some $(u, y) \in V_{in} \times V_{out}$, there is no change to ω_{uy}, then

the subtree rooted at y is pruned from T_a, in line 13 of TWS.* This gives rise to the time bound of $\mathcal{O}(m^*\delta)$ stated by Even and Gazit, where m^* is the number of constraint weights changed.

Other work

We briefly review some other work on incrementally computing shortest paths.

Ausiello, Italiano, Marchetti-Spaccamela, and Nanni (1991) attack the problem for graphs with positive integer constraint weights in a limited range. For constraint weights from the set $\{1, \ldots, C\}$, their approach (which we refer to as AIMN here) takes $\mathcal{O}(Cn^3 \log(nC))$ time for the maximum number of $\mathcal{O}(n^2)$ constraint insertions and $\mathcal{O}(Cn^2)$ constraint weight decreases. They prove a lower bound of $\Omega(Cn^3)$ time for such a sequence of operations, so AIMN's complexity is only a logarithmic factor away from the best possible. Note that these are the only time bounds stated for their approach.

AIMN's improvement with respect to EGR is its clever use of data structures. It exploits the $\mathcal{O}(n^2)$ space available by maintaining all $2n$ shortest path trees in both directions, for each vertex in the graph. When processing an addition or tightening of an arc, these data structures are designed so that they can be kept up to date in $\mathcal{O}(m^*)$ time. With this improvement, the shortest path trees need not be constructed from scratch for each update, which enables achieving the aggregate time bounds stated above.

Though it is clear that AIMN's run time still satisfies the $\mathcal{O}(m^*\delta)$ bound proven for EGR, this is not mentioned anywhere in the paper. In fact, since all shortest path trees are already maintained in memory and each is of size $\mathcal{O}(n)$, the number of dead ends reached, while still bounded by $\mathcal{O}(n^*\delta)$, is also limited to $\mathcal{O}(n)$. This means that we can state another bound on AIMN's time complexity.

Proposition 6.9. *AIMN runs in $\mathcal{O}(n + (n^*)^2)$ time.*

Recall that IFPC also runs within this time bound.

Although Ausiello et al. only consider graphs with positive integer constraint weights, Oddi et al. (2012) mention that they have adapted AIMN for use on the general problem in their job-shop solver (see the case study in Section 6.4). Certainly, the data structures can be used without any problem; only the $\mathcal{O}(Cn^3 \log(nC))$ bound is invalidated, but the other bounds hold.

A more recent publication is Hunsberger's (2008), which presents a complete temporal constraint management system to support planning, scheduling, and real-time execution, based on an STN in which so-called *rigid components* are collapsed. These rigid components are subgraphs where for every edge the weight in one direction equals the negative of the other. Attention is paid to practical space complexity as well as time complexity. The temporal network

*Strictly speaking, T_a falls apart into a forest when edges are removed in line 13. TWS only considers the tree rooted at b. T_a is assumed to be passed by reference, so only a single specimen needs to be kept in memory and no copying is required. Because it is passed by reference, the pruned edges must be added again in line 23.

Table 6.1: Worst-case time bounds on the incremental algorithms

IFPC	$\mathcal{O}(n + (n^*)^2)$
IPPC	$\mathcal{O}(m_c)$ and $\mathcal{O}(n_c^* \delta_c)$
EGR	$\mathcal{O}(n^* \delta + (n^*)^2)$ and $\mathcal{O}(m^* \delta)$
AIMN	$\mathcal{O}(n^* \delta + (n^*)^2)$ and $\mathcal{O}(m^* \delta)$ and $\mathcal{O}(n + (n^*)^2)$

separates the temporal reference time point z into a pair (z_{in}, z_{out}) and includes a 'now' time point; together, these eliminate the need for certain common forms of constraint propagation, especially when considering the passage of time.

The incremental algorithm used is EGR . Although the improvements proposed by Hunsberger do not change its theoretical upper bound, they may improve efficiency in practice. On the other hand, there is also some overhead involved, e.g. in identifying and collapsing rigid components and in the additional arithmetic relating to the split temporal reference point.

Note that Hunsberger's improvements can be applied to any incremental algorithm, including IFPC and IPPC . In this text we therefore choose to compare the basic versions of the algorithms. In specific cases, one may then choose to also implement any subset of Hunsberger's proposals.

Comparison

We can now list the theoretical time complexities of the algorithms presented so far. Recall that we have the bounds listed in Table 6.1. First, note that all of them fit within the trivial $\mathcal{O}(n^2)$ upper bound. Comparing the bounds is difficult because $n_c^* \leq n^* \leq m^* \leq m_c$ and $\delta \leq \delta_c$; also, all of these parameters may be more than a constant factor apart, sometimes even up to a factor of n. So this comparison does not yield a clear overall winner. We can state some strengths and weaknesses, however.

- Among IFPC's weaknesses is that it iterates over all vertices in the graph, even if only very few edges need to be updated. Also, its iteration over all pairs from $V_{in} \times V_{out}$ is rather coarse-grained.

  Its strong point is that the linear loop requires only six array accesses, two additions and two comparisons for vertices not in V^*, and the other parts of its pseudocode are likewise streamlined, making for a very small constant factor hidden in its asymptotic bound.

- IPPC has a big advantage in that it only needs to maintain m_c arcs instead of all $\Omega(n^2)$ distances in a complete graph; however, for this advantage to truly come into play, the difference between m_c and n^2 must be big enough.

  The main shortcoming of IPPC is that it currently does not support addition of new constraint edges, allowing only existing constraints to be tightened.

- EGR takes much care in visiting only affected parts of the graph, which gives it good theoretical bounds. However, building or keeping track of

Table 6.2: Properties of the benchmark sets

Type	n	m	m_c	δ	δ_c
Diamonds	50–379	52–379	82–575	4	10–26
Job shop	5–241	8–3960	9–8164	4–240	4–240
HTN					
– size	32–3358	40–5204	66–7180	6–16	11–128
– landmarks	506–752	568–2290	1009–13741	6–14	11–185

shortest path trees does inflict a higher constant factor on its run time when compared to the very simple pseudocode of IFPC .

- These arguments apply even more strongly to AIMN, whose bounds however indicate that it should be at least on a par with IFPC in the long run.

To find out how these algorithms hold up in practice, we empirically compare their performance in several settings.

6.4 Experimental evaluation

In this section, we empirically evaluate the performance of the new algorithms, in two ways. The first way is to perform an exploratory experimental study on some of the benchmarks also used in the preceding chapters. Then, we discuss the integration of IFPC and IPPC into a deployed solver for job-shop scheduling problems.

Exploratory empirical comparison

We compare IFPC and IPPC against EGR on three benchmarks that were also used in previous chapters. The first benchmark consists of STNs that are extracted from job-shop scheduling problems. Next is the "diamonds" benchmark, that gives rise to sparse networks. Finally, we consider a set of so-called sibling-restricted STNs, obtained from Hierarchical Task Networks (HTNs). The first two benchmarks are again taken from the SMT-LIB benchmark set, whereas the third was randomly generated by ourselves. The properties of these STNs are summarised in Table 6.2; for a more extensive description, see Chapter 4 (pp. 76–79). As discussed there, while these benchmark STNs have the *structure* conforming to the type of networks one would encounter when solving, the constraint weights are modified so as to ensure that each STN is guaranteed to be consistent.

Given these benchmarks, we iteratively solve the INCREMENTAL MINIMALITY problem query (STQ 7) for each constraint. Recall that in its current form, IPPC can only tighten constraints for which an edge is already present in the graph representation; moreover, since the STN must be PPC, this graph must be chordal. When running experiments with IPPC, therefore, we always consider the graph

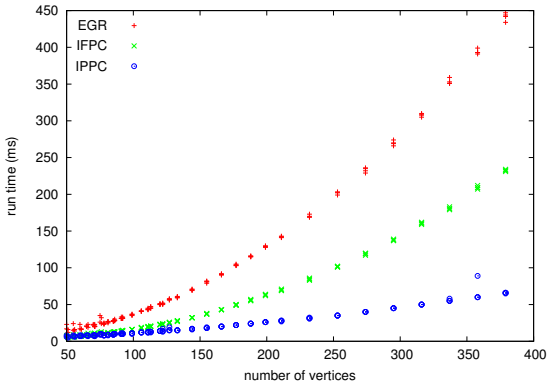


Figure 6.2: On the diamonds benchmark, IPPC performs best while EGR performs worst.

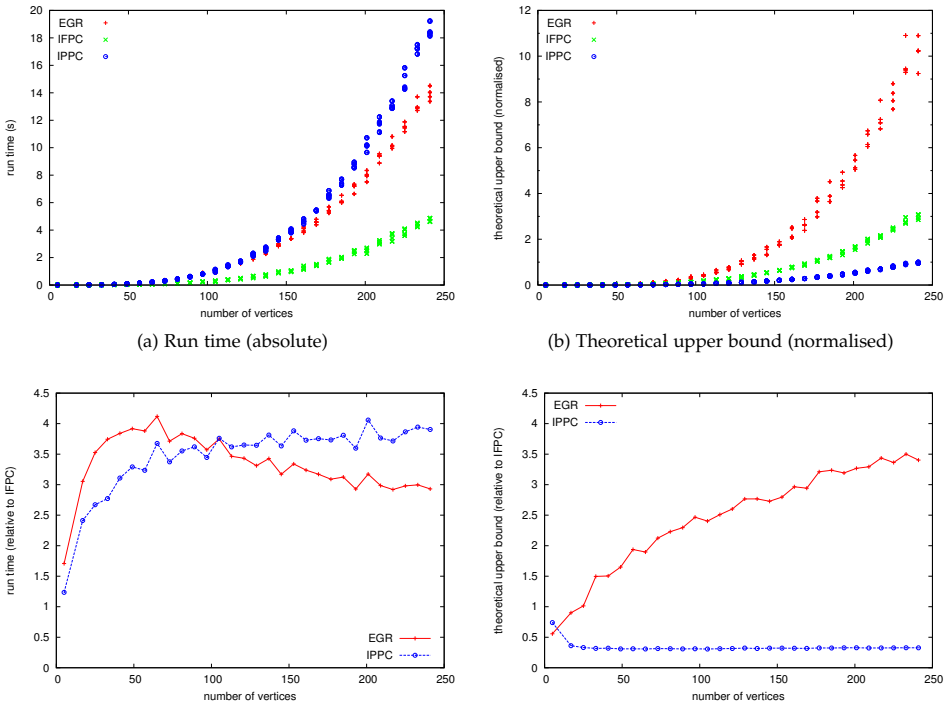
consisting of all constraint edges that will eventually be added, and triangulate it just once with the minimum-fill heuristic (see page 44). As stated before, we reserve incremental triangulation for future work.

The experiments were conducted on a Java virtual machine with 1 GiB of memory on a 2.4 GHz AMD processor. We report the wall-clock time required to add all constraints; triangulation time (in the case of IPPC) is not included.

We first consider the diamonds benchmark, for which results are included in Figure 6.2. On this benchmark, we see a clear and significant separation between the performance the three algorithms, with IPPC performing best and EGR performing worst. It is to be expected that IPPC performs so well, because the STNs in this benchmark are very sparse and the algorithm has far less work to take care of than its competitors, which must maintain the FPC network. However, the fact that IFPC outperforms EGR is surprising, especially considering that these networks have a constant degree $\delta = 4$, which means that the factor can be dropped from EGR's asymptotic time complexity. Thus, its bound simply becomes $\mathcal{O}(m^*)$, which outclasses the $\mathcal{O}(n + (n^*)^2)$ bound on IFPC since $m^* < (n^*)^2$. We can only conclude that the latter's simple pseudocode results in a very small constant hidden by the theoretical bound.

The results of the experiment on the job-shop benchmark can be found in Figure 6.3a. Here, IFPC outperforms both EGR and IPPC . The run times of IPPC and EGR are close, although EGR is slightly better. To be able to distinguish between their performance, consider Figure 6.3c, where we divide their run times by IFPC's. We see that as the size of the instances grows, IPPC's performance gets relatively worse; instead, EGR recovers a little after requiring over 4 times as much time as IFPC for instances with around 60 vertices. The crossover point occurs at around 110 vertices.

These results are all the more surprising because the values of the worst-case bounds as observed on these instances are better for IPPC than for either IFPC or EGR . This can be seen in Figure 6.3b, where we plot the normalised values of the



(a) Run time (absolute)

(b) Theoretical upper bound (normalised)

(c) Run time (relative to IFPC)

(d) Theoretical upper bound (relative to IFPC)

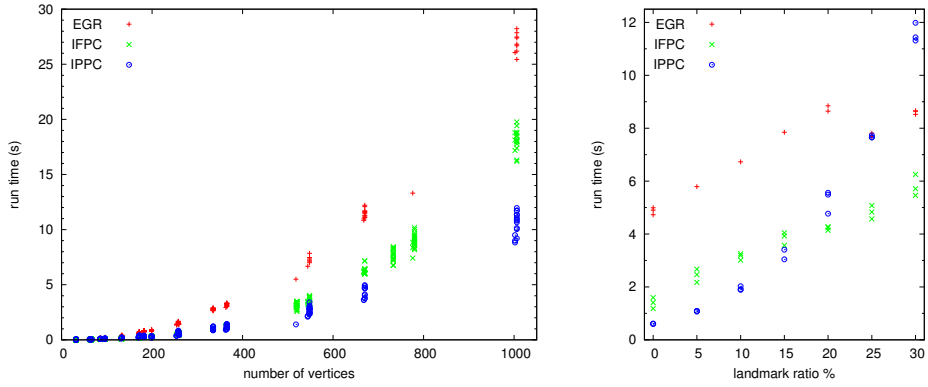
Figure 6.3: IFPC outperforms both EGR and IPPC on the job shop benchmark. However, IPPC's theoretical worst-case time bound is lower than those for both EGR and IFPC .

expressions $n_c^* \cdot \delta_c$, $m^* \cdot \delta$, and $n + (n^*)^2$, corresponding to the respective bounds of the three algorithms; we again show a relativised variant in Figure 6.3d.

It must be noted that the very simple approach taken by IFPC has a much better observed upper bound than the more sophisticated EGR algorithm; for large instances, the relative performance more or less conforms to the relative values of these bounds. In contrast, for IPPC, the worst-case bound paints a much brighter picture than the run times observed in reality. Relative to IFPC's bound, it drops to a value of about 0.3 almost immediately. We conjecture that this is related to the fact that the density* of the triangulated STNs in this benchmark is also around this value, as can be verified from Table 6.2. Note also the structure of the job-shop instances, depicted in Figure 4.4 on page 77. We do not know exactly what causes this discrepancy between theory and practice, but contributing factors may include the fast data structures available for EGR and IFPC and the overhead caused by the larger chordal graph.

We conclude this section with the HTN benchmark, of which we considered two variants. In the first, we varied the size of the networks, varying the branching

*The *density* of a graph is its number of edges relative to the number of edges in a complete graph of the same size, in this case $2m_c / n(n-1)$.



(a) Variable number of tasks

(b) Variable landmark ratio

Figure 6.4: IPPC generally outperforms EGR and IFPC on the HTN benchmarks.

factor between 4 and 10 and the depth of each branch between 3 and 7; the landmark and sibling ratios were fixed at 15% and 50%, respectively. The results for this set are shown in Figure 6.4a. We see that the ranking of the algorithms for this benchmark is equivalent to the diamonds benchmark. Although the separation is smaller, it is still significant, and IPPC is again the best-performing algorithm.

In a the second set of HTN experiments, we varied the landmark ratio from 0% to 30%. The effect of increasing the density of the resulting STNs in this manner is as expected, shown in Figure 6.4b. For landmark ratios up to 15%, IPPC performs best; however, of the three algorithms, it is also the most effected by this parameter. This also becomes apparent from the density of the triangulated network, which grows quickly; see the columns for m_c and δ_c in Table 6.2. IFPC's run time also increases with the landmark ratio, but at a much slower rate, whereas EGR's performance seems to stabilise around the 20% mark.

In summary, these exploratory experiments suggest that IPPC is the most efficient algorithm to use on sparse networks, and that the simple IFPC method should seriously be considered as an alternative to EGR .

Flow-shop scheduling: a case study

A second set of experiments was conducted by integrating our algorithms into an extant solver for job-shop scheduling problems. Several variations of this type of problem exist. Since we present results for benchmark instances of flow-shop scheduling below, we give a formal definition of this problem (see Garey and Johnson, 1979, Problem SS16). We then explain informally how this generalises to the job-shop scheduling problem (ibid., Problem SS18).

Definition 6.1. Flow-shop scheduling.

Instance: *A number $m \geq 1$ of machines and a set J of jobs, each job $j \in J$ consisting of m tasks $t_{j,1}, t_{j,2}, \ldots, t_{j,m}$ with integer lengths $l(t_{j,i}) > 0$, and an integer deadline $D > 0$.*

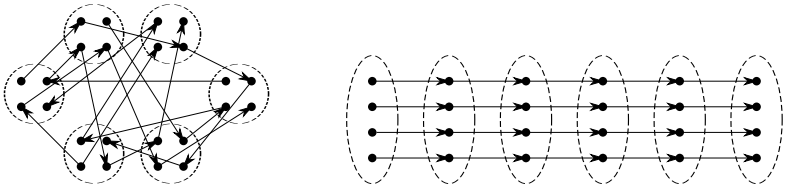
Question: *Is there a flow-shop schedule for J that meets the deadline D? That is, is there a collection of machine schedules $\sigma_i : J \to \mathbb{N}$ for $1 \leq i \leq m$, such that:*
- *each machine $i \in \{1, \ldots, m\}$ processes at most one task at a time, so that for all $j, k \in J$, $\sigma_i(j) \geq \sigma_i(k)$ implies $\sigma_i(j) > \sigma_i(k) + l(t_{j,i})$;*
- *the tasks in each job $j \in J$ progress linearly through the machines, so that for $i \in \{1, \ldots, m-1\}$, we have $\sigma_{i+1}(j) \geq \sigma_i(j) + l(t_{j,i})$; and*
- *the overall deadline is met, so $\sigma_m(j) + l(t_{j,m}) \leq D$ for all jobs $j \in J$?*

Thus, in a schedule for flow-shop scheduling, all jobs must be processed by each of the m machines in order and exactly once. In contrast, in the more general job-shop scheduling problem, the number n_j of tasks comprising each job $j \in J$ may differ, and there is no constraint on which machine must handle which task — in particular, there is no requirement that the jobs "flow" through the machines in a fixed order. Flow shop scheduling is strongly NP-complete for $m \geq 3$, whereas for job-shop scheduling, this is already the case for $m \geq 2$ (see Garey and Johnson, 1979). Often, instead of the decision problem stated here, the NP-hard optimisation variant is considered, where one attempts to find a schedule whose *makespan* (the time of completion of the last task to be processed) lies as close as possible to the minimum value of the deadline D.

Apart from flow-shop scheduling, several other variants of job-shop scheduling exist. Whatever variant is considered, they all have in common that there is contention for usage of the machines, which therefore are a scarce resource. The challenge thus lies in determining an order of precedence of tasks to be processed by each machine which results in an efficient schedule overall.

We describe the approach used by Oddi et al. (2011a,b, 2012) to tackle several variations of the job-shop scheduling problem. First, a schedule is generated through the method of precedence constraint posting (Oddi and Smith, 1997). Although this initial schedule satisfies the constraints, the makespan is generally far from optimal. The schedule is therefore repeatedly refined through iterative flattening search (Cesta et al., 2000), which removes some of the sequencing decisions and then attempts to find a tighter schedule. Therefore, this is basically an "anytime" method that is run for a given amount of time (or alternatively, until no further improvement is found for some number of iterations).

As precedence constraints are added, an STN representation of the schedule is gradually built up. Initially, the only non-universal constraints in this STN are between the subsequent tasks in each job, ensuring that for each $j \in J$, tasks $t_{j,i}$ and $t_{j,i+1}$ are scheduled at least $l(t_{j,i+1})$ time apart. The STN initially contains a universal constraint between each pair t, t' of tasks contending for the same machine. When a precedence $t \prec t'$ is posted, this constraint is tightened to require that task t' be scheduled at least $l(t)$ time later than task t.



(a) Job-shop (b) Flow-shop

Figure 6.5: Example structure of STNs underlying 4×6 job-shop and flow-shop instances. Vertices inside a dashed loop denote tasks requiring the same machine.

Shortest-path information in the STN is used in two ways. The first way is to distinguish between *search decisions*, where a choice must be made which of two tasks t, t' using the same machine is scheduled first, and *mandatory decisions* where the constraints already represented in the STN imply either $t \prec t'$ or $t' \prec t$. If a search decision is to be made, the solver bases this decision on a heuristic value computed from the minimal constraint between t and t' in the STN; this is the other way the shortest-path information is used.

Hence, the temporal reasoning system used by this job-shop solver must support the MINIMAL CONSTRAINT and INCREMENTAL MINIMALITY queries (STQs 5 and 7) for constraints between tasks using the same machine. Moreover, their efficiency is of great influence on the quality of the schedule eventually found by the solver. This is because the solver can run through more cycles in search of a good solution if each cycle (and the queries done in a cycle) takes little time. The solver provided by Oddi et al. has two integrated approaches to answer the INCREMENTAL MINIMALITY query: an implementation of AIMN, and the naive one-line approach from page 127.

Thus, we have another opportunity to test the performance of IFPC and IPPC . We integrated our new algorithms into the job-shop solver to determine their comparative performance in a real-life setting. For IFPC, this task is straightforward, since it can serve as a transparent substitute for either of the existing approaches; however, recall that IPPC in its current form cannot triangulate incrementally. Although we could take the same approach as for the previous set of experiments to deal with this, it would require a bigger modification to the existing solver. Therefore, we decided to restrict our attention to the flow-shop scheduling problem, as explained below.

Consider Figure 6.5, where we depict examples of the structure of an STN underlying the job-shop and flow-shop scheduling problems, both with 6 machines and 4 jobs. The solid arrows denote the precedence constraints that are part of the problem instance, whereas the dashed regions denote machines. When the solver by Oddi et al. tackles such scheduling problems, the MINIMAL CONSTRAINT and INCREMENTAL MINIMALITY queries it asks will only involve pairs of tasks sharing such a region. For IPPC to be able to answer arbitrary queries of this type, the STN must contain explicit (if possibly universal) constraints between each such pair of time points. Now, the great advantage of the STNs underlying flow-shop

instances is captured by the following proposition. Recall the definitions of bandwidth and treewidth from Chapter 3, on pages 35 and 41, respectively.

Proposition 6.10. *An $j \times m$ instance of the flow-shop scheduling problem, where m and j respectively denote the number of machines and jobs, gives rise to an STN with bandwidth and treewidth both equal to j.*

Proof. We prove this result for the example 4×6 instance depicted in Figure 6.5b. The proof is readily extended to the general case.

The STN underlying the example instance must contain explicit constraints for each solid edge and for each pair of vertices sharing a dashed region. Number the vertices in a top-down left-to-right order.

It is not hard to see that triangulating the STN (backwards) along this ordering yields an induced width of 4, with the lower-numbered neighbours of each vertex allocated to the same machine and the preceding machine. The treewidth cannot be lower than this value because all vertices have degree at least 4. Since the numbers given to all neighbouring vertices in the resulting chordal graph are at most 4 apart, it also has a bandwidth of 4. □

Thus, the band matrix is a space-efficient choice of data structure for IPPC, and ensures that the INCREMENTAL MINIMALITY query can be answered in constant time — which is of course also the case for the adjacency matrices underlying the other algorithms. It also follows from the above proof that the degree of the chordal graph is equal to twice the number of jobs. Hence, if the number of jobs can be regarded as a constant, IPPC runs in $\mathcal{O}(n^*)$ time.

Results We ran the job-shop solver by Oddi et al. on three flow-shop benchmark sets inspired by Taillard (1993), of sizes 5×20, 20×50, and 20×100, each of which contains 10 instances.*

For each instance, we ran the solver during 30 minutes of time on 10 random seeds. Thus, each incremental algorithm was used inside the solver 300 times in total. The results are presented in Table 6.3.

In this experiment, it turns out that our two new algorithms outperform the existing incremental solvers across the board. On the smallest instances with 5 jobs, IFPC is the most efficient algorithm, allowing the solver to run for almost four hundred thousand cycles on average. Somewhat surprisingly, AIMN performs worst and is even slower than the naive approach, which may be because its overhead does not pay off on these relatively small instances. Note however that for these small instances, the quality of the schedules is virtually equal for each of these incremental methods: the best schedule found by each of them is the same, i.e. the minimum deviation from the best makespan found

*We write "inspired" because Taillard proposed his benchmarks for a variant called the *permutation* flow-shop scheduling problem, where each machine must process the jobs in the same order. Also, we inadvertently swapped the numbers of jobs and machines. While our problem instances are still perfectly valid and suitable for evaluating the performance of the incremental algorithms, our results cannot be compared against other results and known bounds on Taillard's benchmarks.

Table 6.3: Results of the four incremental algorithms on the flow-shop benchmarks. The best results are in boldface.

		cycles			makespan (deviation %)*		
benchmark	algorithm	min	average	max	min	mean	max
5×20	naive	64.5K	96.2K	120K	0	0.53	5.1
	AIMN	41.4K	48.9K	56.6K	0	0.56	5.1
	IFPC	**313K**	**394K**	**501K**	0	**0.51**	5.1
	IPPC	198K	224K	262K	0	**0.51**	5.1
20×50	naive	25	32.1	40	6.9	12	21
	AIMN	50	59.4	71	4.3	8.7	15
	IFPC	258	298	354	0.24	3.3	8.0
	IPPC	**720**	**810**	**901**	**0**	**1.9**	**4.8**
20×100	naive	2	3.10	4	15	22	31
	AIMN	6	7.51	9	12	19	28
	IFPC	33	42.5	53	4.2	9.7	18
	IPPC	**163**	**194**	**229**	**0**	**2.0**	**5.6**

* For each of the 10 instances in each benchmark set, we recorded the best makespan found over the 40 solver runs (for each combination of algorithm and seed). We report for each algorithm the minimum, geometric mean, and maximum deviation percentages from these measures.

is 0%. In other words, when the solver tackles these small instances, it makes very little difference whether it is run for fifty thousand or five hundred thousand cycles.

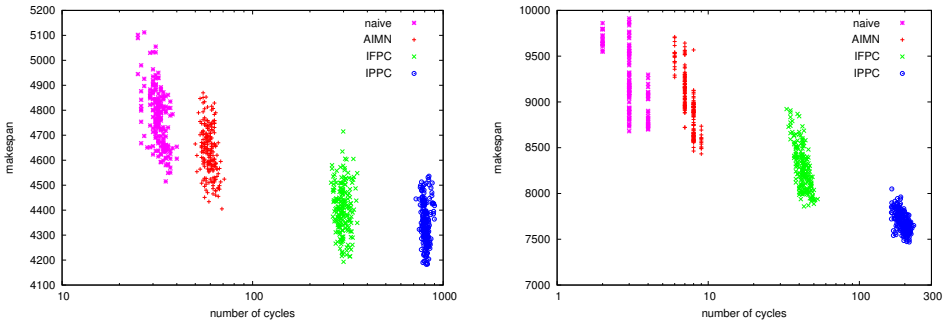
Matters are different for the instances of size 20×50 and 20×100. Here, the number of cycles is at least two to three orders of magnitude lower, and the quality of the schedules does differ between solvers. IPPC is now the most efficient solver, outperforming AIMN and the naive approach by at least an order of magnitude. The naive approach is now clearly the worst, managing to run for only a handful of cycles on the largest benchmark; however, AIMN does not fare much better. As was to be expected, of the four algorithms, IPPC is least affected by the increase in size.

For the larger two sizes of the flow-shop instances, consider also Figure 6.6, where we depict the results graphically. Note that the horizontal axis, for the number of cycles, is in logarithmic scale.

In conclusion, this case study confirms our earlier empirical findings that IFPC and IPPC are very competitive with extant approaches.

6.5 Other related work

In this section, we describe two publications that relate to the topic discussed in this chapter. The first extends the notion of incremental PPC to temporal networks distributed over multiple agents. The second proposes a method for



(a) 20 jobs, 50 machines

(b) 20 jobs, 100 machines

Figure 6.6: When integrated into the flow-shop solver, it performs much better with IPPC and IFPC than with the other approaches.

vertex-incremental PPC, as opposed to the edge-incremental version presented as Algorithm 6.2 in this chapter.

Distributed incremental PPC

Boerkoel, Planken, Wilcox, and Shah (2013) consider the situation where a Simple Temporal Network is distributed over multiple agents. In this setting, each time point is under control by a single agent, and constraints may thus involve time points under the control of two different agents. Here, it is often desirable to enforce and maintain partial path consistancy rather than full path consistency, because the latter requires full coupling between the subnetworks belonging to different agents, which results in loss of efficiency and privacy.

The authors propose two incremental methods for the multi-agent STN. One is DIPPC, a distributed version of IPPC that makes use of the clique tree representation of chordal graphs introduced in Chapter 3. The other is DI△STP, based on the △STP algorithm proposed by Xu and Choueiry (2003) and presented in Chapter 4.

The authors empirically evaluate the performance of these algorithms in a simulated distributed setting. They find that DIPPC is guaranteed to minimise the total amount of effort (i.e. CPU time) across all processors, but requires relatively many messages for coordination and synchronisation. Thus, DIPPC is fastest to propagate changes in the network only if there is very little message latency. As network latency increases, DI△STP outperforms DIPPC in terms of wall-clock time required to propagate changes, despite performing some redundant computation and requiring more messages than DIPPC .

Vertex-incremental PPC

Ten Thije, Planken, and De Weerdt (2011) consider a different take on the problem of incrementally maintaining PPC, namely in the face of a new *event* becoming

available, together with $k \geq 1$ constraints relating it to existing time points. Their method is called vertex-incremental PPC, or VIPPC in short. An important advantage of this approach is that it meshes well with the recent work by Berry et al. (2006) to incrementally triangulate a graph given a new vertex and edges incident on it, which takes $\mathcal{O}(nm)$ time over the course of adding n vertices with m edges in total to an empty graph. To the best of the knowledge of both Berry et al. and VIPPC's authors, it includes the first implementation of this vertex-incremental triangulation method.

Ten Thije et al. tackle the problem in three steps. First, the graph is again made chordal in $\mathcal{O}(m)$ amortised time by possibly adding fill edges incident on the new vertex. Then, DPC is re-enforced on a subnetwork of the STN, namely the new vertex together with its $k' \geq k$ neighbours in the newly chordal graph. This requires $\mathcal{O}(k'w_d^2) \subseteq \mathcal{O}(\delta_c w_d^2)$ time. Finally, the change is propagated outward throughout the complete STN in $\mathcal{O}(m_c)$ time. Thus, the algorithm requires $\mathcal{O}(m_c + \delta_c w_d^2)$ amortised time in total.

VIPPC is then empirically compared against a method consisting of incremental triangulation followed by k invocations of IPPC, once for each constraint involving the new vertex. Although a theoretical bound on this approach is $\mathcal{O}(km_c)$, which would seem worse than VIPPC's bound, the fact of the matter is that the IPPC-based approach is faster. Even when ensuring that $k' = k$, VIPPC does not manage to outperform IPPC.

6.6 Discussion

We presented two new algorithms to answer Simple Temporal Query 7 which asks to compute INCREMENTAL MINIMALITY. They incrementally establish full and partial path consistency, respectively, and thus are named IFPC and IPPC for short. The former is a simple way to maintain all-pairs shortest paths, whereas the latter is the first method — to the best of our knowledge — for incrementally maintaining PPC. Both turned out to be very competitive with the state of the art; in our exploratory experiments, IFPC always outperforms extant approaches, and IPPC is often faster still. Once these algorithms have answered INCREMENTAL MINIMALITY, the query for a MINIMAL CONSTRAINT can be answered in constant or at most logarithmic time.

One may raise the argument that comparing IPPC to the other algorithms is unjustified, because the former solves an easier problem and therefore often has less work to do. In many cases, however, requiring full path consistency it is a waste of effort and maintaining PPC is quite sufficient. We demonstrated this by successfully making use of IPPC to solve flow-shop scheduling problems. This class of problems is very attractive for IPPC because it allows the efficient band matrix to be used.

The IPPC algorithm has an obvious shortcoming which we already alluded to several times. It requires that the STN be chordal, and does not allow new constraint edges to be added. Extending IPPC to support addition of new constraints would greatly enhance its applicability. In the previous section, we noted that

some work has already been done for vertex-incremental triangulation (Berry et al., 2006); also, there exists a method for finding out in linear time whether a new edge preserves chordality (Ibarra, 2008). Future research may start from here.

Another direction of future research would be to decrementally maintain PPC, i.e. to allow for retraction of constraints. An algorithm for this purpose has been proposed by Ten Thije (2011), but it has never been implemented.

Finally, one may look for algorithms that make another trade-off between the time required to answer the queries of INCREMENTAL MINIMALITY and MINIMAL CONSTRAINT, e.g. by decreasing the complexity of the former even further at the cost of making the latter a bit harder to answer. This approach is taken by Chaudhuri and Zaroliagis (2000). Future research may compare IFPC and IPPC against these methods, and investigate whether they can be modified to make a similar trade-off.

CHAPTER 7

Discussion

In this dissertation, we studied the Simple Temporal Problem (STP) in depth. We surveyed its place in relation to other formalisms for temporal reasoning and gave nine different but related formal statements of the STP as a computational problem, whose complexity varies from trivial to P-complete—that is, presumably resistant to parallelisation and requiring significant space and time, but still tractable. We investigated constraint and graph theory and, based on this, we proposed four state-of-the-art algorithms: P^{3}C, Snowball, IFPC, and IPPC. They outperform extant approaches in theory as well as in practice; the latter was empirically established on a wide variety of problem instances and benchmarks.

In this final chapter, we review topics for future work that were identified throughout the dissertation and then broaden our scope to investigate the potential of our research in the fields of temporal coordination and constraint reasoning. The many avenues for future research that we identified along the way in this dissertation can broadly be classified in three categories: (i) perform more extensive empirical validation; (ii) further improve efficiency of our algorithms; and (iii) expand the area of their application. Let us briefly discuss each of these three.

First, although our algorithms have already been subjected to extensive experiments, these were all to greater or lesser extent of an artificial nature; thus, there is still room for more. Specifically, it would be of interest to investigate how all of our algorithms hold up in practical situations, e.g. in a deployed scheduling application. We also identified various more specific types of experiments yet to be performed. For example, in Chapter 4, we identified the grid graph as an interesting boundary case where theory predicts P^{3}C to still outperform its competition. Another type of experiment might investigate in more depth how P^{3}C performs relative to an algorithm for computing multiple-pairs shortest paths; so far, we only established that it is competitive when the set of query pairs coincides with the edges in the graph. Finally, our Snowball algorithm, proposed in Chapter 5, might be compared to algorithms that were specifically designed

for graphs of small treewidth such as those by Pettie (2004) and Chaudhuri and Zaroliagis (2000).

Secondly, throughout the dissertation, we also identified various opportunities for further improving the efficiency our algorithms. The Snowball–separators algorithm is an attempt in this direction. However, although its theoretical complexity is better than the original Snowball, this advantage does not hold up in practice; future research might investigate this. Another very promising line of research, virtually guaranteed to result in improved efficiency, is to let Snowball build on a directed variant of the DPC algorithm as outlined on page 123 in Section 5.5. For our incremental algorithms, presented in Chapter 6, it might pay to investigate whether they may make use of the shortest path trees maintained by the competing algorithms — although for IFPC, this might well negate its biggest strength: simplicity.

Finally, we may study ways to expand the applicability of our algorithms. As a specific example, it would be very desirable for our IPPC algorithm, presented in Chapter 6, to be able to add any constraint edge that was not previously present and thus incrementally triangulate the network in the process. We would also like to study the problem of maintaining PPC decrementally, i.e. when constraints are loosened or removed, even though this problem is in general as hard as enforcing PPC from scratch. More generally, we already noted in Chapter 3 that the properties of full, partial, and directional path consistency were originally proposed for general constraint problems. Although we designed our algorithms for the specific case of the STN, it might well be the case that with minor modifications they can be applied to any constraint problem that has the property of distributivity, as was shown to be the case for the Floyd–Warshall algorithm (see Proposition 3.21 on page 54).

Another direction of branching out is by studying ways of applying our work to a setting that involves multiple agents — whether competitive or of a cooperating nature. A key reason for the efficiency of the algorithms for PPC is that they maintain sparsity of the network. In a distributed setting, this is also a desirable property: it reduces the coupling between the processors or agents involved. An example of the viability of distributing our algorithms was already mentioned in the previous chapter, in the form of the DIPPC and DI△STP algorithms by Boerkoel, Planken, Wilcox, and Shah (2013).

Instead of propagating temporal changes through distributed algorithms, another approach for coordination is to modify the temporal network in such a way that no further processing or communication is required. Hunsberger (2002) proposed the problem of *temporal decoupling*, where one is given an STN together with a partitioning of its time points among a set of agents. The goal is to find a way to share the flexibility in the STN among the agents in such a way that each of them has complete freedom in scheduling the time points in its sub-STN in any locally consistent manner it desires, without risking global inconsistency. In other words, the global, distributed consistency problem is reduced to a number of independent local consistency problems.

Now, besides this search problem that asks for any such distribution of slack,

Hunsberger also proposed a variant called *optimal temporal decoupling* where the goal is to share the flexibility in some optimal way. For example, some sort of fairness might be desired, where the valuations of each of the agents are taken into account. Whereas Hunsberger showed that the search problem is not hard to solve, the complexity of optimal temporal decoupling was left open until we proved that it is NP-hard in general, but tractable for linear valuation functions **(Planken, De Weerdt, and Witteveen, 2010a)**. In the same work, we further demonstrated that through mechanism design ("reverse game theory"), a payment scheme can be designed that gives agents an incentive to truthfully represent their private valuations, thereby preventing manipulation.

This line of research is the more of interest because it has ongoing work that builds in part on the work presented in this dissertation. For example, Boerkoel and Durfee (2013) formally define the multiagent STN, present distributed algorithms for triangulation, decoupling, and PPC, and empirically validate their approach. Their work also contains an extensive list of related work in distributed constraint satisfaction, multiagent planning and scheduling, and resource and task allocation problems.

We conclude this dissertation by expressing our confidence that through it, we have provided a significant contribution to the field of temporal reasoning as well as the broader area of artificial intelligence. On page 1 of this dissertation, at the very beginning of the first chapter, we mentioned the importance of the concept of time in our daily lives. Likewise, in artificial intelligence, temporal reasoning is virtually ubiquitous. It is our hope that our nine Simple Temporal Queries and the notion of normal forms for temporal information, along with our new efficient means to attain these normal forms, will find wide adoption — and expansion — in future work.

APPENDIX A

Approximate Minimum Degree

A fast vertex ordering heuristic that is much used in practice, e.g. by the MATLAB and CPLEX software packages when computing LU decompositions or Cholesky factorisations of matrices, is to compute an approximate minimum degree ordering, abbreviated AMD (Amestoy et al., 1996; Davis et al., 2012; Heggernes et al., 2001). We briefly discussed this heuristic on pages 44–45 in Chapter 3, and mentioned that it is claimed to produce a vertex ordering in $\mathcal{O}(nm)$ time. In this appendix, we show that none of the published versions that we found fully specifies an algorithm that guarantees this bound. After investigating the theory behind AMD, we identify some of the problems with the heuristic's specification in existing literature. Then, we propose a remedy to ensure that the $\mathcal{O}(nm)$ time bound is met.

A.1 Quotient graphs

Instead of working on the graph directly, AMD uses a data structure called a quotient graph. It allows the algorithm to simulate vertex elimination without actually needing to add fill edges, so that no more than the original $\mathcal{O}(m)$ space is required. This is essential when working with matrices whose numbers of rows and columns run into the millions, and AMD takes care not to overstep this bound. A quotient graph $Q = \langle\langle N_s, N_e\rangle, E\rangle$ consists of two sets N_s and N_e of nodes — *s-nodes* and *e-nodes*, respectively — connected by edges. Each s-node can be adjacent to s-nodes as well as e-nodes; in turn, an e-node can only be adjacent to s-nodes. Given an s-node $x \in N_s$, we use $SAdj(x)$ and $EAdj(x)$ to denote the s-nodes and the e-nodes, respectively, that are adjacent to x in Q. For each e-node $e \in N_e$, the set $SAdj(e)$ is defined likewise.

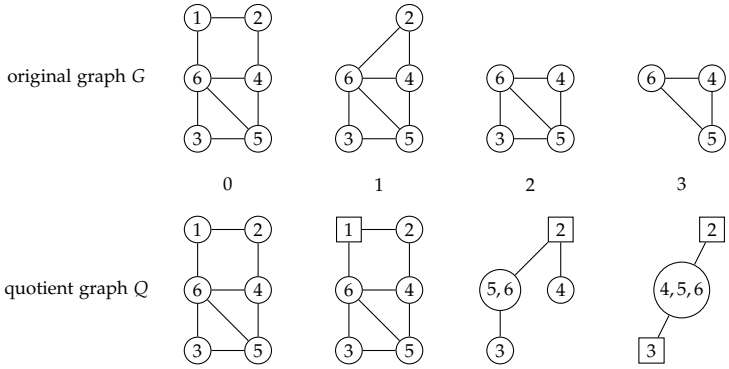


Figure A.1: This example of the evolution of the original graph G and the quotient graph Q during vertex elimination was presented by Heggernes et al. (2001). Circles represent vertices or s-nodes; squares represent e-nodes. Since there are no indistinguishable s-nodes, Q is initially identical to G, with each s-node representing a single vertex.

After the first elimination, s-node 1 is replaced by an e-node. In G, a fill edge is added; in Q, s-nodes 6 and 2 are implicitly connected through the newly-formed e-node.

After the second elimination, s-node 2 is also replaced by an e-node and merged with e-node 1. S-nodes 5 and 6 become indistinguishable and are merged. Also, the edge between s-nodes $\{5,6\}$ and 4 is removed because they are implicitly connected through e-node 2.

Finally, after the third elimination, another e-node is introduced and the remaining two s-nodes are merged. Vertices $\{4,5,6\}$ can now be eliminated (in arbitrary order). Note that e-nodes that are adjacent to only one s-node, such as e-nodes 2 and 3 in this example, lose their function and can be ignored.

Initially, the quotient graph is just a copy of the original graph, with each vertex represented by its own s-node and without any e-nodes. When a vertex in the original graph would be eliminated, the corresponding s-node in the quotient graph is replaced by an e-node instead. Presence of an e-node implies that the s-nodes in its neighbourhood together represent a clique in the original graph, without needing to add fill edges explicitly; put differently, an s-node adjacent to some e-node e is adjacent to all neighbours of e in the quotient graph. Whenever an e-node is formed adjacent to already existing e-nodes, all of them are merged. The neighbourhood of the new e-node then consists of the union of the original neighbourhoods. This merging ensures that to determine the actual neighbourhood of a vertex, only paths with up to one intermediate e-node need be traversed.

A technique employed when working with the quotient graph is to merge s-nodes that have become *indistinguishable*. A pair of s-nodes is said to be indistinguishable if they are adjacent and their neighbourhood is identical. The merged s-node then represents multiple vertices that are pairwise adjacent and therefore form a clique in the original graph. The number of vertices represented by an s-node is called its *weight*; initially, all s-nodes have a weight of 1. If s-nodes become indistinguishable at some point during vertex elimination, they will remain

Algorithm A.1: Approximate Minimum Degree (outline)

Input: A graph $G = \langle V, E \rangle$
Output: An ordering d of V

```
1  Construct a quotient graph Q = ⟨⟨N_s, N_e⟩, E⟩ for G.
2  Merge initial s-nodes where possible.
3  Set initial approximate degree of s-nodes to their actual degree.
4  Set i ← n.
5  while i > 0 do
6  |  Let x be an s-node of minimum approximate degree.
7  |  Set k to the number of vertices represented by x (the weight of x).
8  |  Place the vertices represented by x at positions i through i − k + 1 in d.
9  |  Set i ← i − k.
10 |  Find the set of s-nodes reachable from x (through at most one e-node).
11 |  Convert x to an e-node and merge neighbouring e-nodes.
12 |  Remove edges between all pairs of s-nodes adjacent to x.
13 |  Update approximate degrees for all s-nodes in the reachable set.
14 |  Merge s-nodes in the reachable set that have become indistinguishable.
15 end
16 return d
```

so for the rest of the algorithm (George and Liu, 1981). Also, when one of the vertices in the set represented by an s-node has minimum degree in the original graph, it can be easily shown that the entire set does. Therefore, these vertices can be eliminated together: a technique called *mass elimination*. This improves efficiency further in practice although it has no impact on the worst-case time complexity in general.

We include a side-by-side comparison of the elimination of vertices in a normal graph and a quotient graph in Figure A.1.

A.2 The algorithm

Having introduced the quotient graph data structure, we can present an outline of AMD . In the original publication by Amestoy et al. (1996), the algorithm is presented in a rather condensed way. Heggernes et al. (2001) present the algorithm in a form that is more accessible and also give a more elaborate complexity analysis. Barring small differences in the relative ordering of the steps that have no impact on either correctness or theoretical complexity, the outline we present as Algorithm A.1 fits both presentations. The only exception is line 2, which is only included in the version of Heggernes et al. However, Heggernes et al. do not state how new s-nodes are to be recognised and merged, either in line 2 or 14. Therefore, reference to the original paper is required.

Heuristics that keep track of the exact degree of vertices (or s-nodes) within the confines of only $\mathcal{O}(m)$ space pay for this in their computational complexity: Heggernes et al. prove that at best, such heuristics have a tight bound of $\mathcal{O}(n^2 m)$

on their run time. Thus, the idea behind AMD is to inexpensively compute an upper bound $\Delta(v)$ to the actual degree $\delta(v)$ of each vertex $v \in V$.

As initial approximation, the actual degree is used. Then, in each iteration, after eliminating an s-node x, the approximations for all s-nodes x' reachable from x must be updated. If the heuristic were to compute the exact degree for each of these s-nodes x', it would need to iterate over all s-nodes x'' adjacent to each x' either directly or through a single e-node, and sum their weights (i.e. the number of vertices represented by them). However, enumerating the paths through e-nodes for all members of the reachable set would overstep the $\mathcal{O}(m)$ time bound. Therefore, AMD instead computes for each e-node e, in $\mathcal{O}(m)$ total time, the sum of weights of s-nodes that are adjacent to e but excluded from the reachable set; this sum is then referred to as the weight of e. Now, these weights can be used to approximate the degree of each s-node x' in the reachable set as follows.

Note first that all s-nodes in the reachable set are pairwise adjacent through the e-node x, which has just been formed; any direct edges between them are removed in line 12 of Algorithm A.1. The reachable set is therefore equivalent to $SAdj(x)$. Now, as an an accurate starting point for each x' in the reachable set, AMD takes the summed weights over all s-nodes in $SAdj(x)$ minus the weight of x' itself. Next, it adds the summed weights of all s-nodes $x'' \in SAdj(x')$, none of which are in $SAdj(x)$. Then, the approximation consists in adding the summed weight of all e-nodes (other than x) adjacent to x'. This is possibly an overestimation: the weight of an s-node x'' is counted multiple times if it is outside $SAdj(x)$, but in $SAdj(e_1)$ and $SAdj(e_2)$, for e-nodes $e_1, e_2 \in EAdj(x')$. Finally, the heuristic ensures that the resulting approximation is no higher than either the number of vertices remaining in the graph or the previously approximated value plus the maximum number of (virtual) fill edges added. Denoting by $\|N\|$ the summed weights of all members of some set N of s-nodes or e-nodes, and letting $\Delta_{\text{old}}(x')$ be the previous approximation of the degree of some s-node $x' \in SAdj(x)$, the new approximated value $\Delta(x')$ can formally be specified as follows.*

$$\Delta(x') = \min \left\{ \begin{array}{c} i - 1, \\ \Delta_{\text{old}}(x') + \left\| SAdj(x) \setminus \{x'\} \right\|, \\ \left\| SAdj(x') \right\| + \left\| SAdj(x) \setminus \{x'\} \right\| + \left\| EAdj(x') \setminus \{x\} \right\| \end{array} \right\}$$

For all other s-nodes $y \notin SAdj(x)$, the approximated value is left unchanged: $\Delta(y) = \Delta_{\text{old}}(y)$. Let us now examine s-node merging in more depth.

*The reader may have noted at this point that AMD actually approximates the *external degree*: the number of adjacent vertices except for the ones sharing an s-node (i.e. a clique). This often yields better results in practice (Liu, 1985).

It must further be noted that the approximation as presented by Heggernes et al. (2001) differs from the one specified here and in the original paper by Amestoy et al. (1996), and sometimes results in an underestimation.

A.3 Merging s-nodes: a vulnerability

Since the merge operation is performed during each of the $\mathcal{O}(n)$ iterations, it must run in $\mathcal{O}(m)$ time in order to meet the claimed time bound for the algorithm. Therefore, a careful implementation is required.

AMD bases its verdict of distinguishability of s-nodes in the reachable set on their neighbourhood *in the quotient graph*. This means that s-nodes are considered indistinguishable only if their sets of directly adjacent s-nodes and e-nodes are identical, instead of comparing the union of all s-nodes that are adjacent either directly or through an e-node. Thus, the algorithm is somewhat sloppy, identifying only a subset of the s-nodes that are actually indistinguishable. However, the algorithm remains correct since it checks for a sufficient condition.

It would be prohibitively expensive to compare the sloppy neighbourhoods for all pairs of s-nodes in the reachable set. For this reason, AMD splits the set into separate buckets based on their hash values, computed from each s-node's (sloppy) adjacency. Then, for each pair of s-nodes sharing a hash bucket, the algorithm checks equivalence of their sloppy neighbourhoods by iterating over them. If a hash bucket contains k s-nodes, this requires $\Theta(k^2\delta)$ time. Here, k can be as high as w_d. In general, this means the available time is overstepped. The original authors acknowledge this, and state their assumption that the number of hash collisions per iteration is small:

> *[Our] bound assumes no (or few) [s-node] hash collisions [. . .]. In practice these assumptions seem to hold, but the asymptotic time would be higher if they did not.* — Amestoy et al., 1996, p. 18

The MATLAB software package includes an AMD procedure whose source is available for study (Davis et al., 2012). It uses the same method to merge s-nodes and therefore also exhibits this vulnerability.

A.4 Meeting the claimed time bound

Since s-node detection is merely a technique that can be used to improve practical performance, a way to meet the stated $\mathcal{O}(nm)$ time bound is to be even sloppier and skip merging s-nodes altogether. However, we also found that a careful implementation is possible to perform AMD's sloppy check more efficiently. We propose a detection method that requires $\mathcal{O}(m)$ time for each of the n iterations of AMD . Our approach makes use of *partition refinement* (Habib et al., 1999). After introducing this tool, we explain how it can be applied to merge s-nodes efficiently.

As suggested by its name, a partition refinement data structure iteratively refines a partition $\mathcal{P}$ of some set N, starting from the singleton partition $\{N\}$. The current partition $\mathcal{P}$ is iteratively refined by giving it a *pivot set* S. This operation takes $\mathcal{O}(|S|)$ time and splits each set N_i contained in $\mathcal{P}$ into two parts: $N_i^+ = N_i \cap S$ and $N_i^- = N_i \setminus S$. Note that each member $x \in N_i$ is partitioned into either N_i^+ or N_i^-, depending on whether $x \in S$. If one of N_i^+ and N_i^- is empty,

the refinement is a null operation with regard to N_i and it is left unchanged. It is important to state that the indices i, as well as the labels $+$ and $-$, are used for discussion purposes only; the data structure only keeps track of the current partition.

For AMD's case, we initialise N to the reachable set from the current s-node, determined in line 10 of Algorithm A.1. We adopt the adage of assuming innocence until proven guilty and take s-nodes to be indistinguishable until evidence to the contrary is found. This is reflected in the initial singleton partition $\{N\}$, where all s-nodes in N are assumed to be indistinguishable. Now, to be able to refine this partition and discriminate between s-nodes, we must construct pivot sets. These hinge on a set N' which is initialised to N, after which we let it grow through walking one edge in the quotient graph Q. Recalling that Q is no larger than the original graph, it follows that N' can be built in $\mathcal{O}(m)$ time. For defining N' more formally, we introduce the notions of s- and e-adjacency. Given an s-node $n \in N$, recall that $SAdj(n)$ and $EAdj(n)$ denote the s-nodes and the e-nodes, respectively, that are adjacent to n in Q. Then, N' is the following.

$$N' = N \cup \{s \in SAdj(n) \mid n \in N\} \cup \{e \in EAdj(n) \mid n \in N\}$$

Now, for each $n' \in N'$, we construct a pivot set $S_{n'} = \{n'\} \cup \{n \in SAdj(n')\}$. Although there are many pivots, the sum of their cardinalities $\sum_{n' \in N'} |S_{n'}|$ is no larger than $4m + n$ because each s-node or e-node is contained in at most one pivot, and any edge $\{n_1, n_2\}$ is counted at most four times. The upper bound occurs if both endpoints of the edge are members of S_{n_1} as well as S_{n_2}. Therefore, the partition can be refined in $\mathcal{O}(m)$ time with all of these pivots $S_{n'}$ (in arbitrary order). The following theorem states that we can then merge each set of s-nodes in the resulting partition.

Theorem A.1. *Two s-nodes $n_1, n_2 \in N$ are sloppily indistinguishable if and only if n_1 and n_2 share a set in $\mathcal{P}$ after the above process.*

Proof. ($\Rightarrow$) Assume that n_1 and n_2 are sloppily indistinguishable, so $SAdj(n_1) \cup \{n_2\} = SAdj(n_2) \cup \{n_1\}$ and $EAdj(n_1) = EAdj(n_2)$. Then, it follows by the definition of pivot sets above that for each pivot $S_{n'}$, either both n_1 and n_2 are members of $S_{n'}$ or neither of them is. Since they share a set in the original singleton partition and none of the pivots differentiates between them, they also share a set in the final partition.

($\Leftarrow$) To derive a contradiction, assume that n_1 and n_2 share a set in the final partition $\mathcal{P}$ while they are in fact distinguishable. There are then three possibilities: (i) n_1 and n_2 are not adjacent; (ii) $SAdj(n_1) \neq SAdj(n_2)$; or (iii) $EAdj(n_1) \neq EAdj(n_2)$.

In case (i), note that $n_1 \in N'$, so there is a pivot S_{n_1}. Now, n_2 is not in S_{n_1} because $n_2 \notin SAdj(n_1)$, whereas n_1 is in S_{n_1} since every $n' \in N'$ is a member of its own pivot. Therefore, refining $\mathcal{P}$ with S_{n_1} divorces n_1 from n_2, and we reach a contradiction in this case.

In cases (ii) and (iii), there must exist an s-node (or e-node) n' that is contained in exactly one of $SAdj(n_1)$ and $SAdj(n_2)$ (or $EAdj(n_1)$ and $EAdj(n_2)$). We handle

the case that n' is an s-node; the other case is analogous. If we assume w.l.o.g. that n' is only in $SAdj(n_1)$, the pivot set $S_{n'}$ contains n_1 but not n_2. Refining the partition with this pivot separates n_1 from n_2. Noting that the process of refining partitions is irreversible, n_1 and n_2 cannot be brought together again and we reach a contradiction. □

We conclude that using the approach outlined above, AMD can correctly detect sloppy indistinguishability while running in $\mathcal{O}(nm)$ time.

Bibliography

R. Albert and A.-L. Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47–97, January 2002. ISSN 0034-6861.

J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983. ISSN 0001-0782.

P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4): 886–905, October 1996. ISSN 0895-4798.

L. Anselma, P. Terenziani, S. Montani, and A. Bottrighi. Towards a comprehensive treatment of repetitions, periodicity and temporal constraints in clinical guidelines. *Artificial Intelligence in Medicine*, 38(2):171–195, October 2006. ISSN 0933-3657.

S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a *k*-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8(2):277–284, 1987. ISSN 0196-5212.

G. Ausiello, G. F. Italiano, A. Marchetti-Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. *Journal of Algorithms*, 12(4):615–638, 1991. ISSN 0196-6774.

V. Balachandhran and C. P. Rangan. All-pairs-shortest-length on strongly chordal graphs. *Discrete Applied Mathematics*, 69(1-2):169–182, 1996. ISSN 0166-218X.

L. Barbulescu, Z. B. Rubinstein, S. F. Smith, and T. L. Zimmerman. Distributed coordination of mobile agent teams. In *Proc. of the 9th Int. Conf. on Autonomous Agents and Multiagent Systems*, pages 1331–1338, 2010.

R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. In *Experimental Algorithms*, volume 5038 of *Lecture Notes in Computer Science*, pages 303–318, 2008.

A. Berry, P. Heggernes, and Y. Villanger. A vertex-incremental approach for maintaining chordality. *Discrete Mathematics*, 306(3):318–336, 2006. ISSN 0012-365X.

Ch. Bliek and Dj. Sam-Haroud. Path consistency on triangulated constraint graphs. In *Proc. of the 16th Int. Joint Conf. on Artificial Intelligence*, pages 456–461, 1999.

H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996. ISSN 0097-5397.

H. L. Bodlaender. A partial *k*-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209(1–2):1–45, 1998. ISSN 0304-3975.

J. C. Boerkoel, Jr. and E. H. Durfee. Distributed reasoning for multiagent simple temporal problems. *Journal of Artificial Intelligence Research*, 47:95–156, 2013. ISSN 1076-9757.

J. C. Boerkoel, Jr., L. R. Planken, R. J. Wilcox, and J. A. Shah. Distributed algorithms for incrementally maintaining multiagent Simple Temporal Networks. In *Proc. of the 23rd Int. Conf. on Automated Planning and Scheduling*, pages 11–19, 2013.

B. Bollobás and O. Riordan. Robustness and vulnerability of scale-free random graphs. *Internet Mathematics*, 1(1):1–35, 2004. ISSN 1542-7951.

V. Bouchitté, D. Kratsch, H. Müller, and I. Todinca. On treewidth approximations. *Discrete Applied Mathematics*, 136(2–3):183–196, 2004. ISSN 0166-218X.

J. L. Bresina, A. K. Jónsson, P. H. Morris, and K. Rajan. Activity planning for the mars exploration rovers. In *Proc. of the 15th Int. Conf. on Automated Planning and Scheduling*, pages 40–49, 2005.

V. Brusoni, L. Console, and P. Terenziani. On the computational complexity of querying bounds on difference constraints. *Artificial Intelligence*, 74(2):367–379, 1995. ISSN 0004-3702.

H. H. Bui and N. Yorke-Smith. Efficient variable elimination for semi-structured Simple Temporal Networks with continuous domains. *The Knowledge Engineering Review*, 25(3):337–351, 2010. ISSN 0269-8889.

H. H. Bui, M. Tyson, and N. Yorke-Smith. Efficient message passing and propagation of simple temporal constraints. In *Proc. of Workshop on Spatial and Temporal Reasoning*, 2007.

L. Castillo, J. Fernández-Olivares, and A. González. A temporal constraint network based temporal planner. In *Proc. of the 21st Workshop of the UK Planning and Scheduling Special Interest Group*, pages 99–109, 2002.

A. Cesta, A. Oddi, and S. F. Smith. Iterative flattening: A scalable method for solving multi-capacity scheduling problems. In *Proc. of the 7th Nat. Conf. on Artificial Intelligence*, pages 742–747, 2000.

T. M. Chan. All-pairs shortest paths with real weights in $\mathcal{O}(n^3/\log n)$ time. In *Algorithms and Datastructures*, Lecture Notes in Computer Science, pages 318–324, 2005.

Sh. Chaudhuri and Ch. D. Zaroliagis. Shortest paths in digraphs of small treewidth. Part I: Sequential algorithms. *Algorithmica*, 27(3):212–226, 2000. ISSN 0178-4617.

B. V. Cherkassky and A. V. Goldberg. Negative-cycle detection algorithms. *Mathematical Programming*, 85:277–311, 1999. ISSN 0025-5610.

B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73:129–174, 1996. ISSN 0025-5610.

N. Chleq. Efficient algorithms for networks of quantitative temporal constraints. In *Proc. of the 1st Int. Workshop on Constraint-Based Reasoning*, pages 40–45, April 1995.

B. Y. Choueiry and N. Wilson. Personal communication with Neil Yorke-Smith, 2006. Verified June 2010.

S. A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(1–3):2–22, 1985. ISSN 0019-9958.

Th. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 2nd edition*. MIT Press, 2001. ISBN 0-262-03384-4.

T. A. Davis, P. R. Amestoy, and I. S. Duff. AMD: Approximate minimum degree ordering. `http://www.cise.ufl.edu/research/sparse/amd/`, May 2012. Online; accessed 24 August 2012.

Th. L. Dean and D. V. McDermott. Temporal data base management. *Artificial Intelligence*, 32(1):1–55, 1987. ISSN 0004-3702.

R. Dechter. *Constraint Processing*. Morgan Kaufmann, San Francisco, CA, USA, 2003. ISBN 1-558-60890-7.

R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34(1):1–38, 1987. ISSN 0004-3702.

R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1–3):61–95, 1991. ISSN 0004-3702.

C. Demetrescu and G. F. Italiano. Fully dynamic all-pairs shortest paths with real edge weights. *Journal of Computer and System Sciences*, 72(5):813–837, 2006. ISSN 0022-0000.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. ISSN 0029-599X.

F. F. Dragan. Estimating all-pairs shortest paths in restricted graph families: A unified approach. *Journal of Algorithms*, 57(1):1–21, 2005. ISSN 0196-6774.

Sh. Even and H. Gazit. Updating distances in dynamic graphs. *Methods of Operations Research*, 49:371–387, 1985. ISSN 0078-5318.

N. Fiedler. Analysis of Java implementations of Fibonacci heap. `http://tinyurl.com/fibo-heap` (archived at `http://www.webcitation.org/6DNB1eqBH`), May 2008.

R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962. ISSN 0001-0782.

M. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987. ISSN 0004-5411.

M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0-716-71045-5.

F. Gavril. Personal communication, 1974. Cited by Garey and Johnson (1979), p. 134.

R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proc. of the Int. Workshop on Experimental Algorithms*, pages 319–333, 2008.

A. George and J. W. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1981. ISBN 0-13165-274-5.

M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*, volume 57 of *Annals of Discrete Mathematics*. Elsevier, 1980. ISBN 0-444-51530-5.

R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 1st edition, 1989. ISBN 0-201-14236-8.

R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to parallel computation: P-completeness theory*. Oxford University Press, Inc., New York, NY, USA, 1995.

M. Habib, Ch. Paul, and L. Viennot. Partition refinement techniques: An interesting algorithmic tool kit. *International Journal of Foundations of Computer Science*, 10(02):147–170, 1999. ISSN 0129-0541.

K. Han, Ch. N. Sekharan, and R. Sridhar. Unified all-pairs shortest path algorithms in the chordal hierarchy. *Discrete Applied Mathematics*, 77(1):59–71, 1997. ISSN 0166-218X.

Y. Han. A note of an $\mathcal{O}(n^3/\log n)$-time algorithm for all-pairs shortest paths. *Information Processing Letters*, 105(3):114–116, 2008. ISSN 0020-0190.

P. Heggernes. Minimal triangulations of graphs: A survey. *Discrete Mathematics*, 306(3):297–317, 2006. ISSN 0012-365X.

P. Heggernes, S. C. Eisenstat, G. Kumfert, and A. Pothen. The computational complexity of the minimum degree algorithm. Technical Report 2001–42, NASA Langley Research Center, 2001.

L. Hunsberger. *Group decision making and temporal reasoning*. PhD thesis, Harvard University, 2002.

L. Hunsberger. A practical temporal constraint management system for real-time applications. In *Proc. of the 18th European Conf. on Artificial Intelligence*, pages 553–557, 2008.

L. Ibarra. Fully dynamic algorithms for chordal graphs and split graphs. *ACM Transactions on Algorithms*, 4:40:1–40:20, August 2008. ISSN 1549-6325.

N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, 1988. ISSN 0097-5397.

D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977. ISSN 0004-5411.

D. S. Johnson. A catalog of complexity classes. In *Handbook of Theoretical Computer Science*, volume A, chapter 2, pages 67–161. Elsevier and MIT Press, 1990. ISBN 0-444-88071-2.

R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science*, volume A, chapter 17, pages 869–941. Elsevier and MIT Press, 1990. ISBN 0-444-88071-2.

M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):pp. 81–93, 1938. ISSN 0006-3444.

U. Kjærulff. Triangulation of graphs – algorithms giving small total state space. Technical Report R 90-09, Aalborg University, March 1990.

P. N. Klein, S. Mozes, and O. Weimann. Shortest paths in directed planar graphs with negative lengths: A linear-space $\mathcal{O}(n \log^2 n)$-time algorithm. *ACM Transactions on Algorithms*, 6(2):1–18, 2010. ISSN 1549-6325.

J. W. H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11(2):141–153, 1985. ISSN 0098-3500.

A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1): 99–118, 1977. ISSN 0004-3702.

R. Mohr and Th. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225–233, 1986. ISSN 0004-3702.

J. F. Mondou, T. G. Crainic, and S. Nguyen. Shortest path algorithms: A computational study with the C programming language. *Computers & Operations Research*, 18(8):767–786, 1991. ISSN 0305-0548.

U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7(0):95–132, 1974. ISSN 0020–0255.

N. Muscettola, P. Pandurang Nayak, B. Pell, and B. C. Williams. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103 (1–2):5–47, 1998. ISSN 0004-3702.

B. Nebel and H.-J. Bürckert. Reasoning about temporal relations: A maximal tractable subclass of Allen's interval algebra. *Journal of the ACM*, 42(1):43–66, 1995. ISSN 0004-5411.

A. Oddi and S. F. Smith. Stochastic procedures for generating feasible schedules. In *Proc. of the 14th Nat. Conf. on Artificial Intelligence*, pages 308–314, 1997.

A. Oddi, R. Rasconi, A. Cesta, and S. F. Smith. Iterative flattening search for the flexible job shop scheduling problem. In *Proc. of the 22nd Int. Joint Conf. on Artificial Intelligence*, pages 1991–1996, 2011a.

A. Oddi, R. Rasconi, A. Cesta, and S. F. Smith. Solving job shop scheduling with setup times through constraint-based iterative sampling: An experimental analysis. *Annals of Mathematics and Artificial Intelligence*, 62(3–4):371–402, 2011b. ISSN 1012-2443.

A. Oddi, R. Rasconi, A. Cesta, and S. F. Smith. Iterative improvement algorithms for the blocking job shop. In *Proc. of the 22nd Int. Conf. on Automated Planning and Scheduling*, pages 199–206, 2012.

Ch. H. Papadimitriou. The NP-completeness of the bandwidth minimization problem. *Computing*, 16(3):263–270, 1976. ISSN 0010-485X.

S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312(1):47–74, 2004. ISSN 0304-3975.

L. R. Planken. Temporal reasoning problems and algorithms for solving them (Literature survey). Master's thesis, Delft University of Technology, November 2007.

L. R. Planken. New algorithms for the Simple Temporal Problem. Master's thesis, Delft University of Technology, January 2008a.

L. R. Planken. Incrementally solving the STP by enforcing partial path consistency. In *Proc. of the 27th Workshop of the UK Planning and Scheduling Special Interest Group*, pages 87–94, December 2008b.

L. R. Planken, M. M. de Weerdt, and R. P. J. van der Krogt. P^3C: A new algorithm for the Simple Temporal Problem. In *Proc. of the 18th Int. Conf. on Automated Planning and Scheduling*, pages 256–263, September 2008.

L. R. Planken, M. M. de Weerdt, and C. Witteveen. Optimal temporal decoupling in multiagent systems. In *Proc. of the 9th Int. Conf. on Autonomous Agents and Multiagent Systems*, pages 789–796, 2010a.

L. R. Planken, M. M. de Weerdt, and N. Yorke-Smith. Incrementally solving STNs by enforcing partial path consistency. In *Proc. of the 20th Int. Conf. on Automated Planning and Scheduling*, pages 129–136, 2010b.

L. R. Planken, M. M. de Weerdt, and R. P. J. van der Krogt. Computing all-pairs shortest paths by leveraging low treewidth. In *Proc. of the 21st Int. Conf. on Automated Planning and Scheduling*, pages 170–177, 2011.

L. R. Planken, M. M. de Weerdt, and R. P. J. van der Krogt. Computing all-pairs shortest paths by leveraging low treewidth. *Journal of Artificial Intelligence Research*, 43:353–388, 2012. ISSN 1076-9757.

S. Ranise and C. Tinelli. The SMT-LIB format: An initial proposal. In *Proc. of the 1st Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, July 2003.

H. Rohnert. A dynamization of the all pairs least cost path problem. In *Proc. of the 2nd Symp. on Theoretical Aspects of Computer Science*, pages 279–286, 1985.

E. Schwalb and R. Dechter. Processing disjunctions in temporal constraint networks. *Artificial Intelligence*, 93(1–2):29–61, 1997. ISSN 0004-3702.

M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996. ISBN 0-534-94728-X.

O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding separation formulas with SAT. In *Proc. of the 14th Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 209–222, 2002.

R. Szelepcsényi. The method of forcing for nondeterministic automata. *Bulletin of the European Association for Theoretical Computer Science*, 33:96–100, October 1987. ISSN 0252-9742.

E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993. ISSN 0377-2217.

R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, August 1984. ISSN 0097-5397.

J. O. A. ten Thije. Towards a dynamic algorithm for the Simple Temporal Problem. Master's thesis, Delft University of Technology, August 2011.

J. O. A. ten Thije, L. R. Planken, and M. M. de Weerdt. Maintaining partial path consistency in stns under event-incremental updates. In *Proc. of the 29th Workshop of the UK Planning and Scheduling Special Interest Group*, 2011.

M. Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *Algorithm Theory*, volume 3111 of *Lecture Notes in Computer Science*, pages 384–396, 2004.

I. Tsamardinos and M. E. Pollack. Efficient solution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence*, 151(1–2):43–89, 2003. ISSN 0004-3702.

P. van Beek. Temporal query processing with indefinite information. *Artificial Intelligence in Medicine*, 3(6):325 – 339, 1991. ISSN 0933-3657.

M. Vilain and H. Kautz. Constraint propagation algorithms for temporal reasoning. In *Proc. of the 5th Nat. Conf. on Artificial Intelligence*, pages 377–382, 1986.

I-L. Wang. *Shortest Paths and Multicommodity Network Flows*. PhD thesis, Georgia Institute of Technology, 2003.

I-L. Wang, E. L. Johnson, and J. S. Sokol. A multiple-pairs shortest path algorithm. *Transportation Science*, 39(4):465–476, November 2005. ISSN 1526-5447.

S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962. ISSN 0004-5411.

D. B. West. *Introduction to Graph Theory*. Prentice-Hall, 1996.

L. Xu and B. Y. Choueiry. A new efficient algorithm for solving the Simple Temporal Problem. In *Proc. of the 10th Int. Symp. on Temporal Representation and Reasoning and 4th Int. Conf. on Temporal Logic*, pages 210–220, 2003.

N. Yorke-Smith and H. H. Bui. Efficient variable elimination for semi-structured Simple Temporal Networks with continuous domains. *The Knowledge Engineering Review*, 25(3):337–351, August 2010. ISSN 0269-8889.

Summary

Algorithms for Simple Temporal Reasoning

This dissertation describes research into new methods for automated temporal reasoning. For this purpose, several frameworks are available in literature. Chapter 1 presents a concise literature survey that provides a new overview of their interrelation. In the remainder of the dissertation, the focus is on quantitative frameworks, i.e. temporal constraint networks. Specifically, it is aimed at the subclass of Simple Temporal Networks (STNs) with the goal of assembling an algorithmic toolkit for dealing with the full set of temporal constraint networks. Temporal constraint networks find application in such diverse areas as space exploration, operation of Mars rovers, medical informatics, factory scheduling, and coordination of disaster relief efforts.

Chapter 2 describes properties of the STN, both known and previously unknown, and investigates the question: "What is the Simple Temporal Problem (STP)?" Although the STP is the subject matter of many publications, a formal definition was not yet available. In fact, the various approaches for the STP available in literature do not all solve the same problem — although each of the variants has an STN as underlying object. In the second part of Chapter 2, we therefore present theoretical research in the field of computational complexity and give an inventory of nine different Simple Temporal Queries (STQs), all of which have often been posed implicitly in existing literature. For all of these STQs, we prove membership in a theoretical complexity class. We also present the concept of a normal form for simple temporal information and prove that several STQs become significantly easier once time and effort have been spent for converting an STN into such a normal form.

In Chapter 3, attention is devoted to graph and constraint theory and the links between these fields are explored. Although part of the graph theory we present already formed the basis for existing methods for temporal reasoning, we uncover and prove new relations with constraint networks. By the end of the chapter, we show how the presented concepts of directional, partial, and full path consistency (DPC, PPC, and FPC, respectively) relate to the STQs from Chapter 2. We show that both PPC and FPC STNs can be considered as a normal form for simple temporal information.

The remaining chapters investigate algorithms for enforcing PPC (Chapter 4) and FPC (Chapter 5), and for maintaining either of these normal forms in the

face of new information becoming available (Chapter 6). Each of these chapters follows the same general structure. New algorithms are presented, and their correctness and computational complexity are formally established. These are then compared to existing approaches, both theoretically and empirically. It turns out that each of the new algorithms can be considered as the new state of the art in its respective class for sparse temporal networks, and sometimes even across the board.

Samenvatting

Algoritmen voor Simpel Redeneren over Tijd

Dit proefschrift beschrijft onderzoek naar nieuwe methoden voor geautomatiseerd redeneren over tijd. In de literatuur zijn diverse raamwerken voor dit doel beschikbaar. In hoofdstuk 1 wordt een bondig literatuuronderzoek gepresenteerd waarmee een nieuw overzicht gegeven wordt van hun onderlinge samenhang. In de rest van het proefschrift ligt de nadruk op kwantitatieve raamwerken, dat wil zeggen: netwerken van temporele randvoorwaarden *(temporal constraint networks)*. In het bijzonder spitst het onderzoek zich toe op het deelgebied van het Simpele Temporele Netwerk (STN) opdat een algoritmische gereedschapskist worde samengesteld waarmee de gehele verzameling netwerken van temporele randvoorwaarden aangepakt kan worden. Netwerken van temporele randvoorwaarden worden toegepast op diverse, veelal logistieke gebieden, zoals de planning van medische behandeltrajecten of ruimtevaartmissies, productieplanning in fabrieken en coördinatie van noodhulp in rampgebieden.

Hoofdstuk 2 beschrijft zowel bekende alsook enkele onbekende eigenschappen van het STN en onderzoekt de vraag: „Wat is het Simpele Temporele Probleem (STP)?" Hoewel dit probleem het onderwerp vormt van veel wetenschappelijke literatuur ontbrak tot op heden een formele definitie. Sterker nog: de verschillende oplossingsmethoden voor het STP die in de literatuur te vinden zijn hebben niet alle betrekking op hetzelfde probleem — hoewel elk van de probleemvarianten een STN als onderliggend object heeft. Het tweede deel van hoofdstuk 2 presenteert daarom een theoretisch onderzoek op het vlak van de fundamentele informatica en geeft een inventaris van een negental verschillende probleemvragen *(Simple Temporal Queries, STQ's)* die allemaal impliciet al vaak gesteld werden in bestaande literatuur. Van alle probleemvragen wordt het lidmaatschap in een formele complexiteitsklasse bewezen. Tenslotte presenteren we het concept van een normaalvorm voor temporele informatie. We bewijzen dat diverse STQ's significant makkelijker te beantwoorden zijn nadat er tijd en moeite is geïnvesteerd in het omzetten van een STN in een dergelijke normaalvorm.

In hoofdstuk 3 wordt de aandacht gericht op graaftheorie *(graph theory)* en randvoorwaardentheorie *(constraint theory)*, en worden de onderlinge verbanden tussen deze onderzoeksgebieden beschreven. Hoewel een gedeelte van de gepresenteerde graaftheorie reeds wordt gebruikt als onderbouwing van bestaande methoden voor redeneren over tijd, onthullen en bewijzen we nieuwe verbanden

tussen graaftheorie en netwerken van randvoorwaarden. Aan het eind van het hoofdstuk laten we zien hoe de gepresenteerde concepten van gerichte, partiële en volledige padconsistentie zich verhouden tot de STQ's uit hoofdstuk 2. We tonen aan dat zowel partieel als volledig padconsistente STN's beschouwd kunnen worden als een normaalvorm voor temporele informatie.

In de resterende hoofdstukken worden algoritmen bestudeerd voor het verwezenlijken van partiële en volledige padconsistentie (hoofdstukken 4 respectievelijk 5), evenals algoritmen om deze beide normaalvormen te handhaven wanneer nieuwe informatie beschikbaar komt (hoofdstuk 6). De opbouw van deze drie hoofdstukken is telkens hetzelfde. Nieuwe algoritmen worden gepresenteerd, en hun correctheid en complexiteit worden formeel vastgesteld. Deze algoritmen worden vervolgens vergeleken met bestaande methoden, zowel op het vlak van hun theoretische complexiteit als door middel van empirisch onderzoek. Het blijkt dat de efficiëntie van al onze nieuwe algoritmen, in elk geval voor ijle temporele netwerken en soms zelfs over de gehele linie, als grensverleggend beschouwd kan worden.

Curriculum vitae

Léon Robert Planken was born in Alkmaar on Thursday 24 May 1979. In the years 1991–1997, he followed secondary education at the Murmelliusgymnasium grammar school in Alkmaar and obtained his VWO diploma.

In September 1997, he started his education in Technical Informatics at Delft university, obtaining his B.Sc. degree in 2005; this was followed by a M.Sc. degree in Computer Science in January 2008. He was involved in student life as a member of student rovering crew "de Delftsche Zwervers" and of the student music society "Krashna Musika", and he has served as a board member for both.

Léon worked as a Ph.D. researcher in the Algorithmics group in the faculty of Electrical Engineering, Mathematics, and Computer Science at Delft University of Technology from March 2008 until June 2013. During this period, he published eight peer-reviewed papers as principal author. He was awarded a Honourable Mention for Best Student Paper at the 2011 ICAPS conference and published an article in the JAIR Award-winning Papers track. During his employment at Delft University of Technology, he was also involved in teaching, including the supervision of three students in their M.Sc. thesis work.

He spent October–December 2012 in Rome as a visiting researcher at the Planning and Scheduling Team, part of the Institute for Cognitive Science and Technology (ISTC) of the Italian National Research Council (CNR). Since September 2013, he is employed at West Consulting B.V. in Delft, but he maintains close ties to his alma mater.

Léon has played the French horn since age seven and also enjoys playing his extensive collection of board games with friends.

SIKS Dissertation Series*

1998

01 Johan van den Akker (CWI) *DEGAS – An Active, Temporal Database of Autonomous Objects*

02 Floris Wiesman (UM) *Information Retrieval by Graphically Browsing Meta-Information*

03 Ans Steuten (TUD) *A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective*

04 Dennis Breuker (UM) *Memory versus Search in Games*

05 E. W. Oskamp (RUL) *Computerondersteuning bij Straftoemeting*

1999

01 Mark Sloof (VU) *Physiology of Quality Change Modelling: Automated modelling of Quality Change of Agricultural Products*

02 Rob Potharst (EUR) *Classification using decision trees and neural nets*

03 Don Beal (UM) *The Nature of Minimax Search*

04 Jacques Penders (UM) *The practical Art of Moving Physical Objects*

05 Aldo de Moor (KUB) *Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems*

06 Niek Wijngaards (VU) *Re-design of compositional systems*

07 David Spelt (UT) *Verification support for object database design*

08 Jacques Lenting (UM) *Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation*

2000

01 Frank Niessink (VU) *Perspectives on Improving Software Maintenance*

02 Koen Holtman (TU/e) *Prototyping of CMS Storage Management*

---

*Glossary of abbreviations: CWI: *Centrum Wiskunde & Informatica, Amsterdam;* EUR: *Erasmus Universiteit Rotterdam;* KUB: *Katholieke Universiteit Brabant, Tilburg;* KUN: *Katholieke Universiteit Nijmegen;* OU: *Open Universiteit;* RUG: *Rijksuniversiteit Groningen;* RUL: *Rijksuniversiteit Leiden;* RUN: *Radboud Universiteit Nijmegen;* TUD: *Technische Universiteit Delft;* TU/e: *Technische Universiteit Eindhoven;* UL: *Universiteit van Leiden;* UM: *Universiteit van Maastricht;* UT: *Universiteit Twente, Enschede;* UU: *Universiteit Utrecht;* UvA: *Universiteit van Amsterdam;* UvT: *Universiteit van Tilburg;* VU: *Vrije Universiteit, Amsterdam*

03 Carolien Metselaar (UvA) *Sociaal-organisatorische gevolgen van kennistechnologie: Een procesbenadering en actorperspectief*
04 Geert de Haan (VU) *ETAG, A Formal Model of Competence Knowledge for User Interface Design*
05 Ruud van der Pol (UM) *Knowledge-based Query Formulation in Information Retrieval*
06 Rogier van Eijk (UU) *Programming Languages for Agent Communication*
07 Niels Peek (UU) *Decision-theoretic Planning of Clinical Patient Management*
08 Veerle Coupé (EUR) *Sensitivity Analyis of Decision-Theoretic Networks*
09 Florian Waas (CWI) *Principles of Probabilistic Query Optimization*
10 Niels Nes (CWI) *Image Database Management System Design Considerations, Algorithms and Architecture*
11 Jonas Karlsson (CWI) *Scalable Distributed Data Structures for Database Management*

2001

01 Silja Renooij (UU) *Qualitative Approaches to Quantifying Probabilistic Networks*
02 Koen Hindriks (UU) *Agent Programming Languages: Programming with Mental Models*
03 Maarten van Someren (UvA) *Learning as problem solving*
04 Evgueni Smirnov (UM) *Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets*
05 Jacco van Ossenbruggen (VU) *Processing Structured Hypermedia: A Matter of Style*
06 Martijn van Welie (VU) *Task-based User Interface Design*
07 Bastiaan Schonhage (VU) *Diva: Architectural Perspectives on Information Visualization*
08 Pascal van Eck (VU) *A Compositional Semantic Structure for Multi-Agent Systems Dynamics*
09 Pieter Jan 't Hoen (UL) *Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes*
10 Maarten Sierhuis (UvA) *Modeling and Simulating Work Practice BRAHMS: A multiagent modeling and simulation language for work practice analysis and design*
11 Tom van Engers (VU) *Knowledge Management: The Role of Mental Models in Business Systems Design*

2002

01 Nico Lassing (VU) *Architecture-Level Modifiability Analysis*
02 Roelof van Zwol (UT) *Modelling and searching web-based document collections*
03 Henk Ernst Blok (UT) *Database Optimization Aspects for Information Retrieval*
04 Juan Roberto Castelo Valdueza (UU) *The Discrete Acyclic Digraph Markov Model in Data Mining*
05 Radu Serban (VU) *The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents*
06 Laurens Mommers (UL) *Applied legal epistemology: Building a knowledge-based ontology of the legal domain*
07 Peter Boncz (CWI) *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*
08 Jaap Gordijn (VU) *Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas*
09 Willem-Jan van den Heuvel (UvT) *Integrating Modern Business Applications with Objectified Legacy Systems*
10 Brian Sheppard (UM) *Towards Perfect Play of Scrabble*
11 Wouter Wijngaards (VU) *Agent Based Modelling of Dynamics: Biological and Organisational Applications*
12 Albrecht Schmidt (UvA) *Processing XML in Database Systems*

13 Hongjing Wu (TU/e) *A Reference Architecture for Adaptive Hypermedia Applications*
14 Wieke de Vries (UU) *Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems*
15 Rik Eshuis (UT) *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*
16 Pieter van Langen (VU) *The Anatomy of Design: Foundations, Models and Applications*
17 Stefan Manegold (UvA) *Understanding, Modeling, and Improving Main-Memory Database Performance*

2003

01 Heiner Stuckenschmidt (VU) *Ontology-Based Information Sharing in Weakly Structured Environments*
02 Jan Broersen (VU) *Modal Action Logics for Reasoning About Reactive Systems*
03 Martijn Schuemie (TUD) *Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy*
04 Milan Petkovic (UT) *Content-Based Video Retrieval Supported by Database Technology*
05 Jos Lehmann (UvA) *Causation in Artificial Intelligence and Law – A modelling approach*
06 Boris van Schooten (UT) *Development and specification of virtual environments*
07 Machiel Jansen (UvA) *Formal Explorations of Knowledge Intensive Tasks*
08 Yongping Ran (UM) *Repair Based Scheduling*
09 Rens Kortmann (UM) *The resolution of visually guided behaviour*
10 Andreas Lincke (UvT) *Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture*
11 Simon Keizer (UT) *Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks*
12 Roeland Ordelman (UT) *Dutch speech recognition in multimedia information retrieval*
13 Jeroen Donkers (UM) *Nosce Hostem – Searching with Opponent Models*
14 Stijn Hoppenbrouwers (KUN) *Freezing Language: Conceptualisation Processes across ICT-Supported Organisations*
15 Mathijs de Weerdt (TUD) *Plan Merging in Multi-Agent Systems*
16 Menzo Windhouwer (CWI) *Feature Grammar Systems – Incremental Maintenance of Indexes to Digital Media Warehouses*
17 David Jansen (UT) *Extensions of Statecharts with Probability, Time, and Stochastic Timing*
18 Levente Kocsis (UM) *Learning Search Decisions*

2004

01 Virginia Dignum (UU) *A Model for Organizational Interaction: Based on Agents, Founded in Logic*
02 Lai Xu (UvT) *Monitoring Multi-party Contracts for E-business*
03 Perry Groot (VU) *A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving*
04 Chris van Aart (UvA) *Organizational Principles for Multi-Agent Architectures*
05 Viara Popova (EUR) *Knowledge discovery and monotonicity*
06 Bart-Jan Hommes (TUD) *The Evaluation of Business Process Modeling Techniques*
07 Elise Boltjes (UM) *Voorbeeldig onderwijs: Voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes*
08 Joop Verbeek (UM) *Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiële gegevensuitwisseling en digitale expertise*

09 Martin Caminada (VU) *For the Sake of the Argument: Explorations into argument-based reasoning*
10 Suzanne Kabel (UvA) *Knowledge-rich indexing of learning-objects*
11 Michel Klein (VU) *Change Management for Distributed Ontologies*
12 The Duy Bui (UT) *Creating emotions and facial expressions for embodied agents*
13 Wojciech Jamroga (UT) *Using Multiple Models of Reality: On Agents who Know how to Play*
14 Paul Harrenstein (UU) *Logic in Conflict. Logical Explorations in Strategic Equilibrium*
15 Arno Knobbe (UU) *Multi-Relational Data Mining*
16 Federico Divina (VU) *Hybrid Genetic Relational Search for Inductive Learning*
17 Mark Winands (UM) *Informed Search in Complex Games*
18 Vania Bessa Machado (UvA) *Supporting the Construction of Qualitative Knowledge Models*
19 Thijs Westerveld (UT) *Using generative probabilistic models for multimedia retrieval*
20 Madelon Evers (Nyenrode) *Learning from Design: Facilitating multidisciplinary design teams*

2005

01 Floor Verdenius (UvA) *Methodological Aspects of Designing Induction-Based Applications*
02 Erik van der Werf (UM) *AI techniques for the game of Go*
03 Franc Grootjen (RUN) *A Pragmatic Approach to the Conceptualisation of Language*
04 Nirvana Meratnia (UT) *Towards Database Support for Moving Object data*
05 Gabriel Infante-Lopez (UvA) *Two-Level Probabilistic Grammars for Natural Language Parsing*
06 Pieter Spronck (UM) *Adaptive Game AI*
07 Flavius Frasincar (TU/e) *Hypermedia Presentation Generation for Semantic Web Information Systems*
08 Richard Vdovjak (TU/e) *A Model-driven Approach for Building Distributed Ontology-based Web Applications*
09 Jeen Broekstra (VU) *Storage, Querying and Inferencing for Semantic Web Languages*
10 Anders Bouwer (UvA) *Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments*
11 Elth Ogston (VU) *Agent Based Matchmaking and Clustering – A Decentralized Approach to Search*
12 Csaba Boer (EUR) *Distributed Simulation in Industry*
13 Fred Hamburg (UL) *Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen*
14 Borys Omelayenko (VU) *Web-Service configuration on the Semantic Web: Exploring how semantics meets pragmatics*
15 Tibor Bosse (VU) *Analysis of the Dynamics of Cognitive Processes*
16 Joris Graaumans (UU) *Usability of XML Query Languages*
17 Boris Shishkov (TUD) *Software Specification Based on Re-usable Business Components*
18 Danielle Sent (UU) *Test-selection strategies for probabilistic networks*
19 Michel van Dartel (UM) *Situated Representation*
20 Cristina Coteanu (UL) *Cyber Consumer Law, State of the Art and Perspectives*
21 Wijnand Derks (UT) *Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics*

2006

01 Samuil Angelov (TU/e) *Foundations of B2B Electronic Contracting*
02 Cristina Chisalita (VU) *Contextual issues in the design and use of information technology in organizations*
03 Noor Christoph (UvA) *The role of metacognitive skills in learning to solve problems*

04 Marta Sabou (VU) *Building Web Service Ontologies*
05 Cees Pierik (UU) *Validation Techniques for Object-Oriented Proof Outlines*
06 Ziv Baida (VU) *Software-aided Service Bundling – Intelligent Methods & Tools for Graphical Service Modeling*
07 Marko Smiljanic (UT) *XML schema matching – balancing efficiency and effectiveness by means of clustering*
08 Eelco Herder (UT) *Forward, Back and Home Again – Analyzing User Behavior on the Web*
09 Mohamed Wahdan (UM) *Automatic Formulation of the Auditor's Opinion*
10 Ronny Siebes (VU) *Semantic Routing in Peer-to-Peer Systems*
11 Joeri van Ruth (UT) *Flattening Queries over Nested Data Types*
12 Bert Bongers (VU) *Interactivation – Towards an e-cology of people, our technological environment, and the arts*
13 Henk-Jan Lebbink (UU) *Dialogue and Decision Games for Information Exchanging Agents*
14 Johan Hoorn (VU) *Software Requirements: Update, Upgrade, Redesign – towards a Theory of Requirements Change*
15 Rainer Malik (UU) *CONAN: Text Mining in the Biomedical Domain*
16 Carsten Riggelsen (UU) *Approximation Methods for Efficient Learning of Bayesian Networks*
17 Stacey Nagata (UU) *User Assistance for Multitasking with Interruptions on a Mobile Device*
18 Valentin Zhizhkun (UvA) *Graph transformation for Natural Language Processing*
19 Birna van Riemsdijk (UU) *Cognitive Agent Programming: A Semantic Approach*
20 Marina Velikova (UvT) *Monotone models for prediction in data mining*
21 Bas van Gils (RUN) *Aptness on the Web*
22 Paul de Vrieze (RUN) *Fundaments of Adaptive Personalisation*
23 Ion Juvina (UU) *Development of Cognitive Model for Navigating on the Web*
24 Laura Hollink (VU) *Semantic Annotation for Retrieval of Visual Resources*
25 Madalina Drugan (UU) *Conditional log-likelihood MDL and Evolutionary MCMC*
26 Vojkan Mihajlovic (UT) *Score Region Algebra: A Flexible Framework for Structured Information Retrieval*
27 Stefano Bocconi (CWI) *Vox Populi: Generating video documentaries from semantically annotated media repositories*
28 Borkur Sigurbjornsson (UvA) *Focused Information Access using XML Element Retrieval*

2007

01 Kees Leune (UvT) *Access Control and Service-Oriented Architectures*
02 Wouter Teepe (RUG) *Reconciling Information Exchange and Confidentiality: A Formal Approach*
03 Peter Mika (VU) *Social Networks and the Semantic Web*
04 Jurriaan van Diggelen (UU) *Achieving Semantic Interoperability in Multi-agent Systems: A dialogue-based approach*
05 Bart Schermer (UL) *Software Agents, Surveillance, and the Right to Privacy: A Legislative Framework for Agent-enabled Surveillance*
06 Gilad Mishne (UvA) *Applied Text Analytics for Blogs*
07 Natasa Jovanovic' (UT) *To Whom It May Concern – Addressee Identification in Face-to-Face Meetings*
08 Mark Hoogendoorn (VU) *Modeling of Change in Multi-Agent Organizations*
09 David Mobach (VU) *Agent-Based Mediated Service Negotiation*
10 Huib Aldewereld (UU) *Autonomy vs. Conformity: An Institutional Perspective on Norms and Protocols*
11 Natalia Stash (TU/e) *Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System*

12 Marcel van Gerven (RUN) *Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty*
13 Rutger Rienks (UT) *Meetings in Smart Environments: Implications of Progressing Technology*
14 Niek Bergboer (UM) *Context-Based Image Analysis*
15 Joyca Lacroix (UM) *NIM: A Situated Computational Memory Model*
16 Davide Grossi (UU) *Designing Invisible Handcuffs: Formal investigations in Institutions and Organizations for Multi-agent Systems*
17 Theodore Charitos (UU) *Reasoning with Dynamic Networks in Practice*
18 Bart Orriens (UvT) *On the development an management of adaptive business collaborations*
19 David Levy (UM) *Intimate relationships with artificial partners*
20 Slinger Jansen (UU) *Customer Configuration Updating in a Software Supply Network*
21 Karianne Vermaas (UU) *Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005*
22 Zlatko Zlatev (UT) *Goal-oriented design of value and process models from patterns*
23 Peter Barna (TU/e) *Specification of Application Logic in Web Information Systems*
24 Georgina Ramírez Camps (CWI) *Structural Features in XML Retrieval*
25 Joost Schalken (VU) *Empirical Investigations in Software Process Improvement*

2008

01 Katalin Boer-Sorbán (EUR) *Agent-Based Simulation of Financial Markets: A modular, continuous-time approach*
02 Alexei Sharpanskykh (VU) *On Computer-Aided Methods for Modeling and Analysis of Organizations*
03 Vera Hollink (UvA) *Optimizing hierarchical menus: A usage-based approach*
04 Ander de Keijzer (UT) *Management of Uncertain Data – towards unattended integration*
05 Bela Mutschler (UT) *Modeling and simulating causal dependencies on process-aware information systems from a cost perspective*
06 Arjen Hommersom (RUN) *On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective*
07 Peter van Rosmalen (OU) *Supporting the tutor in the design and support of adaptive e-learning*
08 Janneke Bolt (UU) *Bayesian Networks: Aspects of Approximate Inference*
09 Christof van Nimwegen (UU) *The paradox of the guided user: Assistance can be counter-effective*
10 Wauter Bosma (UT) *Discourse oriented summarization*
11 Vera Kartseva (VU) *Designing Controls for Network Organizations: A Value-Based Approach*
12 Jozsef Farkas (RUN) *A Semiotically Oriented Cognitive Model of Knowledge Representation*
13 Caterina Carraciolo (UvA) *Topic Driven Access to Scientific Handbooks*
14 Arthur van Bunningen (UT) *Context-Aware Querying: Better Answers with Less Effort*
15 Martijn van Otterlo (UT) *The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains*
16 Henriette van Vugt (VU) *Embodied agents from a user's perspective*
17 Martin Op 't Land (TUD) *Applying Architecture and Ontology to the Splitting and Allying of Enterprises*
18 Guido de Croon (UM) *Adaptive Active Vision*
19 Henning Rode (UT) *From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search*
20 Rex Arendsen (UvA) *Geen bericht, goed bericht: Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven*
21 Krisztian Balog (UvA) *People Search in the Enterprise*
22 Henk Koning (UU) *Communication of IT-Architecture*

23 Stefan Visscher (UU) *Bayesian network models for the management of ventilator-associated pneumonia*
24 Zharko Aleksovski (VU) *Using background knowledge in ontology matching*
25 Geert Jonker (UU) *Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency*
26 Marijn Huijbregts (UT) *Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled*
27 Hubert Vogten (OU) *Design and Implementation Strategies for IMS Learning Design*
28 Ildiko Flesch (RUN) *On the Use of Independence Relations in Bayesian Networks*
29 Dennis Reidsma (UT) *Annotations and Subjective Machines – Of Annotators, Embodied Agents, Users, and Other Humans*
30 Wouter van Atteveldt (VU) *Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content*
31 Loes Braun (UM) *Pro-Active Medical Information Retrieval*
32 Trung Bui (UT) *Toward Affective Dialogue Management using Partially Observable Markov Decision Processes*
33 Frank Terpstra (UvA) *Scientific Workflow Design: Theoretical and practical issues*
34 Jeroen de Knijf (UU) *Studies in Frequent Tree Mining*
35 Ben Torben Nielsen (UvT) *Dendritic morphologies: Function shapes structure*

2009

01 Rasa Jurgelenaite (RUN) *Symmetric Causal Independence Models*
02 Willem Robert van Hage (VU) *Evaluating Ontology-Alignment Techniques*
03 Hans Stol (UvT) *A Framework for Evidence-based Policy Making Using IT*
04 Josephine Nabukenya (RUN) *Improving the Quality of Organisational Policy Making using Collaboration Engineering*
05 Sietse Overbeek (RUN) *Bridging Supply and Demand for Knowledge Intensive Tasks – Based on Knowledge, Cognition, and Quality*
06 Muhammad Subianto (UU) *Understanding Classification*
07 Ronald Poppe (UT) *Discriminative Vision-Based Recovery and Recognition of Human Motion*
08 Volker Nannen (VU) *Evolutionary Agent-Based Policy Analysis in Dynamic Environments*
09 Benjamin Kanagwa (RUN) *Design, Discovery and Construction of Service-oriented Systems*
10 Jan Wielemaker (UvA) *Logic programming for knowledge-intensive interactive applications*
11 Alexander Boer (UvA) *Legal Theory, Sources of Law & the Semantic Web*
12 Peter Massuthe (TU/e, Humboldt-Universität zu Berlin) *Operating Guidelines for Services*
13 Steven de Jong (UM) *Fairness in Multi-Agent Systems*
14 Maksym Korotkiy (VU) *From ontology-enabled services to service-enabled ontologies: Making ontologies work in e-science with ONTO-SOA*
15 Rinke Hoekstra (UvA) *Ontology Representation – Design Patterns and Ontologies that Make Sense*
16 Fritz Reul (UvT) *New Architectures in Computer Chess*
17 Laurens van der Maaten (UvT) *Feature Extraction from Visual Data*
18 Fabian Groffen (CWI) *Armada, An Evolving Database System*
19 Valentin Robu (CWI) *Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets*
20 Bob van der Vecht (UU) *Adjustable Autonomy: Controling Influences on Decision Making*
21 Stijn Vanderlooy (UM) *Ranking and Reliable Classification*
22 Pavel Serdyukov (UT) *Search For Expertise: Going beyond direct evidence*
23 Peter Hofgesang (VU) *Modelling Web Usage in a Changing Environment*

24 Annerieke Heuvelink (VU) *Cognitive Models for Training Simulations*
25 Alex van Ballegooij (CWI) *RAM: Array Database Management through Relational Mapping*
26 Fernando Koch (UU) *An Agent-Based Model for the Development of Intelligent Mobile Services*
27 Christian Glahn (OU) *Contextual Support of social Engagement and Reflection on the Web*
28 Sander Evers (UT) *Sensor Data Management with Probabilistic Models*
29 Stanislav Pokraev (UT) *Model-Driven Semantic Integration of Service-Oriented Applications*
30 Marcin Zukowski (CWI) *Balancing vectorized query execution with bandwidth-optimized storage*
31 Sofiya Katrenko (UvA) *A Closer Look at Learning Relations from Text*
32 Rik Farenhorst (VU) and Remco de Boer (VU) *Architectural Knowledge Management: Supporting Architects and Auditors*
33 Khiet Truong (UT) *How Does Real Affect Affect Affect Recognition In Speech?*
34 Inge van de Weerd (UU) *Advancing in Software Product Management: An Incremental Method Engineering Approach*
35 Wouter Koelewijn (UL) *Privacy en Politiegegevens: Over geautomatiseerde normatieve informatie-uitwisseling*
36 Marco Kalz (OU) *Placement Support for Learners in Learning Networks*
37 Hendrik Drachsler (OU) *Navigation Support for Learners in Informal Learning Networks*
38 Riina Vuorikari (OU) *Tags and self-organisation: A metadata ecology for learning resources in a multilingual context*
39 Christian Stahl (TU/e, Humboldt-Universität zu Berlin) *Service Substitution – A Behavioral Approach Based on Petri Nets*
40 Stephan Raaijmakers (UvT) *Multinomial Language Learning: Investigations into the Geometry of Language*
41 Igor Berezhnyy (UvT) *Digital Analysis of Paintings*
42 Toine Bogers (UvT) *Recommender Systems for Social Bookmarking*
43 Virginia Nunes Leal Franqueira (UT) *Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients*
44 Roberto Santana Tapia (UT) *Assessing Business-IT Alignment in Networked Organizations*
45 Jilles Vreeken (UU) *Making Pattern Mining Useful*
46 Loredana Afanasiev (UvA) *Querying XML: Benchmarks and Recursion*

2010

01 Matthijs van Leeuwen (UU) *Patterns that Matter*
02 Ingo Wassink (UT) *Work flows in Life Science*
03 Joost Geurts (CWI) *A Document Engineering Model and Processing Framework for Multimedia Documents*
04 Olga Kulyk (UT) *Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments*
05 Claudia Hauff (UT) *Predicting the Effectiveness of Queries and Retrieval Systems*
06 Sander Bakkes (UvT) *Rapid Adaptation of Video Game AI*
07 Wim Fikkert (UT) *Gesture interaction at a Distance*
08 Krzysztof Siewicz (UL) *Towards an Improved Regulatory Framework of Free Software: Protecting user freedoms in a world of software communities and eGovernments*
09 Hugo Kielman (UL) *A Politiele gegevensverwerking en Privacy: Naar een effectieve waarborging*
10 Rebecca Ong (UL) *Mobile Communication and Protection of Children*
11 Adriaan Ter Mors (TUD) *The world according to MARP: Multi-Agent Route Planning*
12 Susan van den Braak (UU) *Sensemaking software for crime analysis*
13 Gianluigi Folino (RUN) *High Performance Data Mining using Bio-inspired techniques*

14 Sander van Splunter (VU) *Automated Web Service Reconfiguration*
15 Lianne Bodenstaff (UT) *Managing Dependency Relations in Inter-Organizational Models*
16 Sicco Verwer (TUD) *Efficient Identification of Timed Automata, theory and practice*
17 Spyros Kotoulas (VU) *Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications*
18 Charlotte Gerritsen (VU) *Caught in the Act: Investigating Crime by Agent-Based Simulation*
19 Henriette Cramer (UvA) *People's Responses to Autonomous and Adaptive Systems*
20 Ivo Swartjes (UT) *Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative*
21 Harold van Heerde (UT) *Privacy-aware data management by means of data degradation*
22 Michiel Hildebrand (CWI) *End-user Support for Access to Heterogeneous Linked Data*
23 Bas Steunebrink (UU) *The Logical Structure of Emotions*
24 Dmytro Tykhonov *Designing Generic and Efficient Negotiation Strategies*
25 Zulfiqar Ali Memon (VU) *Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective*
26 Ying Zhang (CWI) *XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines*
27 Marten Voulon (UL) *Automatisch contracteren*
28 Arne Koopman (UU) *Characteristic Relational Patterns*
29 Stratos Idreos (CWI) *Database Cracking: Towards Auto-tuning Database Kernels*
30 Marieke van Erp (UvT) *Accessing Natural History – Discoveries in data cleaning, structuring, and retrieval*
31 Victor de Boer (UvA) *Ontology Enrichment from Heterogeneous Sources on the Web*
32 Marcel Hiel (UvT) *An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems*
33 Robin Aly (UT) *Modeling Representation Uncertainty in Concept-Based Multimedia Retrieval*
34 Teduh Dirgahayu (UT) *Interaction Design in Service Compositions*
35 Dolf Trieschnigg (UT) *Proof of Concept: Concept-based Biomedical Information Retrieval*
36 Jose Janssen (OU) *Paving the Way for Lifelong Learning: Facilitating competence development through a learning path specification*
37 Niels Lohmann (TU/e) *Correctness of services and their composition*
38 Dirk Fahland (TU/e) *From Scenarios to components*
39 Ghazanfar Farooq Siddiqui (VU) *Integrative modeling of emotions in virtual agents*
40 Mark van Assem (VU) *Converting and Integrating Vocabularies for the Semantic Web*
41 Guillaume Chaslot (UM) *Monte-Carlo Tree Search*
42 Sybren de Kinderen (VU) *Needs-driven service bundling in a multi-supplier setting – the computational e3-service approach*
43 Peter van Kranenburg (UU) *A Computational Approach to Content-Based Retrieval of Folk Song Melodies*
44 Pieter Bellekens (TU/e) *An Approach towards Context-sensitive and User-adapted Access to Heterogeneous Data Sources, Illustrated in the Television Domain*
45 Vasilios Andrikopoulos (UvT) *A theory and model for the evolution of software services*
46 Vincent Pijpers (VU) *e3alignment: Exploring Inter-Organizational Business-ICT Alignment*
47 Chen Li (UT) *Mining Process Model Variants: Challenges, Techniques, Examples*
48 *withdrawn*
49 Jahn-Takeshi Saito (UM) *Solving difficult game positions*
50 Bouke Huurnink (UvA) *Search in Audiovisual Broadcast Archives*
51 Alia Khairia Amin (CWI) *Understanding and supporting information seeking tasks in multiple sources*
52 Peter-Paul van Maanen (VU) *Adaptive Support for Human-Computer Teams: Exploring the Use of Cognitive Models of Trust and Attention*

53 Edgar Meij (UvA) *Combining Concepts and Language Models for Information Access*

2011

01 Botond Cseke (RUN) *Variational Algorithms for Bayesian Inference in Latent Gaussian Models*
02 Nick Tinnemeier (UU) *Organizing Agent Organizations: Syntax and Operational Semantics of an Organization-Oriented Programming Language*
03 Jan Martijn van der Werf (TU/e) *Compositional Design and Verification of Component-Based Information Systems*
04 Hado van Hasselt (UU) *Insights in Reinforcement Learning: Formal analysis and empirical evaluation of temporal-difference learning algorithms*
05 Base van der Raadt (VU) *Enterprise Architecture Coming of Age – Increasing the Performance of an Emerging Discipline*
06 Yiwen Wang (TU/e) *Semantically-Enhanced Recommendations in Cultural Heritage*
07 Yujia Cao (UT) *Multimodal Information Presentation for High Load Human Computer Interaction*
08 Nieske Vergunst (UU) *BDI-based Generation of Robust Task-Oriented Dialogues*
09 Tim de Jong (OU) *Contextualised Mobile Media for Learning*
10 Bart Bogaert (UvT) *Cloud Content Contention*
11 Dhaval Vyas (UT) *Designing for Awareness: An Experience-focused HCI Perspective*
12 Carmen Bratosin (TU/e) *Grid Architecture for Distributed Process Mining*
13 Xiaoyu Mao (UvT) *Airport under Control: Multiagent Scheduling for Airport Ground Handling*
14 Milan Lovric (EUR) *Behavioral Finance and Agent-Based Artificial Markets*
15 Marijn Koolen (UvA) *The Meaning of Structure: The Value of Link Evidence for Information Retrieval*
16 Maarten Schadd (UM) *Selective Search in Games of Different Complexity*
17 Jiyin He (UvA) *Exploring Topic Structure: Coherence, Diversity and Relatedness*
18 Mark Ponsen (UM) *Strategic Decision-Making in complex games*
19 Ellen Rusman (OU) *The Mind ' s Eye on Personal Profiles*
20 Qing Gu (VU) *Guiding service-oriented software engineering – A view-based approach*
21 Linda Terlouw (TUD) *Modularization and Specification of Service-Oriented Systems*
22 Junte Zhang (UvA) *System Evaluation of Archival Description and Access*
23 Wouter Weerkamp (UvA) *Finding People and their Utterances in Social Media*
24 Herwin van Welbergen (UT) *Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior*
25 Syed Waqar ul Qounain Jaffry (VU) *Analysis and Validation of Models for Trust Dynamics*
26 Matthijs Aart Pontier (VU) *Virtual Agents for Human Communication – Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots*
27 Aniel Bhulai (VU) *Dynamic website optimization through autonomous management of design patterns*
28 Rianne Kaptein (UvA) *Effective Focused Retrieval by Exploiting Query Context and Document Structure*
29 Faisal Kamiran (TU/e) *Discrimination-aware Classification*
30 Egon van den Broek (UT) *Affective Signal Processing (ASP): Unraveling the mystery of emotions*
31 Ludo Waltman (EUR) *Computational and Game-Theoretic Approaches for Modeling Bounded Rationality*
32 Nees-Jan van Eck (EUR) *Methodological Advances in Bibliometric Mapping of Science*
33 Tom van der Weide (UU) *Arguing to Motivate Decisions*
34 Paolo Turrini (UU) *Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations*
35 Maaike Harbers (UU) *Explaining Agent Behavior in Virtual Training*

36 Erik van der Spek (UU) *Experiments in serious game design: A cognitive approach*
37 Adriana Burlutiu (RUN) *Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference*
38 Nyree Lemmens (UM) *Bee-inspired Distributed Optimization*
39 Joost Westra (UU) *Organizing Adaptation using Agents in Serious Games*
40 Viktor Clerc (VU) *Architectural Knowledge Management in Global Software Development*
41 Luan Ibraimi (UT) *Cryptographically Enforced Distributed Data Access Control*
42 Michal Sindlar (UU) *Explaining Behavior through Mental State Attribution*
43 Henk van der Schuur (UU) *Process Improvement through Software Operation Knowledge*
44 Boris Reuderink (UT) *Robust Brain-Computer Interfaces*
45 Herman Stehouwer (UvT) *Statistical Language Models for Alternative Sequence Selection*
46 Beibei Hu (TUD) *Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work*
47 Azizi Bin Ab Aziz (VU) *Exploring Computational Models for Intelligent Support of Persons with Depression*
48 Mark Ter Maat (UT) *Response Selection and Turn-taking for a Sensitive Artificial Listening Agent*
49 Andreea Niculescu (UT) *Conversational interfaces for task-oriented spoken dialogues: Design aspects influencing interaction quality*

2012

01 Terry Kakeeto (UvT) *Relationship Marketing for SMEs in Uganda*
02 Muhammad Umair (VU) *Adaptivity, emotion, and Rationality in Human and Ambient Agent Models*
03 Adam Vanya (VU) *Supporting Architecture Evolution by Mining Software Repositories*
04 Jurriaan Souer (UU) *Development of Content Management System-based Web Applications*
05 Marijn Plomp (UU) *Maturing Interorganisational Information Systems*
06 Wolfgang Reinhardt (OU) *Awareness Support for Knowledge Workers in Research Networks*
07 Rianne van Lambalgen (VU) *When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions*
08 Gerben de Vries (UvA) *Kernel Methods for Vessel Trajectories*
09 Ricardo Neisse (UT) *Trust and Privacy Management Support for Context-Aware Service Platforms*
10 David Smits (TU/e) *Towards a Generic Distributed Adaptive Hypermedia Environment*
11 J. C. B. Rantham Prabhakara (TU/e) *Process Mining in the Large: Preprocessing, Discovery, and Diagnostics*
12 Kees van der Sluijs (TU/e) *Model Driven Design and Data Integration in Semantic Web Information Systems*
13 Suleman Shahid (UvT) *Fun and Face: Exploring non-verbal expressions of emotion during playful interactions*
14 Evgeny Knutov (TU/e) *Generic Adaptation Framework for Unifying Adaptive Web-based Systems*
15 Natalie van der Wal (VU) *Social Agents: Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes*
16 Fiemke Both (VU) *Helping people by understanding them – Ambient Agents supporting task execution and depression treatment*
17 Amal Elgammal (UvT) *Towards a Comprehensive Framework for Business Process Compliance*
18 Eltjo Poort (VU) *Improving Solution Architecting Practices*
19 Helen Schonenberg (TU/e) *What's Next? Operational Support for Business Process Execution*
20 Ali Bahramisharif (RUN) *Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing*
21 Roberto Cornacchia (TUD) *Querying Sparse Matrices for Information Retrieval*

22 Thijs Vis (UvT) *Intelligence, politie en veiligheidsdienst: Verenigbare grootheden?*
23 Christian Muehl (UT) *Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction*
24 Laurens van der Werff (UT) *Evaluation of Noisy Transcripts for Spoken Document Retrieval*
25 Silja Eckartz (UT) *Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application*
26 Emile de Maat (UvA) *Making Sense of Legal Text*
27 Hayrettin Gurkok (UT) *Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games*
28 Nancy Pascall (UvT) *Engendering Technology Empowering Women*
29 Almer Tigelaar (UT) *Peer-to-Peer Information Retrieval*
30 Alina Pommeranz (TUD) *Designing Human-Centered Systems for Reflective Decision Making*
31 Emily Bagarukayo (RUN) *A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure*
32 Wietske Visser (TUD) *Qualitative multi-criteria preference representation and reasoning*
33 Rory Sie (OU) *Coalitions in Cooperation Networks (COCOON)*
34 Pavol Jancura (RUN) *Evolutionary analysis in PPI networks and applications*
35 Evert Haasdijk (VU) *Never Too Old To Learn – On-line Evolution of Controllers in Swarm and Modular Robotics*
36 Denis Ssebugwawo (RUN) *Analysis and Evaluation of Collaborative Modeling Processes*
37 Agnes Nakakawa (RUN) *A Collaboration Process for Enterprise Architecture Creation*
38 Selmar Smit (VU) *Parameter Tuning and Scientific Testing in Evolutionary Algorithms*
39 Hassan Fatemi (UT) *Risk-aware design of value and coordination networks*
40 Agus Gunawan (UvT) *Information Access for SMEs in Indonesia*
41 Sebastian Kelle (OU) *Game Design Patterns for Learning*
42 Dominique Verpoorten (OU) *Reflection Amplifiers in self-regulated Learning*
43 *withdrawn*
44 Anna Tordai (VU) *On Combining Alignment Techniques*
45 Benedikt Kratz (UvT) *A Model and Language for Business-aware Transactions*
46 Simon Carter (UvA) *Exploration and Exploitation of Multilingual Data for Statistical Machine Translation*
47 Manos Tsagkias (UvA) *Mining Social Media: Tracking Content and Predicting Behavior*
48 Jorn Bakker (TU/e) *Handling Abrupt Changes in Evolving Time-series Data*
49 Michael Kaisers (UM) *Learning against Learning – Evolutionary dynamics of reinforcement learning algorithms in strategic interactions*
50 Steven van Kervel (TUD) *Ontologogy driven Enterprise Information Systems Engineering*
51 Jeroen de Jong (TUD) *Heuristics in Dynamic Sceduling: A practical framework with a case study in elevator dispatching*

2013

01 Viorel Milea (EUR) *News Analytics for Financial Decision Support*
02 Erietta Liarou (CWI) *MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing*
03 Szymon Klarman (VU) *Reasoning with Contexts in Description Logics*
04 Chetan Yadati (TUD) *Coordinating autonomous planning and scheduling*
05 Dulce Pumareja (UT) *Groupware Requirements Evolutions Patterns*
06 Romulo Goncalves (CWI) *The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience*

07 Giel van Lankveld (UvT) *Quantifying Individual Player Differences*
08 Robbert-Jan Merk (VU) *Making enemies: Cognitive modeling for opponent agents in fighter pilot simulators*
09 Fabio Gori (RUN) *Metagenomic Data Analysis: Computational Methods and Applications*
10 Jeewanie Jayasinghe Arachchige (UvT) *A Unified Modeling Framework for Service Design*
11 Evangelos Pournaras (TUD) *Multi-level Reconfigurable Self-organization in Overlay Services*
12 Marian Razavian (VU) *Knowledge-driven Migration to Services*
13 Mohammad Safiri (UT) *Service Tailoring: User-centric creation of integrated IT-based homecare services to support independent living of elderly*
14 Jafar Tanha (UvA) *Ensemble Approaches to Semi-Supervised Learning Learning*
15 Daniel Hennes (UM) *Multiagent Learning – Dynamic Games and Applications*
16 Eric Kok (UU) *Exploring the practical benefits of argumentation in multi-agent deliberation*
17 Koen Kok (VU) *The PowerMatcher: Smart Coordination for the Smart Electricity Grid*
18 Jeroen Janssens (UvT) *Outlier Selection and One-Class Classification*
19 Renze Steenhuizen (TUD) *Coordinated Multi-Agent Planning and Scheduling*
20 Katja Hofmann (UvA) *Fast and Reliable Online Learning to Rank for Information Retrieval*
21 Sander Wubben (UvT) *Text-to-text generation by monolingual machine translation*
22 Tom Claassen (RUN) *Causal Discovery and Logic*
23 Patricio de Alencar Silva (UvT) *Value Activity Monitoring*
24 Haitham Bou Ammar (UM) *Automated Transfer in Reinforcement Learning*
25 Agnieszka Anna Latoszek-Berendsen (UM) *Intention-based Decision Support: A new way of representing and implementing clinical guidelines in a Decision Support System*
26 Alireza Zarghami (UT) *Architectural Support for Dynamic Homecare Service Provisioning*
27 Mohammad Huq (UT) *Inference-based Framework Managing Data Provenance*
28 Frans van der Sluis (UT) *When Complexity becomes Interesting: An Inquiry into the Information eXperience*
29 Iwan de Kok (UT) *Listening Heads*
30 Joyce Nakatumba (TU/e) *Resource-Aware Business Process Management: Analysis and Support*
31 Dinh Khoa Nguyen (UvT) *Blueprint Model and Language for Engineering Cloud Applications*
32 Kamakshi Rajagopal (OU) *Networking For Learning: The role of Networking in a Lifelong Learner's Professional Development*
33 Qi Gao (TUD) *User Modeling and Personalization in the Microblogging Sphere*
34 Kien Tjin-Kam-Jet (UT) *Distributed Deep Web Search*
35 Abdallah El Ali (UvA) *Minimal Mobile Human Computer Interaction*
36 Than Lam Hoang (TU/e) *Pattern Mining in Data Streams*
37 Dirk Börner (OU) *Ambient Learning Displays*
38 Eelco den Heijer (VU) *Autonomous Evolutionary Art*
39 Joop de Jong (TUD) *A Method for Enterprise Ontology based Design of Enterprise Information Systems*
40 Pim Nijssen (UM) *Monte-Carlo Tree Search for Multi-Player Games*
41 Jochem Liem (UvA) *Supporting the Conceptual Modelling of Dynamic Systems: A Knowledge Engineering Perspective on Qualitative Reasoning*
42 Léon Planken (TUD) *Algorithms for Simple Temporal Reasoning*

A magician wandered along the beach, but no one needed him. ⟨+⟩