Towards a Dynamic Algorithm for the Simple Temporal Problem

Jan Otto Arie ten Thije

Towards a Dynamic Algorithm for the Simple Temporal Problem

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jan Otto Arie ten Thije
born in Amsterdam, the Netherlands

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

Towards a Dynamic Algorithm for the Simple Temporal Problem

Author: Jan Otto Arie ten Thije
Student id: [redacted]
Email: ottenthije@gmail.com

Abstract

Efficient management and propagation of temporal constraints is important for temporal planning as well as for scheduling. During plan development, many solvers employ a heuristic-driven backtracking approach, over the course of which they maintain a so-called Simple Temporal Network (STN) of events and constraints.

Recent research has shown that partial path consistency (PPC) can be used to efficiently propagate temporal information in such networks. This insight was applied in the IPPC algorithm, which enforces PPC in an incremental fashion when new constraints are introduced.

We present two new algorithms that efficiently enforce PPC in modified STNs. *Vertex-IPPC* allows the incremental introduction of an event and all its associated constraints at once. Conversely, *Support-DPPC* allows the removal or loosening of existing constraints. To the best of our knowledge, this is the first *decremental* algorithm for enforcing PPC.

Our ultimate goal is a fully dynamic algorithm for PPC, supporting on-line deletions as well as additions. This will allow solvers to efficiently explore the solution space, rather than solving entire networks after each update.

Thesis Committee:

Chair:	Prof. dr. C. Witteveen, Faculty EEMCS, TU Delft
Supervisor:	Dr. M. M. de Weerdt, Faculty EEMCS, TU Delft
Member:	Dr. D. de Ridder, Faculty EEMCS, TU Delft
Member:	L. R. Planken, Faculty EEMCS, TU Delft

Acknowledgements

First of all I would like to thank my parents, for supporting me and for always standing by with encouragement and good advice. Most relevant in the current context is perhaps their advice to just bite the bullet and get started with my thesis work already, without which my graduation might have been a bit longer in the making. My sincere thanks also go out to my brothers, for providing diversions when I was home and for their words of encouragement during the final push to get everything done.

I would like to thank my lunch mates in Delft for putting up with my sometimes incessant blathering about clique trees, support graphs and related terminology, which I am sure must have annoyed *someone* after two weeks of nothing else. On a more serious note, I am grateful for their reviews of my work and also their more general input on problems I encountered over the course of my thesis work. A special thank you goes out to Tim van Heugten, who was always willing to brainstorm when I got stuck.

Thanks also go out to Cees Witteveen, for his always enjoyable visits to the graduates' room and his feedback on my intermediate presentation, and to Dick de Ridder for taking the time to read this report and completing my thesis committee.

Last – but most certainly not least – I would like to thank my supervisors Mathijs de Weerdt and Leon Planken for their great feedback and guidance, the many excellent brainstorm sessions and our in-depth discussions that ultimately led to this document.

Ot ten Thije
Delft, August 2011

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Schedules of all kinds greatly affect our daily lives, be it train schedules, time tables used in schools and universities, or even our own working hours. While a schedule itself is usually a fairly straightforward list showing at what time specific events should take place, *creating* these schedules is a far from trivial task in general.

From a computer science perspective, schedules are solutions to problems from a general class of so-called *Temporal Constraint Satisfaction Problems*, first formally described by Dechter et al. (1991). "Temporal" since we aim to assign events a point in time, "Constraint Satisfaction" since the resulting schedule needs to obey certain constraints to make sense in the real world. In a railroad setting for example, if two trains use the same platform, the departure of train one should occur before the arrival of train two.

Many scheduling problems involve *choices*, meaning a solution is fine as long as it satisfies one of a number of possible constraints. In the train example, it may be that a train can arrive at a station either between 14:00 and 14:15 or between 14:30 and 14:45, but not from 14:15 to 14:30, because at that time another train occupies the platform. Scheduling problems with choices like these are called *Disjunctive Temporal Problems* (DTP), introduced by Stergiou and Koubarakis (2000).

Perhaps counter-intuitively, the fact that the DTP allows these choices can make finding a schedule *more* difficult. The problem is that we can combine the various options in an exponential number of ways, but we do not know beforehand whether any of these combinations allows a valid schedule. Algorithms solving DTP instances therefore fall back on a basic back-tracking approach. They exploit the fact that if there are *no* choices, the DTP reduces to a *Simple* Temporal Problem (STP), which *can* be solved efficiently.

Specifically, DTP solvers start with an unconstrained STP instance. They then use heuristics to select one option for a constraint from the DTP and add it to the STP instance, creating a simplified version of the original problem. The solver then checks if there is a solution to this STP, and, if so, simply includes an option for another constraint. It continues doing so until we either have a solution satisfying

one option for every constraint in the original DTP, or find that our STP becomes *inconsistent*, i.e. no longer has a solution.

If the schedule is inconsistent, the DTP solver records this and updates its heuristics. It then takes a step *back* by removing a constraint, and starts over, using its updated heuristics to select a different constraint. Taking a step back, however, is not as simple as it might seem at first glance. While current DTP solvers can efficiently *add* constraints to the schedule and check if any valid schedules remain, they can not do the same when a constraint is *removed*. Instead, they may have to re-check the entire STP from scratch.

In this thesis, we present two new algorithms for dynamically updating STP instances. The first algorithm allows us to efficiently update an existing STP instance to include an entire new *event* and all its adjacent constraints, rather than just a single *constraint* as previous algorithms have done. Our second algorithm addresses the problem described above: it can be used to efficiently remove or loosen a constraint in an existing STP instance.

The remainder of this chapter provides an overview of the contents in this thesis. In Chapter 2 we formally define the Simple Temporal Problem and review the concept of chordal graphs, which is essential for the algorithms that form the current state of the art for solving these problems. We review these and other existing approaches to solving the STP in Chapter 3.

Having introduced the context of our work, we present our main contributions in Chapters 4 and 5. Chapter 4 describes *Vertex-IPPC*, our new algorithm for incrementally solving the STP. In Chapter 5 we turn our attention to the problem of loosening or removing constraints, and present our algorithm *Support-DPPC*, which performs this task efficiently.

Having demonstrated the theoretical correctness of our algorithms, we perform a limited empirical evaluation of Vertex-IPPC in Chapter 6. Finally, in Chapter 7 we conclude and provide interesting directions for future work.

Chapter 2

Problem definition

Having sketched the broader context in which the Simple Temporal Problem (STP) finds its use in Chapter 1, we now turn to consider this problem itself more closely.

We start by providing a more specific practical use case for the STP in Section 2.1. Using this example as an illustration, we formally define the STP in Section 2.2, and introduce our notation. In this section we also note that there is more than one way to solve an STP, and proceed to introduce thee different kinds of solutions.

Since most algorithms in this thesis operate on graphs, we specify our notation for general graph properties in Section 2.3. We conclude with an introduction to the class of *chordal graphs* in Section 2.4, and discuss the properties that make them vital tools in approaches for solving the STP.

2.1 Example

Suppose we are tasked with finding a schedule for arrivals and departures of trains in a railway station. In a particular part of the afternoon schedule, we need to handle two trains, under the following conditions:

The first train is an express train arriving between 16:15 and 16:30 and can stay in the station for only 5 to 10 minutes before departing again. Meanwhile, train two is a local train, so we are allowed to hold it at the platform for 5 to 20 minutes, but it has to depart between 16:00 and 16:20. Finally, we want to allow passengers to switch from the local into the express train. To keep their travel time short, we do not want to hold them up for more than 15 minutes. On the other hand, they should not have to run, so we want to give them at least 10 minutes to change platforms.

To help reason about a possible solution, we can capture the essence of this story in a graph. First, we create a node for every time variable. For example, let us say that the arrival and departure times of train one are a_1 and d_1 respectively, and a_2 and d_2 represent the arrival and departure of train two. We also add an additional variable x_0, called the *temporal reference point*, which allows us to convert the relative values in our schedule to absolute time. In this example, we fix x_0 at 16:00. Finally, we

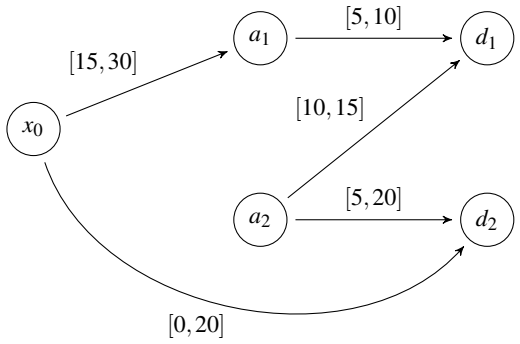


Figure 2.1: Example of a Simple Temporal Network.

represent the constraints as edges connecting events. For our story, this yields a network like the one shown in Figure 2.1.

To check whether a schedule is valid, we can now simply assign its values to the time points and see if they satisfy all constraints. For example, the reader can verify that $(x_0 = 0, a_1 = 15, d_1 = 20, a_2 = 5, d_2 = 10)$ is a valid schedule.

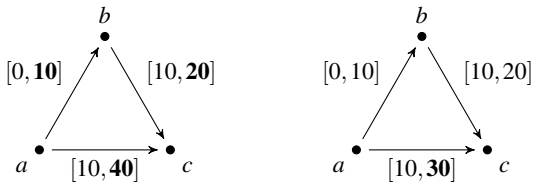
The network just presented is called a *Simple Temporal Network* (STN), which models an instance from the class of *Simple Temporal Problems* (STP).In a seminal paper, Dechter et al. (1991) introduced the STP as an important subclass of the more general class of constraint satisfaction problems.

The "Temporal" part of these names stems from the fact that they concern constraints between events in time: temporal constraints. Furthermore, they are "Simple" in the sense that the constraints can be expressed as a binary relation between two time variables, which is not generally the case in constraint satisfaction problems. In spite of this apparent simplicity there are many practical applications for the STP, one of which we have just seen. Having intuitively established the usefulness of STNs, we further formalize them in the next section.

2.2 Simple Temporal Networks

A Simple Temporal Network $\mathcal{S}$ consists of a set $X = \{x_1, x_2, \ldots, x_n\}$ containing *n events* and a set C containing *m constraints*. Each constraint $c_{i\to j} \in C$ has a *weight* $w_{i\to j}$, which represents an upper bound on the time that may elapse between events x_i and x_j. A constraint can therefore also be interpreted as the inequality $x_j - x_i \leq w_{i\to j}$. Multiplying both sides of this equation by -1 we have $x_i - x_j \geq -w_{i\to j}$, so $-w_{i\to j}$ is also the *lower* bound on the time difference between x_j and x_i.

As demonstrated in the example, STNs can be naturally modelled as directed graphs. An event x_i is represented as a node, and for each constraint $c_{i\to j}$ there is a directed edge from node x_i to node x_j, labelled by $w_{i\to j}$. Alternatively, both $c_{i\to j}$ and



(a) Original STN **(b)** Tightened STN

Figure 2.2: Example of constraints implying a stricter value for another constraint. The combination of constraints $c_{a\to b}$ and $c_{b\to c}$ implies a tighter bound on $c_{a\to c}$ than originally specified.

$c_{j\to i}$ can be shown as a single directed edge from x_i to x_j. This edge is then labelled by $[-w_{j\to i}, w_{i\to j}]$, the exact range which the time difference $x_j - x_i$ is constrained to. The latter notation is used in Figure 2.1.

2.2.1 Solutions

A simple way to define the solution of an STN is the one we mentioned in our introductory example: a schedule satisfying all constraints specified by the STN instance. Formally, such a solution is a *schedule* τ, which assigns a moment in time to every event in X, such that all constraints in C are satisfied. If no such schedule exists, the network is said to be *inconsistent*.

However, we can generalize this notion. In particular, provided an STN is consistent, there are usually many different schedules that will satisfy all constraints. For example, $(x_0 = 0, a_1 = 20, d_1 = 25, a_2 = 10, d_2 = 15)$ is also a valid solution for the STN in Figure 2.1. It would therefore be interesting to have a representation of *all* valid schedules.

Generally, an unprocessed STN will contain *implicit* information about constraints. Figure 2.2 illustrates this phenomenon. Here, event c can occur at most 20 minutes after event b, which in turn can occur at most 10 minutes after a. Therefore, c can occur at most 30 minutes after a, which is stricter than the upper limit of 40 specified in the original STN.

Dechter et al. (1991) showed that if the network is a complete graph and all constraints are as tight as possible, any valid schedule can be extracted efficiently using a backtrack-free algorithm. Hence, this tight network can be considered a solution to an STN as well.

In fact, this new solution gives rise to a number of new questions we can ask ourselves regarding tight STNs. Recall that the STN instances we solve are ultimately intermediate steps generated by a DTP solver. Typically, instances from consecutive steps will differ in only a single constraint, leaving the rest of the network untouched. It seems wasteful to re-solve an instance from scratch, when we already have a tight network at our disposal, which may only require a small change to conform to the updated constraint. We therefore distinguish three types of problems regarding *updates* on STNs.

Definition 2.1. Given a consistent, tight STN $\mathcal{S}$ and a new weight $w'_{a\to b}$ for a constraint $c_{a\to b}$, with the original weight $w_{a\to b}$. We distinguish three types of problems regarding the STN $\mathcal{S}'$ incorporating $w'_{a\to b}$.

- INCREMENTAL-STP: compute tight constraints in $\mathcal{S}'$, given that $w'_{a\to b} < w_{a\to b}$, or return INCONSISTENT if the new weight makes the problem unsolvable.
- DECREMENTAL-STP: compute tight constraints in $\mathcal{S}'$ given that $w'_{a\to b} > w_{a\to b}$.
- DYNAMIC-STP: compute tight constraints in $\mathcal{S}$ without restrictions on $w'_{a\to b}$, returning INCONSISTENT if the new weight makes the problem unsolvable.

Note that the weight of $c_{a\to b}$ may have been infinite, in which case lowering $w_{a\to b}$ is equivalent to adding an entirely new constraint to the graph. Conversely, raising $w_{a\to b}$ to infinity is equivalent to removing the constraint from the graph.

This is the reason for the perhaps somewhat confusing naming of the incremental and decremental problems: "Incremental" does not refer to a weight increase, but to the fact this method can be used to *increase the complexity* of an STN, by adding new constraints or further restricting the possible values of existing ones. In the same vein "Decremental" algorithms can be use to *decrease* the complexity of the problem, by loosening existing constraints or removing them altogether.

2.2.2 Intermediate STNs

It turns out that there are multiple variants of the "tight" STNs discussed in the previous section, each of which can be used as input for the greedy algorithm to construct a schedule. These variants are sub-classes of a more general class of *Intermediate* STNs (ISTNs), surveyed by Planken et al. (2011b). ISTNs explicitly record the information that can be inferred by tightening constraints.

Planken et al. identify five different types of ISTNs, that differ in how much implicit information they make explicit. Since our interest for the STN originates in solvers for the DTP, we are particularly interested in ISTNs that contain the information these solver require. In particular, they need to compute the space of possible schedules, which is captured by the tight STNs just discussed. Two of the five ISTN types can immediately be used as input for Dechter et al.'s greedy schedule construction algorithm, so we restrict our discussion to these two types.

1. A *Fully Path Consistent* STN, is a complete graph that contains the tightest possible constraints between any two events in the schedule. This ISTN corresponds to the notion of a *minimal network* from constraint literature. Figure 2.3 shows the minimal network corresponding to the STN shown in Figure 2.1.
2. A *Partially Path Consistent* STN, first specified by Bliek and Sam-Haroud (1999), is defined only for so-called *chordal graphs*. It contains the tightest

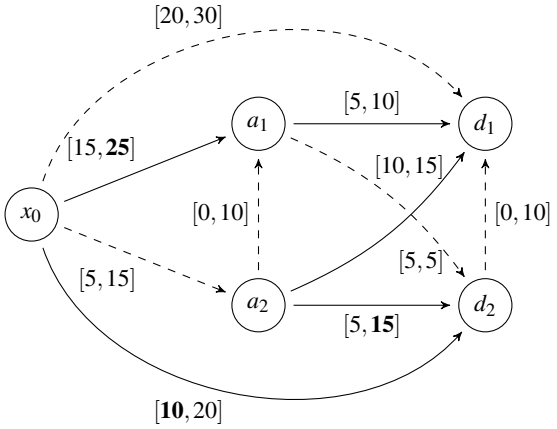


Figure 2.3: A Fully Path Consistent STN equivalent to the STN $\mathcal{S}$ from Figure 2.1. Dashed constraints were added to make the graph complete, constraints highlighted in bold were tightened using implicit information from $\mathcal{S}$.

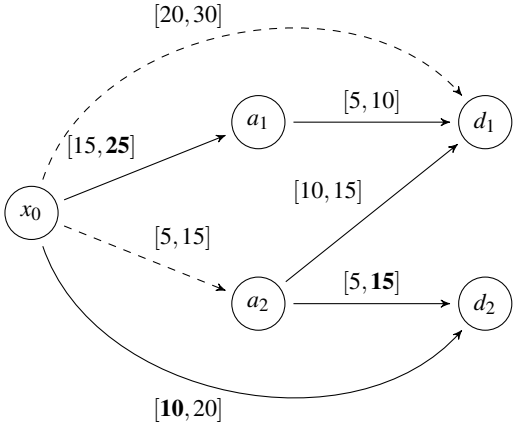


Figure 2.4: A Partially Path Consistent STN equivalent to the STN $\mathcal{S}$ from Figure 2.1. Dashed constraints were added to make the graph chordal, constraints highlighted in bold were tightened using implicit information from $\mathcal{S}$.

possible constraints as well, but only between events that are directly connected by an edge. Figure 2.3 shows a Partially Path Consistent equivalent of the STN from our example.

As illustrated by Figures 2.3 and 2.4, chordal graphs are in general far sparser than complete graphs on the same number of vertices, so it seems reasonable that enforcing partial path consistency would be less costly than enforcing full path consistency.

Since algorithms enforcing Partial Path Consistency rely heavily on special properties of chordal graphs, we discuss them in more detail in Section 2.4. Before doing so however, we shortly specify our notation for general properties of graphs in Section 2.3.

2.3 Notation for graph properties

A graph $G = \langle V, E \rangle$ consists of $|V| = n$ vertices and $|E| = m$ edges. Each edge is a pair of two vertices $u, v \in V$. For *directed* graphs, these are ordered pairs, denoted (u, v). In *undirected* graphs, the pairs are unordered and we use braces instead of parentheses, i.e. $\{u, v\}$.

In an undirected graph the *neighbours* of a vertex u are the vertices v such that there is an edge between u and v in E. The *neighbourhood* $N(u)$ of u is the set of these vertices, i.e. $N(u) = \{v \mid \{u, v\} \in E\}$. The *degree* $\delta(u)$ of the vertex u is the size of its neighbourhood, i.e. $\delta(u) = |N(u)|$. If no argument is given, δ refers to the *degree of the graph*, which is the size of the largest neighbourhood in G, i.e. $\delta = \max_{v \in V} \delta(v)$.

For a directed graph, we define the *out-neighbourhood* $N_{out}(u)$ and *in-neighbourhood* $N_{in}(u)$, where $N_{\text{out}}(u) = \{v \mid (u, v) \in E\}$ and $N_{\text{in}}(u) = \{v \mid (v, u) \in E\}$. As with undirected graphs, the *in-degree* $\delta_{in}(u)$ and *out-degree* $\delta_{out}(u)$ of vertex u are defined as the size of the in- and out neighbourhood respectively.

Finally, given a graph $G = \langle V, E \rangle$ and a subset $V' \subseteq V$, the *induced graph* $G_{V'}$ is the subgraph of G obtained by removing all vertices $v \notin V'$ and their adjacent edges. Formally, we define $G_{V'} = \langle V', \{\{u, v\} \in E \mid u, v \in V'\} \rangle$.

2.4 Chordal graphs

Many recent advances in the state of the art for STN solving algorithms were made by exploiting specific properties of chordal graphs (e.g. Planken et al. (2008, 2011a); Xu and Choueiry (2003)). Given the importance of chordal graphs to these algorithms, we now explore this concept in more detail.

After formally defining what makes a graph chordal in Section 2.4.1, we discuss the two properties most relevant for STN algorithms: *clique trees* in Section 2.4.2 and *simplicial elimination orderings* in Section 2.4.3. We conclude in Section 2.4.4

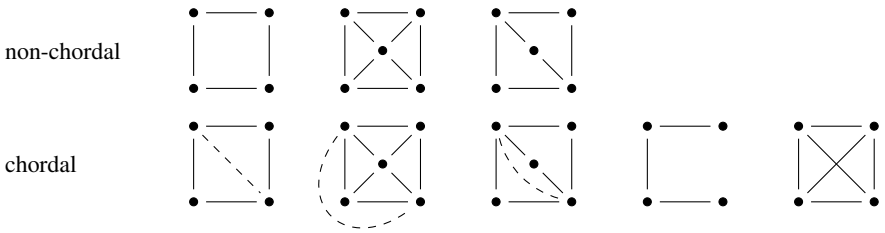


Figure 2.5: Examples of chordal and non-chordal graphs. Dashed lines represent *fill edges* added to the upper graphs to make them chordal.

with remarks on methods to obtain a chordal equivalent of an arbitrary STN, which make algorithms for chordal STNs more generally applicable.

2.4.1 Definition

The defining property of a chordal graph is that for every cycle of length greater than three, there is an edge between two non-adjacent vertices on that cycle. We can formally specify this as follows:

Definition 2.2. Let $G = \langle V, E \rangle$ be an undirected graph. If $C = (v_1, v_2, \ldots, v_k = v_1)$ with $k > 3$ is a cycle, then any edge $\{v_i, v_j\}$ with $1 < j - i < k - 1$ is a *chord* of this cycle. If every cycle of G with length greater than three has a chord, G is a *chordal graph*.

Definition 2.2 implies that the longest chordless cycles in a chordal graph have length three, or in other words, that they are triangles. Chordal graphs are therefore also referred to as *triangulated* graphs.[1] Figure 2.5 gives some examples of chordal and non-chordal graphs.

2.4.2 Clique trees

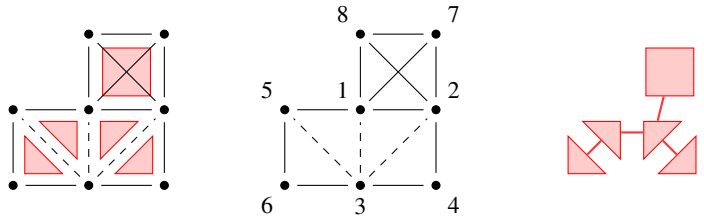
Most people know a "clique" as a tightly knit group of friends, all of which know each other. The graph-theoretical clique is not much different, if slightly more abstract. Here, a clique is a group of vertices that are all directly connected to each other. Formally:

Definition 2.3. A *clique* in a graph $G = (V, E)$ is set of vertices $K \subseteq V$ such that $\{u, v\} \in E$ for all $u, v \in K$ (with $u \neq v$). A clique K of G is *maximal* if there is no other clique K' in G such that $K \subset K'$.

As we will see, chordal graphs are closely related to the notion of cliques. One way in which this relationship becomes apparent is through the notion of *clique trees*.

A clique tree is an additional structure beyond the normal edges and vertices making up a graph. Each node in a clique tree represents a clique of vertices in

[1] The term "triangulated" is also used in context of planar graphs, which do not concern us here.



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

Figure 2.6: Characterizations of a chordal graph. Edges of the original graph are drawn as solid lines, fill edges are dashed. Each shape shaded red represents a maximal clique of the graph, containing the vertices at its corners.

the underlying graph, and if two nodes are connected by a clique edge, the vertex sets they represent overlap. We capture the defining properties of a clique tree in Definition 2.4.

Definition 2.4. A tree $T = (K, F)$ is a *maximal clique tree* of a graph $G = (V, E)$ if it satisfies the following conditions:

1. *Vertex coverage*: Every vertex $v \in V$ is contained in at least one clique $C \in K$.
2. *Edge coverage*: For every edge $\{u, v\} \in E$ there is a clique $C \in K$ such that both $u \in C$ and $v \in C$.
3. *Coherence*: If two cliques C_1 and C_2 in K both contain a vertex v, then all nodes on the (unique) path between C_1 and C_2 in T contain v as well.
4. *Clique maximality*: Every node $C \in K$ represents a maximal clique of G.

Figure 2.6c illustrates how a clique tree relates to the underlying graph. Note that in general clique trees are not unique. Since vertices can be part of multiple cliques, there can be instances where many cliques overlap. In such cases we can choose which of these overlaps to represent by an edge, provided we do not break coherence.

Intuitively, clique trees are useful because they introduce a notion of *locality*. They allow us to decompose a graph into several subgraphs, each of which has the special property that it is a clique. Algorithms can exploit this locality by first decomposing the graph, then performing operations on each individual subgraph and finally merging the results.

The following theorem allows us to exploit this notion for chordal graphs:

Theorem 2.5 (Buneman (1974); Gavril (1974); Walter (1972))**.** *A graph is chordal if and only if it has a maximal clique tree.*

2.4.3 Simplicial elimination orderings

Clique trees can be seen as a high-level property of chordal graphs: the tree provides a generalized structure, allowing us to consider entire cliques at once rather than every vertex and edge individually. However, since we are ultimately interested in properties of individual edges, it is useful to have a lower level view as well. Such a low-level perspective is provided by the notion of *simplicial elimination orderings*.

Simplicial elimination orderings are based on an interesting property of chordal graphs. When we eliminate vertices from a chordal graph one by one in a particular order, we can always find at least two vertices whose remaining neighbours induce a clique. Formally, we define this as follows.

Definition 2.6. Given a graph $G = \langle V, E \rangle$, we can define the following concepts:

- A vertex v is *simplicial* if its neighbouring vertices $N(v) = \{u \mid \{u, v\} \in E\}$ induce a clique.
- Let $d = (v_n, v_{n-1}, \ldots, v_1)$ be an ordering of V and let G_k be the graph induced by the vertices $\{v_1, \ldots, v_k\}$. If every vertex v_k is simplicial in G_k, d is a *simplicial elimination ordering* of G, and its reverse $d' = (v_1, v_2, \ldots, v_n)$ is a simplicial *construction* ordering of G.
- If $d = (v_n, v_{n-1}, \ldots v_1)$ is a simplicial elimination ordering of G, the *induced width* w_d of d is the number of neighbours in the largest induced clique, i.e.:

$$w_d = \max_i |\{\{v_i, v_j\} \in E \mid j < i\}|$$

Simplical orderings are particularly useful because they provide a natural way to exploit the chordality of a graph when iterating over its vertices. The fact that the neighbourhood of each vertex in a simplicial construction ordering induces a clique when it is added the graph makes such an ordering very useful in combination with induction proofs.

For chordal graphs, the induced width w_d has the same value along any simplicial elimination ordering d, and is exactly equal to the graph's *tree width* w^*. The tree width is a measure for the "tree-likeness" of the graph: graphs of low tree width can be decomposed into many small cliques, mutually connected in a tree-like structure. In particular, if $w^* = 1$ the graph must be a tree, in which all cliques are of size two. Graphs with the same number of vertices but higher tree widths consist of fewer, but larger cliques, and the tree they span must therefore be less intricate.

The dependency on the ordering is relevant for the process of triangulation, which computes a chordal graph that is similar in structure to general graph. We will discuss this in more detail in Section 2.4.4. The induced width also provides a numeric property of a graph which crops up frequently in run-time bounds of algorithms based on elimination orderings.

As with clique trees, it has been shown that the existence of a simplicial elimination ordering is a necessary and sufficient proof of graph chordality:

Theorem 2.7 (Fulkerson (1964))**.** *A graph is chordal if and only if it has a simplicial elimination ordering.*

Note that just as a chordal graph may have many clique trees, it may have different simplicial elimination orderings. For example, while $(8,7,6,5,4,3,2,1)$ is a valid elimination ordering of the graph shown in Figure 2.6b, so are $(6,5,7,8,4,3,2,1)$ and $(4,8,6,5,3,1,2,7)$. However, $(8,7,1,6,5,4,3,2)$ is *not* a valid simplicial elimination ordering. When vertices 8 and 7 are eliminated, the set of remaining neighbours of vertex 1 is $\{5,3,2\}$, but this does not induce a clique: there is no edge between vertices 2 and 5.

There are various algorithms that produce a simplicial elimination ordering of a chordal graph. They can be implemented to run in time $O(m_c)$, where m_c is the number of edges in the chordal graph. In Section 4.2 we will discuss one such algorithm in more detail.

2.4.4 Triangulation

Given the properties just discussed, it would be useful if we could transform problems on arbitrary graphs into equivalent problems on chordal graphs. An important tool for such transformations is the notion of a *triangulation*, which produces a chordal graph that is structurally similar to an arbitrary input graph. This can be achieved in two ways: we can either add so-called *fill edges* to the original graph, which serve as chords for any chordless cycle, or we can remove edges from the graph until no more chordless cycles remain.

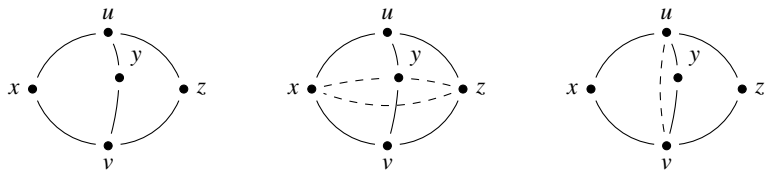
In our case edges represent constraints, and deleting edges is equivalent to ignoring constraints. Since we want our final solution to obey all constraints, ignoring them is not an option. We will therefore only consider adding fill edges, which can be formally defined as follows:

Definition 2.8. A *triangulation* G' of the graph $G = (V,E)$ adds a set of *fill edges* F such that $G' = (V, E \cup F)$ is chordal, where $F \cap E = \varnothing$.

To differentiate between the properties of the original graph and those of its chordal counterparts, the latter are assigned an (additional) subscript c. For example, the number of edges in the chordal graph is m_c, and the degree of vertex u in a chordal graph is $\delta_c(u)$.

Minimum and minimal triangulations

Triangulations are not unique: there are many ways in which chords can be added to all chordless cycles. A trivial solution is to simply add all possible edges, and return a complete graph: a complete graph has no chordless cycles and is therefore chordal.



(a) Original graph **(b)** Minimal triangulation **(c)** Minimum triangulation

Figure 2.7: Examples of triangulation classes. Edges from the original graph are drawn solid, fill edges are dashed.

However, generally we would like to be more clever and add as few edges as possible. Ideally, we want to find a triangulation such that no other triangulation achieves chordality using fewer edges. Such a triangulation is called a *minimum* triangulation. Formally, we say:

Definition 2.9. A triangulation $G = (V, E \cup F)$ is a *minimum triangulation* if there is no other triangulation $G' = (V, E \cup F')$ such that $|F'| < |F|$.

Unfortunately, it turns out that the problem of finding a minimum triangulation for an arbitrary graph is NP-hard (Yannakakis, 1981). However, there is a weaker condition on triangulations that can be enforced in polynomial time. We say that a triangulation is *minimal* if no strict subset of the fill edges it adds would suffice to triangulate the graph. Formally:

Definition 2.10. A triangulation $G = (V, E \cup F)$ is a *minimal triangulation* if there is no other triangulation $G' = (V, E \cup F')$ such that $F' \subset F$.

We illustrate the difference between minimal and minimum triangulations in the following example.

Example 2.11. Consider the graph in Figure 2.7a. This graph contains three cycles of length four: (u, x, v, y, u), (u, x, v, z, u) and (u, y, v, z, u).

One way to break these cycles is to insert the edges $\{x, y\}$, $\{y, z\}$ and $\{x, z\}$, as shown in Figure 2.7b. Note that each edge serves as a chord for exactly one of the three cycles, so removing any of them will leave a 4-cycle without chord. Therefore this is a minimal triangulation, of size three.

However, it is not a *minimum* triangulation. As shown in Figure 2.7c, all three cycles can also be broken by inserting just *one* edge: $\{u, v\}$. Since we need to add at least one edge, we cannot improve over this solution, and it is therefore a minimum triangulation. □

On a related note, Example 2.11 illustrates a pattern that can be used to show that a minimal triangulation may use $O(n^2)$ more edges than the minimum triangulation. We create a graph consisting of two vertices u and v, and $n-2$ vertices $w_1, \ldots, w_{n-2}$. These vertices w_i are connected to u and v with the edges $\{\{u, w_i\}, \{w_i, v\}\}$.

One possible minimal triangulation of this graph is obtained by adding fill edges until the subgraph consisting of all w_i is complete, i.e. we add $\Theta(n^2)$ edges. However, the minimum triangulation of just inserting the one edge $\{u, v\}$ is also valid, so the difference between minimal and maximal triangulations can indeed be $O(n^2)$.

Triangulation algorithms

There are a number of algorithms that produce minimal triangulations, which are excellently surveyed by Heggernes (2006). The best run time bounds for these algorithms are $O(nm)$ and $O(n^{2.69})$.

Ultimately, these algorithms either visit the vertices in the graph in a particular ordering d, or traverse the clique tree. Algorithms using the ordering-based approach add fill edges such that this order becomes a simplicial elimination or construction ordering. In other words, for each vertex v they add fill edges such that the vertices in the neighbourhood of v that come after it in the chosen ordering d induce a clique. Since the number of vertices following v depends on d, different triangulation methods may yield different *induced* tree widths w_d.

Algorithms based on clique trees construct an incomplete tree and then add fill edges such that each node represents a clique. Here the choice of the original incomplete tree determines how large the largest clique is, and therefore what the *value* of the tree width w_d becomes. In Section 4.6 we will review one particular technique, by Berry et al. (2006), in more detail.

If we let go of the condition of minimality, we can fall back on heuristics that will produce a reasonable *approximation* of a minimal triangulation, i.e. include some redundant fill edges. We shortly discuss two of these that are frequently used.

The *minimum-fill* heuristic takes $O(n^2)$ time, and iteratively eliminates the vertex whose elimination results in the fewest fill edges to be added. The *minimum-degree* heuristic is even simpler. It always eliminates the vertex with the lowest degree, which can be done in $O(n)$ time.

Chapter 3

Known approaches

In the previous chapter we found that, for our purposes, solving an STN equates to computing either its fully path consistent ISTN or its partially path consistent ISTN. We now review algorithms that perform this computation in different settings.

Section 3.1 discusses the basic problem of computing an ISTN from scratch, given an arbitrary STN as input. In section Section 3.2 we review more specialized *incremental* algorithms, which ensure the tightness of an existing ISTN when the weight of a single constraint is lowered.

3.1 Statically computing intermediate networks

The two kinds of ISTN introduced in Section 2.2.2 can be computed in different ways. In this section we review algorithms that compute these ISTNs using no other information than the constraints and events as specified by original STN instance.

We start with a brief overview of methods to compute intermediate networks. In Section 3.1.1 and Section 3.1.2 we review methods for FPC and PPC networks respectively. Since we are particularly interested in algorithms for partially path consistent networks, Section 3.1.3 gives a more in-depth discussion of P^3C, the state of the art in this area.

3.1.1 Fully Path Consistent STNs

Dechter et al. (1991) show that enforcing Full Path Consistency is equivalent to calculating the weight of the shortest paths between any pair of events in an STN. The standard way to enforce this property is to run one of the many well-known All-Pairs Shortest Paths (APSP) algorithms. A critical requirement for their use in computing STNs is that these algorithms must be able to deal with negative weights. In particular, an algorithm should terminate even if the network is inconsistent, i.e. if it contains a negative cycle.

The two classic APSP algorithms are the Floyd-Warshall algorithm, published independently by Floyd (1962) and Warshall (1962), and Johnson's algorithm (Johnson, 1977). Floyd-Warshall is quite simple to implement and runs in $\Theta(n^3)$ time.

Johnson is somewhat more complex but runs in $O(nm + n^2 \log n)$, which is more efficient for sparse graphs containing relatively few edges.

Recently even more advanced algorithms have appeared that exploit more specific properties. Pettie (2003) describe an algorithm which achieves a bound of $O(mn + n^2 \log\log n)$. Finally, the *Snowball* algorithm published by Planken et al. (2011a) requires $O(n^2 w_d)$ time, which is efficient for graphs with low tree widths.

3.1.2 Partially Path Consistent STNs

The problem of enforcing Partial Path Consistency on graphs has received significant attention in recent times. The original algorithm for this problem was proposed by Bliek and Sam-Haroud (1999) for general constraint satisfaction problems. It relies on the key insight that the tightest constraints in a chordal graph can be found by adjusting a constraint only when a tighter value is implied by a directly neighbouring triangle.

Their general idea is to maintain a queue of triangles that contain an edge whose weight has been lowered, and then check whether this lowered weight induces a lower weight in an other edge of the triangle. If so, all triangles adjacent to that edge are added to the queue. This process stops when there are no more triangles in which a shorter weight can be enforced. Bliek and Sam-Haroud show that at this point all weights have their lowest possible value.

The time required by this algorithm is bounded by $O(\delta \cdot m \cdot \partial^2)$, where δ is the maximum degree of the graph and ∂ is the maximum *domain size*, i.e. the number of distinct values any variable can assume (for example, the domain size of a Boolean problem would be 2). In practice however many problems take values from domains of infinite size (e.g. the domain of real numbers) limiting the usefulness of this approach.

Xu and Choueiry (2003) improved upon this idea in their algorithm $\triangle$STP, which is more careful in selecting the triangles to be added to the queue. This allows their algorithm to solve problems on continuous domains. Although it performed well in practice, Xu and Choueiry did not offer a theoretical upper bound on the performance $\triangle$STP. However, Planken et al. (2008) showed that in pathological cases the algorithm may need time quadratic in the number of triangles.

In the same paper, Planken et al. present the P^{3}C algorithm, which *does* have a theoretical upper bound: it requires time at most linear in the number of triangles in the graph. P^{3}C is the current state of the art in enforcing Partial Path Consistency, and is the most efficient known method to solve the STP from scratch. Since it is closely related to the new algorithms presented in this work, we will now discuss it in more detail.

3.1.3 The P^{3}C algorithm

The P^{3}C algorithm requires the STN to be triangulated, and a simplicial elimination ordering d to be known. It performs two sweeps along d: in the *inward* sweep, a

Algorithm 3.1: DPC

Input: A chordal STN $\mathcal{S} = \langle V, E \rangle$ with associated weights $w_{i \to j}$ and an ordering $d = (v_n, v_{n-1}, \ldots, v_1)$.

Output: CONSISTENT if $\mathcal{S}$ has been made DPC along d, or INCONSISTENT otherwise.

```
1 for k ← n to 1 do
2     foreach i < j < k such that {v_i, v_k}, {v_j, v_k} ∈ E do
3         w_{i→j} ← min{w_{i→j}, w_{i→k} + w_{k→j}}
4         w_{j→i} ← min{w_{j→i}, w_{j→k} + w_{k→i}}
5         if w_{i→j} + w_{j→i} < 0 then
6             return INCONSISTENT
7 return CONSISTENT
```

property called *Directed* Path Consistency (DPC) is enforced. During this phase the algorithm checks whether the STN is consistent. The *outward* sweep then uses this property to calculate the shortest paths.

Inward sweep: Directed Path Consistency

Given an ordering $d = (v_n, v_{n-1}, \ldots, v_1)$, the DPC algorithm ensures that every edge (v_i, v_j) is labelled with the weight of the shortest $v_i - v_j$ path through the subgraph induced by all vertices that precede both v_i and v_j in the ordering d. In other words, when DPC is enforced there is no path π from v_i to v_j consisting only of vertices v_k with $k > \max(i, j)$ such that the total weight of π is less than the weight $w_{v_i \to v_j}$. This allows us to formally define Directed Path Consistency as follows:

Definition 3.1. A chordal network $\mathcal{S}$ is *DPC along the simplicial elimination ordering* $d = (v_n, v_{n-1}, \ldots, v_1)$ if $w_{v_i \to v_j} \leq w_{v_i \to v_k} + w_{v_k \to v_j}$ for all $i, j < k$ where $v_i, v_j \in N(v_k)$.

Algorithm 3.1 shows the algorithm enforcing Directed Path Consistency, which was discovered by Dechter et al. (1991).

Outward sweep

In the outward sweep, shown in lines 3 to 7 of Algorithm 3.2, we exploit the fact that graph is already DPC to find the weights of the shortest paths through the entire graph. In particular, we already know the weights $w_{1 \to 2}$ and $w_{2 \to 1}$ of the globally shortest paths between v_1 and v_2. We use this knowledge to inductively calculate the globally shortest paths for the other edges. That is, assuming that we know the globally shortest paths between any two vertices v_i and v_j given $i, j < k$, we can derive the weight of the globally shortest paths between v_k, v_i and v_j.

Algorithm 3.2: P^{3}C

Input: A chordal STN $\mathcal{S} = \langle V, E \rangle$ with associated weights $w_{i \to j}$ and a simplicial elimination ordering $d = (v_n, v_{n-1}, \ldots, v_1)$.

Output: The PPC network $\mathcal{S}$, or INCONSISTENT.

```
1 call DPC(S, d)
2 return INCONSISTENT if DPC did
3 for k ← 1 to n do
4     foreach i, j < k such that {v_i, v_k}, {v_j, v_k} ∈ E do
5         w_{i→k} ← min{w_{i→k}, w_{i→j} + w_{j→k}}
6         w_{k→j} ← min{w_{k→j}, w_{k→i} + w_{i→j}}
7 return S
```

For (v_k, v_i) the idea is as follows. Let $\pi = v_k \to v_{j_1} \to \ldots \to v_{j_\ell} \to v_i$ be the shortest path from v_k to v_i. Since the graph is DPC, we can replace any part of π outside G_k by a shortcut edge that lies within G_k. Also, since v_i and v_{j_1} are neighbours of v_k, by DPC there must be an edge $v_{j_1} \to v_i$, so we can reduce the path to $\pi' = v_k \to v_{j_1} \to v_i$. Since i and j_1 are less than k, we know by our assumption that $v_{j_1} \to v_i$ is labelled with the weight of shortest path. The loop in lines 4 to 6 then checks all possible v_{j_1} to find the weight of the shortest path from v_k to v_i.

3.2 Incrementally maintaining networks

Once an STP instance $\mathcal{S}$ has been solved, an interesting question is how it responds to changes. Specifically, DTP solvers may produce a new network $\mathcal{S}'$ by further restricting the value of an existing constraint, or even by introducing new constraints between events that were not previously connected.

A naive approach would be to simply add the constraint to the network, and then compute the solution for $\mathcal{S}'$ from scratch using one of the methods discussed in the previous section. However, it seems that in many cases the solutions for this modified network $\mathcal{S}'$ should not differ too much from the solutions for $\mathcal{S}$. At the very least we know that every constraint will be at least as tight in the solution for $\mathcal{S}'$ as it will be in the solution for $\mathcal{S}$. Furthermore, since we only deal with a single edge, it seems intuitive that an update may be less costly than re-computing all constraints from scratch.

In this section we discuss two algorithms that realize this intuition. We first discuss an algorithm that incrementally maintains full path consistency in Section 3.2.1, and then turn to an algorithm for partially path consistent STNs in Section 3.2.2.

3.2.1 Fully path consistent STNs

In practice the task of incrementally updating an STN boils down to the following: Given a new weight $w'_{a\to b}$ for an edge (a,b), adjust the weight of every edge (i,j) for which there is a new path $i \dashrightarrow a \to b \dashrightarrow j$ with a lower total weight than the previously shortest path from i to j.

Tsamardinos and Pollack (2003) mention the *IFPC* algorithm to accomplish this, citing Mohr and Henderson (1986). However, this paper only presents a static $O(n^3d^3)$ approach to path consistency rather than an incremental one (with d the domain size mentioned earlier). Pseudocode for a version of the IFPC algorithm meeting the $O(n^2)$ bound on run time stated by Tsamardinos and Pollack was first published in full by Planken (2008). This algorithm runs in $O(n+n^{*2})$ time, where n^* is the number of vertices attached to an edge whose weight must change.

This variant of the IFPC algorithm works as follows. We start with a sweep over all vertices v_k other than v_a and v_b. For each such v_k, we update the shortest paths $v_k \dashrightarrow v_b$ and $v_a \dashrightarrow v_k$ to take the lower weight on $v_a \to v_b$ into account. During this sweep we track for which v_k either of the paths was actually updated. In the second phase of the algorithm we update the weights for all pairs (v_i, v_j) of affected vertices as determined in the first sweep, by setting their corresponding weights to $w_{i\to a} + w_{a\to j}$.

On a somewhat higher level of abstraction, we use the following observation: for any pair of events (i,j) affected by the change, the new path must be $i \dashrightarrow a \to b \dashrightarrow j$. In the first phase, we update $i \dashrightarrow b$ and $a \dashrightarrow j$ accordingly. Finally, since the network was already minimal, the weights for $i \dashrightarrow a$ and $b \dashrightarrow j$ cannot change, and we calculate the correct weight for $i \dashrightarrow j$ in the second phase.

3.2.2 Partially path consistent STNs

The problem statement for incrementally updating Partial Path Consistent networks is nearly identical with that for incrementally updating Fully Path Consistent networks, with the notable distinction that in the PPC case we only need to update weights on edges that already existed in the triangulated graph.

This has only recently become an area of interest, and the author is aware of only one algorithm to perform such an update: the *IPPC* algorithm developed by Planken et al. (2010).

The idea behind this algorithm is similar to the one behind IFPC, with some tweaks to exploit the chordality of the underlying graph. Again, we know that when a new, tighter, weight $w'_{a\to b}$ is given for the constraint $c_{a\to b}$, all constraints whose tightest value depends on $c_{a\to b}$ need to be tightened even further.

The critical insight used by IPPC is that a PPC graph is DPC along *any* simplicial elimination ordering. This allows us to efficiently construct an elimination ordering d which visits a and b last. Since the original graph was already PPC and the weight of $a \to b$ is lowered, we know that the changed graph remains DPC along d. Therefore, we only need to execute the outward sweep of P^{3}C.

However, the IPPC algorithm is even more clever. When we visit a vertex k during the construction of the simplicial construction ordering, we can maintain the weights of the shortest paths $k \dashrightarrow a$ and $b \dashrightarrow k$. Since every new shortest $i - j$ path has to be of the form $i \dashrightarrow a \to b \dashrightarrow j$ we can use this information to update the weight of the path $i \dashrightarrow j$.

All these steps can be executed in a single outward sweep that visits each edge of the chordal graph at most once, so the run time of this algorithm is $O(m_c)$. A further optimization stops the outward sweep when we are certain no more edges will be affected by a change, which allows us to bound the run time by $O(n^*\delta_c)$. Here n^* is again the number of vertices adjacent to an edge whose weight was modified, and δ_c is the degree of the chordal graph.

However, the IPPC algorithm has some weaknesses. The first issue is that it requires that the edge $a \to b$ is already part of the (chordal) STN. The problem here is that adding a new edge may create a new chordless cycle of length greater than three, thus breaking the chordality of the graph. Note that IFPC does not suffer from this problem, for the simple reason that it operates on a *complete* graph, which already contains all possible edges.

Secondly, in an empirical evaluation of the performance of the IPPC algorithm, Planken et al. (2010) concluded that the gains it offered compared to IFPC were rather limited. The results for IFPC were competitive on most benchmarks, and it outperformed IPPC for the class of so-called job-shop instances.

Chapter 4

Vertex-incremental PPC

As discussed in Section 3.2.2, the IPPC algorithm by Planken et al. (2010) has two weaknesses. It requires that the edge (a,b) whose weight is lowered is already explicitly part of the triangulated graph, and in practice its performance does not improve over the Incremental Full Path Consistency algorithm.

In this chapter we present a new algorithm, dubbed *Vertex-IPPC*, for incrementally enforcing new or tighter constraints on an existing STN. To emphasize the difference between our approach and that of Planken et al., we will refer to the original IPPC as *Edge*-IPPC from here onward.

The general idea for our new algorithm is presented in Section 4.1. Sections 4.2 to 4.4 discuss the algorithms Vertex-IPPC relies on and demonstrate the correctness of their use in the current context. In Section 4.5 we use the proofs from earlier parts of this chapter to prove the correctness of the entire algorithm. Finally, Section 4.6 concludes by discussing how Vertex-IPPC integrates nicely with an existing vertex-incremental triangulation algorithm, thus addressing one of the problems with Edge-IPPC mentioned above.

4.1 Algorithm idea

The idea behind the new algorithm is that instead of adding constraints one at a time, we add a single new *event* a and all its adjacent constraints C_a at once. In terms of the underlying network, we add a vertex instead of an edge.

We show that a large part of the network is still DPC when updates of this type are executed. More specifically, we can use the *Lex-BFS* algorithm to find a simplicial elimination ordering in which we need only re-enforce DPC in the graph induced by a and its neighbours $N(a)$. We therefore need only do a little work to enforce DPC on the entire graph, and then we can simply re-run the outward sweep of the P^3C algorithm to re-enforce partial path consistency.

However, we can improve efficiency even more when we use the following observation. Recall that Edge-IPPC can efficiently calculate the new weights of tightened constraints (i,j) when the weight for (a,b) is lowered, because we know that the

Algorithm 4.1: Vertex-IPPC

Input: A chordal STN $\mathcal{S}'$ obtained by extending the PPC STN $\mathcal{S}$ with (1) a new event a, (2) a set of constraints connecting a to the events in $U \subseteq V$ with weights $C_a = \{w_{a\to u}, w_{u\to a} \mid u \in U\}$, and (3) a set of constraints with infinite weight needed to make $\mathcal{S}'$ chordal.

Output: INCONSISTENT if the new constraints yield inconsistency, CONSISTENT if PPC has been enforced on the modified network $\mathcal{S}'$.

```
1 d ← Lex-BFS (S', a)
2 call DPC (S'_{N(a)∪{a}}, d)
3 return INCONSISTENT if DPC did
4 call SSSP-PPC (S', d, a)
5 return CONSISTENT
```

shortest path corresponding to any such tightened constraint must be of the form $i \dashrightarrow a \to b \dashrightarrow j$. A similar fact holds for Vertex-IPPC: if constraint (i, j) needs to be tightened, the new shortest path must be of the form $i \dashrightarrow a \dashrightarrow j$, i.e. it must contain the newly added vertex a.

Hence, if we maintain the so-called Single-Source Shortest Paths from a to every vertex, and the Single-Sink Shortest Paths every vertex to a, we can tighten all constraints in one outward sweep, just as with Edge-IPPC. Since at this point the network is already DPC, we can use an algorithm that does this in $O(m_c)$ time, which is more efficient than the $O(nw_d^2)$ required by P^{3}C's outward sweep. Algorithm 4.1 shows the high-level pseudo-code for the algorithm just described.

It is possible that the new constraints C_a break the chordality of the original STN $\mathcal{S}$. We therefore require that chordality is re-enforced on the network in a preprocessing step before we run Vertex-IPPC. In Section 4.6 we discuss an algorithm to accomplish this.

We now discuss each step of the algorithm in more detail.

4.2 Lexicographic Breadth-First Search

The Lexicographic Breadth-First Search (Lex-BFS) algorithm was discovered by Rose and Tarjan (1975) and can be used to construct a simplicial elimination ordering of a chordal graph. We reproduce the high-level pseudocode of this algorithm in Algorithm 4.2.

Recall that when we visit a node v in a generic breadth-first search, we are required to visit all neighbours of v before proceeding with any node that is more than one hop away from v, i.e. before visiting any node that is a neighbour of a neighbour. Lex-BFS is a specialised version of this algorithm, which additionally requires that the neighbours of v be visited in a particular order.

Specifically, the algorithm maintains a label for every unvisited vertex and a sin-

Algorithm 4.2: Lex-BFS

Input: A chordal STN $\mathcal{S} = \langle V, E \rangle$ and a vertex $a \in V$.
Output: A simplicial elimination ordering d of the vertices in $\mathcal{S}$, ending in a.

```
1 foreach v ∈ V do
2 |   LABEL[v] ← ∅
3 |   TAGGED[v] ← FALSE
4 call Tag(d, a, n)
5 for i ← n − 1 to 1 do
6 |   pick vertex v with the lexicographically largest label
7 |   call Tag(d, v, i)
8 return d
```

Procedure Tag($d, v \in V, i \in 1 \ldots n$)

```
1 d(i) ← v
2 TAGGED[v] ← TRUE
3 foreach u ∈ N(v) such that ¬TAGGED[u] do
4 |   LABEL[u] ← LABEL[u] ∪ {i}
```

gle global counter i, initialized with n, the number of vertices in the graph. Whenever the algorithm visits a vertex v, the label of every unvisited neighbour of v is extended with the current value of the counter. The counter is then decreased and the next vertex is selected.

This next vertex must have a *lexicographically largest* label of all unvisited vertices. A label $a = a_1 a_2 \ldots a_k$ is lexicographically larger than $b = b_1 b_2 \ldots b_\ell$ when there is a $j > 0$ such that both $a_i = b_i$ for all $i < j$ and $a_j > b_j$. Note that there may be multiple vertices with identical lexicographic labels. In that case one of these may be selected arbitrarily. The order in which this algorithm visits the vertices is exactly the *reverse* of a simplicial elimination ordering. However, since i counts down, we obtain a simplicial elimination ordering by simply inserting the visited vertex at $d(i)$. This yields the following result:

Theorem 4.1 (Rose and Tarjan (1975))**.** *The ordering d produced by* Lex-BFS(G, a) *is a simplicial elimination ordering of the chordal graph $G = \langle V, E \rangle$ for any $a \in V$.*

In practice, implementations of Lex-BFS do not maintain the labels explicitly. Rather, they use a modified queue data structure, which maintains sets of vertices with equal lexicographic labels. During the execution of the algorithm these sets are split and vertices are moved between them, such that the set at the head of the queue always contains the vertices with the currently largest label. With careful implementation, these algorithms require $O(m_c)$ time to compute a full ordering, where m_c is the number of edges in the chordal graph.

Remember that for Vertex-IPPC we want to re-enforce DPC in the neighbourhood of the newly added vertex a. Since DPC is enforced along a simplicial elimination ordering, we want such an ordering that visits a and its neighbourhood *last*: in that case, we show that we do not need to do work for any of the vertices earlier in the ordering. Recall that $\delta_c(a)$ is the degree of a in a chordal graph. The following lemma shows that the ordering produced by Lex-BFS fulfils the requirement just stated:

Lemma 4.2. *Given a chordal graph $G = \langle V, E \rangle$ and a vertex $a \in V$, the simplicial elimination ordering d of G produced by* Lex-BFS *starting at a satisfies $d(n) = a$ and $\{d(n - \delta_c(a)), \ldots, d(n-1)\} = N(a)$.*

Proof. The ordering d is the reverse of the order in which Tag is called on the vertices of G. The first vertex to be tagged is a (on line 4) and it is assigned the last position in d on line 1 of Tag. Consequently the label of the neighbours $N(a)$ of a is updated to start with n on line 4.

Note that since the loop counter i of Lex-BFS decreases from $n-1$, the label assigned to any vertex not in $N(a)$ can at most start with $n-1$. Since such labels are lexicographically smaller than labels starting with n, all neighbours of a are tagged before any other vertices. Hence the slots $d(n - \delta_c(a) - 1)$ through $d(n-1)$ are taken by $N(a)$. □

4.3 DPC on induced subgraphs

In the previous section we saw that we can obtain a simplicial elimination ordering in which we visit the new vertex a and its neighbourhood last. We now use this information to demonstrate that running the DPC algorithm along this ordering in the subgraph induced by $\{a\} \cup N(a)$ *only* is sufficient to re-enforce DPC for the entire graph.

This is done in three steps. First we demonstrate that the subsequence d' of a simplicial elimination ordering d is a valid simplicial elimination ordering for the subgraph induced by the vertices in d'. We then revisit the definition of DPC, and show that it can also be expressed as a property of vertices relative to an ordering. Finally, we show that running DPC along the ordering in the subgraph induced by $\{a\} \cup N(a)$ re-enforces this property for all vertices, thus re-enforcing DPC on the entire graph.

As stated above, we start by showing that a subsequence d of a simplicial elimination ordering d' is a valid elimination ordering of the subgraph induced by the vertices in d'. Let $G \backslash \{v\}$ be a shorthand for $G_{V \backslash \{v\}}$, the graph induced by all vertices in V other than v. In other words, $G \backslash \{v\}$ is the graph from which the vertex v and all edges connected to v have been removed. We then have the following result:

Lemma 4.3. *Let $d = (v_n, v_{n-1}, \ldots, v_1)$ be a simplicial elimination ordering of the chordal graph G. Then the ordering $d' = (v_n, \ldots, v_{k+1}, v_{k-1}, \ldots, v_1)$ is a simplicial elimination ordering of the graph $G' = G \backslash \{v_k\}$ for every $1 \leq k \leq n$.*

Proof. First consider the vertices v_i with $i > k$. Since d is a simplicial elimination ordering of G, the vertices $C_i = \{v_j \mid v_j \in N(v_i), j < i\}$ induce a clique in G. A subset of vertices in a clique also induces a clique, so $C_i \backslash \{v_k\}$ is a clique in G'. Therefore v_i remains simplicial in d'.

Furthermore, for v_i with $i < k$, we know that C_i is a clique in G, which cannot contain v_k since $i < k$. Since the removal of v_k is the only difference between G and G', this clique remains unchanged in G', and v_i remains simplicial in d'.

Since v_k itself does not occur in d' and all other vertices remain simplicial, d' must indeed be a valid simplicial elimination ordering of G'. □

Let $d' = d \backslash V'$ be a sequence of all vertices $v \in V \backslash V'$, appearing in the same order as they appear in d. Repeatedly applying Lemma 4.3 for different vertices of the graph G yields the following corollary:

Corollary 4.4. *Given a simplicial elimination ordering d of a chordal graph $G = \langle V, E \rangle$ and an arbitrary subset $V' \subseteq V$. Then the ordering $d' = d \backslash V'$ is a simplicial elimination ordering for the graph $G' = G_{V \backslash V'}$.*

Given an elimination ordering for the entire graph G, Corollary 4.4 allows us to derive an elimination ordering for the subgraph $G_{\{a\} \cup N(a)}$ induced by $\{a\} \cup N(a)$. We now need to show that re-enforcing DPC in this subgraph suffices to re-enforce DPC in the entire graph.

Unfortunately, Definition 3.1 (repeated below for convenience) defined DPC in terms of an ordering spanning the *entire* graph. For Vertex-IPPC, we want to be able to reason about DPC in the context of an induced *subgraph*. Definition 4.5 addresses this issue by providing an alternative definition of DPC at the level of individual vertices.

Definition 3.1. A chordal network $\mathcal{S}$ is *DPC along the simplicial elimination ordering* $d = (v_n, v_{n-1}, \ldots, v_1)$ if $w_{v_i \to v_j} \leq w_{v_i \to v_k} + w_{v_k \to v_j}$ for all $i, j < k$ where $v_i, v_j \in N(v_k)$.

Definition 4.5. A vertex v_k of an STN $\mathcal{S}$ has the *DPC property relative to the ordering* $d = (v_n, v_{n-1}, \ldots, v_1)$ if $w_{v_i \to v_j} \leq w_{v_i \to v_k} + w_{v_k \to v_j}$ for all $i, j < k$ where $v_i, v_j \in N(v_k)$.

The following proposition shows that these definitions express the same notion of Directed Path Consistency.

Proposition 4.6. *A network $\mathcal{S}$ is DPC along the ordering d if and only if all its vertices have the DPC property relative to d.*

Proof. ($\Rightarrow$) If $\mathcal{S}$ is DPC along d, Definition 3.1 requires that for every vertex v_k the inequality $w_{v_i \to v_j} \leq w_{v_i \to v_k} + w_{v_k \to v_j}$ holds, and hence that every v_k has the DPC property relative to d.

($\Leftarrow$) For the other direction, if every vertex has the DPC property relative to d, we know by Definition 4.5 that $w_{v_i \to v_j} \leq w_{v_i \to v_k} + w_{v_k \to v_j}$ holds for every v_k, and therefore that $\mathcal{S}$ is DPC along d. □

We use the shorthand *d-DPC* both to indicate that a vertex has the DPC property relative to d and to indicate that an STN is DPC along d. Which of the two definitions applies will always be clear from context.

The concept of the DPC property of a vertex allows us to prove the following intuitive notion. Suppose the first k vertices of an ordering d already have the DPC property. Then we need only run the DPC algorithm along the remaining unvisited vertices of d to enforce DPC on the entire graph. We formalise this in Lemma 4.7, where we use $\mathcal{S}_{d_2}$ to denote the subnetwork induced by the vertices in d_2 only.

Lemma 4.7. *Given an STN $\mathcal{S}$ and an elimination ordering $d = (v_n, v_{n-1}, \ldots, v_1)$ of which the vertices $d_1 = (v_n, v_{n-1}, \ldots, v_{k+1})$ have the DPC property along d. Then running the DPC algorithm along $d_2 = (v_k, v_{k-1}, \ldots, v_1)$ in $\mathcal{S}_{d_2}$ makes $\mathcal{S}$ d-DPC.*

Proof. Because d_2 is a subsequence of d, it follows from Corollary 4.4 that d_2 is a simplicial elimination ordering of $\mathcal{S}_{d_2}$. This means that we can run the DPC algorithm in $\mathcal{S}_{d_2}$ to enforce the DPC property along d_2 on all vertices in d_2.

When all vertices in d_2 are d_2-DPC, we know that all weights $w_{v_i \to v_j}$ are correctly set for $i, j < \ell$, where $v_i, v_j \in N(v_\ell)$ and $1 \leq \ell \leq k$. Since the vertices in d_1 are still d-DPC, the same holds for $k+1 \leq \ell \leq n$. Together, this means that the weights are correctly set for all $1 \leq \ell \leq n$, and therefore that $\mathcal{S}$ is d-DPC. □

We now have enough information to prove the principal lemma of this section. Let $\mathcal{S} = (V, C)$ be a PPC STN and consider the STN $\mathcal{S}' = (V', C')$ obtained by adding to $\mathcal{S}$ a new vertex a connected to $N(a) \subseteq V$ by the set of constraints C_a, i.e. with $V' = V \cup \{a\}$ and $C' = C \cup C_a$. We show that running the DPC algorithm along a simplicial elimination ordering visiting only a and its neighbours re-enforces DPC on the entire network.

Lemma 4.8. *Let $\mathcal{S}'$ be an STN as just defined. Given a simplicial elimination ordering $d = (v_n, v_{n-1}, \ldots v_1)$ of $\mathcal{S}'$ such that $\{v_{\delta_c(a)+1}, \ldots, v_2\} = N(a)$ and $v_1 = a$. Then running the DPC algorithm along $d' = (v_{\delta_c(a)+1}, \ldots, v_1 = a)$ in $\mathcal{S}'_{\{a\} \cup N(a)}$ makes $\mathcal{S}'$ d-DPC.*

Proof. It suffices to show that each vertex $v_k \in (v_n, v_{n-1}, \ldots, v_{\delta_c(a)+2})$ has the DPC property along d in $\mathcal{S}'$. The remainder of the proof then follows immediately from Lemma 4.7.

Assume for a contradiction that one such v_k is *not* d-DPC; then there must be some v_i, v_j such that $i, j < k$ and v_i, $v_j \in N(v_k)$, but $w_{i \to j} > w_{i \to k} + w_{k \to j}$. Since $\mathcal{S}$ was PPC already and none of the weights in $\mathcal{S}$ have changed, this inequality cannot hold if v_i, v_j and v_k are all in $\mathcal{S}$. Hence one of them must be a, the only vertex not in $\mathcal{S}$.

Since $k > \delta_c(a) + 1$ and $a = v_1$, v_k cannot be a. Furthermore, since $k > \delta_c(a) + 1$ and all neighbours v_ℓ of a have $\ell \leq \delta_c(a) + 1$, v_k cannot be a neighbour of a. But

Algorithm 4.3: SSSP-PPC($\mathcal{S}, d, a$)

Input: A DPC STN $\mathcal{S} = \langle V, E \rangle$ which is PPC on $\mathcal{S} \backslash \{a\}$ and has associated weights $\{w_{i \to j}, w_{j \to i} \mid \{i, j\} \in E\}$, and a simplicial elimination ordering $d = (v_n, v_{n-1}, \ldots, v_2, v_1 = a)$.

Output: The PPC network $\mathcal{S}$.

```
1 D_{a→}[a] ← 0; D_{a←}[a] ← 0
2 VISITED[a] ← TRUE
3 for k ← 2 to n do
4 |   D_{a→}[v_k] ← ∞; D_{a←}[v_k] ← ∞
5 |   call Visit(v_k)
```

since v_i and v_j are neighbours of v_k, this means that v_i and v_j cannot be a either, a contradiction. Thus, vertices $(v_n, v_{n-1}, \ldots, v_{\delta_c(a)+2})$ must indeed be d-DPC in $\mathcal{S}'$. □

4.4 PPC with Single-Source Shortest Paths

Now that DPC has been re-enforced on the extended network $\mathcal{S}'$, the final step is exploiting this property to re-enforce partial path consistency. This can be done with a slightly modified version of the DPC-SSSP algorithm designed by Planken et al. (2011b).

The original DPC-SSSP algorithm calculates the Single-Source Shortest Paths (SSSP) distances from a single vertex a to every other vertex in the graph. By exploiting the knowledge that the graph is already DPC, DPC-SSSP needs to visit every edge in the graph only twice, achieving a runtime bound of $O(m_c)$.

Our modified version is extended to track the length of both the shortest incoming and shortest outgoing paths to and from a. As we stated earlier, we know that if the insertion of a and its adjacent constraints causes a change in weight for the edge (i, j), the new shortest path from i to j must go through a. In other words, the new shortest $i - j$ path must be of the form $i \dashrightarrow a \dashrightarrow j$.

Since DPC-SSSP calculates exactly the weights of these paths $i \dashrightarrow a$ and $a \dashrightarrow j$ for all i and j, we can re-enforce partial path consistency in the same sweep. Our modified algorithm, dubbed SSSP-PPC, is shown in Algorithm 4.3.

We prove the correctness of SSSP-PPC in two steps. First, in Lemma 4.9 we adapt the correctness proof of DPC-SSSP to show that SSSP-PPC does in fact calculate the correct weights for the shortest paths between a and any other vertex. This information is then used in Lemma 4.10 to show that a run of SSSP-PPC enforces partial path consistency on the network we encounter in line 4 of Vertex-IPPC.

Lemma 4.9. *When* VISITED$[v]$ *is* TRUE, $D_{a\leftarrow}[v]$ *and* $D_{a\rightarrow}[v]$ *are set to the total weight of the shortest paths from v to a and from a to v respectively.*

Procedure Visit($v \in V$)

1 **foreach** $u \in N(v)$ **such that** VISITED$[u]$ = TRUE **do**
2 $\quad D_{a\rightarrow}[v] \leftarrow \min\{D_{a\rightarrow}[v], D_{a\rightarrow}[u] + w_{u\rightarrow v}\}$
3 $\quad D_{a\leftarrow}[v] \leftarrow \min\{D_{a\leftarrow}[v], w_{v\rightarrow u} + D_{a\leftarrow}[u]\}$
4 VISITED$[v] \leftarrow$ TRUE
5 **foreach** $u \in N(v)$ **such that** VISITED$[u]$ = TRUE **do**
6 $\quad w_{u\rightarrow v} \leftarrow \min\{w_{u\rightarrow v}, D_{a\leftarrow}[u] + D_{a\rightarrow}[v]\}$
7 $\quad w_{v\rightarrow u} \leftarrow \min\{w_{v\rightarrow u}, D_{a\leftarrow}[v] + D_{a\rightarrow}[u]\}$

Proof. The proof is by induction along the simplicial *construction* ordering $d^{-1} = (a = v_1, v_2, \ldots, v_n)$. For the base case, $D_{a\leftarrow}[a]$ and $D_{a\rightarrow}[a]$ are set correctly in line 1 of SSSP-PPC. Now for the induction step, assume that the weight $D_{a\rightarrow}[v]$ is set correctly for all $v \in \{v_1, \ldots, v_k\}$; we show that SSSP-PPC sets the correct weight for $D_{a\rightarrow}[v_{k+1}]$. The proof for $D_{a\leftarrow}[v_{k+1}]$ is analogous.

Assume for a contradiction that VISITED$[v_{k+1}]$ is TRUE but that $D_{a\rightarrow}[v_{k+1}]$ does not hold the weight of shortest path. Then there must be some path $\pi = a \rightarrow v_{j_1} \rightarrow \cdots \rightarrow v_{j_\ell} \rightarrow v_{k+1}$ with weight $w_\pi < D_{a\rightarrow}[v_{k+1}]$. If there is a vertex on π such that $\max_i j_i > k$, we know by DPC that we can replace this vertex by a shortcut from its predecessor to its successor, without increasing the weight. By repeating this procedure we can remove all such v_{j_i} from the path π without increasing its total weight.

In particular, this means that $j_\ell \leq k$, which implies that VISITED$[v_{j_\ell}]$ is TRUE and therefore, by the induction hypothesis, that $D_{a\rightarrow}[v_{j_\ell}]$ has been correctly set. But then the value of $D_{a\rightarrow}[v_{k+1}]$ is correctly updated by the assignment on line 2 of Visit, contradicting our assumption that $w_\pi < D_{a\rightarrow}[v_{k+1}]$. □

Lemma 4.10. *Given a d-DPC STN $\mathcal{S}$, where a is the last vertex of d and $\mathcal{S}\backslash\{a\}$ is PPC. Then the network $\mathcal{S}'$ produced by running SSSP-PPC on $\mathcal{S}$ along the reverse of d is PPC.*

Proof. First consider the edges adjacent to a, i.e. the edges $\{a, v\}$ with $a \in N(v)$. Before the second loop of Visit(v) is executed, VISITED$[a]$ and VISITED$[v]$ are both TRUE, by lines 2 of SSSP-PPC and 4 of Visit respectively. Hence, by Lemma 4.9, $D_{a\rightarrow}[v]$ and $D_{a\leftarrow}[v]$ hold the weights of the corresponding minimum paths between v and a. So the weights of $w_{a\rightarrow v}$ and $w_{v\rightarrow a}$ are correctly updated in lines 6 and 7 of Visit.

Now consider any other edge $\{u, v\}$ neither endpoint of which is a. Since $\mathcal{S}\backslash\{a\}$ was PPC, the only way in which the weight $w_{u\rightarrow v}$ can be further reduced is if there is some shorter path $u \dashrightarrow a \dashrightarrow v$ through a. Without loss of generality, assume that Visit(u) is called before Visit(v). Then VISITED$[u]$ and VISITED$[v]$ are both TRUE before the execution of the second loop in Visit(v). By the same reasoning as used above for a and its neighbours, line 6 of Visit correctly updates $w_{u\rightarrow v}$. The proof for $w_{v\rightarrow u}$ is analogous, reversing all arcs and using line 7 of Visit instead. □

4.5 Correctness and efficiency

Having shown the correctness of the individual steps of the Vertex-IPPC algorithm, we now complete our analysis by demonstrating that these steps do indeed re-enforce partial path consistency when executed in sequence.

Theorem 4.11. *The* Vertex-IPPC *algorithm correctly re-enforces PPC or decides inconsistency in* $O(m_c + \delta_c(a)w_d^2)$ *time.*

Proof. The simplicial elimination ordering d of $\mathcal{S}'$ is obtained using Lex-BFS, and therefore, by Lemma 4.2, the last elements in d are the neighbours of a, followed by a itself as final element. Since $\mathcal{S}$ was PPC, by Lemma 4.8 the call to DPC on the sub-graph consisting of a and its neighbours re-enforces DPC along d for the new $\mathcal{S}'$, or concludes inconsistency. Finally, by Lemma 4.9, running SSSP-PPC along $\mathcal{S}'$ re-enforces PPC on $\mathcal{S}'$.

We now turn to the time complexity. Lex-BFS takes $O(m_c)$ time to construct the ordering d, and running DPC on an induced graph with $\delta_c(a)+1$ nodes takes $O(\delta_c(a)w_d^2)$ time. Finally, SSSP-PPC takes $O(m_c)$ time: Visit is called once per vertex and all operations in Visit take amortized constant time per edge. □

4.6 Incremental triangulation

So far we have focussed exclusively on the *correctness* of Vertex-IPPC. We now discuss an important *advantage* of Vertex-IPPC over Edge-IPPC: it integrates well with an incremental triangulation algorithm.

Recall that Edge-IPPC requires that the edge (a,b), representing the constraint $c_{a\to b}$ to be tightened, is already part of the underlying structure. If the edge is not present, a new triangulation must be found in which it does exist. Note that we cannot simply include (a,b), since it may introduce a new chordless cycle This would break the chordality required for PPC algorithms to work.

For Vertex-IPPC, we can address this issue by using a recently discovered algorithm by Berry et al. (2006). This algorithm performs *vertex-incremental minimal triangulation*: given a chordal graph $G = \langle V, E\rangle$, a new vertex a, and a set of edges E_a connecting a to existing vertices of G, it can compute the *minimal* fill F such that the graph $G' = \langle V \cup \{a\}, E \cup E_a \cup F\rangle$ is chordal.

Their algorithm needs $O(n)$ time to insert one edge, and since there are m edges it takes $O(nm)$ time to triangulate a graph from scratch. An interesting observation here is that if we average this time over the number of vertices, we obtain an amortized bound of $O(m)$ time to insert one vertex. Since $m = O(m_c)$, this fits comfortably in the $O(m_c + \delta_c(a)w_d^2)$ bound for one run of Vertex-IPPC.

While Berry et al. give an extensive description of their algorithm and proved its correctness, they did not provide an implementation. Since vertex-incremental triangulation aligns well with Vertex-IPPC, we contacted the original authors to ask whether they were aware of any existing implementations. It turned out that to the

best of their knowledge, no such implementation existed. We therefore decided to create our own.

In the remainder of this section, we discuss the high-level idea behind Berry et al.'s algorithm and the special data structure they use to achieve a time bound of $O(n)$ for inserting an edge. We conclude with the discussion of an important implementation detail that was not mentioned in the original paper.

4.6.1 Algorithm idea

The algorithm relies on the notion of *separators* in the graph. A set of vertices V' is called a $u-v$ *separator* if there is no longer any path from u to v when all vertices in V' have been removed from the graph. More generally, we say that V' is separator of the graph G if G dissolves into two or more disconnected components after V' is removed. If there is no strict subset of $V'' \subset V'$ such that V'' is still a separator, V' is a *minimal separator*.

Recall that each node in a clique tree represents a clique of vertices in the underlying graph, and that if there is an edge between two node in the clique tree, the cliques represented by the nodes overlap. Therefore, every edge in the clique tree can be labelled by the overlap between the cliques on its endpoint. A special property of clique trees is that the overlap corresponding to an edge is a separator of the underlying graph.

The algorithm is based on two key observations. First, the authors prove that when an edge $\{u,v\}$ is inserted, the set of fill edges required to ensure chordality of the graph is exactly the set of edges between u and any vertex in a minimal $u-v$ separator. Second, they show that we obtain a *minimal* triangulation if we insert the fill edges required by the previous observation for each edge attached to the new vertex a. This is not necessarily the case when we use the algorithm to insert an edge connecting two vertices already in G.

At the highest level, their algorithm for adding each individual edge $\{a,v\}$ attached to a follows the following steps:

1. Use the clique tree to find the union S of all minimal $a-v$ separators.
2. Insert the corresponding fill edges $\{a,s\}$ for all $s \in S$ in the underlying graph.
3. Update the clique tree to reflect the new structure.

In order to find S in step one, we find the shortest path P through the clique tree between the closest two cliques K_a and K_v, such that $a \in K_a$ and $v \in K_v$. Each of the clique edges on this path is linked to a separator that should be included in S. Some of these separators may not be *minimal*, but this can be detected while traversing the path, at which point the clique tree can be restructured. Following this restructuring, we know that each clique edge on P represents a unique, minimal $a-v$ separator.

The second step is straightforward, but the third step may require some explanation. The idea here is to traverse P again and insert a into every clique along the way, since it is now connected to at least one vertex in that clique. However, this may cause a to be inserted into a clique with some other vertex q it is not actually connected to. Fortunately, this too can efficiently detected, and if we encounter such a clique, we split it in two, such that one contains a and the other contains q.

Finally, it may be that in the third step one clique becomes a subset of another, which would cause the clique tree to contain a non-maximal clique. However, this can only occur at the head of the path P, where it can be easily detected. Simply removing the non-maximal clique re-establishes maximality of the clique tree.

4.6.2 Data structure

There is one problem with the algorithm as described above. In a naive clique tree data structure, we may annotate each clique edge with the set of all vertices in the minimal separator it represents. However, each such separator can be of size $\Theta(n)$, such that merely scanning through all sets on the path in step 1 may cost more than $O(n)$ time. Since this work has to be done for each edge, and there are m edges, the algorithm would not be competitive with other $O(mn)$ minimal triangulation algorithms.

To circumvent this problem, the authors use another representation of a clique tree. Rather than explicitly maintaining the contents of each separator, we annotate a clique edge with the *difference* between its two endpoints, in so-called *diff-lists*. More specifically, we maintain *add* and *remove* lists, such that when we traverse the edge from clique K_x to clique K_y, we know that $K_y = (K_x \cup \mathit{add}(K_x, K_y)) \setminus \mathit{remove}(K_x, K_y)$.

Using the coherency property of clique trees, we can derive that if a vertex v appears on the path P, all cliques containing v will form a connected sub-path of P. Therefore, when traversing P each vertex occurs at most once on an *add* and at most once on a *remove* list, and we can collect the separators on a path in $O(n)$ time.

Finally, we also add a *root node* K_r, which we connect to one arbitrary clique K in the tree, and set $\mathit{add}(K_r, K)$ to be the contents of K. Now, if we want to determine the contents of a clique node K', we find the path from the root node K_r to K'. We can determine the contents of K' by starting with the empty set, and then iterating over the clique edges on the path. For every edge (K_i, K_{i+1}) we add the vertices on $\mathit{add}(K_i, K_{i+1})$ to our set and remove the vertices in $\mathit{remove}(K_i, K_{i+1})$. When we arrive at K', the set will hold the contents of K'. Moreover, since we only read the diff-lists on a path, by the same reasoning as above this operation takes only $O(n)$ time.

We include a slightly more detailed version of the pseudo-code for the algorithm in Appendix A. For the full details and correctness proof we refer the reader to Berry et al. (2006).

4.6.3 Implementation details

A detail left open by the authors of the algorithm is exactly how we find a clique K_v containing the vertex v in the first step of the algorithm. Since the structure of the clique tree can change during the execution of the algorithm, a static lookup table does not suffice: as we have seen, a clique can be split in parts, or may even be deleted. These operations may cause references in a static table to point at incorrect cliques, or even cliques that no longer exist.

A first approach to circumvent this is to simply search through the tree. Since a clique tree has $O(n)$ nodes and $O(n)$ edges, a breadth-first search will also complete in $O(n)$ time. Furthermore, in naive clique trees it is an $O(1)$ operation to check whether a clique node contains a specific vertex.

However, we do *not* have a naive clique tree: we need to check the *diff-lists* to determine whether a clique contains a vertex. Moreover, while coherence guarantees that the total size of the diff-lists along a *path* is $O(n)$, for a *tree-traversal* as performed by a BFS the total size may well be $\Omega(n)$. Therefore this solution will not do.

Fortunately, we can solve this issue by falling back on a lookup-table, but being more clever about maintaining it. Specifically, we have to be careful about the way cliques are split and removed in step three.

Recall that we split a clique K when it contains a vertex q not connected to the inserted vertex u in the underlying graph. We observe that instead of splitting K, we can just insert another clique $K' = K \setminus \{q\} \cup \{u\}$, connect K to K' in the tree and replace K with K' on the path. This way, we do not have to update any references in the table, since the original clique K still exists.

Regarding removal, this only happens when a clique becomes a subset of some other clique that is its direct neighbour on the path P. As mentioned earlier we can find the contents of the clique to be removed in $O(n)$ time, and updating the references in the table to point at the neighbouring clique can also be done in $O(n)$ time.

We will revisit our implementations of Berry et al.'s vertex-incremental triangulation algorithm and of Vertex-IPPC in Chapter 6, where we subject them to a limited empirical evaluation. Before doing so however, we present our second new algorithm in Chapter 5.

Chapter 5

Support-based decremental PPC

As discussed in Section 3.2.2, there is at least one method to efficiently maintain partial path consistency when an edge weight is *lowered*. However, we are not aware of an algorithm to accomplish this is in the case when an edge weight is *raised*.

The crucial difference between lowering and raising weights is as follows. When an edge is shortened, we can simply check the weights of the shortest paths in the graph, and see if we can reduce them even further using the edge whose weight was just lowered. Note that we only need to know the *weights* of the paths, not the paths themselves.

When the weight of an edge is *raised* however, it is not immediately obvious which shortest paths are affected by the change. Moreover, even *if* we know what paths have changed, in the new situation an entirely different path may have become shortest.

Nonetheless, it seems plausible that the effects of raising the weight of a local edge will be unlikely to affect the weights of an edge very far away. Conversely, the edges whose weights *are* affected are probably relatively close to the edge whose weight was raised to begin with.

In this chapter we present an algorithm that exploits this intuition to efficiently re-enforce partial path consistency in an STN, given that the weight of one of its edges is raised. Specifically, in Section 5.1 we introduce the *weight support graph*, which tracks the relations between the weights of edges at a local level. In Section 5.2 we show how this support graph can be efficiently initialized for a partially path consistent STN.

The subsequent sections then describe our new algorithm. We start with a high-level overview in Section 5.3. Section 5.4 discusses the details of our method for finding the affected edges. These affected edges are then used in Section 5.5, where we show how to re-enforce partial path consistency and update the weight support graph accordingly. In Section 5.6 we use the results from the preceding sections to prove our overall claims on correctness and efficiency. Finally, in Section 5.7 we discuss possible extensions to improve the applicability and run time of our algorithm.

5.1 The weight support graph

We will need to reason about both weights that were originally assigned to constraints and the minimum weights that are computer for them when partial path consistency is enforced. To be able to succinctly refer to the right kind of weight, we introduce the following notation:

Definition 5.1. Given a chordal STN $\mathcal{S}$, we define the following two weights for each constraint $c_{i\to j}$ between events x_i and x_j. The *original weight* $w_{i\to j}$ is the weight the constraint was initialized with, and the *minimum weight* $\omega_{i\to j}$ is the weight assigned to $c_{i\to j}$ when partial path consistency is enforced, i.e. the lowest cumulative weight along any path from x_i to x_j.

Now, observe that there are two ways we can arrive at the minimum weight $\omega_{u\to v}$ of a constraint (u,v). First, it may be that there is no path with a total weight lower than the original weight $w_{u\to v}$ of (u,v). The other option is that there is a path $u \dashrightarrow x \dashrightarrow v$ through a vertex x neighbouring on both u and v, such that $\omega_{u\to x} + \omega_{x\to v} = \omega_{u\to v}$. Note that since shortest paths are not unique, both options may be true at the same time.

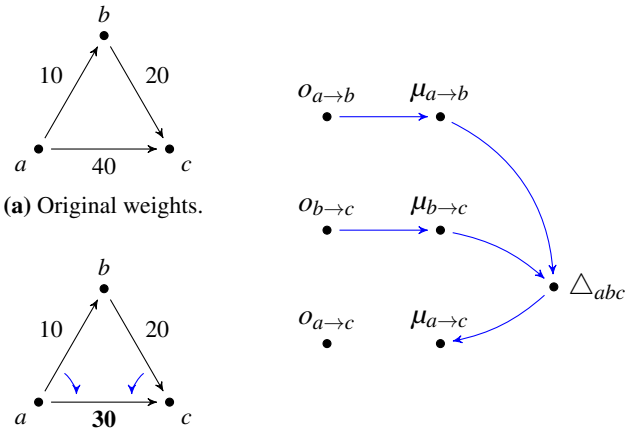
Both options above contain an implicit notion of *support*. If the minimum weight of (u,v) is due to its original weight, we can say that $\omega_{u\to v}$ is supported by $w_{u\to v}$. In the second case, the minimum weight of (u,v) is due to the minimum weights of (u,x) and (x,v), and therefore $\omega_{u\to v}$ is supported by $\omega_{u\to x}$ and $\omega_{x\to v}$. It now seems reasonable that if one of these supporting weights changes, its support falls away. Furthermore, if that means the minimum weight of (u,v) is no longer supported, $\omega_{u\to v}$ should change as well.

5.1.1 Formal definition

We capture this intuitive notion of support in the definition of a *weight support graph* D, the structure of which reflects how the minimum weights of the constraints in the underlying network $\mathcal{S}$ support each other. For each constraint (i,j) in the network $\mathcal{S}$, D contains a *minimum node* $\mu_{i\to j}$ representing the minimum weight $\omega_{i\to j}$ of (i,j) and an *original node* $o_{i\to j}$ representing the original weight $w_{i\to j}$ of (i,j). Furthermore, for every ordered triangle (i,k,j) such that $\omega_{i\to k} + \omega_{k\to j} = \omega_{i\to j}$, D contains a *triangle node* $\triangle_{ikj}$.

We can now model the weight supports in the network $\mathcal{S}$ by drawing directed edges between these nodes in D, as follows:

- If $\omega_{i\to j} = w_{i\to j}$, we insert an edge from $o_{i\to j}$ to $\mu_{i\to j}$, showing that the minimum weight of (i,j) is supported by its original weight.
- If $\mathcal{S}$ contains a triangle (i,k,j) such that $\omega_{i\to j} = \omega_{i\to k} + \omega_{k\to j}$, we insert three edges:



(a) Original weights.

(b) Tightened weights.

(c) Support graph for Figure 5.1b.

Figure 5.1: Example of a support graph.

- One edge $(\triangle_{ikj}, \mu_{i\to j})$, showing that the minimum weight of the constraint (i, j) is supported by the presence of the triangle, and
- Two edges $(\mu_{i\to k}, \triangle_{ikj})$ and $(\mu_{k\to j}, \triangle_{ikj})$, showing that this support exists only because of the minimum weight of the constraint (i,k) and (k, j).

We summarize the above in the following formal definition of the weight support graph:

Definition 5.2. Let $\mathcal{S}$ be a PPC STN with events X and constraints C, such that each constraint $c_{i\to j}$ has the original weight $w_{i\to j}$ and the minimum weight $\omega_{i\to j}$. Then the graph $D = \langle V', E' \rangle$ is the *weight support graph* of $\mathcal{S}$, where V' and E' are defined as follows:

$$\begin{aligned} V' =& \{o_{i\to j}, \mu_{i\to j} \mid c_{i\to j} \in C\} \\ & \cup \{\triangle_{ikj} \mid c_{i\to k}, c_{k\to j}, c_{i\to j} \in C \wedge \omega_{i\to k} + \omega_{k\to j} = \omega_{i\to j}\} \\ E' =& \{(o_{i\to j}, \mu_{i\to j}) \mid \omega_{i\to j} = w_{i\to j}\} \\ & \cup \{(\mu_{i\to k}, \triangle_{ikj}), (\mu_{k\to j}, \triangle_{ikj}), (\triangle_{ikj}, \mu_{i\to j}) \mid \omega_{i\to k} + \omega_{k\to j} = \omega_{i\to j}\} \end{aligned}$$

Example 5.3. Consider the network in Figure 5.1a. As we can see, there are two paths from vertex a to c: following the constraint (a,c) directly, for a weight of $w_{a\to c} = 40$, or by following the path through constraints (a,b) and (b,c), for a total weight of $w_{a\to b} + w_{b\to c} = 30$. Since the path through b is shorter, it supports a tighter weight of $\omega_{a\to c} = 30$ on the constraint (a,c), as shown in Figure 5.1b.

Figure 5.1c shows the corresponding support graph. Since the weights of (a,b) and (b,c) retain their original values, there are edges from the original nodes $o_{a\to b}$ and $o_{b\to c}$ to the minimum nodes $\mu_{a\to b}$ and $\mu_{b\to c}$ respectively. The minimum weight

on the constraint (a,c) on the other hand is *not* supported by its original weight, so there is no edge from $o_{a\to c}$ to $\mu_{a\to c}$.

Instead, its weight is supported by the minimum weights of the constraints (a,b) and (b,c). Therefore, there is an incoming support edge on $\mu_{a\to c}$ from the triangle node $\triangle_{abc}$, which in turn has edges incoming from the minimum nodes $\mu_{a\to b}$ and $\mu_{b\to c}$. □

5.1.2 Properties

In Section 5.4 we present an algorithm to determine which other constraints are affected when the weight of one constraint is increased. This algorithm requires that the support graph is acyclic. Since the existence of a cycle implies that a constraint ultimately supports its *own* weight, this may seem absurd at first. When we adhere strictly to Definition 5.2 however, a cycle can in fact occur, if the underlying network contains a cycle with zero cumulative weight.

Fortunately, we are able to show that this is in fact the *only* case in which the support graph may contain a cycle. Hence, if we restrict ourselves to networks without such zero-cycles, our algorithm will work. In Section 5.7.1 we present ideas to lift this restriction.

Our proof here is in two stages. First we show that the existence of a cycle in D implies the existence of at least one circuit in the network $\mathcal{S}$. We then show that the cumulative weight of these circuits is exactly 0, from which our result follows.

In the first part of our proof we need to relate the edges in D to the constraints $\mathcal{S}$. In particular, we need the following property:

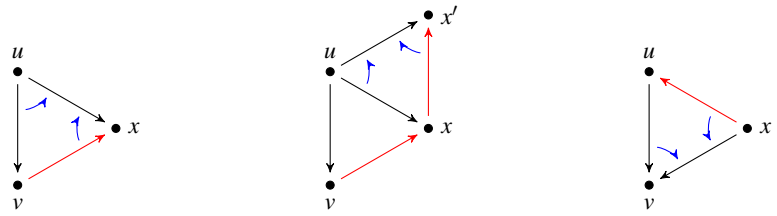
Definition 5.4. Let (u,v) be a constraint in $\mathcal{S}$ and let $\mu_{u\to v}$ be its corresponding minimum node in D. Then, if $P = \triangle_i \to \mu_{u\to v} \to \triangle_{i+1}$ is a path in D, we call (u,v) a *propagating constraint on P* between $\triangle_i$ and $\triangle_{i+1}$. Note that the triangle $\triangle_{i+1}$ has another incoming edge from some minimum node $\mu_{x\to y}$, which is not on P. We call the constraint (x,y) corresponding to this node a *non-propagating constraint on P*.

We are now ready for the first part of our proof.

Lemma 5.5. *Let D be the weight support graph of the PPC STN $\mathcal{S}$. Then, if there is a cycle in D, there is at least one circuit in $\mathcal{S}$, consisting of non-propagating edges on C.*

Proof. Let C be the cycle in D, i.e. $C = \mu_1 \to \triangle_1 \to \mu_2 \to \cdots \to \triangle_k \to \mu_{k+1} = \mu_1$. Now consider any subsequence $C' = \triangle_i \to \mu_{u\to v} \to \triangle_{i+1}$ of C. The existence of this subsequence implies that $\triangle_i$ supports the minimum weight of (u,v), which, together with some other constraint *not* in $\triangle_i$, forms the triangle $\triangle_{i+1}$ supporting the weight of some next constraint.

Consider the connection between $\mu_{u\to v}$ and $\triangle_{i+1}$. Note that the triangle represented by $\triangle_{i+1}$ has to include u and v, and some other vertex x. There are two possibilities, shown in Figures 5.2a and 5.2c.



(a) Option I: (u,v) and (v,x) support (u,x).

(b) Option I continued: (u,x) and (x,x') support (u,x').

(c) Option II: (x,u) and (u,v) support (x,v).

Figure 5.2: Possible constraint configurations in $\mathcal{S}$ if $\mu_{u\to v} \to \triangle_{i+1}$ is a subpath of a cycle in D. Black constraints are propagating, red constraints non-propagating; the blue arrows show support relations.

First, we can have that $\triangle_{i+1} = \triangle_{uvx}$, in which case constraints (u,v) and (v,x) support the minimum weight of (u,x), as shown in Figure 5.2a. Since $\triangle_{i+1}$ is on the cycle, and there is only one outgoing edge from any triangle node in D, this means that $\mu_{u\to x}$ is also on the cycle. This in turn means that (u,x) is propagating between $\triangle_{i+1}$ and $\triangle_{i+2}$, and, therefore, that (v,x) is *not* propagating between these two. Note that (v,x) is outgoing from v, and that the next *propagating* constraint (u,x) again starts from the starting vertex u of (u,x).

The other option is that $\triangle_{i+1} = \triangle_{xuv}$, i.e. constraints (x,u) and (u,v) support the minimum weight of (x,v), as shown in Figure 5.2c. By the same reasoning as above, this means that (x,v) is propagating between $\triangle_{i+1}$ and $\triangle_{i+2}$, and that (x,u) is not propagating. Note that in this case the non-propagating constraint is *outgoing* from u, and that the next propagating constraint (x,v) ends in v, the terminal vertex of (u,v).

If we apply this reasoning to subsequent sequences $\triangle_{i+1} \to \mu_{i+2} \to \triangle_{i+2}$ etc., we observe that we construct two directed paths through G, consisting of non-propagating constraints. Specifically, at each step, the non-propagating constraint is either:

1. *Ap*pended to a path *starting* in v, connecting *terminal* vertices of propagating constraints (illustrated in Figure 5.2b), or
2. *Pre*pended to a path *ending* in u, connecting *starting* vertices of propagating constraints.

Now, note that the cycle in D eventually has to come full circle and return to $\mu_{u\to v}$. So consider the connection between $\triangle_i$ and $\mu_{u\to v}$. We know that $\triangle_i$ contains u, v and some other vertex y, and has to support the weight of (u,v). Hence, we must have $\triangle_i = \triangle_{uyv}$ and constraints (u,y) and (y,v) support the weight of (u,v). Thus we again have two possibilities, shown in Figure 5.3: either (u,y) or (y,v) must be supported by $\triangle_{i-1}$, the previous triangle on C.

If $\triangle_{i-1}$ supports (u,y), then (y,v) is *not* supported by $\triangle_{i-1}$ and thus not propagating between $\triangle_{i-1}$ and $\triangle_i$. So in this case, (y,v) must be a non-propagating

(a) Option I: (u,y) is propagating. **(b)** Option II: (y,v) is propagating.

Figure 5.3: Possible edge configurations in $\mathcal{S}$ if $\triangle_i \to \mu_{u\to v}$ is a subpath of a cycle in D.

constraint *incoming* on v, as shown in Figure 5.3a. Since y is a terminal vertex of a propagating constraint on C, by the construction in the first part of this proof there is a path from v to y. The constraint (y,v) connects the endpoints of this path, thus creating a circuit.

Similarly, if $\triangle_{i-1}$ supports (y,v), then (y,v) is propagating between $\triangle_{i-1}$ and $\triangle_i$, and (u,y) is not. So here we have an *outgoing* non-propagating constraint from u, as in Figure 5.3b. Since y is a starting vertex of the propagating constraint (y,v), by the first part of this proof, there is a path from y to u. The non-propagating constraint (u,y) connects the endpoints of this path, thus creating a circuit.

The above shows that there is at least one circuit through G. It may be that either u or v is shared between *all* triangles on the path, in which case one of the circuits degenerates to consist of just that one vertex. However, since we are discussing *triangles*, there is always a third vertex involved, and thus there is always at least one non-propagating constraint to construct a circuit with.

Finally, we only claim *circuits* and not strict *cycles*, since it may be the case that some non-propagating constraint participates in multiple triangles: the above only shows that these triangles cannot be *consecutive* on C. □

Lemma 5.6. *If the consistent PPC STN $\mathcal{S}$ does not contain any cycles of cumulative weight 0, its corresponding weight support graph D is acyclic.*

Proof. We will prove the contrapositive of the statement above: if D contains a cycle, there must be a zero-weight cycle in $\mathcal{S}$. So consider a weight support graph D that contains a cycle $C = \mu_1 \to \triangle_1 \to \mu_2 \to \cdots \to \triangle_k \to \mu_{k+1} = \mu_1$. By Lemma 5.5, we know that $\mathcal{S}$contains one or more circuits consisting of non-propagating constraints. We now show that the cumulative weight of these circuits is exactly zero.

By the definitions of D and of propagating constraints on a path, we have the following two facts. First, every triangle $\triangle_i$ on C has incoming support from one propagating constraint p_i and from one non-propagating constraint $\bar{p}_i$. The triangle itself supports a *leaving* constraint ℓ_i, which is the constraint propagating to the next triangle $\triangle_{i+1}$.

Second, the existence of the triangle $\triangle_i$ implies that $\omega_{p_i} + \omega_{\bar{p}_i} = \omega_{\ell_i}$. We can rewrite this to $\omega_{p_i} + \omega_{\bar{p}_i} - \omega_{\ell_i} = 0$. Summing these equations over all triangles on C,

we have:

$$\sum_{1\leq i\leq k} \omega_{p_i} + \sum_{1\leq i\leq k} \omega_{\bar{p}_i} - \sum_{1\leq i\leq k} \omega_{\ell_i} = 0 \tag{5.1}$$

Since the leaving constraint ℓ_i of every triangle $\triangle_i$ is the incoming propagating constraint p_{i+1} of $\triangle_{i+1}$, we have $\omega_{\ell_i} = \omega_{p_{i+1}}$ for $1 \leq i < k$ and $\omega_{\ell_k} = \omega_{p_1}$. Therefore, we have $\sum_{1\leq i\leq k} \omega_{p_i} = \sum_{1\leq i\leq k} \omega_{\ell_i}$, which means that the first and third term in Equation (5.1) cancel out. This leaves:

$$\sum_{1\leq i\leq k} \omega_{\bar{p}_i} = 0$$

So the total weight of the circuits spanned by the non-propagating constraints is exactly zero. A circuit may consist of many cycles, but since $\mathcal{S}$ was consistent, none of these cycles can be negative. Since the cumulative weight of the circuit is zero, this means that no cycle can have positive weight either, and therefore all component cycles of the circuit have weight 0. □

5.2 Initializing the weight support graph

Provided the network $\mathcal{S}$ is partially path consistent, we can initialize its weight support graph $D = \langle V', E' \rangle$ using the algorithm Construct-Support-Graph, shown in Algorithm 5.1. To do so, we construct a preliminary support graph D containing only the original nodes $o_{u\to v}$ and the minimum nodes $\mu_{u\to v}$ for every constraint (u, v), and *mark* every minimum node.

We then re-trace the steps of the P^{3}C algorithm, in the subprocedure Update-Support-Graph, shown in Algorithm 5.2. Every triangle in the graph is visited again, allowing us to reconstruct how the weights of its constraints depend on each other. This information is captured to produce a weight support graph conforming to Definition 5.2.

The notion of marking is important when processing decremental updates, since it gives us fine-grained control over the constraints whose support is updated. For our current purpose of initialisation however, we want to obtain the support information for *all* constraints, and therefore simply mark all minimum nodes.

Lemma 5.7 below proves a key property of Update-Support-Graph. We use this property to show the correctness of the initialization in Theorem 5.8.

Lemma 5.7. *Given a PPC STN $\mathcal{S}$,* Update-Support-Graph *correctly inserts exactly all edges in E' that either end in a marked minimum node, or end in a triangle supporting a marked minimum node. The run time of* Update-Support-Graph *is bounded by $O(nw_d^2)$.*

Proof. First of all, since $\mathcal{S}$ is already PPC, we know that all minimum weights are correctly set. We now show that all support relations incoming on a marked minimum node in D indeed conform to Definition 5.2.

Algorithm 5.1: Construct-Support-Graph

Input: A PPC STN $\mathcal{S} = \langle V, E \rangle$ with each constraint $c_{i \to j} \in C$ annotated by its original weight $w_{i \to j}$ and its minimum weight $\omega_{i \to j}$, and a simplicial elimination ordering $d = (v_n, v_{n-1}, \ldots, v_1)$.

Output: The weight support graph D for $\mathcal{S}$.

```
1 V', E' ← ∅
2 foreach (u, v) ∈ E do
3     add o_{u→v} and μ_{u→v} to V'
4     mark μ_{u→v}
5 D ← (V', E')
6 call Update-Support-Graph(S, d, D)
7 return D
```

Algorithm 5.2: Update-Support-Graph

Input: (1) A PPC STN $\mathcal{S} = \langle V, E \rangle$ with each constraint $c_{i \to j} \in C$ annotated by its original weight $w_{i \to j}$ and its minimum weight $\omega_{i \to j}$, and a simplicial elimination ordering $d = (v_n, v_{n-1}, \ldots, v_1)$. (2) A preliminary weight support graph $D = \langle V', E' \rangle$ in which no edges are adjacent to any marked minimum node.

Output: A weight support graph D with correct supports for all minimum nodes that were originally marked.

```
1  foreach (u, v) ∈ E do
2      if ω_{u→v} = w_{u→v} and μ_{u→v} is marked then
3          add (o_{u→v}, μ_{u→v}) to E'
4  for k ← 1 to n do
5      foreach i, j < k such that {v_i, v_k}, {v_j, v_k} ∈ E do
6          call find-support(i, k, j)
7          call find-support(k, j, i)
8          call find-support(i, j, k)
9  foreach (u, v) ∈ E do clear mark on μ_{u→v}
10 return D
```

Procedure find-support(x, y, z)

```
1 if μ_{x→y} is marked and ω_{x→y} = ω_{x→z} + ω_{z→y} then
2     add △_{xzy} to V'
3     add (μ_{x→z}, △_{xzy}), (μ_{z→y}, △_{xzy}) and (△_{xzy}, μ_{x→y}) to E'
```

First, observe that in the loop on lines 1 to 3, we insert exactly those edges between original and marked minimum nodes required by the definition.

Now consider the second loop, from line 4 through line 8. We show that we visit each (directed) constraint, and add the correct support if its corresponding minimum node is marked. Note that any triangle $\{x,y,z\}$ contains six directed constraints. W.l.o.g. assume that $z > x,y$ in the elimination ordering d. In that case, the body of the inner loop visits $\{x,y,z\}$ twice: once for $(i,j,k) = (x,y,z)$ and once for $(i,j,k) = (y,x,z)$.

During the first iteration through the loop, find-support considers the constraints (x,z), (z,y) and (x,y) in $\mathcal{S}$, during the second iteration, it considers (y,z), (z,x) and (y,x). Thus, we consider every directed constraint in every triangle. The first part of the condition on line 1 of find-support ensures that we only add support incoming on marked minimum nodes. Since the second part of the condition is exactly the condition from Definition 5.2, if $\mu_{x\to y}$ is marked we add exactly those triangle nodes and support edges that the definition requires.

Regarding the time complexity, the first loop visits each edge of the chordal graph once, in $O(m_c)$ time, while the second loop performs the same iterations as P^{3}C and therefore takes $O(nw_d^2)$ time. Since $m_c = O(nw_d^2)$, we arrive at a total run time of $O(nw_d^2)$. □

Theorem 5.8. *Given a PPC STN $\mathcal{S}$,* Construct-Support-Graph *computes the weight dependency graph D corresponding to $\mathcal{S}$ in $O(nw_d^2)$ time.*

Proof. The preliminary support graph D constructed in lines 1 to 5 contains only original and minimum nodes, and all minimum nodes are marked. Note that by definition all edges in the weight support graph must either end in a minimum node or in a triangle node. Since *all* minimum nodes in the preliminary graph D are marked, by Lemma 5.7, Update-Support-Graph adds the correct edges to all minimum nodes, and therefore the resulting weight support graph D is correct.

Constructing the preliminary graph takes $O(m_c)$ time, which is dominated by the $O(nw_d^2)$ time required by Update-Support-Graph. The overall time required is therefore bounded by $O(nw_d^2)$. □

5.3 Processing decremental updates

The weight support graph gives us a powerful tool to determine what needs to be done when the weight of a constraint (a,b) is raised. We now use it for the main contribution of this chapter: an algorithm capable of re-enforcing partial path consistency, taking the raised weight of (a,b) into account.

As we noted earlier, the minimum weight $\omega_{i\to j}$ of a constraint (i,j) is valid only if it is supported by either its original weight or some triangle. We can now make this requirement more specific:

Lemma 5.9. *The minimum weight $\omega_{i\to j}$ of the constraint $(i,j) \in \mathcal{S}$ is only correct if its corresponding minimum node $\mu_{i\to j} \in D$ has* at least one *incoming edge.*

Proof. Recall that there are only two ways in which the value of $\omega_{i\to j}$ can be derived: either it is implied by an original weight, or it is implied by some triangle. Now suppose for a contradiction that $\omega_{i\to j}$ is correctly set, but $\mu_{i\to j}$ has *no* incoming edges. By the definition of D, this means that neither $w_{i\to j} = \omega_{i\to j}$, since then there would be an incoming edge from $o_{i\to j}$, nor is there any triangle (i,k,j) such that $\omega_{i\to k} + \omega_{k\to j} = \omega_{i\to j}$, since then there would be an incoming edge from $\triangle_{ikj}$. □

This suggests the following algorithm for updating the weights in an STN, given that the weight $w_{a\to b}$ of the constraint (a,b) is increased to $w'_{a\to b} > w_{a\to b}$. First, note that we only need to do any work if there is an edge between the original node $o_{a\to b}$ and the minimum node $\mu_{a\to b}$. If there is no such edge, the weight of the shortest path from a to b does not depend on the weight of the edge (a,b), so when we raise it none of the shortest paths in the graph need to change.

If the edge $(o_{a\to b}, \mu_{a\to b})$ *does* exist in D, we remove it, since with the increased weight it is no longer true that $w'_{a\to b} = \omega_{a\to b}$. At this point we can use Lemma 5.9: if there are no other incoming edges on $\mu_{a\to b}$, apparently its minimum weight is no longer valid. Therefore, any triangles that depended on $\mu_{a\to b}$ can no longer be valid either, so we remove them from the graph. This removal causes more edges to disappear from D, and may cause other minimum nodes to lose all support, causing the removal of yet more triangles.

We continue this process until there are no new minimum nodes that lose all support. By tracking the endpoints of any constraint (u,v) whose corresponding minimum node $\mu_{u\to v}$ is removed, we obtain a set V^* of all vertices adjacent to a constraint whose weight needs to change. In other words, we need only re-enforce PPC in the subgraph $\mathcal{S}_{V^*}$ induced by the vertices in V^*. When we traverse the constraints to be changed, we make sure to reset their minimum weights to their original values. The new minimum weights will be computed when we re-enforce PPC.

Unfortunately, simply running the P^{3}C algorithm on $\mathcal{S}_{V^*}$ will not suffice to accomplish this. Specifically, when the weight of (a,b) is raised, the new shortest path π_{xy} between the two endpoints x and y of an edge in $\mathcal{S}_{V^*}$ may now visit some vertex z outside V^*. In Theorem 5.22 we show that we can include knowledge about the shortest paths external to V^* by including all vertices that neighbour on at least two vertices in V^*. It therefore suffices to run the P^{3}C algorithm on the graph induced by this extended set to enforce partial path consistency on the entire graph. Once partial path consistency is re-established, we update the weight support graph by calling Update-Support-Graph.

The entire process just discussed is summarized in Algorithms 5.3 to 5.5. Support-DPPC, shown in Algorithm 5.3, performs the pre-processing and contains the high-level control flow. The process of finding V^* is formalized in Algorithm 5.4, dubbed Affected-Endpoints. Finally, Algorithm 5.5 shows the algorithm used to determine the set of vertices that neighbour on at least two vertices in V^*.

Recall that the sets $N_{\text{in}}(\mu_{u\to v})$ and $N_{\text{out}}(\mu_{u\to v})$ used in Affected-Endpoints and line 6 represent the sets of incoming and outgoing neighbours respectively. Furthermore,

note that by Definition 5.2 all outgoing edges of a minimum node point at triangle nodes, and hence we need not apply any filtering on line 4 of Affected-Endpoints.

In the following two sections, we prove the correctness of these algorithms and give bounds on their run time. Section 5.4 discusses the Affected-Endpoints algorithms, the construction of the extended set V^+ and the re-enforcement of partial path consistency are presented in Section 5.5.

5.4 Finding the affected endpoints

The difficulty in proving the correctness of Affected-Endpoints lies in the fact that even if the minimum weight $\omega_{u\to v}$ for the constraint (u,v) is supported by multiple shortest $u-v$ paths, it may still be the case that *each* of these paths contains the constraint (a,b) whose weight is raised. In that case u and v should of course be included in V^*, but we need to show how the algorithm makes progress in tearing down the different paths supporting $\omega_{u\to v}$, until it ultimately concludes that no such paths remain.

In Section 5.4.1 we introduce the notion of a *hop-maximal shortest path*, which provides exactly such a measure. After proving three supporting lemmas in Section 5.4.2, we use this notion to inductively prove the correctness of Affected-Endpoints in Section 5.4.3. Finally, in Section 5.4.4 we exploit this proof to demonstrate the correctness of a slightly more complex version of the algorithm, which has the benefit of being more efficient.

5.4.1 Hop-maximal shortest paths

The measure of progress needed for our induction proof is provided by the concept of *hop-maximal shortest $u-v$ paths*. Let $|P|$ denote the number of edges in the path P.

Definition 5.10. Let π be a shortest $u-v$ path. Then π is *hop-maximal shortest* if there is no shortest $u-v$ path π' such that $|\pi'| \geq |\pi|$. We let $hms(u,v)$ denote the set of all hop-maximal shortest $u-v$ paths.

In other words, if $\pi \in hms(u,v)$ is a hop-maximal shortest $u-v$ path, there is no $u-v$ path consisting of *more* edges whose cumulative weight is still $\omega_{u\to v}$.

As we will see, Affected-Endpoints visits edges (u,v) in order of increasing number of edges on the hop-maximal shortest $u-v$ paths. To prove the correctness of Affected-Endpoints in Theorem 5.17, we will exploit the following property of hop-maximal shortest paths:

Lemma 5.11. *Let $u \dashrightarrow x \dashrightarrow v$ be a shortest $u-v$ path and let π_{uv}, π_{ux} and π_{xv} be hop-maximal shortest $u-v$, $u-x$ and $x-v$ paths respectively. Then π_{uv} has at least as many edges as the sum of the edges on π_{ux} and π_{xv}, i.e. $|\pi_{ux}| + |\pi_{xv}| \leq |\pi_{uv}|$.*

Algorithm 5.3: Support-DPPC

Input: (1) A PPC STN $\mathcal{S} = \langle V, E \rangle$ with original weights $w_{i\to j}$, lowest weights $\omega_{i\to j}$, and a simplicial elimination ordering d. (2) The weight support graph $D = \langle V', E' \rangle$ of $\mathcal{S}$. (3) A new weight $w'_{a\to b} > w_{a\to b}$ for the edge (a, b).

Output: PPC has been re-enforced in $\mathcal{S}$ given $w_{a\to b} = w'_{a\to b}$, and D has been updated to reflect the new weight supports.

```
1  if (o_{a→b}, μ_{a→b}) ∉ E' then
2      w_{a→b} ← w'_{a→b}
3      return
4  w_{a→b} ← w'_{a→b}
5  remove (o_{a→b}, μ_{a→b}) from E'
6  if N_in(μ_{a→b}) ≠ ∅ then return
7  V* ← Affected-Endpoints(a, b)
8  V+ ← V* ∪ Shared-Neighbours(V*)
9  call P³C(S_{V+}, d_{V+})
10 call Update-Support-Graph(S_{V+}, d_{V+})
```

Algorithm 5.4: Affected-Endpoints(a, b)

```
1  while there is an unmarked μ_{u→v} with N_in(μ_{u→v}) = ∅ do
2      mark μ_{u→v} ; add u and v to V*
3      reset ω_{u→v} to w_{u→v}
4      foreach △_{xyz} ∈ N_out(μ_{u→v}) do
5          remove △_{xyz} and edges attached to it from D
6  return V*
```

Algorithm 5.5: Shared-Neighbours(V^*)

```
1  result ← ∅
2  foreach u ∈ V* do
3      foreach v ∈ N(u) such that v ∉ V* do
4          increment COUNT[v]
5          if COUNT[v] = 2 then
6              add u to result
7  return result
```

Proof. We will assume that there exist $\pi_{ux} \in hms(u,x)$ and $\pi_{vx} \in hms(x,v)$ such that $|\pi_{ux}| + |\pi_{xv}| > |\pi_{uv}|$, and arrive at a contradiction.

Consider the path π'_{uv} created by concatenating π_{ux} and π_{xv}. By our assumption, we know that $|\pi'_{uv}| > |\pi_{uv}|$. Since π_{ux} and π_{xv} are shortest paths to and from x, they have weights $\omega_{u\to x}$ and $\omega_{x\to v}$, so π'_{uv} is a shortest $u-v$ path. But π'_{uv} contains more hops than π_{uv}, so π_{uv} was not a *hop-maximal* shortest $u-v$ path, contradicting the condition from the lemma statement. Therefore π'_{uv} cannot exist. □

5.4.2 Supporting lemmas

Before constructing the induction proof itself, we need a general property of shortest paths, and two observations on the behaviour of the Affected-Endpoints algorithm.

Lemma 5.12. *Suppose that the weight of the edge (u,v) must be raised due to an increase of the weight of the edge (a,b) in a network which contains no zero-length cycles. Then, if there is a shortest path $u \dashrightarrow x \dashrightarrow y \dashrightarrow v$, such that $a \to b$ is on $x \dashrightarrow y$, and there exists an edge (x,y), the weight of (x,y) must be increased.*

Proof. First of all, note that neither $u \dashrightarrow x$ nor $y \dashrightarrow v$ can include $a \to b$. If either did, (a,b) would repeat, and the path either would not be shortest, or there would be a zero-length cycle.

Now suppose for the purpose of obtaining a contradiction that the weight of (x,y) need not be raised. Then there must be some other shortest $x-y$ path π that does not include $a \to b$. But then the concatenation of $u \dashrightarrow x$, π and $y \dashrightarrow v$ would be a shortest $u-v$ path whose weight is independent of $a \to b$. This would mean that the weight of (u,v) need not be raised if the weight of (a,b) is raised, contradicting the statement of the lemma.

Therefore, all shortest $x-y$ paths must contain $a \to b$, and the weight of (x,y) must increase if the weight of (a,b) increases. □

Lemma 5.13. *During the main loop of* Affected-Endpoints*, if the weight of edge (u,v) needs to change following a weight increase on (a,b), every incoming edge on $\mu_{u\to v}$ in D originates from some triangle node $\triangle_{uxv}$ (with $x \in V$).*

Proof. Observe that all edges in D are either between original and minimum nodes or between triangle and minimum nodes. We therefore need only show the absence of the edge $(o_{u\to v}, \mu_{u\to v})$ for every edge (u,v) whose weight needs to be raised.

There are two cases: either (u,v) is (a,b), or it is not. If $(u,v) = (a,b)$, the edge $(o_{a\to b}, \mu_{a\to b})$ is explicitly removed on line 5 of Support-DPPC, and therefore does not exist when the main loop of Affected-Endpoints is executed.

Now consider the case where $(u,v) \neq (a,b)$. If there were an edge $(o_{u\to v}, \mu_{u\to v})$, this would imply that the minimum weight for (u,v) is equal to its original weight. Furthermore, since $(u,v) \neq (a,b)$, this original weight does not change. But then the minimum weight for (u,v) would not need to change, contradicting the statement that it *does* need to change. Therefore, the edge $(o_{u\to v}, \mu_{u\to v})$ cannot exist. □

Lemma 5.14. *If* Affected-Endpoints *does not mark a node $\mu_{u\to v}$ corresponding to a changed edge (u,v), $\mu_{u\to v}$ has at least one incoming edge.*

Proof. This follows immediately from lines 1 and 2: if $\mu_{u\to v}$ had no incoming edge, it would be marked. □

5.4.3 Correctness proof

We now have sufficient information to complete the proof of correctness of Affected-Endpoints. As mentioned earlier, we will use an induction proof, over the number of edges in the hop-maximal shortest $u-v$ paths for edges whose weight must change.

Lemma 5.15 proves the induction step. In Lemma 5.16 we then show that (a,b) is the only edge we need to consider for the base. Finally, these two are combined in the induction proof for Theorem 5.17.

Lemma 5.15. *Assume* Affected-Endpoints *marks all nodes $\mu_{u'\to v'}$ corresponding to edges (u',v') whose weight must change and whose hop-maximal shortest paths contain* fewer *than k edges. Then* Affected-Endpoints *also marks all nodes $\mu_{u\to v}$ corresponding to edges (u,v) whose weight must change and whose hop-maximal shortest paths contain* exactly *k edges.*

Proof. Suppose, for the purpose of obtaining a contradiction, that there is some node $\mu_{u\to v}$ not marked by Affected-Endpoints, even though its corresponding edge (u,v) must be changed and every hop-maximal shortest path $\pi_{uv} \in hms(u,v)$ has $|\pi_{uv}| = k$.

By Lemma 5.14, $\mu_{u\to v}$ has at least one incoming edge, and by Lemma 5.13 this edge originates at a triangle node. In other words, there must be an $x \in V$ such that $\mu_{u\to x}$ and $\mu_{x\to v}$ point to $\triangle_{uxv}$. This means that there must be a shortest $u-v$ path $u \dashrightarrow x \dashrightarrow v$ through x. Since the weight for (u,v) must change, every shortest $u-v$ path must contain $a \to b$, so either $u \dashrightarrow x$ or $x \dashrightarrow v$ must contain $a \to b$. We assume $u \dashrightarrow x$ contains $a \to b$; the case for $x \dashrightarrow v$ is analogous.

Consider any hop-maximal shortest $u-v$ path $\pi_{uv} \in hms(u,v)$, and let π_{ux} and π_{xv} be hop-maximal shortest $u-x$ and $x-v$ paths respectively. By Lemma 5.11, we know that $|\pi_{ux}| + |\pi_{xv}| \leq |\pi_{uv}|$. Since any hop-maximal shortest path consists of at least one edge, we have $|\pi_{ux}| < |\pi_{uv}| = k$. Furthermore, since $u-x$ is a subpath of $u \dashrightarrow x \dashrightarrow v$ containing $a \to b$, and (u,x) is an edge of G, we have by Lemma 5.12 that the weight on (u,x) must change.

So, since $|\pi_{ux}| < k$ and the weight of (u,x) must change, by the assumption in the statement of this lemma, $\mu_{u\to x}$ must have been marked by Affected-Endpoints. But then all $\triangle_{uxy}$ and $\triangle_{yux}$ were removed from D in lines 4 through 5. In particular, $\triangle_{uxv}$ and its outgoing edge $(\triangle_{uxv}, \mu_{u\to v})$ must have been removed, contradicting the assumption that this edge blocked the algorithm from marking $\mu_{u\to v}$.

This argument applies to *any* triangle node with an edge pointing at $\mu_{u\to v}$, so $\mu_{u\to v}$ can have no edge originating from a triangle node. Since by Lemma 5.13 all incoming edges on $\mu_{u\to v}$ originate at triangles, we conclude that $\mu_{u\to v}$ has no

incoming edges. Therefore the condition on line 1 will eventually be fulfilled and Affected-Endpoints will mark $\mu_{u\to v}$. □

Lemma 5.16. *The edge (a,b) is the only affected edge whose hop-maximal shortest path has length 1.*

Proof. Any shortest path for any other edge (u,v) whose weight must increase must contain (a,b) as a strict subpath. Therefore the hop-maximal shortest path $u-v$ path of such an edge will have length at least 2.

The hop-maximal shortest path for (a,b) must have length 1, for otherwise there is some shortest path that does not use the direct edge, and the weight for (a,b) need not have changed. □

Theorem 5.17. Affected-Endpoints(a,b) *marks all minimum nodes corresponding to edges whose weight must be raised given that the weight of (a,b) has increased. Moreover, the set V^* it returns is exactly the set of vertices that are adjacent to at least one edge whose weight must be raised.*

Proof. We will prove this by induction over the number of edges in the hop-maximal shortest $u-v$ path.

For the base case, consider the edges whose hop-maximal shortest $u-v$ path consists of a single edge, which by Lemma 5.16 can only be (a,b). By Lemma 5.13, any incoming edge on $\omega_{a\to b}$ must be a triangle edge. However, since we only enter Affected-Endpoints when we *know* (a,b) must be changed and by Lemma 5.6 D contains no cycles, there can be no such incoming triangle edge on $\mu_{a\to b}$. Hence, we have $N_{\text{in}}(\mu_{a\to b}) = \varnothing$ and the condition on line 1 is fulfilled, which causes $\mu_{a\to b}$ to be marked on line 2. So all edges whose hop-maximal shortest $u-v$ paths have length 1 are marked.

For the induction step, we assume that Affected-Endpoints marks all nodes $\mu_{u\to v}$ corresponding to edges (u,v) whose weight must change and whose hop-maximal shortest paths contain fewer than k edges. By Lemma 5.15, the algorithm will then also mark all edges (u',v') that need to change and whose hop-maximal shortest $u'-v'$ paths contain k edges. By induction, Affected-Endpoints therefore marks *all* edges (u,v) whose weight must change.

Finally, note that the endpoints of any changed edge are added to V^* on line 2, right after its corresponding minimum node is marked. Therefore, the set V^* returned by Affected-Endpoints does indeed contain all vertices attached to at least one edge whose weight must be raised, given that the weight of (a,b) has increased. □

5.4.4 Efficiency

Having shown the correctness of Affected-Endpoints, we now consider the amount of time it requires to find all affected edges. Note that while the pseudocode shown in Algorithm 5.4 finds the correct edges, it requires us to repeatedly scan the entire support graph for minimum nodes whose in-degree has dropped to 0. Since there are

Algorithm 5.6: Affected-Endpoints-Advanced(a,b)

```
1  enqueue μ_{a→b} in the queue Q
2  while Q is not empty do
3      dequeue μ_{u→v} from Q
4      if N_in(μ_{u→v}) = ∅ then
5          mark μ_{u→v} ; add u and v to V*
6          reset ω_{u→v} to w_{u→v}
7          foreach △_{xyz} ∈ N_out(μ_{u→v}) do
8              enqueue μ_{x→z} in Q
9              remove △_{xyz} and edges attached to it from D
10 return V*
```

$O(m_c)$ minimum nodes and in the worst case all of them change, this naive implementation may require $O(m_c^2)$ time.

To improve over this, we make the following observation: If the in-degree of minimum node drops to 0, its last incoming support edge must have been removed in some earlier iteration. Moreover, by Lemma 5.13, that support edge must have originated at a triangle.

In other words, the above means that when a minimum node μ loses all support, we know the only other minimum nodes that could be directly affected by this are exactly those nodes who were supported by some triangle supported by μ. We can therefore maintain an explicit *queue* of minimum nodes that we still need to visit, which means we no longer need to scan the entire support graph in any iteration. An advanced version of the Affected-Endpoints algorithm exploiting this idea is shown in Algorithm 5.6.

We now show that Affected-Endpoints-Advanced has the same end result as Affected-Endpoints. This will, again, be done by induction over the lengths hop-maximal shortest paths. However, in this case we need a slightly higher-level property, which we prove in Lemma 5.18. Lemma 5.19 then shows the equivalence of Affected-Endpoints-Advanced and Affected-Endpoints, following which we determine a bound on its run time in Theorem 5.20.

Lemma 5.18. *If an edge (u,v) supports the weight of some other edge (u',v'), the hop-maximal shortest path from u to v has fewer edges than the hop-maximal shortest path from u' to v'.*

Proof. Assume for a contradiction that there is some edge (u,v) whose hop-maximal shortest $u-v$ path π_{uv} has length k, such that the weight of (u,v) is supported by some edge (x,y) whose hop-maximal shortest $x-y$ path π_{xy} is of length $\ell \geq k$. Since (u,v) is supported by (x,y), we either have that $x = u$ and there is a path $\pi'_{uv} : u = x \dashrightarrow y \dashrightarrow v$, or we have $y = v$ and there is a path $\pi''_{uv} : u \dashrightarrow x \dashrightarrow y = v$. Both π'_{uv}

and π''_{uv} are shortest $u-v$ paths, but contain at least one more edge than π_{uv}. But then π_{uv} cannot have been a *hop-maximal* shortest $u-v$ path, a contradiction. □

Lemma 5.19. Affected-Endpoints *and* Affected-Endpoints-Advanced *mark the exact same minimum nodes and remove the same triangle nodes.*

Proof. We will prove this by induction over the length of the hop-maximal shortest paths of affected edges.

For the base case, we consider (a,b), which by Lemma 5.16 is the only edge whose hop-maximal shortest $u-v$ path consists of a single edge. On line 1 we explicitly add the minimum node corresponding to (a,b) to the queue. Since we only enter Affected-Endpoints-Advanced when we know (a,b) must change, $\mu_{a\to b}$ has no incoming support and therefore it is marked on line 5.

For the induction step, assume that Affected-Endpoints and Affected-Endpoints-Advanced mark the exact same minimum nodes $\mu_{u'\to v'}$ for all edges (u',v') whose hop-maximal shortest $u'-v'$ paths have length $k-1$ or less. Note that the same minimum node $\mu_{x\to y}$ may occur in the queue multiple times: once for each triangle node $\triangle_i$ that supported it. By Lemma 5.18 each such $\triangle_i$ must itself have been supported by some minimum node $\mu_{x'\to y'}$ whose hop-maximal shortest path has fewer edges than the hop-maximal shortest $x-y$ path.

Consider the last time any node $\mu_{u\to v}$ is dequeued, where the hop-maximal $u-v$ path is of length k. By the induction hypothesis, all minimum nodes whose hop-maximal shortest path has fewer edges than k have been removed, and therefore, by line 9, all edges incoming on $\mu_{u\to v}$ from triangles supported by such nodes have been removed as well. By Lemma 5.13, the only possible remaining support at this point is from triangles supported by minimum nodes whose corresponding hop-maximal shortest paths have length at least k. But by Lemma 5.18 these nodes cannot support $\mu_{u\to v}$. Therefore, no incoming edges remain, and $\mu_{u\to v}$ is correctly marked on line 5.

By induction, the above shows that Affected-Endpoints and Affected-Endpoints-Advanced mark the same minimum nodes. Finally, note that both Affected-Endpoints and Affected-Endpoints-Advanced remove all triangles adjacent to a minimum node immediately after it is marked. Since both algorithms mark the same minimum nodes, they therefore also remove the same triangle nodes, concluding the proof. □

Theorem 5.20. Affected-Endpoints-Advanced *performs the same modifications on* $\mathcal{S}$ *and* D *as* Affected-Endpoints *in* $O(t^*)$ *time, where* t^* *is the number of triangles that contain an affected edge.*

Proof. The equivalence of the modifications follows from Lemma 5.19, so we focus on the run time.

After adding $\mu_{a\to b}$, a minimum node $\mu_{u\to v}$ can only be added to the queue if it is supported by a triangle about to be removed, i.e. on lines 8 and 9. Every such triangle is visited because it was supported by an edge whose weight needed to change, and is removed from the graph immediately after $\mu_{u\to v}$ is added to the queue. Hence, every

triangle containing an affected edge causes exactly *one* entry in the queue, and the outer loop is executed at most $O(t^*)$ times.

Moreover, the inner loop in lines 7 to 9 is executed exactly once for each triangle containing an affected edge. Therefore, amortized over all affected edges, this loop is also executed at most $O(t^*)$ times during the entire run of Affected-Endpoints-Advanced. We show below how to remove a triangle from D in constant time, so we can execute the body of the inner loop in constant time as well, and therefore the entire procedure takes at most $O(t^*)$ time. □

To delete triangle nodes from D in constant time, we use the following implementation. We store the outgoing neighbours of every minimum node in a linked list. For each triangle, we keep a reference to the list nodes in the linked lists of its incoming and outgoing minimum nodes. To remove the triangle, we simply remove these list nodes from their linked lists, which can be done in constant time.

5.5 Re-enforcing partial path consistency

Having demonstrated how to efficiently obtain the set of affected endpoints, we now consider the next phase in the algorithm: re-enforcing partial path consistency. Essentially, we have to perform two tasks. We need to make sure that the minimum weights are correctly set, and we need to update the weight support graph to reflect the new support relations.

We start with the first task, re-enforcing partial path consistency. Recall from our general introduction in Section 5.3 that this requires the extended set V^+. Lemma 5.21 shows that the Shared-Neighbours algorithm find the vertices that we need in addition to V^*.

Lemma 5.21. *The set computed by* Shared-Neighbours *contains all vertices neighbouring on at least two vertices in* V^*.

Proof. Consider any vertex $x \notin V^*$ neighbouring on vertices $u, v \in V^*$. W.l.o.g. suppose that u is the first of the neighbours of x to be visited in the outer loop of Shared-Neighbours, and v the second. When the neighbours of u are visited in the inner loop, $\text{COUNT}[x]$ is incremented to 1. Then, when the outer loop visits v, $\text{COUNT}[x]$ is incremented to 2, and consequently added to the result, as required. □

We are now ready to prove that our algorithm re-enforces partial path consistency.

Theorem 5.22. *After running Support-DPPC, the minimum weight* $\omega_{u\to v}$ *associated with any constraint* (u,v) *is the lowest cumulative weight along any path from u to v, i.e. the network is partially path consistent.*

Proof. Note that unless (u,v) is an affected edge, its minimum weight is already correctly set before we run the algorithm. So we only need to show that the minimum weights for affected edges (u,v) are set correctly. Hence, consider any such edge

(u,v). We distinguish three cases, based on the number of edges in the $u-v$ path with lowest cumulative weight.

First, this path may simply be the edge (u,v) itself. The minimum weight for (u,v) was reset to its original weight in line 3 of Affected-Endpoints, and since this is the minimum weight P^{3}C will not lower it further, nor increase it. So in this case, our theorem holds.

Secondly, the path of lowest weight may have consisted of two edges, i.e. it was of the form $u \to x \to v$ for some x. Since (u,v) is affected, u and v must be in V^*. But then x is a common neighbour of two vertices in V^*, and therefore we must have that $x \in V^+$. This means that the path is fully contained within $\mathcal{S}_{V^+}$, and therefore running P^{3}C on this subgraph will set the correct minimum weight.

Finally, it may be that the path π of lowest weight has more than two edges. We now show that in this case π can be reduced to a path π', which contains one edge less. Since π contains more than two edges, there must be a cycle $C = u-\pi-v-u$ of more than three edges (disregarding directionality). Hence, the graph induced by the vertices on this cycle must have a simplicial vertex x. We distinguish two sub-cases: either x is one of u or v, or x is neither.

First, suppose x is neither u or nor v. Let x' and x'' be the neighbours of x on C. Since x is simplicial, there must be an edge between x' and x''. If any two of the three vertices x, x' and x'' are in V^*, the remaining vertex must be in V^+. Therefore, P^{3}C will correctly derive the minimum weight on the edge between x' and x'', and we can eliminate x from the cycle. On the other hand, if at most one of the three is in V^*, none of the edges can be affected and we do not need to adjust any weights. So in this case as well we can eliminate x, by taking the shortcut between x' and x'' instead. Observe that in both cases, we reduce π to a path π' with equivalent weight, but containing one edge fewer.

Now suppose x is one of u or v. W.l.o.g. we assume $x = u$, and let u' be u's neighbour on C other than v. Since u is simplicial, there is an edge between u' and v. Hence, u' is connected to two vertices in V^*, and is therefore in V^+. Note that the $u'-v$ path of lowest weight is a subpath of the $u-v$ path of lowest weight, but one edge shorter. There are two cases: either (u,v') is itself affected or it is not. If it is not, we immediately know that the minimum weight for (u',v) is correct, and P^{3}C will enforce the correct weight on (u,v). If (u,v') *is* affected, we can recursively apply our reasoning to the lowest weight $u'-v$ path, which is one edge shorter than π. Since this ultimately reduces to a path of length two, for which we already know we derive the correct weight, we must also arrive at the correct weight for (u',v). P^{3}C then enforces the correct weight on (u,v).

Note that regardless of whether x is u, v, or neither, we have shown that either (1) P^{3}C determines the correct weight, or (2) that this weight can be obtained from a reduced path π', one edge shorter than the original $u-v$ path. By repeating this reduction, any path eventually reduces to length two, in which case we have already shown we provide the correct answer. Hence, every edge (u,v) is assigned the minimum weight of any $u-v$ path, and the resulting network is partially path consistent. □

Finally, we show that we correctly update the weight support graph for use with future updates. For the bound on the run time, we let n^+ denote the number of vertices in V^+.

Lemma 5.23. *Running* Update-Support-Graph *on $\mathcal{S}_{V^+}$ on line 10 of* Support-DPPC *ensures D is a correct weight support graph for $\mathcal{S}$. This run takes $O(n^+ w_d^2)$ time, following which all minimum nodes in D are unmarked.*

Proof. First of all, we note that running P^{3}C can never cause an affected edge e to support the weight of a non-affected edge e'. If e was the only support for e', then e' should have been affected as well. Since it is not, there must be some other path π' that supports the weight of e' but does not include any affected edge. The weight of π' does not change, but the weight of any path π containing e must increase, so e can never support e'.

Since the graph outside V^* was already PPC, the support relations there need not change. Moreover, the above implies that any new support relations within V^* must be incoming on an affected edge. These edges correspond exactly to the minimum nodes marked by Affected-Endpoints. By Lemma 5.7, running Update-Support-Graph therefore yields the correct support edges in D. Moreover, since we run it on an induced subgraph of n^+ nodes, Lemma 5.7 shows that we require $O(n^+ w_d^2)$ time.

Finally, note that line 9 clears the mark on all minimum nodes corresponding to an edge in $\mathcal{S}_{V^+}$. Since all affected edges are in $\mathcal{S}_{V^+}$, and only affected edges were marked, this line therefore clears all marks in the graph. □

5.6 Correctness and efficiency

At this point, the correctness of all components of the algorithm has been proven. We are now ready to demonstrate the correctness of the algorithm as a whole and provide bounds on the run time of each individual update. We conclude with remarks on possible future improvements of our algorithm.

Let n^* denote the number of vertices in V^*, n^+ the number of vertices in V^+ and recall that δ_c is the degree of the vertex with the most neighbours in the chordal graph. We then have the following:

Theorem 5.24. *After initialization by* Construct-Support-Graph *in $O(nw_d^2)$ time,* Support-DPPC *can correctly process any sequence of weight increases on edges in a PPC STN $\mathcal{S}$ without zero-weight cycles, each in time $O(n^*\delta_c + n^+ w_d^2)$ and $O(nw_d^2)$.*

Proof. By Theorem 5.8, running Construct-Support-Graph takes $O(nw_d^2)$ time and correctly constructs the weight support graph of $\mathcal{S}$. Moreover, by line 9 of Update-Support-Graph all minimum nodes in the graph are unmarked.

Now consider any update processed by Support-DPPC. When its execution is complete, $\mathcal{S}$ is PPC by Theorem 5.22, with the updated weight taken into account. By Lemma 5.23 the weight support graph is updated accordingly, and all minimum

nodes are again unmarked. Thus, both the network $\mathcal{S}$ and its weight support graph D are in the correct state required by the next update.

We now consider the time required by each such update. The initial checks and preliminary modifications on lines 1 through 6 of Support-DPPC can all be performed in constant time.

By Lemma 5.19, we can compute the set of affected endpoints in $O(t^*)$ time. Note that each affected triangle contains at least one affected edge and hence at least two vertices from V^*. The final vertex of any such triangle must be a common neighbour of the vertices in V^* and thus each affected triangle is fully contained in V^+. Then, since V^+ contains at most $O(n^+w_d^2)$ triangles, we have $t^* = O(n^+w_d^2)$.

To compute V^+ we visit all neighbours of every vertex in V^*, for a total time of $O(n^*\delta_c)$. Alternatively, Shared-Neighbours traverses each edge in the network at most twice: once for each endpoint. With this analysis, the upper bound is $O(m_c) \subseteq O(nw_d^2)$. Finally, running P^{3}C takes $O(n^+w_d^2)$ time, and by Lemma 5.23 so does running Update-Support-Graph.

Summarizing the above, we require time of order $O(1) + O(n^+w_d^2) + O(n^*\delta_c) + O(n^+w_d^2)$, which sums to $O(n^*\delta_c + n^+w_d^2)$, as claimed. We obtain the bound of $O(nw_d^2)$ by observing that $n^* \leq n^+ \leq n$ by the definition of V^* and V^+, and that we can replace $O(n^*\delta_c)$ by $O(nw_d^2)$ as discussed earlier. □

5.7 Possible further improvements

With the proof of Theorem 5.24, we formally established the correctness and efficiency of the Support-DPPC algorithm. However, there are still ways in which our algorithms can be further improved, which we present in this section. In Section 5.7.1 we discuss ways to extend the applicability of our algorithm to networks that include zero-cycles. Section 5.7.2 presents ways to reduce the execution time of the algorithm. We have not yet fully investigated these ideas, but we mention them here as directions for future research.

5.7.1 Handling zero-cycles

The only restriction on the applicability our algorithms is that the underlying network may not contain any zero-cycles, cycles of cumulative weight zero. However, we conjecture that we can work around this issue. In particular, we sketch two approaches that could be used to remove the zero cycles from the graph, without affecting the correctness of the computed minimum weights.

A first way would be to pre-process the graph, raising the weight of each constraint by a very small amount ε. After our algorithm completes we would then perform a post-processing step to remove all epsilons from the computed minimum weights.

The idea behind this method is that by increasing the weights, the cumulative weight along any cycle that was originally a zero-cycle will be raised to some multiple of ε, i.e. to a weight strictly greater than zero. Hence, the processed graph no longer contains any zero cycles, and by Lemma 5.6 its weight support graph is acyclic.

There is an issue with this approach however. In the original graph, the weight of a constraint may have been supported by multiple paths, consisting of different numbers of edges. In the pre-processed graph, paths with more edges accumulate more weight increases, and thus their weight is higher than that of paths with fewer edges. Hence, the processed graph does not capture all support relations that were present in the original graph.

This might cause us to incorrectly deduce that the weight for some constraint is no longer supported. However, note that we can never incorrectly assume that an edge is still supported, and need not change. Therefore the algorithm would still be correct, but may perform some unnecessary work.

A second option emerges when we consider what a zero-cycle represents in terms of our original problem. When a constraint between two events x_i and x_j has weight zero, this means that x_j can occur no more than zero time units after x_i. However, if there is a zero cycle, we also know that x_i can occur no more than zero time units after x_j. In other words x_j and x_i must *coincide*.

Our second idea then is to search for zero-cycles in the graph. If we find one, we know that all events on that cycle must coincide, and can thus replace them all by a *single* new event. We can repeat this method to eliminate all zero-cycles in the graph, yet retain a network equivalent to the input network. The pre-processed network can then be used in our algorithm.

Unfortunately, this method too has a catch. It may be that our algorithm is asked to increase the weight of a constraint that previously participated in a zero-cycle. At this point, we may need to expand the previously collapsed cycle, and adjust the network and support graph accordingly. It is not immediately clear how this might be accomplished.

5.7.2 Reducing the extended set

The size of V^+ determines the size of the subgraph on which we run P^{3}C and Update-Support-Graph. These algorithms are quite costly, so any reduction of this set could yield a significant speed-up in overall execution. In this section we briefly mention two possible improvements that would reduce the size of V^+.

First of all, while the current method to select V^+ is correct, we observe that it does not produce the *minimum* set required to correctly enforce PPC. Specifically, if the vertex $u \notin V^*$ is connected to two vertices $v_1, v_2 \in V^*$, but the weight of neither of the edges (v_1, v_2) or (v_2, v_1) needs to change, we can show that u need not be included in V^+.

Secondly, even if no such *vertices* exist, it may be possible to omit a specific set of *edges*. In particular, if the graph induced by V^+ contains an edge e between two vertices that are not in V^*, we know that the weight of e cannot change. Moreover, since neither endpoint of e is in V^*, we know that e is not even part of a *triangle* one of the edges of which must change. Nonetheless, in the current implementation e is still visited, which is essentially a waste of time. If we had a way to eliminate these edges, we could further reduce the run time of Support-DPPC, to be bound by the number of triangles containing an affected edge. The correctness of this second optimization is less certain however, since it is unclear whether the graph would still be chordal if we removed such edges.

Chapter 6

Experimental evaluation

In Chapter 4 we proved the correctness of our new Vertex-IPPC algorithm and derived a theoretic bound on its asymptotic run time. We also observed that it would integrate well with a vertex-incremental triangulation algorithm.

This chapter presents a limited experimental evaluation of our implementations of these two algorithms. In Section 6.1 we evaluate the run-time characteristics of Berry et al.'s triangulation method. Section 6.2 then discusses a similar experiment performed to demonstrate that our implementation of Vertex-IPPC meets the run time bound we claimed in Chapter 4. We also compare the performance of our algorithm to that of a simple extension of Edge-IPPC.

6.1 Incremental triangulation

This experiment aims to assess the run-time characteristics of our implementations of Berry et al.'s incremental triangulation algorithm in terms of the number of vertices in the input graph. As we mentioned in Section 4.6, the authors prove that their algorithm inserts a single edge in $O(n)$ time.

In Section 4.6.3 we presented two different implementations of this algorithm: one using breadth-first search and one using a dynamic lookup table. This in fact represents our implementation history: we first implemented the breadth-first search approach, and later switched to dynamic lookup tables, based on experimental results. In other words, we ran experiments to find evidence supporting the following two hypotheses:

Hypothesis 6.1. *Our implementation of the vertex-incremental triangulation method by Berry et al. using breadth-first search performs an edge insertion in $O(n)$ time.*

Hypothesis 6.2. *Our implementation of the vertex-incremental triangulation method by Berry et al. using a dynamic lookup table implementation performs an edge insertion in $O(n)$ time.*

Tasks	1000
Branch depth [min,max]	$[3, 6]$
Branches [min,max]	$[3, 10]$
Landmark ratio	0.20
Probability of SR-edge	0.50

Table 6.1: Parameters for generating HTN graphs.

6.1.1 Data set

Since we aim to assess asymptotic behaviour, we require graphs with large numbers of vertices. Moreover, since we concern ourselves with problems related to scheduling, we would like our input graphs to stem from this domain as well.

We therefore used a generator for so-called *Hierarchical Task Networks* (HTNs), a planning paradigm formalised by Erol et al. (1994). The generator we used was created by Planken et al. (2010) for the empirical evaluation of the Edge-IPPC algorithm. We briefly summarize their discussion of HTNs here, for more details we refer to the original papers.

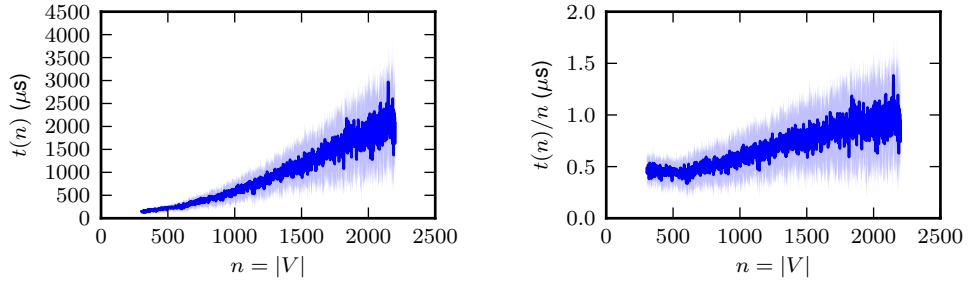
In short, an HTN represents a hierarchy of tasks, in which high-level tasks are progressively decomposed into collections of smaller tasks as we go further down the hierarchy. Constraints in an HTN occur only between a parent task and its children, or between sibling tasks. The latter constraints are so-called sibling-restricted (SR) edges. Given this restriction, it is not possible to coordinate the execution of tasks in the different branches of the hierarchy when we strictly adhere to the HTN format. In order to circumvent this, the definition of an HTN may be extended to include so-called *landmark variables*, which do allow synchronisation between branches. Finally, each task is modelled as a pair of events in the network: it start and its end.

In the interest of reproducibility, the settings we used to generate our HTNs are listed in Table 6.1. Using these settings, each generated graph contained 2200 vertices and between 9500 and 9800 edges. Our benchmark set consisted of 80 such graphs.

6.1.2 Method

In order to assess the run time as function of the number of vertices, we then proceeded as follows. First we generated a random permutation of the vertices in the graph by shuffling a sequence of all vertices. We then created a new, empty graph, on which the experiment was to be performed. During the actual experiment we used the triangulation algorithm to insert the vertices from the generated graph into the experiment graph one by one, as enumerated in the shuffled sequence. We timed the performance of every *edge* insertion, keeping track of the number of nodes at the time of insertion.

The experiments were run on an Intel Core i7 Q740 with four cores clocked at 1.73 GHz and 4 GB RAM. The Java VM (build 1.6.0_24-b07) was given 1.5 GB of



(a) Time needed to insert one edge and its required fill edges at given graph sizes.

(b) The data from Figure 6.1a, normalized by graph size.

Figure 6.1: Run time for Berry et al.'s triangulation algorithm, using breadth-first search to find cliques. The dark blue lines plot the median values over 400 runs, the light bands show the range between the 25$^{\text{th}}$ and 75$^{\text{th}}$ percentile.

memory. Since it takes some time before the Just-In-Time optimizer in the Java VM starts up, we ignored the results for the first 300 vertex insertions and focused only on the asymptotic behaviour.

6.1.3 Results and discussion

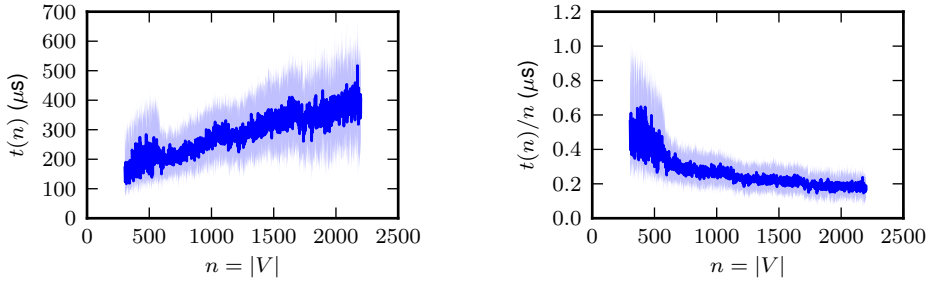
Breadth first search Figure 6.1 shows the results for running the algorithm using the implementation based on breadth-first search. Plotting the size of the graph against the time required to insert an edge and its accompanying fill-edges, we observe a slight upwards curve in Figure 6.1a, suggesting supra-linear run times.

This suspicion is confirmed by Figure 6.1b, where we normalize the time by the size of the graph at the time the vertex was inserted. For an algorithm with performance linear in the number of vertices, we expect a horizontal line here, averaging the constant for our implementation. We observe a clear upwards trend however, indicating that this implementation does not realize the $O(n)$ bound we aimed for. Hence we reject Hypothesis 6.1.

Dynamic lookup table To improve on this negative result, we replaced the implementation using breadth-first search with a version using a dynamically maintained lookup table. Keeping all other factors constant, we performed the exact same experiment, which yielded the results shown in Figure 6.2.

Our first observation is that Figure 6.2a does not show the curvature observed when using breadth-first search. Reviewing the normalized graph in Figure 6.2b shows that in this case we do achieve a horizontal line. In fact, if anything the normalized graph appears to show a decreasing trend. Hence, this experiment provides evidence that Hypothesis 6.2 holds, and we do not reject it.

A second observation can be made by comparing the scale on the y-axes of Figures 6.1a and 6.2a. As we see, the implementation using a lookup table requires far



(a) Time needed to insert one edge and its required fill edges at given graph sizes.

(b) The data from Figure 6.2a, normalized by graph size.

Figure 6.2: Run time for Berry et al.'s triangulation algorithm, using a lookup table to find cliques. The dark blue lines plot the median values over 400 runs, the light bands show the range between the 25$^{\text{th}}$ and 75$^{\text{th}}$ percentile.

less time than the breadth-first search version, especially when the size of the graph increases. This is of course to be expected, as we have replaced a supra-linear search operation by a constant time lookup operation. Nonetheless, we point out that the performance gains are impressive.

6.2 Vertex-IPPC

Having evaluated the expected bounds of our implementation of the vertex-incremental triangulation algorithm, we now turn to assess our own contribution, the Vertex-IPPC algorithm.

In particular, we want to answer two questions. One, does our implementation meet the run time bound of $O(m_c + \delta_c(a)w_d^2)$ derived in Chapter 4, and two, how does its performance compare to Edge-IPPC?

Regarding the second question, comparing a vertex-incremental algorithm to an edge-incremental one may seem strange. However, we observe that Edge-IPPC can be used to enforce PPC after a vertex has been inserted, using the following simple approach. After the graph has been pre-processed such that it contains a and will still be chordal after all new constraints are added, we simply run Edge-IPPC once for every constraint attached to a. Since there are exactly $\delta(a)$ such constraints and each run of Edge-IPPC takes $O(m_c)$ time, this simple algorithm re-enforces PPC in $O(\delta(a) \cdot m_c)$ time.

Hypothesis 6.3 and Hypothesis 6.4 present the expected answers to respectively the first and second question posed above.

Hypothesis 6.3. *The run time of our implementation of Vertex-IPPC meets the theoretical bound of $O(m_c + \delta_c(a)w_d^2)$.*

Hypothesis 6.4. *When inserting a new vertex a in a graph, running Vertex-IPPC is faster than running Edge-IPPC $\delta(a)$ times.*

6.2.1 Data sets

To assess the runtime characteristics of the Vertex-IPPC algorithm, we generated HTN graphs with the same values for the parameters used to generate the graphs for our experiments with incremental triangulation. However, since these graphs were intended for more computationally intensive experiments, the number of tasks was restricted to 100. This resulted in 80 graphs of 220 vertices each.

To assign the weights, we used a utility provided with the HTN generator by Planken et al. (2010). We configured this utility to first ensure a solution exists, by assigning a random time between -50 and 100 to each event. Then, the weight for each constraint in the network was calculated as follows. First, the utility determined the minimum possible weight such that the random solution was still possible. It then increased this weight by a value selected randomly from the range $[0, 150]$. Hence the original solution still existed, but had been "obscured" by the randomly added slack.

In order to test the influence of the graph *structure* on performance, we also generated a set of chordal graphs. This set was also created with the HTN generator and the weight setting utility. However, for this set we first added the fill edges as inserted by the minimum degree heuristic, before assigning weights.

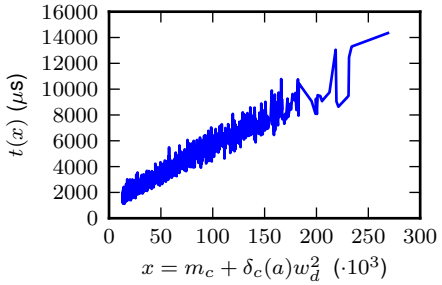
6.2.2 Method

In order to assess performance, we used the following procedure. For each experiment we incrementally constructed three graphs: two graphs used to benchmark Vertex-IPPC and Edge-IPPC, and one control graph on which we ran the P^3C algorithm to verify that our implementation was correct.

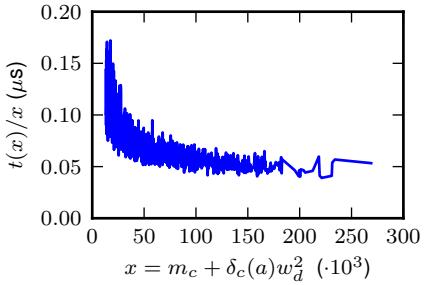
All three graphs were initially empty. During the experiment we inserted the vertices from the input graph into the three experiment graphs one by one, as well as all constraints connecting the new vertex to the other vertices already in the graph. The order in which vertices were inserted was chosen randomly.

When a vertex and its constraints were inserted, we first used Berry's vertex-incremental triangulation scheme to ensure the resulting graph was chordal. We then re-enforced partial path consistency by three different methods. On the first graph we then ran Vertex-IPPC, starting from the newly added vertex. In the second graph, we ran Edge-IPPC once for each constraint attached to the new vertex a, for a total of $\delta(a)$ runs. Finally, we ran P^3C on the third graph.

At the end of each step, we made sure the weights in the graphs maintained by Vertex- and Edge-IPPC were equivalent to the weights enforced by P^3C, thus verifying that the implementation was correct.

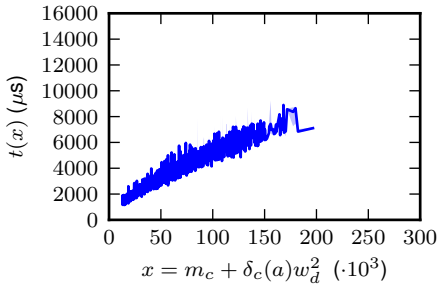


(a) Run time as function of input complexity.

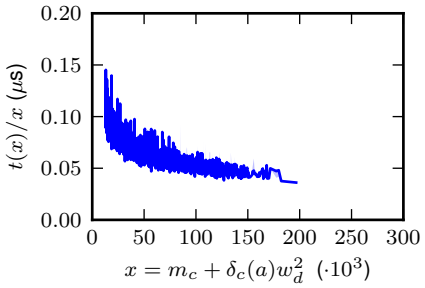


(b) Run time normalized by input complexity.

Figure 6.3: Run time for Vertex-IPPC on HTN graphs consisting of 220 vertices.



(a) Run time as function of input complexity.



(b) Run time normalized by input complexity.

Figure 6.4: Run time for Vertex-IPPC on pre-triangulated HTN graphs consisting of 220 vertices.

6.2.3 Results and discussion

Verification of theoretical bounds

The aim of our first experiment was to verify our implementation against the theoretical upper bound of $O(m_c + \delta_c(a)w_d^2)$ derived in Chapter 4. We therefore plot the run time required to insert a vertex as function of the complexity of the graph at the time of insertion. By taking $x = m_c + \delta_c(a)w_d^2$ as measure of complexity, we should see a straight line if our implementation meets this bound on the given input graphs.

Figure 6.3a shows this plot for the normal HTN graphs, Figure 6.4a for HTN graphs that were previously triangulated. We observe that both figures show a line with a clear linear trend.

In order to more closely assess whether we meet the theoretical bound, we include normalized plots for both normal and pre-triangulated HTN graphs in Figure 6.3b and Figure 6.4b respectively. If our implementation satisfies the theoretical bounds, these plots should show a constant horizontal line or a decreasing trend. This is indeed the behaviour we observe. In other words, our experiments support Hypothesis 6.3 and we conclude that we cannot reject it.

Comparison to Edge-IPPC

In our second experiment we compare the performance of using Vertex-IPPC to insert a vertex at once, to the performance of running Edge-IPPC once for every constraint adjacent to the new vertex. We initially ran this experiment on the set of normal HTN graphs and obtained the results summarized in Figure 6.5a.

Rather surprisingly, we find that Edge-IPPC is significantly faster than Vertex-IPPC in this benchmark. Hence, Hypothesis 6.4 does not hold in general, and we reject it.

A possible explanation for this somewhat disappointing result is that the degrees of the new vertices in the chordal graph may have been much higher than their degrees in the original graph. Recall that Edge-IPPC needs to be executed only once for each constraint in the *original* graph, and completes in time linear in the number of edges. On the other hand, the run time of the DPC algorithm used by Vertex-IPPC depends on the size of the neighbourhood in the *triangulated* graph. Running DPC is relatively expensive, so if there are only a few real constraints but many fill edges, this investment may not pay off.

In particular, we may have the following pathological case. Suppose the new vertex a has only two constraints involving older vertices in the original graph. However, these constraints create a cycle involving all n vertices, and thus after triangulation, a is connected to *all* vertices in the chordal graph. In this case we need only run Edge-IPPC twice, which can be done in $O(m_c)$ time, while the DPC step alone takes $O(nw_d^2)$ time. Conversely however, if nearly all edges in the chordal graph correspond to actual constraints, the Edge-IPPC method will take $O(nm_c)$ time, which is worse than the $O(nw_d^2)$ that dominates the run time of Vertex-IPPC in this case.

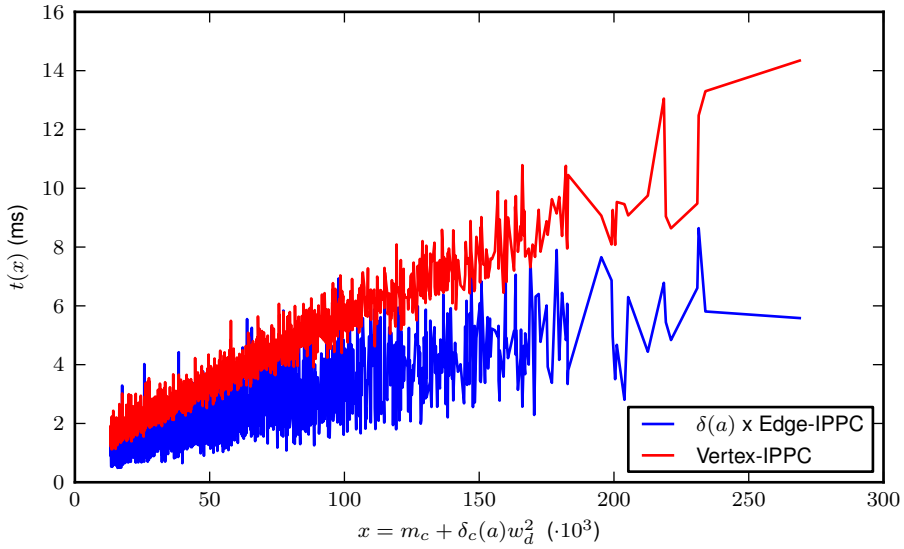
We summarise the above in the following new hypothesis, a more specific form of Hypothesis 6.4:

Hypothesis 6.5. *Given that $\delta(a) = \delta_c(a)$ when a new vertex a is inserted in a graph, running Vertex-IPPC is faster than running Edge-IPPC $\delta(a)$ times.*

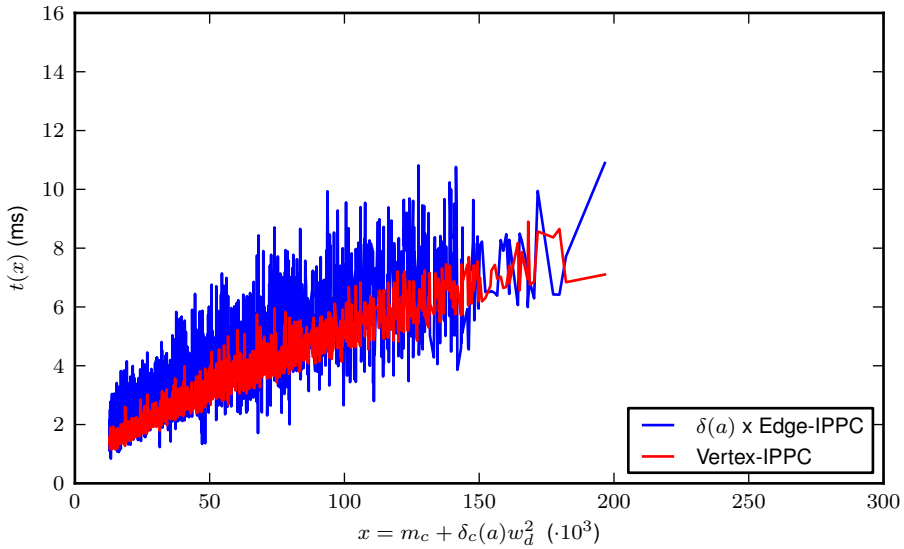
To test this hypothesis, we ran our experiment on the input set consisting of pre-triangulated HTNs. Since these graphs are already triangulated, we do not need to add *any* fill edges, which should give Vertex-IPPC an advantage over Edge-IPPC. The results for this experiment are shown in Figure 6.5b.

This figure does indeed show a better relative performance for Vertex-IPPC compared to Edge-IPPC. However, the difference is only small, and for greater input complexities Edge-IPPC seems to be gaining the upper hand again. Since Vertex-IPPC does not clearly perform better than Edge-IPPC, we have to reject Hypothesis 6.5 as well.

A final possibility to explain these results is the following. It may be that the consecutive runs of Edge-IPPC in the neighbourhood of the new vertex "help" each other. More specifically, it may be that two constraints c_1 and c_2 imply a lower weight for some constraint c_3 than c_3 is itself labelled with in the source graph. If Edge-IPPC



(a) Run time on normal HTN graphs.



(b) Run time on pre-triangulated HTN graphs.

Figure 6.5: Run time comparison between Vertex-IPPC and Edge-IPPC on HTN graphs.

enforces c_1 and c_2 first, it can then immediately detect that the weight of c_3 need not change, thus reducing the execution cost. If this happens frequently, then Edge-IPPC essentially gets a few runs "for free". Vertex-IPPC on the other hand always needs to pay the cost of running DPC, whether constraints subsume each other or not.

While this hypothesis does not offer a way to improve the performance of Vertex-IPPC, it can be used to improve that of *Edge*-IPPC. In particular, it would be interesting to see whether there is some way to put the constraints inserted by Edge-IPPC in such an order that the "help" is maximized. We will not pursue this idea here, but note it as an interesting direction for future work.

Chapter 7

Conclusions and future work

In this final chapter, we first summarize our contributions and draw conclusions. We then present some interesting directions for future work.

7.1 Summary and conclusions

In Chapter 2 we introduced the Simple Temporal Problem (STP) and reviewed the different solution types for this problem. We also paid attention to the concept of chordal graphs, the properties of which are key to the efficiency of the current state of the art in algorithms solving the STP.

Such algorithms were discussed in more detail in Chapter 3, which reviewed existing methods to compute the different solution types introduced in the preceding chapter. We showed that these methods can be classified into two groups: methods enforcing Full Path Consistency operate on complete graphs, while methods enforcing Partial Path Consistency operate on chordal graphs. We paid particular attention to the P^3C algorithm, which computes a solution for a STP instance by enforcing PPC and constitutes the current state of the art in this area.

In Chapter 4 we presented our first contribution: the *Vertex-IPPC* algorithm. This algorithm allows us to efficiently re-enforce partial path consistency when we update an existing STP instance by adding a new event and all its adjacent constraints. We demonstrated that Vertex-IPPC integrates well with a recently discovered method for vertex-incremental triangulation, designed by Berry et al.. Moreover, to the best of our knowledge – having corresponded with the original authors – we created the first actual implementation of this triangulation method. In doing so we contributed the implementation of one step not made explicit in the original paper.

Our most important contribution is made in Chapter 5, where we present the *Support-DPPC* algorithm. To the best of our knowledge this is the first algorithm to efficiently process *decremental* updates for a partially path consistent STP instance. To do so we introduced the notion of the weight support graph, which tracks the origins for the tightest constraints in the graph. We showed that our algorithm efficiently exploits and maintains this graph in parallel with the original network. Moreover, we

concluded that even in the worst case the asymptotic bound on the run time for our algorithm is no worse than that of P^3C, and we proved a second bound showing that our algorithm may in fact outperform P^3C practice.

Finally, in Chapter 6 we presented a limited empirical evaluation of our implementations of the vertex-incremental triangulation method by Berry et al. and our own Vertex-IPPC algorithm. We conclude that both our implementations meet their respective theoretical bounds. Unfortunately, a second set of experiments comparing Vertex-IPPC to a straightforward extension of the existing Edge-IPPC algorithm showed that Vertex-IPPC did not yield the improved execution time we expected. Due to limited time, we have not yet implemented and benchmarked Support-DPPC.

7.2 Future work

In this section we give an overview of topics for future research, some of which have already been mentioned in the preceding chapters. The topics are grouped by the nature of the algorithms they relate to.

7.2.1 Vertex-incremental approaches

It would be interesting to further explore the properties of the naive algorithm for vertex-incremental partial path consistency based on Edge-IPPC we benchmarked in Chapter 6. As discussed in Section 6.2.3, a first avenue for improvement is the ordering in which the weights of the new constraints are enforced. A heuristic that would maximize the number of "free" constraints could significantly reduce the run time of the algorithm, particularly in cases where the new vertex has many adjacent constraints.

Furthermore, recall from Section 3.2.2 that Edge-IPPC itself was outperformed by IFPC, the Incremental Full Path Consistency algorithm. Therefore it would be interesting to see if we can perhaps exploit the properties of a complete graph to create a vertex-incremental version of IFPC, which well may outperform the variant using Edge-IPPC. Note that this approach may also benefit from a heuristic as described in the previous paragraph.

7.2.2 Decremental PPC

Since the Support-DPPC is the first decremental algorithm for enforcing partial path consistency, there are many interesting directions for further research.

An obvious first step is to implement Support-DPPC and benchmark its performance on practical problems. Interesting candidates to compare its performance against would be the P^3C algorithm itself, but also the method for fully dynamic *full* path consistency on graphs with non-negative weights discovered by Demetrescu and Italiano (2004).

Furthermore, we would like to eliminate our dependency on the fact that the underlying network may not contain cycles of cumulative length zero. It would therefore be interesting to investigate the pre-processing approaches suggested in Section 5.7.1, and determine whether they can be used to eliminate all zero-cycles from the network.

Finally, in Section 5.7.2 we observed that the current way the extended set V^+ is created results in the inclusion of vertices and edges that are not strictly necessary to re-enforce PPC. Further research may yield more advanced methods, which should reduce the size of V^+. Since the the size of V^+ determines the size of the subgraph on which we perform expensive computations, any reduction of this graph could yield a significant speed-up in overall execution.

Another idea only tangentially related to decremental PPC is to use the notion of support graphs for decremental *triangulation*. In the context of triangulation, we also find the notion of "support": there is a sense that a "hard" edge in the original graph "supports" a fill edge required to break some cycle in which the hard edge participates. If a fill edge is not supported by any hard edge, it can be removed. If we find a way to efficiently capture these relations in a support graph or dependency graph, we should therefore be able to immediately remove fill edges if they are no longer needed.

To the best of our knowledge, no *truly* decremental triangulation algorithms exist yet. There is an algorithm by Ibarra (2008), but this algorithm explicitly *fails* if an edge cannot be removed without breaking chordality. An approach based on support graphs could implicitly *defer* the deletion of an edge in this case, until the last hard edge requiring its existence is removed.

7.2.3 Dynamic PPC

While we have not achieved a fully dynamic algorithm for enforcing partial path consistency, we have come quite close. In particular, we believe it should be possible to combine Support-DPPC with an incremental algorithm in order to obtain a fully dynamic algorithm for enforcing partial path consistency. The most important aspect here is that we maintain the weight support graph correctly. An interesting observation in this regard is that when the weight of an edge is lowered, we know we can remove any pre-existing incoming support. After all, such pre-existing support relations represented support for the previous, higher weight, which has now changed.

A simple first approach might be to modify Edge-IPPC to update the support graph in this way. If we then determine the subgraph in which the updates took place, we can run the Update-Support-Graph algorithm to correctly re-establish the weight support graph.

Finally, provided the approach outlined above works, it would be interesting to embed the resulting algorithm in a DTP solver. As we stated in the introduction, current DTP solvers have to recompute the solution for an STP from scratch when

they back-track. If Support-DPPC is used instead, performing these backtracking steps may take less time, thus improving performance.

As the title of this thesis indicates, we have worked towards a fully dynamic algorithm for the Simple Temporal Problem. Especially in the light of the approach sketched in this last section, we expect that such an algorithm is indeed feasible, and it will be interesting to see how it performs in practice.

Bibliography

A. Berry, P. Heggernes, and Y. Villanger. A vertex incremental approach for maintaining chordality. *Discrete Mathematics*, 306(3):318–336, 2006.

C. Bliek and D. Sam-Haroud. Path consistency on triangulated constraint graphs. In *International Joint Conference on Artificial Intelligence*, volume 16, pages 456–461, 1999.

P. Buneman. A characterization of rigid circuit graphs. *Discrete Math*, 9(205-212): 18, 1974.

R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.

C. Demetrescu and G. Italiano. A new approach to dynamic all pairs shortest paths. *Journal of the ACM (JACM)*, 51(6):968–992, 2004. ISSN 0004-5411.

K. Erol, J. Hendler, and D. Nau. UMCP: A sound and complete procedure for hierarchical task-network planning. In *Proc. 2nd Intl. Conf. on AI Planning Systems (ICAPS-94)*, pages 249–254, 1994.

R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5:345, June 1962.

D. Fulkerson. Incidence matrices and interval graphs. Technical report, DTIC Document, 1964.

F. Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, 16(1):47–56, 1974.

P. Heggernes. Minimal triangulations of graphs: A survey. *Discrete Mathematics*, 306(3):297–317, 2006.

L. Ibarra. Fully dynamic algorithms for chordal graphs and split graphs. *ACM Trans. Algorithms*, 4:40:1–40:20, August 2008.

D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24:1–13, January 1977. ISSN 0004-5411.

R. Mohr and T. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28(2):225–233, 1986.

S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312(1):47–74, 2003.

L. R. Planken. Incrementally solving the stp by enforcing partial path consistency. In *Proceedings of the 27th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2008)*, pages 87–94, 2008.

L. R. Planken, M. M. de Weerdt, and R. P. J. van der Krogt. P3C: A New Algorithm for the Simple Temporal Problem. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 256–263, Menlo Park, CA, USA, September 2008.

L. R. Planken, M. M. de Weerdt, and N. Yorke-Smith. Incrementally Solving STNs by Enforcing Partial Path Consistency. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS-10)*, pages 129–136, Menlo Park, CA, USA, May 2010.

L. R. Planken, M. M. de Weerdt, and R. P. J. van der Krogt. Computing all-pairs shortest paths by leveraging low treewidth. In *Proceedings of the Twenty-first International Conference on Automated Planning and Scheduling (ICAPS-11)*, May 2011a.

L. R. Planken, M. M. de Weerdt, and R. P. J. van der Krogt. Solving the simple temporal problem. Working Paper, 2011b.

D. Rose and R. Tarjan. Algorithmic aspects of vertex elimination. In *Proceedings of seventh annual ACM symposium on Theory of computing*, pages 245–254. ACM, 1975.

K. Stergiou and M. Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. *Artificial Intelligence*, 120(1):81–117, 2000.

I. Tsamardinos and M. Pollack. Efficient solution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence*, 151(1-2):43–89, 2003.

J. Walter. *Representations of rigid cycle graphs*. PhD thesis, Wayne State University, 1972.

S. Warshall. A theorem on boolean matrices. *J. ACM*, 9:11–12, January 1962.

L. Xu and B. Y. Choueiry. A new efficient algorithm for solving the simple temporal problem. In *Proceedings of the 10th International Symposium on Temporal Representation and Reasoning and Fourth International Conference on Temporal Logic*, 2003.

M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2(1):77–79, 1981.

Appendix A

Pseudo-code incremental triangulation

As discussed in Section 4.6, while Berry et al. (2006) prove that their vertex-incremental triangulation algorithm is correct, they did not create an implementation. Moreover, their paper does not contain explicit pseudocode. Instead, the algorithm is described in text form, mixed with proofs of correctness.

When building our implementation of the algorithm, we had some initial difficulties to maintain a high-level overview of how it worked. This became much easier when we extracted our own version of the pseudocode from the text, which we include in Algorithm A.1.

Our main purpose for including it here is that it may serve as a high-level guide for any other implementers of the algorithm. In that spirit, we would like to point out that Section 4.6 discusses an important implementation detail on the process of finding the cliques K_u and K_v mentioned in line 1.

Algorithm A.1: Add edge (u, v) to G and update the clique forest F

```
1  find any cliques K_u, K_v ∈ F such that u ∈ K_u and v ∈ K_v
2  find a path P_uv between K_u and K_v in the clique forest
3  if such a path exists then                    K_u, K_v are nodes of the same tree
4      trim P_uv so only the first and last clique contain u and v respectively
5      reduce P_uv so each edge is a minimal u, v separator
6      extract minimal separators S from P_uv
7      call update-tree(u, P_uv)
8      insert edges (u, s) for all s ∈ S into the underlying graph
9  else
10     create a new clique K_uv containing u and v
11     connect K_u and K_v to K_uv in F
12 insert (u, v) in the underlying graph
```

Procedure update-tree(u, P_{uv})

```
1  remove u from remove-list of the first edge of P_uv
2  foreach clique C_i ∈ P_uv do
3      update incoming add-lists of C_i to include u
4      if C_i now contains both u and a non-neighbour x of u then
5          split C_i into two cliques, one containing x, one containing u
6  if head of P_uv ⊆ second clique in P_uv then
7      remove head of P_uv
```