Utrecht University
Algorithmic Design and Complexity

Master Thesis

Improved Algorithms for the Computation of Special Junction Trees

Author:
M.E.R. van Boxel

Supervisors:
dr. H.L. Bodlaender

July 13, 2014

Abstract

Several intractable (e.g. $\mathcal{NP}$-complete) graph problems can be solved efficiently with help of junction trees without large bags. Graph parameters that measure the suitability of a junction tree include the well known notion of Treewidth, and the related notions of Pathwidth and Treecost. In this thesis, exact algorithms to compute the Pathwidth and Treecost of a given graph are studied. We improve upon the current state algorithms using techniques formerly used for exact algorithms for Treewidth. For the techniques, we discuss whether these are suitable for the computation of Pathwidth and/or Treecost. We give both a theoretical analysis of the running time of our algorithms, and present experimental results. Amongst others, these show that one of our algorithms is sufficiently efficient to compute the Pathwidth exactly of graphs with at most 45 vertices. For TREECOST we give an algorithm for finding triangulations of optimal cost that uses $\mathcal{O}\left(2^{|V|-\omega(G)}\right)$ time, with $\omega(G)$ the size of the maximum clique in G, assuming this maximum clique is known.

Contents

1 Introduction

In the year 2000 Clay Mathematics Institute of Cambridge introduced seven mathematical problems to the world [12]. The problems where called *The Millennium Prize Problems*. What made these seven problems so special? First of all, any one who can solve one of these problem would receive one million dollars. To solve a problem you either need to prove the correctness of the statement, or give a counter example. Further the problems had experts guessing on the outcome already for many years. Up until the creation of this report, there are still six problems unsolved. One of these infamous problems is the $\mathcal{P}$ versus $\mathcal{NP}$ problem, first introduce by Cook [14]. Since 1971 experts are trying to prove if it is possible to solve $\mathcal{NP}$-complete problems in polynomial time. Until now the problem has yet to be solved.

Solving the $\mathcal{NP}$-complete problems tend to be difficult. For several $\mathcal{NP}$-complete problems approaches have complexities of order $\mathcal{O}(2^n)$, where n is equal to the input size. For many problems there exist algorithms that reduce the exponential complexity to $\mathcal{O}(c^n)$, where $c \leq 2$ and n equal to the input size, see e.g. [23, 31, 4, 20]. So some of the $\mathcal{NP}$-complete problems can be solved in $\mathcal{O}(c^n)$ time but are still exponential.

A different approach for solving some of the $\mathcal{NP}$-complete problems is Fixed-Parameter Tractability [18], FPT. Algorithms that are classified as FPT have a complexity of $\mathcal{O}\left(f(k) * n^{\mathcal{O}(1)}\right)$, where n is equal to the input size and $f(k)$ is a function of some property that depends on the parameter k. An example of such a solution is given by Chen et. al. [13]. They found an algorithm which uses a complexity $\mathcal{O}\left(1.2738^k + k * n\right)$ for solving the Vertex Cover problem, where k equals the decision variable. Solving the problem with the FPT results in much quicker solutions then the naive approach. For different $\mathcal{NP}$-complete problems there are theoretical FPT algorithms which are not practical because they have large constant factors.

For some graph $\mathcal{NP}$-complete problems there exist FPT algorithms that use the parameter k equal to the Treewidth or Pathwidth, see e.g. [21, 1, 2, 3, 5, 15, 16]. Therefore having a low Treewidth or Pathwidth on an arbitrary graph G will reduce the computation time of many other $\mathcal{NP}$-complete problems.

The first step of an FPT algorithm for a problem on graphs with small Treewidth or Pathwidth is to determine the treewidth or pathwidth of the input graph and find a corresponding Junction Tree or Pathdecomposition. However computing these problems are proven to be $\mathcal{NP}$-complete. The first definition of Treewidth and Pathwidth are from Robertson and Seymour [29, 28]. Since then many experts have tried to solve these two problems efficient. One of the first algorithms is from Robertson and Seymour [30]. After that many followed, an overview has been created by Bodlaender [6, 7].

In this report we first look at exact algorithms for Pathwidth. A theoretical investigation of this problem was made by Fomin and Villangers [22]. The currently best known theoretical bound is an algorithm with complexity $\mathcal{O}^*(1.7346^n)$. We will however construct our own algorithm based on Bodlaender and Fomin [9]. We show that the Pathwidth problem can be solved efficiently in $\mathcal{O}^*(2^n)$ time. We carried out experiments that show that the algorithm to solves the problem efficiently for graphs of size up to 45. For larger graphs the bottleneck appears to be the memory, see Section 3.

In the second part of this report we will look at Treecost. This is a different parameter for FPT algorithms which tend to give a better indication of the running time of certain FPT algorithms. The computation of Treecost of a graph is also NP-hard [32]. Already in

1990, Wen [32] and Kjaerulff [24, 25] gave heuristics based on simulated annealing to compute the TREECOST. Several other, simpler and faster heuristics for TRIANGULATION (see [17]) are also often used, e.g., iteratively eliminating the vertex with minimum degree. For an overview, see [19]. We look at exact algorithms for the TREECOST problem. Bodlaender and Fomin [8] made a theoretical investigation of the problem. Ottosen and Vomlel [27] gave an exact algorithm for TREECOST with several improvements, including an intricate method to dynamically maintain the maximal cliques. This was improved by Li and Ueno [26], with a depth-first-search branch and bound algorithm with a running time of $\mathcal{O}^*((n-1)!)$ for networks with n vertices.

We will show that this running time can be reduced to $\mathcal{O}^*\left(2^{n-\omega(G)}\right)$, with $\omega(G)$ the size of the maximum clique in G, assuming this clique is already known. Tests show that our algorithm is able to solve the problem efficiently for graphs of size up to 30. For larger graphs the bottleneck appears to be both memory and time, see Section 4.

This report is organized as follows. First in Section 2 we will give the terminology for the rest of the report. In Section 3 and 4 the PATHWIDTH and TREECOST problem are researched and methods will be provided for solving the problems respectively. After, in Section 5 we will give conclusion of the methods we used and provide further research possibilities. At the end in Section B we provide the reader with the pseudo code of the implemented algorithms.

2 Preliminaries

We assume that the reader is known with the basic notations of graph theory. Throughout this report the following notations will be used.

For a graph $G(V, E)$ we denote V also known as $V(G)$ to be the vertices and E also written as $E(G)$ to be the edges. Further for a tree $T(V, E)$ we denote vertices $V(T)$ and edges $E(T)$. The tree $T(V, E)$ contains no cycles. A path $P(V, E)$ is denoted by vertices $V(P)$ and edges $E(P)$. The path $P(V, E)$ contains no cycles and is connected. For each v in $V(P)$ we have that the branching factor is less or equal to 2. Of all these graphs we have $|V| = n$ and $m = |E|$. All edges are undirected and denoted by $\{e(u, v)|u, v \in E, u! = v\}$.

An ordering Π is denoted by $\Pi = \{x_1, x_2, .., x_n\}$ such that x_i is a vertex of $V(G)$ and for all i, j with $i \neq j$ we have $x_i \neq x_j$. From this ordering we can obtain a sub ordering. Let Π_i be a sub ordering of Π, such that $\Pi_i = \{x_1, x_2, .., x_i\}$.

A clique is a set S such that $S \subseteq V$ and for each pair of vertices x_i and x_j in S there exists an edge connecting both vertices x_i and x_j. All maximal cliques from a graph $G(V, E)$ are obtained by all cliques from a graph that are not a subset of any other clique. The maximum clique is the largest clique of all maximal cliques, i.e., the set contains the most vertices.

3 Pathwidth

In this section we will discuss the *pathwidth* problem. First we will give a number of definitions, followed by the problem description. Next we describe the different methods we will apply to tackle the problem. We will do this by giving motivations why they are useful and how they will reduce the running time. After this we introduce different variants of algorithms for solving the *minimum pathwidth* problem. Finally we present experimental results , give our conclusions and future work possibilities.

3.1 Pathwidth

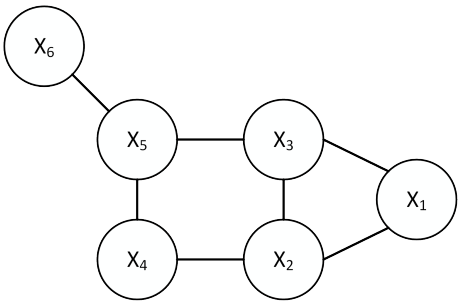

Figure 1: Example graph $G(V, E)$, with vertices $V(G) = \{x_1, x_2, x_3, x_4, x_5, x_6\}$ and edges $E(G) = \{(x_1, x_2), (x_1, x_3), (x_2, x_3), (x_2, x_4), (x_3, x_5), (x_4, x_5), (x_5, x_6)\}$

To understand pathwidth we first need to know the definition of *treewidth*. The pathwidth problem is closely related to the treewidth problem, due to that a solution of the pathwidth solution is a special case of a solution of the treewidth problem. While each pathwidth solution is a valid solution for treewidth, we should note that it is not guaranteed to be a minimum solution. First we will give the definition of treewidth. A solution to the treewidth problem is a *Junction Tree* where the junction tree has the conditions from Definition 3.1. For the examples in this section we will use the graph from Figure 1.

Definition 3.1 *A* JUNCTION TREE *of a graph $G(V, E)$ is a tree $T(V, E)$, with nodes $V = \{X_1, ..., X_n\}$, with for each i, X_i is a set of vertices from $V(G)$ such that the following conditions are satisfied,*

- $\bigcup_{X_i \in X} X_i = V(G)$
- $\forall_{v \in V(G)} Z_v$ *form a connected subtree of T, where* $Z_v := \{X_i | v \in X_i, i = 1..n\}$
- $\forall_{e(v,w) \in E(G)} \Rightarrow \exists X_i$ *such that* $v, w \in X_i$

Junction trees are also called *tree decomposition*. A junction tree is a tree $T(V, E)$ constructed from a graph $G(V, E)$ with the following properties. First of all the tree T cannot have any cycles. Second each node X_i in $V(T)$ is a collection of vertices from $V(G)$. Further we have that each vertex x_i in $V(G)$ is at least contained in one node X_j from $V(T)$. We also represent the edges from $E(G)$ in tree T. This is done such that for each edge $e(x_i, x_j)$ in $E(G)$ we have, there is at least one node X_k that contains both x_i and x_j, where X_k is from $G(T)$. Last we have that for each x_i in $V(G)$, we must satisfy that all nodes X_j in $V(T)$ that contain vertex x_i are connected.

Two examples of a junction tree of the graph of Figure 1 can be seen in Figure 2. We can verify that both examples satisfies the three conditions from Definition 3.1. Therefor multiple junction trees exist for each graph. In fact the number of unique junction trees is infinite for each graph $G(V, E)$, where $V(G)$ is not the empty set.

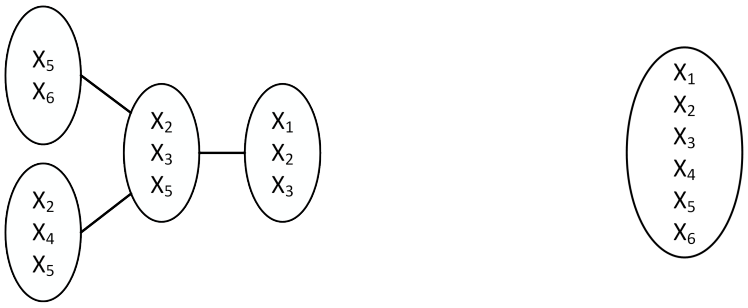

Figure 2: Two *junction trees* constructed by the graph from Figure 1

When we have constructed a junction tree we can easily find the corresponding treewidth. The TREEWIDTH is defined by the size of the largest node in the junction tree $T(V, E)$ minus one. Definition 3.2 represents the formula for finding the treewidth of a junction tree $T(V, E)$. If we apply the treewidth formula on the two junction trees in Figure 2, we obtain values 2 and 5 respectively.

Definition 3.2 *The* TREEWIDTH *of a junction tree* $T(V, E)$ *is the integer defined by*

$$TW(G) = \min_{\forall X \in T(V)} (|X| - 1)$$

Instead of finding the treewidth of a junction tree we would like to find the treewidth of a graph. The minimum treewidth of a graph $G(V, E)$ is defined as the minimum treewidth over all possible junction trees, see Definition 3.3. For our example from Figure 1 we have that the minimum treewidth equals two and a corresponding junction tree is the left tree in Figure 2.

Definition 3.3 *The* MINIMUM TREEWIDTH *of a graph* $G(V, E)$ *is the integer defined by*

$$TW(G) = \min_{junction\ trees\ T(V,E)\ from\ G(V,E)} treewidth(T(V, E))$$

As mentioned before, the pathwidth is a variant of treewidth. The difference between these values are obtained by not using a junction tree but a path decomposition. The definition of path decomposition compared to the definition of junction tree has one additional condition, see Definition 3.4. Instead of constructing a tree we construct a path. Therefor the branching factor must be at most two. It is still not allowed to create a cycle, so a path decomposition a beginning node and an ending node.

Definition 3.4 *A* PATH DECOMPOSITION *of a graph $G(V, E)$ is a path $P(V, E)$, with nodes $V = \{X_1, ..., X_n\}$, with for each i, X_i is a set of vertices from $V(G)$. Such that the following conditions are satisfied.*

- $\bigcup_{X_i \in X} X_i = V(G)$
- $\forall_{v \in V(G)} Z_v$ *form a connected subpath of T, where* $Z_v := \{X_i | v \in X_i, i = 1..n\}$
- $\forall_{e(v,w) \in E(G)} \Rightarrow \exists X_i$ *such that* $v, w \in X_i$
- $\forall_{X_i \in T} X_i \Rightarrow \triangle X_i \leq 2$

Two examples of path decompositions can be seen in Figure 3. Here we can also verify that both path decompositions satisfy the conditions from Definition 3.4. It is also possible to create infinity many unique path decompositions for every graph $G(V, E)$, where $V(G)$ is not the empty set.

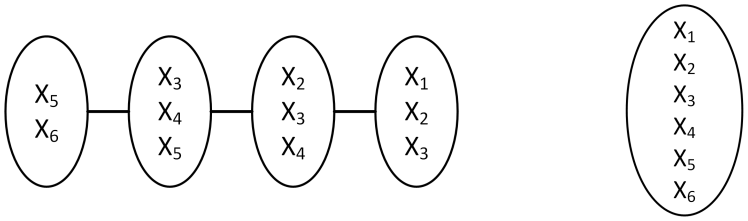

Figure 3: Two *path decompositions* constructed by the graph from Figure 1

To calculate the pathwidth of a path decomposition $P(V, E)$, we take the size of the largest node from $V(P)$ minus one, see Definition 3.5. This is exactly the same as calculating the treewidth. If we apply the pathwidth formula on the two path decompositions in Figure 3, we obtain values 2 and 5 respectively.

Definition 3.5 *The* PATHWIDTH *of a path decomposition $P(V, E)$ is a single number defined by*

$$\text{PATHWIDTH} = \max_{X \in P(V)} (|X| - 1)$$

The minimum pathwidth of a graph $G(V, E)$ is defined as the minimum pathwidth over all possible path decompositions, see Definition 3.6. For our example from Figure 1 we have that the minimum pathwidth equals two and a corresponding path decomposition is the left path in Figure 3.

Definition 3.6 *The* MINIMUM PATHWIDTH *of a graph $G(V, E)$ is a single number defined by*

$$\text{MINIMUM PATHWIDTH} = \min_{path\ decomposition\ P(V,E)\ from\ G(V,E)} treewidth(P(V, E))$$

3.1.1 Problem Description

Our objective in this section will be to improve the current state algorithms to calculate the minimum pathwidth, given a arbitrary graph $G(V, E)$. The pathwidth of a graph is the maximum over the pathwidth of its connected component. Thus, we assume that the input graph is connected. Otherwise we can run the algorithms separately on each connected component.

3.2 Theory

In this section we will introduce two new theorems for calculating the minimal pathwidth. We will give the correctness prove of these two theorems and explain how they can be used in practise.

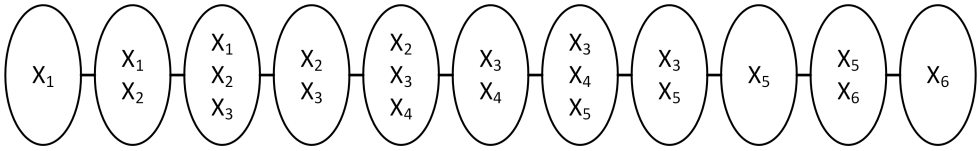

Figure 4: Example path decomposition

The two theorems are based on creating an *elimination ordering* for a given path decomposition. We will give the intuition by an example. Assume the graph in Figure 1, a possible path decomposition from this graph and can be seen in Figure 4, do note that this is according to Definition 3.4.

We want to express the path decomposition in a more compact form so we can reconstruct this particular path or its equivalent. If we look at the nodes from left to right in the path decomposition we see that each node next to one other either adds a vertex or removes a vertex. Such a path decomposition is known as a nice path decomposition introduced by Kloks [10]. This inspires us to create an ordering that represents this sequence. The first method, $\overrightarrow{PW}$, is based on the order of removal of vertices, which we call an elimination ordering. When we apply the elimination ordering from left to right, we create the ordering $\Pi^1 = [x_1, x_2, x_4, x_3, x_5, x_6]$. From this ordering we want to construct a path decomposition that is equivalent to Figure 4.

To do this we need to reconstruct the nodes X_i from the ordering. If we look at the ordering, when the first vertex x_1 is removed, all the neighbors of this vertex are included inside the node. When the second vertex x_2 is removed, we have all neighbors of x_1 and x_2 in the node. This continues onward until all the vertices have been eliminated. So according to our description we would get a *path decomposition* that looks like Figure 5.

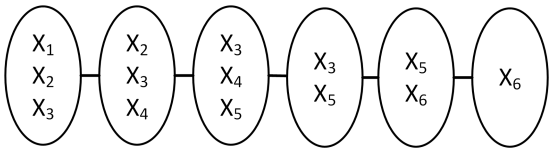

Figure 5: Example left to right

Instead of going from left to right we will now look the at the path decomposition from right to left. We want to construct an ordering again with a different technique. The second method, $\overleftarrow{PW}$, is based on the addition of vertices, which we call an introduce ordering. When we construct the introduce ordering from right to left, we create the ordering $\Pi^2 = [x_6, x_5, x_3, x_4, x_2, x_1]$.

To reconstruct the path decomposition from this ordering we will have to do something different in comparison to the previous idea. What we are going to do is the following: we keep on adding the vertices from the ordering. But after each new vertex we have a set X containing all vertices that are added until that point, we check for each vertex x_i in X if all his neighbors are also contained in X. If this is the case we do not include x_i in the upcoming nodes. For our example we add x_6, we check if all the neighbors are already used, which is not the case. So we add the second vertex x_5. Now vertex x_6 has all the neighbors used, so we do not include from this point vertex x_6 in any of the upcoming nodes. If we continue this we construct a path decomposition that looks like Figure 6.

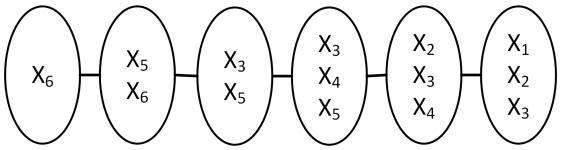

Figure 6: Example right to left

We have introduced the base for two methods of how to create an path decomposition from an ordering, either by elimination or introduction. If we look at both orderings the following is observed. The elimination ordering Π^1 reversed is the introduce ordering Π^2. This suggests that the results of both methods are each others complement.

In the rest of this section we will work out the ideas into general approaches. The approaches will be proven to give the minimum pathwidth.

3.2.1 Minimum pathwidth by elimination ordering

To create a path decomposition using the elimination ordering we need to know all the neighbors of previously eliminated vertices. This will be expressed by Definition 3.7. The reason why we need this is because we need to simulate the creation of a *clique*.

Definition 3.7 *The neighbors of a given set $S \subseteq V$ and graph $G(V, E)$ is denoted by*

$$N(S) = \{x \mid x \in \ (V - S),\ \exists s\ \in S \text{ and } \exists e(x, s) \in E(G)\}$$

To construct a node of the path decomposition we use the following Definition 3.8. Here we represent x as the vertex to be eliminated and W to be the set of already eliminated vertices. The node that is created contains x and all vertices $V - W$ that have an edge to any vertex in W or to x. This function will be used as subroutine to calculate the pathwidth of an ordering on an arbitrary graph.

Definition 3.8 *Let $G(V, E)$ be a graph. $\overrightarrow{PW'}(x, W)$ is the function with arguments set $W \subseteq V$ and a vertex $x \notin W$, defined by*

$$\overrightarrow{PW'}(x, W) = N(W \cup \{x\}) \cup \{x\}$$

Using Definition 3.8 we can find the pathwidth of a given ordering Π, see Definition 3.9. We construct all the nodes from the path decomposition corresponding to ordering Π. On this path decomposition we apply Definition 3.5 to obtain the corresponding pathwidth.

Definition 3.9 *Let $G(V, E)$ be a graph. $\overrightarrow{PW}_{ord}(\Pi)$ is the function with arguments an ordering $\Pi = \{x_1, .., x_n\}$.*

$$\overrightarrow{PW}_{ord}(\Pi) = \max_{i=1,..,n} (|\overrightarrow{PW'}(x_i, \Pi_{i-1})| - 1)$$

To obtain the minimum pathwidth of a connected graph $G(V, E)$, we need to construct all possible path decompositions and take the minimum of all pathwidths, see Definition 3.10. For this we have used Definition 3.6.

Definition 3.10 *Let $G(V, E)$ be a graph. $\overrightarrow{PW}(S)$ is the function with arguments a set $S \subseteq V$.*

$$\overrightarrow{PW}(S) = \min_{x \in S}(\max(|\overrightarrow{PW'}(x, V - S)| - 1, \overrightarrow{PW}(S - \{x\})$$

We claimed that $\overrightarrow{PW}$ gives the minimum pathwidth. We will show that this statement is correct, see Theorem 3.11. This result will be proven by Lemmas 3.12 and 3.13.

Theorem 3.11 *Let $G(V, E)$ be a graph, $S \subseteq V$ a set of vertices and let $k \geq 0$. Then the following statements are equivalent:*

1. $\overrightarrow{PW}(S) \leq k$
2. *There exists a path decomposition of $G[V]$, with pathwidth $\leq k$*

Lemma 3.12 *Let $S \subseteq V$ and suppose $\overrightarrow{PW}(S) \leq k$. Then there exists a path decomposition of $G[S]$, with pathwidth $\leq k$*

Proof: *We will prove this by looking at the three conditions of path decomposition:*

1. *For all $x \in S$, there exists a node B_z such that $x \in B_z$*
2. *For all $x \in S$, the nodes B^x form a connected subset*
3. *For all edges $e(i, j) \in E(G)$, there exists a Bag B_e such that, $x_i, x_j \in B_z$*

Let Π be the ordering of S such that $\overrightarrow{PW'}(\Pi) \leq k$.
1) We know by definition that $\overrightarrow{PW'}_{ord}$ creates for each $x \in \Pi$ a node that contains x
2) We prove the statement by induction.
First note that for $\overrightarrow{PW'}(\Pi_1, \emptyset)$ the statement is valid,
For all j with $2 \leq j \leq |S|$, the following holds:

$$\begin{aligned}
\overrightarrow{PW'}(x_j, \Pi_{j-1}) &= N(\{x_j\} \cup \Pi_{j-1}) \cup \{x_j\} \\
&= N(\{x_j\} \cup \{x_{j-1}\} \cup \Pi_{j-2}) \cup \{x_j\} \\
&= N(\{x_{j-1}\} \cup \Pi_{j-2}) \cup (N(\{x_j\}) - \Pi_{j-1}) \cup \{x_j\} \\
&= N(\{x_{j-1}\} \cup \Pi_{j-2}) \cup \{x_{j-1}\} \cup (N(\{x_j\}) - \Pi_{j-1}) \cup \{x_j\} - \{x_{j-1}\} \\
&= \overrightarrow{PW'}(x_{j-1}, \Pi_{j-2}) \cup (N(\{x_j\}) - \Pi_{j-1}) \cup \{x_j\} - \{x_{j-1}\}
\end{aligned}$$

So vertex x_j will not be contained in any of the nodes k, where $k > j$. Thus for all $x \in \Pi$, the nodes B^x form a connected subset
3) For all $e(x_i, x_j) \in E(G)$, we chose x_i to be before x_j in the ordering Π. Now we know that $\overrightarrow{PW'}(x_i, \Pi_{i-1})$ contains both x_i and x_j. We have proven that if $\overrightarrow{PW}(S) \leq k$ for $S \subseteq V$ then there exists a path decomposition of $G[S]$ with pathwidth $\leq k$

□

Lemma 3.13 *Let $S \subseteq V$. Suppose there exists a path decomposition $G[S]$ with pathwidth $\leq k$. Then $\overrightarrow{PW}(S) \leq k$*

Proof: *By definition of path decomposition there exists an ordering* Π_{elim} *that has a path width* $\leq k$*. We can take the ordering of* Π_{elim} *as the ordering for* $\overrightarrow{PW}(S)$*. This will give us* $\overrightarrow{PW}_{ord}(\Pi_{elim})$*. We will prove the correctness by induction,*
1) We pick $x_1 \in \Pi_{elim}$*, then* $\overrightarrow{PW'}(x_1, \emptyset) = N(\{x_1\}) \cup \{x_1\} \subseteq Bag_1$*.*
2) For all $j, 2 \leq j \leq |S|$*, this will give us the following equations,*

$$\begin{aligned}
Bag_j &\supseteq (Bag_{j-1} \cup N(\{x_j\}) \cup \{x_j\}) - (\bigcup_{i=1}^{j-1} \{x_i\}) \\
&\supseteq (N(\{x_1\}) \cup \{x_1\}) \cup \bigcup_{i=2}^{j-1} (N(\{x_i\}) \cup \{x_i\} - (\bigcup_{k=1}^{i-1} \{x_k\})) \cup N(x_j) \cup \{x_j\}) - (\bigcup_{i=1}^{j-1} x_i) \\
&\supseteq (N(\bigcup_{i=1}^{j} (\{x_i\})) \bigcup_{i=1}^{j} \{x_j\} - (\bigcup_{i=2}^{j} \bigcup_{k=1}^{i-1} \{x_k\})) \cup N(\{x_{i+1}\})\}) - (\bigcup_{i=1}^{j-1} x_i) \\
&\supseteq N(\bigcup_{i=1}^{j} (\{x_i\})) \bigcup_{i=1}^{j} \{x_j\} - (\bigcup_{i=1}^{j-1} \{x_i\}) \\
&\supseteq N(\Pi_{elim_j}) \cup \{x_j\} \\
&\supseteq \overrightarrow{PW'}(\Pi_{elim_j})
\end{aligned}$$

We have proven that if there exists a path decomposition $G[S]$ *with pathwidth* $\leq k$ *where* $S \subseteq V$ *then there exists a* $\overrightarrow{PW}(S) \leq k$*.*

□

We have proven that $\overrightarrow{PW}(S)$ denotes the width of a valid path decomposition of minimum pathwidth for any set $S \subseteq V$. If we take S equal to V we can construct a path decomposition containing all vertices. By our previous statement we define the minimum pathwidth for the entire graph, see Corollary 3.14.

Corollary 3.14 *Given a graph* $G(V, E)$*, then there exists a path decomposition of* V *with pathwidth* $\leq k$ *if and only if* $\overrightarrow{PW}(V) \leq k$

In Section 3.3.5 an algorithm is constructed for the calculation of the minimum treewidth based on Corollary 3.14.

3.2.2 Minimum pathwidth by introduction ordering

We will construct in the same way a path decomposition for the introduce ordering. To construct a node we use the following Definition 3.15. Here is x the vertex that is introduced and W the set of vertices already introduced. The node created contains x and all vertices in W that have an edge to any vertex in $V - W$. This function will be used in the construction of a path decomposition.

Definition 3.15 *Let* $G(V, E)$ *be a graph.* $\overleftarrow{PW'}(x, W)$ *is the function with arguments* $W \subseteq V$*,* $x \notin W$*s.*

$$\overleftarrow{PW'}(x, W) = \{w | w \in W, \exists z \in (V - W) \text{ and } \exists e(w, z) \in E(G)\} \cup \{x\}$$

Using Definition 3.15 we can construct a path decomposition of $G(V, E)$, fulfilling all conditions. From this path decomposition we can determine the pathwidth. This is done by combining Definition 3.15 with Definition 3.5, see Definition 3.16.

Definition 3.16 *Let* $G(V, E)$ *be a graph.* $\overleftarrow{PW}_{ord}(\Pi)$ *is the function with arguments an ordering* $\Pi = \{x_1, .., x_n\}$*.*

$$\overleftarrow{PW}_{ord}(\Pi) = \max_{i=1,..,n}(|\overleftarrow{PW'}(x_i, \Pi_{i-1})| - 1)$$

To obtain the minimum pathwidth of a complete graph $G(V, E)$ we have to try all possible combinations of introduce orderings. From these orderings we can construct the path decompositions to finally obtain the pathwidth. This recursive function can be seen in Definition 3.17.

Definition 3.17 *Let $G(V, E)$ be a graph. $\overleftarrow{PW}(S)$is the function with arguments set $S \subseteq V$.*

$$\overleftarrow{PW}(S) = \min_{x \in S}(\max(|\overleftarrow{PW'}(x, V - S)| - 1, \overleftarrow{PW}(S - \{x\})$$

For the function $\overleftarrow{PW}$ we claim that the result equals the minimum pathwidth, see Theorem 3.18. This statement will be proven with Lemmas 3.19 and 3.20.

Theorem 3.18 *Let $G(V, E)$ be a graph, $S \subseteq V$ and let $k \geq 0$. Then the following statements are equivalent:*

1. $\overleftarrow{PW}(S) \leq k$
2. *There exists a path decomposition of $G[V]$, with pathwidth $\leq k$*

Lemma 3.19 *Let $S \subseteq V$ and suppose there exists a $\overleftarrow{PW}(S) \leq k$. Then there exists a path decomposition of $G[S]$, with pathwidth $\leq k$*

Proof: *We will prove three conditions:*

1. *For all $x \in S$, there exists a Bag B_z such that $x \in B_z$*
2. *For all $x \in S$, the Bags B_x form a connected subset*
3. *For all edges $e(i, j) \in E(G)$, there exists a Bag B_z such that, $x_i, x_j \in B_z$*

Let Π be the ordering of S such that $\overleftarrow{PW'}(\Pi) \leq k$. 1) We know by definition that $\overleftarrow{PW'}_ord$ creates for each $x \in \Pi$ a node that contains x
2) We prove the statement by induction.
First note that for $\overleftarrow{PW'}(\Pi_1, \emptyset)$ the statement is valid,
For all j with $2 \leq j \leq |S|$, the following holds:

$$\begin{aligned}
\overleftarrow{PW'}(x_j, \Pi_{j-1}) &= \{w | w \in W, \exists z \in (V - \Pi_{j-1}) \text{ and } \exists e(w, z) \in E(G)\} \cup \{x_j\} \\
&= \{w | w \in W, \exists z \in (V - \Pi_{j-2}) \text{ and } \exists e(w, z) \in E(G)\} \cup \{x_{j-1}\} \cup \{x_j\} - \\
&\quad \{w | w \in \Pi_{j-1}, \nexists y \in (V - \Pi_{j-1}) \text{ and } \nexists e(w, y) \in E(G)\} \\
&= \overleftarrow{PW'}(x_{j-1}, \Pi_{j-2}) \cup \{x_{j-1}\} \cup \{x_j\} - \\
&\quad \{w | w \in \Pi_{j-1}, \nexists y \in (V - \Pi_{j-1}) \text{ and } \nexists e(w, y) \in E(G)\}
\end{aligned}$$

So vertex x_j will not be contained in any of the nodes k, where $k < j$. Thus for all $x \in \Pi$, the nodes B^x form a connected subset
3) For all $e(x_i, x_j) \in E(G)$, we chose x_i to be before x_j in the ordering Π. Now we know that $\overleftarrow{PW'}(x_j, \Pi_{i-j})$ contains both x_i and x_j. We have proven that if $\overleftarrow{PW}(S) \leq k$ for $S \subseteq V$ then there exists a path decomposition of $G[S]$, with pathwidth $\leq k$

□

Lemma 3.20 *Let $S \subseteq V$ and suppose there exists a path decomposition $G[S]$ with pathwidth $\leq k$. Then $\overleftarrow{PW}(S) \leq k$*

Proof: *By definition of path decomposition there exists an ordering Π_{intro} that has a path width $\leq k$. We can take the ordering of Π_{intro} as the ordering for $\overleftarrow{PW}(S)$. This will give us $\overleftarrow{PW}_{ord}(\Pi_{intro})$. We will prove the correctness by induction,*

1) We pick $x_1 \in \Pi_{intro}$, then $\overleftarrow{PW'}(x_1, \emptyset) = \{x_1\} \subseteq Bag_1$.
2) For all $j, 2 \leq j \leq |S|$, this will give us the following equations,

$$\begin{aligned}
Bag_j &\supseteq Bag_{j-1} \cup \{x_j\} \ - \ \{w | w \in \Pi_{j-1}, \nexists y \in (V - \Pi_{j-1}) \text{ and } \nexists e(w,y) \in E(G)\} \\
&\supseteq (\bigcup_{i=1}^{j-1} (\bigcup_{k=1}^{i} \{x_k\}) \ - \ \{w | w \in \Pi_{i-1}, \nexists y \in (V - \Pi_{i-1}) \text{ and } \nexists e(w,y) \in E(G)\}) \cup \\
&\quad \{x_j\} \ - \ \{w | w \in \Pi_{j-1}, \nexists y \in (V - \Pi_{j-1}) \text{ and } \nexists e(w,y) \in E(G)\} \\
&\supseteq (\bigcup_{k=1}^{j} \{x_k\}) \ - \ (\bigcup_{i=1}^{j} \{w | w \in \Pi_{i-1}, \nexists y \in (V - \Pi_{i-1}) \text{ and } \nexists e(w,y) \in E(G)\}) \\
&\supseteq \overleftarrow{PW'}(x_j, \Pi_{i-j})
\end{aligned}$$

We have proven that if there exists a path decomposition $G[S]$ with pathwidth $\leq k$ where $S \subseteq V$ then there exists a $\overleftarrow{PW}(S) \leq k$.

□

We have proven that $\overleftarrow{PW}(S)$ denotes the width of a valid path decomposition that has the least pathwidth for any set $S \subseteq V$. Suppose we take S equal to V then $\overleftarrow{PW}(V)$ defines the minimum pathwidth. This has been proven by Theorem 3.18, see Corollary 3.21.

Corollary 3.21 *Given a graph $G(V, E)$ and a $S \subseteq V$ then there exists a path decomposition of $S \subseteq V$ with pathwidth $\leq k$ if and only if $\overleftarrow{PW}(S) \leq k$*

In Section 3.3.5 an algorithm is constructed for the calculation of the minimum treewidth based on Corollary 3.21.

3.2.3 Minimum pathwidth by a hybrid method

We have seen in Sections 3.2.1 and 3.2.2 two different methods that construct path decompositions of optimal width. In Section 3.2 we have given the intuition that both methods can be combined, see Theorem 3.22. The idea is to combine the two methods to construct a path decomposition for $V(G)$. This is done by creating two path decomposition of sets S and $V - S$ by the different methods and paste the decompositions together. We will prove that this technique results in the creation of an optimal path decomposition with minimal pathwidth, see Lemmas 3.23 and 3.24.

Theorem 3.22 *Let $G(V, E)$ be a graph, $S \subseteq V$ and let $k \geq 0$. Then the following statements are equivalent:*

1. *$\overrightarrow{PW}(S) \leq k$ and $\overleftarrow{PW}(V - S) \leq k$*
2. *There exists a path decomposition of $G[V]$, with pathwidth $\leq k$*

Lemma 3.23 *Let $S \subseteq V$ and suppose $\overrightarrow{PW}(S) \leq k$ and $\overleftarrow{PW}(V - S) \leq k$. Then there exists a path decomposition of $G[V]$, with pathwidth $\leq k$*

Proof: *We will prove three conditions:*

1. *For all $x \in S$, there exists a Bag B_z such that $x \in B_z$*
2. *For all $x \in S$, the Bags B_x form a connected subset*
3. *For all edges $e(i,j) \in E(G)$, there exists a Bag B_z such that, $x_i, x_j \in B_z$*

Let Π^1 be the ordering of S such that $\overrightarrow{PW}(\Pi^1) \leq k$ and let Π^2 be the ordering of $(V-S)$ such that $\overleftarrow{PW}(\Pi^2) \leq k$.

1) We know by definition that for all $x_i \in S$, $\overrightarrow{PW'}_{ord}$ creates a node that contains x_i and when $x_i \in V - S$, $\overleftarrow{PW'}_{ord}$ creates a node that contains x_i.
2) We know that for all $x_i \in S, x_i$ form a connected subset and denote k equal to $|S|$ and j equal to $|V-S|$. We will prove that the last bag of $\overrightarrow{PW}(S)$ equals $\overleftarrow{PW'}(x_k, (V-S))$. This will give us the following equations,

$$\begin{aligned}
\overrightarrow{PW'}(x_k, \Pi^1_{k-1}) &= N(\{x_k\} \cup \Pi^1_{k-1}) \cup \{x_k\} \\
&= \{z | z \in (V-S), \exists s \in S \text{ and } \exists e(z,s) \in E(G)\} \cup \{x\} \\
&= \overleftarrow{PW'}(x_k, \Pi^2_j)
\end{aligned}$$

We now know that for all $x_i \in (V-S), x_i$ form a connected subset by definition of $\overleftarrow{PW}$.
3) By definition of $\overrightarrow{PW}$ and $\overleftarrow{PW}$, all edges $e(i,j) \in E(G)$ it is either contained in $\overrightarrow{PW}$ or $\overleftarrow{PW}$.

We have proven that if $\overrightarrow{PW}(S) \leq k$ and $\overleftarrow{PW}(V-S) \leq k$. Then there exists a path decomposition of $G[V]$, with pathwidth $\leq k$.

□

Lemma 3.24 *Suppose there exists a path decomposition $G[V]$ with pathwidth $\leq k$. Then there exists a set $S \subseteq V, |S| = \frac{1}{2}|V|$ such that $\overrightarrow{PW}(S) \leq k$ and $\overleftarrow{PW}(V-S) \leq k$*

Proof: *By definition of path decomposition there exists an ordering Π_{elim} that has pathwidth $\leq k$. We choose $\Pi^1 = \{x_1, .., x_j\}$ and $\Pi^2 = \{x_n, .., x_{j+1}\}$ where $j = \frac{1}{2}|V|$ and x_i is the i^{th} vertex of ordering Π. This will give us $\overrightarrow{PW}_{ord}(\Pi^1) \leq k$ and $\overleftarrow{PW}_{ord}(\Pi^2) \leq k$ by definition. We know that the last bag of $\overrightarrow{PW}_{ord}$ contains x_j and vertices $N(\Pi^1)$. The last bag of $\overrightarrow{PW}_{ord}$ contains all vertices of Π^2 that have an edge between a vertex that is not in Π^2 which equals $N(\Pi^1)$. Therefore we have that if there exists a path decomposition $G[V]$ with pathwidth $\leq k$, then there exists a set $S \subseteq V, |S| = \frac{1}{2}|V|$ such that $\overrightarrow{PW}(S) \leq k$ and $\overleftarrow{PW}(V-S) \leq k$.*

□

We have proven that we can construct every path decomposition with the combinations of the two methods. The following Corollary 3.25 is a result of this theorem.

Corollary 3.25 *Given a graph $G(V,E)$ and a $S \subseteq V$ then there exists a path decomposition of $S \subseteq V$ with pathwidth $\leq k$ if and only if $\overrightarrow{PW}(S) \leq k$ and $\overleftarrow{PW}(V-S) \leq k$*

3.3 Algorithm

In this section the algorithms will be discussed and explained. First we give the general approach for solving the minimum pathwidth problem. After that we provide additional methods to increase the computing time of the algorithms. In the Appendix B, we give the pseudo code for the algorithms.

3.3.1 Search space

In this section we will explain what the corresponding search tree is if we want to compute the minimum pathwidth. Further we describe how we can reduce the search space using dynamic programming and how we can use this technique in our algorithms. First we recall two definitions from section 3.2.

Definition 3.26

$$\begin{aligned}\overrightarrow{PW}(S) &= \min_{x\in S}(\max(|N(S\cup\{x\})\cup\{x\}|-1, \overrightarrow{PW}(S-\{x\})\\ \overleftarrow{PW}(S) &= \min_{x\in S}(\max(|\{s|s\in S, \exists z\in(V-S)\ and\ \exists e(s,z)\in E(G)\}\cup\{x\}|-1, \overleftarrow{PW}(S-\{x\})\end{aligned}$$

To compute all possible combinations for an arbitrary graph $G(V, E)$ would be too time consuming, this means that the complexity of our algorithms will become $\mathcal{O}^*(|V|!)$. This means that for graphs with $|V| \geq 11$, at least 40 billion computations need to be computed which will not be of any practical use. So we would like to find an algorithm based on these definitions that will have a running time smaller then $\mathcal{O}^*(|V|!)$. To find such an algorithm, dynamic programming techniques will be applied to Definition 3.26.

The idea of a general dynamic-programming approach is to cut the problem into subproblems of smaller size. Then solve these subproblems to finally combine them together to get a solution for the original problem. A dynamic-programming approach will only solve each subproblem once and store the corresponding answer to avoid recalculation; this technique is called memorization. The question arises, how can we apply this technique to Definitions 3.26?

Suppose we have a graph where $V = \{1, 2, 3\}$. If Definition 3.26 is applied naively; then a search tree would be created that will look like Figure 7.

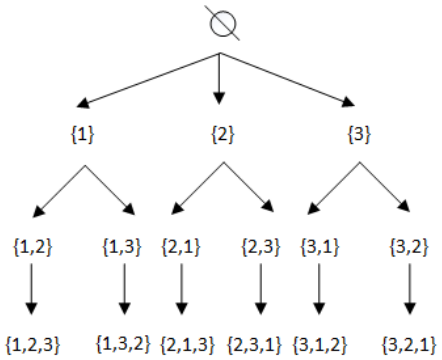

Figure 7: Example: using definition 1 with $S = \{1, 2, 3\}$

At the bottom, six (3!) different orderings have been created. To create these orderings we must do 15 calculations. However if the vertices are represented as a binary string, we are

able to create the same search tree but with different designations, see Figure 8.

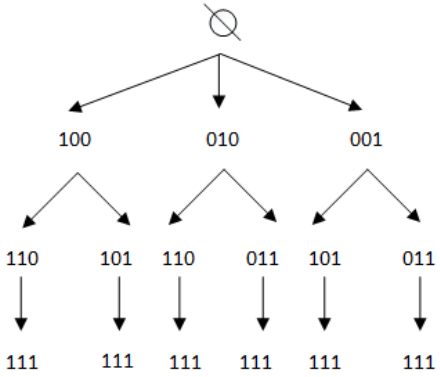

Figure 8: Example: using definition 1 with $S = \{1, 2, 3\}$, binary notation

Still we create six different orderings and need 15 calculations to do so. However there are a lot of the same subsets calculated. If we look to Definition 1 we observe the following situation. We compute $\overrightarrow{PW}(S - \{x\})$ and $\overleftarrow{PW}(S - \{x\})$, where $S \subseteq V$ and $x \in S$. We can also pick an $S' \subseteq V$ and $y \in V$ such that $S - \{x\} = S' - \{y\}$ where $S' = (S - \{x\}) \cup \{y\}$. This statement holds for all $y \in (V - S)$.

From this observation we can conclude that the same sets S with $\overrightarrow{PW}$ and $\overleftarrow{PW}$ are calculated multiple times. If we store S once with corresponding treewidth value, we can remove a lot of unnecessary calculations. By doing this we reduced the search tree to figure 9.

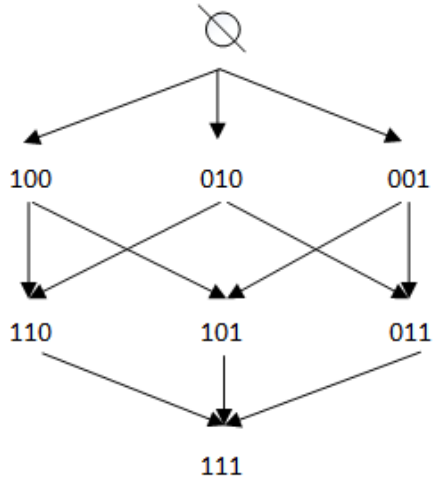

Figure 9: Example: using definition 1 with $S = \{1, 2, 3\}$, binary notation with dynamic programming

One ordering is created with only 7 calculations. By reducing the search tree we can conclude that this approach will reduce the running time from $\mathcal{O}^*(|V|!)$ to $\mathcal{O}^*\left(2^{|V|}\right)$. The dynamic programming approach will be used to construct the algorithms.

3.3.2 Expanding the search space

In Subsection 3.3.1 the search space is given on how to efficiently determine the minimum pathwidth. Dynamic programming is a great way for reduction of the search space. In this

section the focus will be on how to explore the search space.

We recall Definition 3.26. The following situation occurs. We take the maximum value of either the size of the node constructed with vertex x or the minimum pathwidth value that is calculated for all possible subsets. From this we can conclude that Lemma 3.27 holds.

Lemma 3.27 *Let PW be any of the functions from Definition 3.26, $S \subseteq V$ and $x \in V - S$ then the following holds:*

$$PW(S) \leq PW(S \cup \{x\})$$

This observation can be used to find ways to explore the search space. Two ways of exploring the search space will be provided to obtain the minimum pathwidth with a running time of complexity $\mathcal{O}^*\left(2^{|V|}\right)$.

3.3.2.1 Tabulation

The first exploration is called *tabulation*. As we have seen in Figure 9 each layer is dependent only on the first layer above it. This gives us the idea that we can explore the search space, layer by layer. A data structure like Figure 10 can be created this way. Here each layer in the table contains all subsets of the same size. The idea is to start with the empty set in layer 0 and then compute from this layer all other layers in order and one-by-one. If we apply this technique, we can guarantee that the minimum pathwidth is found when the n^{th} layer is reached.

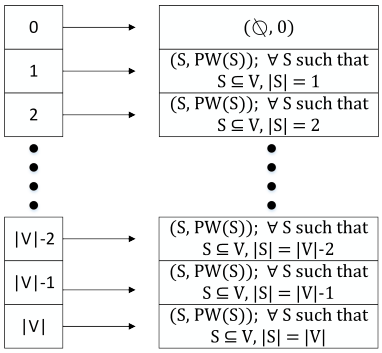

Figure 10: Tabulation construction, layer by layer storing the ordering and minimum pathwidth value

3.3.2.2 Prioritize

The second exploration is called *prioritize*. For this, the property from Definition 3.27 is used. The idea is to expand the node that has the least pathwidth value of the entire search tree. By expanding this node X_{low} with least pathwidth value, we know that we cannot construct a path decomposition that has a lower pathwidth than node X_{low}. This is due to the parent nodes that have a lower pathwidth value have already been expanded. They did not result in a path decomposition with lower pathwidth value than the current value of node X_{low}. For the parents of X_{low} that have a higher pathwidth we know that they will not result in a better pathwidth value, because of Definition 3.27.

3.3.3 Pathwidth at the end

A different method to decrease the computation time is *pathwidth at the end*. This increase can be directly obtained from the definitions of PW in Definition 3.26. The idea is to not fully calculate the search space obtained by applying dynamic programming. Some of the final layers do not need to be computed.

We recall the computation of the the node in $\overrightarrow{PW}$ path decompositions, see Definition 3.28. We know that the set $N(W \cup \{x\})$ is at most $V - W$. Thus we know that $\overrightarrow{PW'}(x, W) \leq |(V - W)| + 1$.

Definition 3.28 *(Node creation for $\overrightarrow{PW}$ in path decomposition)*

$$\overrightarrow{PW'}(x, W) = N(W \cup \{x\}) \cup \{x\}$$

The same can be done for $\overleftarrow{PW}$ in an only slightly different way, see Definition 3.29. Each node contains $|\{w|w \in W, \exists z \in (V - W) \text{ and } \exists e(w, z) \in E(G)\} \cup \{x\}|$ elements. We know that the pathwidth can be increased at most by $(V - W)$. Thus we know that $\overleftarrow{PW'}(x, W) \leq |(V - W)| + |\{w|w \in W, \exists z \in (V - W) \text{ and } \exists e(w, z) \in E(G)\} \cup \{x\}|$.

Definition 3.29 *(Node construction for $\overleftarrow{PW}$ in path decomposition)*

$$\overleftarrow{PW'}(x, W) = \{w|w \in W, \exists z \in (V - W) \text{ and } \exists e(w, z) \in E(G)\} \cup \{x\}$$

Using these properties will mean we can stop with the computation of the functions from Definition 3.26 early. Namely for $\overrightarrow{PW}(S)$ when the pathwidth equals $|(V - S)| + 1$. And for $\overleftarrow{PW}(S)$ when the pathwidth equals $|(V - S)| + |\{s|s \in S, \exists z \in (V - S) \text{ and } \exists e(s, z) \in E(G)\} \cup \{x\}|$.

3.3.4 Incremental increase

All $\mathcal{NP}$-complete problems are decision problems. The decision version of the pathwidth problem is given in Definition 3.30. For pathwidth the decision notation is shown in Definition 3.30. We can use this decision to find the minimum pathwidth. In the rest of this section we will address how this is done and how we can apply the technique in our algorithms.

Definition 3.30 *Let $G(V, E)$ be a graph, does there exists a path decomposition with pathwidth $\leq k$?*

When we construct a path decomposition that has a pathwidth greater than this upper bound k, we will not expand the node further in the search tree. Doing so will reduce the search space. To find the minimum pathwidth we will need to run our algorithm multiple times. Each time with a different upper bound k. The question is, how will we choose this upper bound such that it will result in finding the minimum pathwidth? The first thing that comes to mind is apply a binary search. This will ensure that we find the minimum pathwidth. What we do when we find a path decomposition that has at least a pathwidth of k, we decrease the upper bound. If such a path decompositions cannot be constructed we increase the upper bound.

To test the running time in practice, we have calculated the time it takes to create the entire search tree with a given upper bound for two graphs. The result of this test can be seen in Table 1.

Upper bound	Queen6_6	Mainuk	Upper bound	Queen6_6	Mainuk
1	90	175	16	284	125110
2	91	418	17	479	132766
3	90	1031	18	362	143219
4	89	1686	19	364	162847
5	94	4998	20	347	184759
6	95	10187	21	432	216826
7	93	40153	22	685	261387
8	91	50817	23	690	291086
9	90	60612	24	1135	334448
10	86	69156	25	1893	321060
11	85	83603	26	4359	365195
12	82	94799	27	7625	398262
13	77	103269	28	1024	438488
14	75	107909	29	1582	473995
15	338	118954	30	230	533886

Table 1: The time to calculate all possible subsets in milliseconds for given upper bound using pathwidth at the end. Graphs: Queen6_6, vertices 36, minimum pathwidth 25 and Mainuk, vertices 48, minimum pathwidth 7

We notice that for low upper bound values, the time to calculate the entire search tree is very small compared to the higher upper bound values. If we apply binary search on both graphs we get the following computation time. For Mainuk we address upper bounds, (24, 12, 6, 9, 7) and Queen6_6, (19, 27, 23, 25, 26). This will result in a computation time of Mainuk: 540199 and Queen6_6: 11707. Instead of using a binary approach we can also increase the upper bound each time by 1 (*incremental increase*). The computation time for the last case will be, Mainuk: 58648 and Queen6_6: 8237. This implies that using a binary search on the upper bound is not the best idea. Because of this observation it seems that a better way to use the upper bound is to incremental increase it. We start the upper bound at 1 and run our algorithm. If a path decomposition cannot be constructed we increase the upper bound by 1. The first time we are able to create a path decomposition, it is guaranteed that the upper bound equals the minimum pathwidth.

3.3.5 Implemented algorithms

We have implemented 5 different algorithms based on combinations of previously discussed methods. In the different paragraphs we will explain the choices of the combinations that are used.

3.3.5.1 Algorithm 1

The first algorithm we propose, will make use of Definition 3.10. We apply the dynamic programming technique together with the tabulation method. The algorithm will represent the naive approach without using any other optimizations. It will be used for comparison of the others. The pseudo code can be seen in Appendix B, under Algorithm 1.

3.3.5.2 Algorithm 2

Algorithm 2 will contain all possible optimizations discussed previously. It will use dynamic programming with tabulation and Definition 3.10. The optimization techniques we use are incremental increase and pathwidth at the end. We assume that incremental increase will decrease the computation time significantly. The algorithm can be seen in Appendix B, under Algorithm 2.

3.3.5.3 Algorithm 3

This algorithm will be the same as algorithm 1. However Definition 3.17 is used. The algorithm is for the comparison of algorithm 4, to see how the optimization techniques work for this definition. The algorithm can be seen in Appendix B, under Algorithm 3.

3.3.5.4 Algorithm 4

This algorithm will be the same as algorithm 2. However we will use Definition 3.17. To see the effects of incremental increase and pathwidth at the end for Definition 3.17. The algorithm can be seen in Appendix B, under Algorithm 4.

3.3.5.5 Algorithm 5

For the last algorithm we combine breath-first-search with depth-first-search. We use dynamic programming with priority. We do not implement the incremental increase, because the algorithm will only expend nodes with the least pathwidth. So starting over again when we are not able to find a path decomposition, will result in many computations that are unnecessary. Thus using this technique will only slow the algorithm down. We also do not need to use the pathwidth at the end. Because the pathwidth value is not increase by this technique. It will take at most n computations to compute the minimum pathwidth if such a case occurs. The algorithm can be seen in Appendix B, under Algorithm 5.

3.3.5.6 Overview

In Table 2 we have the overview of the described algorithms. These algorithms will be tested and the results can be found in Section 3.4.

All implementations use boolean vectors to store the different states of the search tree. The graphs edges are stored with a double vector containing boolean values.

#Alg	PW method	DP	Incremental increase	Tabulation	Priority	Pathwidth at the end
1	$\overrightarrow{PW}$	X	-	X	-	-
2	$\overrightarrow{PW}$	X	X	X	-	X
3	$\overleftarrow{PW}$	X	-	X	-	-
4	$\overleftarrow{PW}$	X	X	X	-	X
5	$\overrightarrow{PW}$	X	-	-	X	-

Table 2: Implemented algorithms for finding the minimum pathwidth.

3.4 Experiments

We will present the experiments obtained by running our algorithms from Section 3.3.5. The results can be found in Table 6 and Appendix A . It is done on a Windows 7 ultimate OS with Intel i7 970 quad core, 3.60GHz processor and with maximum of 12 GB internal memory available. The algorithm was implemented in C++. The library for graphs that have been used is Treewidth[1]. The algorithms are implemented without the use of multi-threading and with a maximum running time of two hours, which will be represented by ∞ if it would take longer. When the program uses more memory then there is available we use the notation OoM; Out of Memory.

[1]Developed by Thomas van Dijk Jan-Pieter van den Heuvel and Wouter Slob, it can be found at: http://www.treewidth.com/

				time in seconds				
Graph	|V|	|E|	PW	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5
Alarm	37	65	4	OoM	0.275	OoM	13.39	0.275
Barley	48	126	7	OoM	2.652	OoM	OoM	0.33
Mainuk	48	198	7	OoM	0.148	OoM	OoM	0.023
Mildew	35	80	5	OoM	0.044	OoM	34.786	0.013
Myciel2	5	5	2	0.001	0.001	0.001	0.001	0.001
Myciel3	11	20	5	0.035	0.001	0.031	0.026	0.001
Myciel4	23	71	10	296	1.221	301	92.327	0.026
Myciel5	47	236	-	OoM	OoM	OoM	OoM	OoM
Queen5_5	25	320	18	∞	0.285	∞	1228	0.024
Queen6_6	36	580	25	OoM	0.612	OoM	OoM	0.05
Queen7_7	49	952	35	OoM	OoM	OoM	OoM	0.784
Queen8_8	64	1456	45	OoM	OoM	OoM	OoM	4.498
Queen9_9	81	2112	-	OoM	OoM	OoM	OoM	OoM
Raster2x4	8	10	2	0.004	0.001	0.004	0.001	0.001
Raster4x6	24	38	4	632	0.004	670	0.461	0.003
Raster6x9	54	93	6	OoM	1.003	OoM	OoM	0.097
Water	32	123	10	OoM	16.5	OoM	OoM	0.11

Table 3: Experiments for the computation of pathwidth using five algorithms

We notice a couple of things straight away. We can see that for smaller graphs, with less than 15 vertices, all the algorithms compute the minimum pathwidth under one second. So all the approaches can be applied for small graphs. When the graph has more than 26 vertices, we see that the naive approaches are running out of memory. This is because there is no optimization included for memory usage. From this point on, the incremental increase methods out preforms the naive approach as expected. The incremental increase method is however still limited to the memory usage. The more vertices a graph has the more states in the search space have to be expanded. Therefore the structure of the graph has a great influence on the computation time for the incremental increase algorithms 2 and 4.

The difference between $\overrightarrow{PW}$ and $\overleftarrow{PW}$ is showing. Algorithm 4 preforms much worse than algorithm 2. The number of states calculated by algorithm 4 is much higher than of algorithm 2. It seems that there is a lot of overhead available when the algorithm makes use of $\overleftarrow{PW}$. This is the reason why we have not implemented the hybrid algorithm.

The experiments indicate the best algorithm we have tested. This is namely the prioritized algorithm. We can clearly see that it has no problem calculating the pathwidth of vertices up to 45. It can even calculate the pathwidth of $Queen8\_8$ which has 64 vertices. The power of this algorithm is that it does not expand unnecessary states with high pathwidth value. If the algorithm expands one state equal to the minimum pathwidth, a depth first search from that point on will preformed. It reduces the overhead calculation that is preformed by the incremental increase, therefor obtaining much lower computational times. The algorithm is however vulnerable to bad graph structures.

3.5 Conclusion

We have shown that using *pathwidth*, we can efficiently construct a *path decomposition* of any given ordering Π and arbitrary graph $G(V, E)$. Further we have constructed two breath-first search algorithms and one best-first search algorithm, with complexity $\mathcal{O}^*\left(2^{|V|}\right)$ for both time and memory usage, for finding the *minimum pathwidth*.

The experiments show that the best-first search is the best algorithm for solving our problem. It is done very quickly for larger graphs, with 50 vertices. However the structure of the graphs is important for the computation time. We can guarantee that the algorithm has a lower computational time than the others.

The proposed optimization methods for improving the algorithms actually decrease the computation time. *Incremental increase* in combination with *pathwidth at the end* is a good way for reducing the memory usage and get faster results compared to the naive approach.

The main problem that is experienced, is that the algorithms are running out of memory. If we can improve the memory usage such that it is of a less complexity then $\mathcal{O}^*\left(2^{|V|}\right)$ then we could probably solve instances with more vertices. However this will most likely not be possible without increasing the complexity of the algorithms.

To even further improve the computation time we can increase the number of threads. Every algorithm is implemented with a single thread. Multi-threading will reduce the computation time and there are a lot of possibilities in the algorithms that can be multi threaded.

4 Treecost

In this section we will present the *treecost* problem. First we will describe a number of necessary definitions, followed by the problem description. Next the techniques will be presented for solving the *minimum treecost*. This is done by giving motivations why and how they will influence the complexity and running time of the algorithms. These techniques are combined into two different algorithms for calculating the minimum treecost problem. Finally we present experimental results and give our conclusions.

4.1 Treecost description

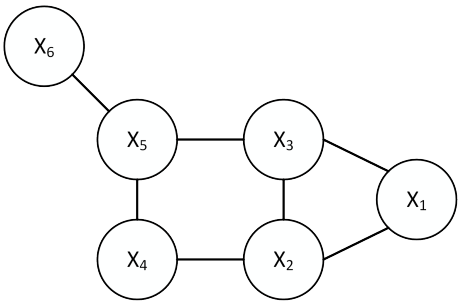

Figure 11: Example graph $G(V, E)$, with vertices $V(G) = \{x_1, x_2, x_3, x_4, x_5, x_6\}$ and edges $E(G) = \{(x_1, x_2), (x_1, x_3), (x_2, x_3), (x_2, x_4), (x_3, x_5), (x_4, x_5), (x_5, x_6)\}$

The example graph we will use in this section can be seen in Figure 11. Note that this is the same example as in Section 3.1.

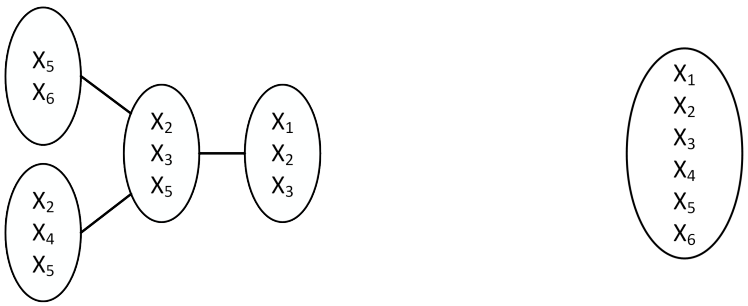

Figure 12: Two *junction trees* constructed by the graph from Figure 1

In Section 3.1 we have given the Definition 3.1 for creating a *junction tree* and finding the minimum treewidth, see Definition 3.3. We recall two examples of a junction tree, see Figure 12. The difference between computing the minimum treewidth problem and minimum treecost problem is the measurement function of the junction tree. Each node X_i in a junction tree, has a value equal to $2^{|X_i|}$. The treecost of the entire junction tree is the summation of each node instead of finding the maximum, see Definition 4.1.

From this definition we can conclude that the treecost value is between $|V| \leq treecost \leq 2^{|V|}$. The minimum value can be obtained by a graph with $|V|$ vertices and 0 edges. In this case we obtain an optimal *junction tree* with $|V|$ nodes, all with size 1. The maximum value can be obtained for a graph $G(V, E)$ where V is equal to the maximum clique. In this case the optimal junction tree has 1 node which contains all vertices. This junction tree has a treecost of $2^{|V|}$.

Definition 4.1 *The* TREECOST *of a junction tree* $T(V, E)$ *is the integer defined by*

$$\text{TREECOST} = \sum_{\forall X \in T(V)} (2^{|X|})$$

The minimum treecost of a graph $G(V, E)$ is defined as the minimum treecost over all possible junction trees with corresponding treecost. Of all the treecosts we take the minimum value to get the minimum treecost of a graph $G(V, E)$, see Definition 4.2. When we apply the definition to Figure 12 the values 28 and 64 are obtained respectively.

Definition 4.2 *The* MINIMUM TREECOST *of a graph* $G(V, E)$ *is the integer defined by*

$$TC = \min_{junction\ trees\ T(V,E)} treecost(T(V, E))$$

4.1.1 Problem description

The objective in the rest of the section is to improve upon the current algorithms for calculating the minimum treecost, given an arbitrary graph $G(V, E)$. The treecost of a graph is the sum over the treecost of its connected component. Thus, we assume that the input graph is connected. Otherwise we can run the algorithms separately on each connected component.

4.2 Theory

In this section we will introduce a method for calculating the minimum treecost for arbitrary graph $G(V, E)$. The method we introduce is based on the same idea as presented in Subsection 3.2. We can create a junction tree from an elimination ordering but this is done in a different way than the construction of a path decomposition. We will explain the reconstruction of a junction tree with the following example.

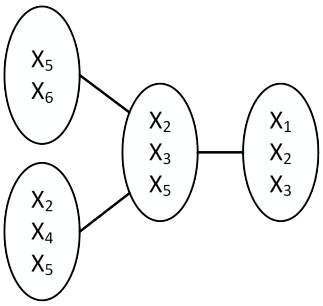

Figure 13: One *junction tree* constructed from the graph from Figure 1

Suppose we have the graph from Figure 11 and the elimination ordering $\Pi = \{x_6, x_4, x_5, x_3, x_2, x_1\}$. We can construct the junction tree, see Figure 13, in the following way. If we eliminate vertex x_6 we create a node which contains x_6 and all the neighbors of vertex x_6. The second vertex x_4 is not connected to x_6. Because we are constructing a junction tree, we do not need to include all neighbors from previously eliminated vertices. Therefor a node which contains x_4 and all the neighbors from x_4 is constructed. The first two eliminations can be observed in the following figures.

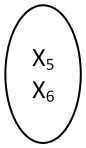

Figure 14: Elimination of vertex x_6

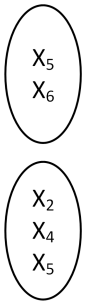

Figure 15: Elimination of vertex x_4

The next vertex to eliminate is x_5. We notice that both vertices x_4 and x_6, that we already have eliminated, are connected to x_5. Therefor we construct a node which contains all neighbors of x_4, x_5, x_6 together with x_5. The created node is connected to both the previously constructed nodes as can be seen in Figure 4.2. This is to ensure the properties of the junction tree.

When we eliminate x_3 we notice that it is connected to all previously eliminated vertices. So we construct a node containing all the neighbors of vertices x_3, x_4, x_5, x_6 and node x_3, see Figure 4.2.

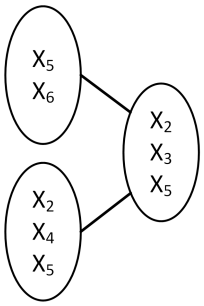

Figure 16: Elimination of vertex x_6

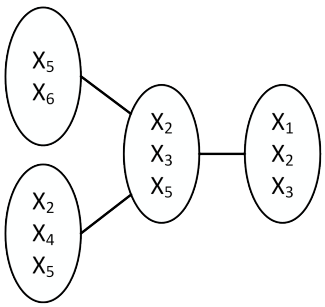

Figure 17: Elimination of vertex x_4

The last two vertices in the elimination ordering Π do not construct a new node for the junction tree. This is because the nodes constructed by eliminating vertices x_1 and x_2 are a subset of the node constructed by eliminating x_3 also known as *simplicial vertex*, see Section 4.4.3. Therefor adding these nodes will result in a different junction tree with higher treecost.

4.2.1 Minimum treecost by elimination ordering

In the previous example we have seen a reconstruction of a junction tree using an *elimination ordering*. To construct a node of vertex x in the junction tree using an elimination ordering, we need to find all the previous eliminated vertices in Π that are connected to the eliminated

vertex x. Of all these vertices we need to collect the neighbors. Definition 4.3 is used for finding the connected component of a vertex x given a set S. In Definition 4.4 we recall the neighbor function of a given set S. Given these two definitions we can construct two definitions for finding the treecost of a set S and the minimum treecost of an elimination ordering.

Definition 4.3 *The connected component of a given vertex x, set $S \subseteq V$ and graph $G(V, E)$ is denoted by*

$$R(x, S) = \{s' \mid s' \in (S \cup \{x\}),\ s' \sim_s x\}$$

Definition 4.4 *The neighbors of a given set $S \subseteq V$ and graph $G(V, E)$ is denoted by*

$$N(S) = \{x \mid x \in\ (V - S),\ \exists s\ \in S \text{ and } \exists\ e(x, s) \in E(G)\}$$

To construct a node of an eliminated vertex x and set S we will use Definition 4.5. We have the case that if a vertex is simplicial, the created node is not needed in the junction tree. Therefore we simply say that this node is the empty set. If the eliminated vertex is not simplicial, a node is created with the vertex and the neighbors of the connected component in S. This definition will be used as subroutine to calculate the treecost of an ordering.

Definition 4.5 *Let $G(V, E)$ be a graph. $TC'(x, S)$ is the function with arguments a set $S \subseteq V$ and vertex $x \notin S$.*

$$TC'(x, S) = \begin{cases} \emptyset & , \textit{vertex } x \textit{ is a simplicial vertex} \\ N(R(x, S)) \cup \{x\}) & , \textit{otherwise} \end{cases}$$

To construct the entire junction tree from an elimination ordering Π, we use TC' to construct all the individual nodes. We are however only interested in the treecost, so we apply the treecost function on each node created, see Definition 4.6.

Definition 4.6 *Let $G(V, E)$ be a graph. $TC_{ord}(\Pi)$ is the function with arguments an ordering $\Pi = \{x_1, .., x_n\}$.*

$$TC_{ord}(\Pi) = \sum_{i=1}^{n} 2^{|(TC'(x_i, \Pi_{i-1}))|}$$

To compute the minimum treecost of an arbitrary graph $G(V, E)$ we have to try all possible elimination orderings. Of all these orderings we can determine the treecost and then take the minimum . This way we compute the minimum treecost of the entire graph, see Definition 4.7.

Definition 4.7 *Let $G(V, E)$ be a graph. $TC(S)$ is the function with arguments a $S \subseteq V$.*

$$TC(S) = \min_{x \in S}\{2^{|TC'(x,S)|} + TC(S - \{x\})\}$$

We will prove that TC, Definition 4.7 with subroutine TC' Definition 4.5, will lead to a *junction tree* with minimal treecost. The correctness of this statement will be proven in two steps, see Theorem 4.8.

Theorem 4.8 *Let $G(V, E)$ be a graph, $S \subseteq V$ a set of vertices and let $k \geq 0$. Then the following statements are equivalent:*

1. $TC(S) \leq k$
2. *There exists a junction tree of S, with treecost $\leq k$*

In Lemma 4.9 we will prove that the recursion TC leads to a valid junction tree with minimum treecost. Furthermore in Lemma 4.10 the prove is given that for every elimination ordering Π, we can construct a junction tree with equal or better treecost using our recursion function TC.

Lemma 4.9 *Let $S \subseteq V$ and suppose $TC(S) \leq k$. Then there exists a junction tree of $G[S]$, with treecost $\leq k$*

Proof: *We will prove the following three conditions:*

1. *For all $x \in S$, there exists a Bag B_z such that $x \in B_z$*
2. *For all $x \in S$, the Bags B_x form a connected subset*
3. *For all edges $e(i,j) \in E(G)$, there exists a Bag B_z such that, $x_i, x_j \in B_z$*

1) Let Π be the ordering of S such that $TC(\Pi) \leq k$. We know by definition that for all $x_i \in S, x_i$ is contained in Bag_i.
2) We prove the statement by induction.
We pick $Bag_0 = \emptyset$,
For all j with $1 \leq j \leq |S|$, the following holds:

$$\begin{aligned} TC'(\{x_j\}) &= N(R(x_j, S)) \cup \{x_j\} \\ &= N((R(x_j, S) - \{x_j\}) \cup \{x_j\}) \cup \{x_j\} \\ &= N(R(x_j, S) - \{x_j\}) \cup (N(\{x_j\}) - R(x_j, S)) \cup \{x_j\} \\ &= (\bigcup_i Bag_i - \{x_i\}) \cup (N(\{xj\} - R(x_j, S)) \cup \{x_j\} \\ &\subseteq Bag_j \end{aligned}$$

Vertex x_j will not be contained in Bag_i where $i > j$. So for all $x \in S$, the Bags B_x form a connected subset.
When x_j is a contained in a earlier bag we have a trivial case, because Bag_j is contained in a Bag_k with $k < j$
3) For all i, j with $i < j$ and there exists an edge $e(i,j) \in E(G)$. Then in we have that $x_i, x_j \in Bag_i$.
We have proven that if $TC(S) \leq k$. Then there exists a junction tree of $G[S]$, with treecost $\leq k$

□

Lemma 4.10 *Let $S \subseteq V$. Suppose there exists a junction tree $G[S]$ with treecost $\leq k$. Then $TC(S) \leq k$*

Proof: *By definition of junction tree there exists an ordering Π_{forget} that has a treecost $\leq k$. We can take the ordering of Π as the ordering for $TC_{ord}(S)$. This will give us $TC_{ord}(\Pi)$. We will prove the correctness in two steps.*
1) We pick $x_1 \in \Pi$, then $N(x_1) \cup x_1 \subseteq Bag_1 \leq k$.
2) For all $j, 2 \leq j \leq |S|$, this will give us the following equations,

$$
\begin{aligned}
Bag_j &\supseteq (\bigcup_i Bag_i - \{x_i\}) \cup (N(x_j) - \bigcup_i) \cup \{x_i\}) \\
&\supseteq (\bigcup_i Bag_i - \{x_i\}) \cup (N(\{xj\} - R(x_j, S)) \cup \{x_j\} \\
&\supseteq N(R(x_j, S) - \{x_j\}) \cup (N(\{x_j\}) - R(x_j, S)) \cup \{x_j\} \\
&\supseteq N((R(x_j, S) - \{x_j\}) \cup \{x_j\}) \cup \{x_j\} \\
&\supseteq N(R(x_j, S)) \cup \{x_j\} \\
&\supseteq TC'(\{x_j\})
\end{aligned}
$$

In the case that x_j is contained in an earlier bag, $TC'(\{x_j\})$ is empty.
We have proven that if there exists a junction tree $G[S]$ with treecost $\leq k$ where $S \subseteq V$. Then there exists a $TC(S) \leq k$.

□

We have given the prove that $TC(S)$ denotes the cost of a valid junction tree that has the least treecost for any set $S \subseteq V$. If we take S equal to V we can compute the treecost of the entire graph, see Corollary 4.11.

Corollary 4.11 *Given a graph $G(V, E)$, then there exists a junction tree of V with treecost $\leq k$ if and only if $TC(V) \leq k$*

In Section 4.4 two algorithms are constructed using TC.

4.3 Minimum treecost by introduction ordering

In the computation of the minimum pathwidth we have discussed a second approach, see Section 3.2.2. It will not be able to use this approach for the minimum treecost. The main problem of the approach is that it constructs bags that are redundant. All nodes add a value to the treecost which is not desirable. We can not know by forehand if a node is used in the final junction tree or not. Therefore the approach will not be used.

4.4 Algorithm

In this section the algorithms will be discussed and explained. First we give the general approach for solving the minimum treecost problem. After that we provide additional methods to decrease the computing time of the algorithms. In the Appendix B, we give the pseudo code for the algorithms.

4.4.1 Search space

In the computation of minimum pathwidth we have defined the search space. To reduce the complexity, dynamic programming is used to obtain a complexity of $\mathcal{O}^*\left(2^{|V|}\right)$. On this search space two methods were defined to explore the space. These methods where called *tabulation* and *priority*, see Section 3.3.1. For both tabulation and priority we will construct an algorithm to see how well the methods perform on the computation of treecost.

We recall in particularly Definition 4.12. This definition is also valid for the computation of treecost. This definition will have an even bigger effect for treecost. We have that for each vertex being eliminated and not equal to the empty set the treecost will decrease.

Lemma 4.12 *Let TC be defined by Definition 4.7, $S \subseteq V$ and $x \in V - S$ then the following holds*

$$TC(S) \leq TC(S \cup \{x\})$$

4.4.2 Clique at the end

The *clique at the end* has some similarity to *pathwidth at the end*, see Section 3.3.3. The main idea is to find the largest clique in the original graph. We know for a fact that this clique is contained in a node of the optimal juncation tree, because of the conditions of such a tree. Therefore we can handle the vertices of the largest clique separately.

During our algorithm the vertices of the clique are not allowed to be eliminated. This will reduce the complexity by the size of the largest clique. After our algorithm finishes we check if the clique is contained in any of the constructed nodes of the junction tree. We do this by verifying if any of the vertices of the clique is simplicial or not. If one vertex is simplicial we add the clique otherwise we do not. This approach will result in the construction of a junction tree which gives the minimum treecost.

4.4.3 Simplicial vertex

A vertex x is *simplicial* when all the neighbors are connected, also known as vertex x is only part of one maximal clique of a graph, see Definition 4.13.

Definition 4.13 *A vertex x is simplicial if*

$(N(\{x\}) \cup \{x\})$ *is a clique*

If we detect a simplicial vertex we know that when we eliminate this vertex the treecost will not increase, see Definition 4.5. Therefore we can safely eliminate this vertex without having to expand all other children and eliminating a simplicial vertex will never increase the junction tree. This statement has been proven by Bodlaender and Koster [11]. So instead of branching all the children of a state that has a simplicial vertex that is not yet eliminated. We can only branch on this vertex to improve the computation time.

4.4.4 Implemented algorithms

In total there are three implemented algorithms. Two of which have been constructed from the theory described earlier. The last algorithm is implemented based on the descriptions of Li and Ueno [26].

4.4.4.1 Algorithm 6

The first algorithm uses dynamic programming to obtain a complexity of $\mathcal{O}^*\left(2^{|V|}\right)$. It uses the techniques tabulation with *clique at the end*. Because of the layer structure we do not

include the detection of simplicial vertex. Therefore we will branch all the children even if we detect a simplicial vertex. This is to ensure we will get the minimum treecost for each set. The algorithm can be seen in Appendix B, under Algorithm 6.

4.4.4.2 Algorithm 7

The second algorithm uses dynamic programming as well. To explore the search space the priority approach is used. During the computation, simplicial vertices are detected and will eliminated according to Section 4.4.3. This will reduce the overall running time of the algorithm. The clique at the end will be used to reduce the input size of the graph for even less computations. The algorithm can be seen in Appendix B, under Algorithm 7.

4.4.4.3 Li and Ueno

The algorithm from Li and Ueno uses a depth-first search approach with branch and bound. They have found a method for calculating all maximal cliques at run-time to obtain a lower bound for each state in the search space. The upper bound is maintained by eliminating all vertices and keeping track of the lowest treecost value. In the algorithm they include the elimination of a simplicial vertex in order to decrease the computation time. The algorithm can be found in Li and Ueno [26].

4.4.4.4 Overview

In Table 4 we have the overview of the described algorithms with the techniques included. These algorithms will be tested and the results can be found in Section 4.5.

#Alg	Method	DP	Tabulation	Priority	Clique at the end	Simplical vertex
6	*TC*	X	X	-	X	-
7	*TC*	X	-	X	X	X
8	Li and Ueno	-	-	-	-	X

Table 4: Implemented algorithms for finding the *minimum treecost*.

4.5 Experiments

We will present the experiment obtained by running algorithms 6 and 7. Furthermore we have implemented the algorithm from Li and Ueno. The results can be found in Table 5. It is done on a Windows 7 ultimate OS with Intel i7 970 quad core, 3.60GHz processor and with maximum of 12 GB internal memory available. The algorithm was implemented in Java with VM arguments -Xmx9g -Xss9g -Xms9g[2]. The library for graphs that have been used is

[2]Both the maximum and minimum heap size are set to 9gb

Treewidth[3]. The algorithms are implemented without the use of multi-threading and with a maximum running time of two hours, which will be represented as ∞ if it would take longer. When the program uses more memory then there is available we use the notation *OoM*; Out of Memory.

				time in seconds		
Graph	|V|	|E|	Treecost	Alg 6	Alg 7	Alg 8
Alarm	37	65	264	421.949	OoM	1.718
Barley	47	126	-	∞	∞	∞
BN_29	24	49	192	0.440	0.710	0.046
mildew	35	80	524	326	OoM	339
myciel2	5	5	24	0.001	0.001	0.003
myciel3	11	20	144	0.040	0.020	0.042
myciel4	23	71	4704	3.821	0.736	84
funguik	15	36	144	0.026	0.014	0.013
oesoca	39	67	284	∞	OoM	0.362
oow_bas	27	54	352	9.333	12.618	18.611
oow_solo	40	87	-	∞	∞	∞
oow-trad	33	72	808	893	OoM	∞
queen5_5	25	320	34873344	0.310	0.090	∞
queen6_6	36	580	2529230848	12.506	0.166	∞
raster2x4	8	10	48	0.010	0.001	0.017
raster4x6	24	38	448	30.336	23.157	1516
ship_ship	50	114	-	∞	OoM	∞
water	32	123	7360	∞	∞	879
wilsonhugin	21	27	108	0.740	0.926	0.040
weeduk	15	49	440	0.007	0.005	0.031

Table 5: Experiments for the computation of treecost using three algorithms

The first thing to notice is that the difference between our proposed algorithm and of Li and Ueno [26] is not straight forward. We notice that some instances do get solved very quickly by Li and Ueno; these are the larger graphs with more vertices. However the instances which have high treecost do take much longer in comparison to our proposed algorithm as we can see in

[3]Developed by Thomas van Dijk Jan-Pieter van den Heuvel and Wouter Slob, it can be found at: http://www.treewidth.com/

the example *queen5_5, queen6_6*. This has everything to do with the time it takes to calculate a large clique in comparison to finding all maximal cliques of smaller size. The advantage of choosing the algorithm of Li and Ueno is that this method can solve larger instancesand is more dependant on the graph structure, where our algorithms are more limited on the instance size.

The difference between the methods priority and tabulation is shown in the results. For larger graph instances we can see that the minimum treecost is solved relatively quickly. However, the problem of the priority method is that it is running out of memory much faster than the tabulation method. This can be explained, because all the instances that have been calculated but not expanded are still in the cache memory. So the memory usage is higher than of the tabulation method. The priority method addresses multiple layers at the same time which can result in more memory usage than of the tabulation method.

Our proposed tabulation algorithm has a guaranteed complexity of $\mathcal{O}^*\left(2^{|V|-\omega(G)}\right)$, with $\omega(G)$ the size of the maximum clique in G. Meaning it will calculate all search tree states. Therefore when $|V|-\omega(G) > 30$ solving the treecost with the tabulation algorithm becomes unfeasible. The priority method however has a worst case complexity of $\mathcal{O}^*\left(2^{|V|-\omega(G)}\right)$, with $\omega(G)$ the size of the maximum clique in G. So this algorithm depends on the structure of the graph and is limited to $|V|-\omega(G) > 30$, not by running time but by memory capacity.

4.6 Conclusion

We have shown that using the *treecost* we can efficiently create a *triangulation* of any given elimination ordering Π and arbitrary graph G. We have successfully constructed two breath-first search algorithms for finding the minimum *junction tree* regarding the treecost. Because of a technique used in treewidth by Bodlaender and Fomin [9], we can reduce the search space from $|V|-1$ to, $|V|-\omega(G)$, with $\omega(G)$ the size of the maximum clique in G.

The proposed *priority* and *tabulation* algorithms have a disadvantage that the memory usage is of complexity $\mathcal{O}^*\left(2^{|V|}\right)$. Therefore calculating the junction tree regarding the treecost will not be feasible for large graphs. But the complexity of both algorithms is $\mathcal{O}^*\left(2^{|V|-\omega(G)}\right)$, with $\omega(G)$ the size of the maximum clique in G.

The experiments show that smaller instances are solved significantly faster then the algorithm proposed by Li and Ueno [26]. Therefore the proposed algorithms can be used for subroutines in existing heuristics to improve the lower bound calculation.

The proposed priority and tabulation algorithms can still be improved. There are a lot of possibilities for multi-threading. Especially in the tabulation algorithm we can run multiple instances for each table layer concurrent, which will reduce the computational time drasticly.

A different problem of the breath-first search technique is that the memory needed for computing the junction tree is of order $\mathcal{O}^*\left(2^{|V|}\right)$. So computing junction trees for large graphs is not possible with this approach yet. A possibility is to find a better data structure that will reduce the memory usage of each state in the search space.

5 Conclusion

In this report we have successfully introduced two methods for computing the *minimum pathwidth*. We have shown that we can construct a *path decomposition* based on an ordering using two different methods, namely $\overrightarrow{PW}$ and $\overleftarrow{PW}$. From these methods we constructed efficient algorithms for the computation of the minimum pathwidth for arbitrary graphs G. These algorithms all have a computational time and memory usage of $\mathcal{O}^*\left(2^{|V|}\right)$. The best algorithm we constructed uses a combination of breath first search and depth first search. The algorithm is able to compute the minimum pathwidth of graphs with up to 45 vertices *within a second*. It is not practical to compute larger instances due to the exponential increase of the memory usage. Further we have shown that different techniques are useful for reducing the total computation time.

We introduced a method TC for the construction of a *junction tree* for arbitrary graphs G. Using this method we created two algorithms for the computation of the *minimum treecost*. We have introduced the method *clique at the end* to reduce the number of vertices from $|V|$ to $|V| - \omega(G)$, with $\omega(G)$ the size of the maximum clique in G. The proposed algorithms therefor have a running time complexity of $\mathcal{O}^*\left(2^{|V|-\omega(G)}\right)$. However the memory usage is of complexity $\mathcal{O}^*\left(2^{|V|}\right)$. The experiments show that small instances of vertices up to 25 are solved very quickly. However, for larger instances the memory usage becomes the bottleneck.

All the proposed algorithms are limited to memory usage. To compute either the *minimum pathwidth* or *minimum treecost* of larger arbitrary graphs G either more memory should be used or a better technique should be constructed. A possibility to reduce the total computation time of the algorithms is multi-threading, since every algorithm is implemented with a single thread. Much computational improvement can be obtained with the use of multi-threading.

References

[1] S. Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability – A survey. *BIT*, 25:2–23, 1985.

[2] S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12:308–340, 1991.

[3] S. Arnborg and A. Proskurowski. Linear time algorithms for np-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, 23(1):11–24, 1989.

[4] R. Beigel and D. Eppstein. 3-coloring in time $o(2^{1.3289*n})$. *Journal of Algorithms*, 54(2):168–204, 2005.

[5] H. L. Bodlaender. Dynamic programming algorithms on graphs with bounded tree-width. In T. Lepistö and A. Salomaa, editors, *Proceedings of the 15th International Colloquium on Automata, Languages and Programming, ICALP'88*, volume 317 of *Lecture Notes in Computer Science*, pages 105–119. Springer Verlag, 1988.

[6] H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–23, 1993.

[7] H. L. Bodlaender. Treewidth: Algorithmic techniques and results. In I. Privara and P. Ruzicka, editors, *Proceedings of the 22nd International Symposium on Mathematical Foundations of Computer Science, MFCS'97*, pages 19–36. Springer Verlag, Lecture Notes in Computer Science, vol. 1295, 1997.

[8] H. L. Bodlaender and F. V. Fomin. Tree decompositions with small cost. *Discrete Applied Mathematics*, 145:143–154, 2004.

[9] H. L. Bodlaender, F. V. Fomin, A. M. C. A. Koster, D. Kratsch, and D. M. Thilikos. On exact algorithms for treewidth. *ACM Transactions on Algorithms*, 9(1), 2012.

[10] H. L. Bodlaender and T. Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *Journal of Algorithms*, 21:358–402, 1996.

[11] H. L. Bodlaender, A. M. C. A. Koster, F. v. d. Eijkhof, and L. C. van der Gaag. Preprocessing for triangulation of probabilistic networks. In J. S. Breese and D. Koller, editors, *Proceedings of the 17th Annual Conference on Uncertainty in Artificial Intelligence, UAI 2001*, pages 32–39. Morgan Kaufmann, 2001.

[12] J. A. Carlson, A. Jaffe, and A. Wiles. *The millennium prize problems*. American Mathematical Soc., 2006.

[13] J. Chen, I. A. Kanj, and G. Xia. Improved parameterized upper bounds for vertex cover. In *Mathematical Foundations of Computer Science 2006*, pages 238–249. Springer, 2006.

[14] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual Symposium on Theory of Computing, STOC'71*, pages 151–158, New York, 1971. ACM.

[15] B. Courcelle. The monadic second-order logic of graphs I: Recognizable sets of finite graphs. *Information and Computation*, 85:12–75, 1990.

[16] B. Courcelle and M. Mosbah. Monadic second-order evaluations on tree-decomposable graphs. *Theoretical Computer Science*, 109:49–82, 1993.

[17] A. Darwiche. *Modeling and Reasoning with Bayesian networks*. Cambridge Universal Press, 2009.

[18] R. G. Downey and M. R. Fellows. *Parameterized Complexity*, volume 3. springer Heidelberg, 1999.

[19] M. J. Flores and J. A. Gámez. A review on distinct methods and approaches to perform triangulation for Bayesian networks. In P. Lucas, J. A. Gámez, and A. Salmerón, editors, *Advances in Probabilistic Graphical Models*, volume 214 of *Studies in Fuzziness and Soft Computing*, pages 127–152. Springer Verlag, 2007.

[20] F. V. Fomin, F. Grandoni, and D. Kratsch. Measure and conquer: a simple o($2^{0.288*n}$) independent set algorithm. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 18–25. ACM, 2006.

[21] F. V. Fomin and D. M. Thilikos. New upper bounds on the decomposability of planar graphs and fixed paramater algorithms. Technical Report LSI-02-56-R, Universitat Politècnica de Catalunya, Spain, 2002.

[22] F. V. Fomin and Y. Villanger. Finding induced subgraphs via minimal triangulations. In J.-Y. Marion and T. Schwentick, editors, *Proceedings 27th International Symposium on Theoretical Aspects of Computer Science, STACS 2010*, volume 5 of *Dagstuhl Seminar Proceedings*, pages 383–394, Schloss Dagstuhl, Germany, 2010. Leibniz-Zentrum für Informatik.

[23] H. Kim. Finding a maximum independent set in a permutation graph. *Information Processing Letters*, 36:19–24, 1990.

[24] U. Kjærulff. Triangulation of graphs — algorithms giving small total state space. Research Report R-90-09, Dept. of Mathematics and Computer Science, Aalborg University, 1990.

[25] U. Kjærulff. Optimal decomposition of probabilistic networks by simulated annealing. *Statistics and Computing*, 2:2–17, 1992.

[26] C. Li and M. Ueno. A depth-first-search algorithm for optimal triangulation of Bayesian network. In *Proceedings of the 6th European Workshop on Probabilistic Graphical Networks, PGM 2012*, pages 187–194, 2012.

[27] T. J. Ottosen and J. Vomlel. All roads lead to Rome — New search methods for the optimal triangulation problem. *International Journal of Approximate Reasoning*, 53(9):1350–1366, 2012.

[28] N. Robertson and P. D. Seymour. Graph minors. I. Excluding a forest. *Journal of Combinatorial Theory, Series B*, 35:39–61, 1983.

[29] N. Robertson and P. D. Seymour. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36:49–64, 1984.

[30] N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7:309–322, 1986.

[31] J. M. M. van Rooij, J. Nederlof, and T. C. van Dijk. Inclusion/exclusion meets measure and conquer: Exact algorithms for counting dominating sets. In A. Fiat and P. Sanders, editors, *Proceedings of the 17th Annual European Symposium on Algorithms, ESA 2009*, pages 554–565. Springer Verlag, Lecture Notes in Computer Science, vol. 5757, 2009.

[32] W. X. Wen. Optimal decomposition of belief networks. In P. P. Bonissone, M. Henrion, L. N. Kanal, and J. F. Lemmer, editors, *Proceedings of the 6th Annual Conference on Uncertainty in Artificial Intelligence, UAI'90*, pages 245–256. Elsevier, 1990.

Appendix A Algorithms

				time in seconds				
Graph	|V|	|E|	PW	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5
Fungiuk	15	36	4	0.45	0.005	0.455	0.032	0.001
Oesoca	39	67	4	OoM	1.6	OoM	31.538	0.267
Oow_bas	27	54	4	OoM	0.011	OoM	1.031	0.003
Oow_solo	40	87	6	OoM	0.255	OoM	670.16	0.034
Oow_trad	33	72	6	OoM	0.065	OoM	450.65	0.022
Ship_ship	50	114	9	OoM	OoM	OoM	OoM	16.07
Weeduk	15	49	7	0.444	0.014	0.476	0.021	0.003
Wilson-hugin	21	27	3	56.482	0.052	59.618	0.674	0.009

Table 6: Experimental resuls for the computation of pathwidth using the five algorithms described in Section 3.3.5

Appendix B Algorithms

Algorithm 1 Naive tabulation $\overrightarrow{PW}$ DP

Input: A graph G = (V,E)
Output: The minimum pathwidth of G

PW_0 will contain $(\varnothing, 0)$.
for $i = 1$ **to** n **do**
  for all $S \in PW_{i-1}$ **do**
    for all vertex $x \in V \setminus S$ **do**
      Width = $max\{|N(S \cup \{x\}) \cup \{x\}| - 1, PW_{i-1}(S)\}$
      if $PW_i(S \cup \{x\}) \geq Width$ **then**
        Insert $((S \cup \{x\}), Width)$ in PW_i
      end if
    end for
  end for
end for
return PW_n

Algorithm 2 Incremental increase tabulation $\overrightarrow{PW}$ DP

Input: A graph G = (V,E)
Output: The minimum pathwidth of G

PW_0 will contain $(\varnothing)$.
for $t = 1$ **to** n **do**
  for $i = 1$ **to** $n - t$ **do**
    for all $S \in PW_{i-1}$ **do**
      for all vertex $x \in V \setminus S$ **do**
        if $|N(S \cup \{x\}) \cup \{x\}| - 1 < t$ **then**
          Insert $(S \cup \{x\})$ in TW_i
        end if
      end for
    end for
  end for
  if TW_n contains an element **then**
    return t
  end if
end for

Algorithm 3 Naive tabulation $\overleftarrow{PW}$ DP

Input: A graph G = (V,E)
Output: The minimum pathwidth of G

1: PW_0 will contain $(\varnothing, 0)$.
2: **for** $i = 1$ **to** n **do**
3: **for all** $S \in PW_{i-1}$ **do**
4: **for all** vertex $x \in V \setminus S$ **do**
5: Width $= max\{|\{s|s \in S, \exists z \in (V - S)\ and\ \exists e(s, z) \in G(E)\} \cup \{x\}| - 1, PW_{i-1}(S)\}$
6: **if** $PW_i(S \cup \{x\}) \geq Width$ **then**
7: Insert $((S \cup \{x\}), Width)$ in PW_i
8: **end if**
9: **end for**
10: **end for**
11: **end for**
12: **return** PW_n

Algorithm 4 Incremental increase tabulation $\overleftarrow{PW}$ DP

Input: A graph G = (V,E)
Output: The minimum pathwidth of G

PW_0 will contain $(\varnothing)$.
for $t = 1$ **to** n **do**
  for $i = 1$ **to** $n - t$ **do**
    for all $S \in PW_{i-1}$ **do**
      for all vertex $x \in V \setminus S$ **do**
        if $|\{s|s \in S, \exists z \in (V - S)\ and\ \exists e(s, z) \in G(E)\} \cup \{x\}| - 1 < t$ **then**
          Insert $(S \cup \{x\})$ in TW_i
        end if
      end for
    end for
  end for
  if TW_n contains an element **then**
    return t
  end if
end for

Algorithm 5 Priority $\overrightarrow{PW}$ DP

Input: A graph G = (V,E)
Output: The minimum pathwidth of G

1: PW will contain $(\varnothing, 0)$.
2: **while** PW is not empty **do**
3: $State = PW$ first element
4: **if** $State.S = V$ **then**
5: **return** $State.pathwidth$
6: **end if**
7: **for all** vertex $x \in V \setminus State.S$ **do**
8: Width $= max\{|N(State.S \cup \{x\}) \cup \{x\}| - 1, State.pathwidth\}$
9: Insert $((S \cup \{x\}), Width)$ in PW
10: **end for**
11: **end while**

Algorithm 6 Algorithm TC Tabulation (Graph G = (V,E))

TC_0 will contain $(\varnothing, 0)$.
for $i = 1$ **to** n **do**
  for all $(S, cost) \in TC_{i-1}$ **do**
    for all $x \in V - S$ **do**
      if x is contained in a earlier bag **then**
        $TC' = \min(cost, TC_i(S \cup \{x\}))$
      else
        $TC' = \min(2^{N|R(x,S)|} + cost, TC_i(S \cup \{x\}))$
      end if
      $TreeCost = \min(TC_i(S \cup \{x\}), TC')$
      Insert $(S \cup \{x\}, TC'$ in $TC_i)$
    end for
  end for
end for
if TC_n contains an element **then**
  return TC_n
end if

Algorithm 7 Algorithm TC Priorityqueue (Graph G = (V,E))

TC will contain $(\varnothing, 0)$.
while TC != empty **do**
  $(S, cost) = dequeueTC$
  if $S = V$ **then**
    return *cost*
  end if
  S first time addressed
  for all $x \in V - S$ **do**
    if x is contained in a earlier bag **then**
      $TreeCost = \min(cost, TC_i(S \cup \{x\}))$
    else
      $TreeCost = \min(2^{N|R(x,S)|} + cost, TC_i(S \cup \{x\}))$
    end if
    Insert $(S \cup \{x\}, TreeCost$ in $TC)$
  end for
end while