Utrecht University
Institute of Information and Computer Science

Treecost-based Preprocessing for Probabilistic Networks

Master Thesis

Myrna van de Burgwal

Supervisors:
Hans Bodlaender
Silja Renooij

Utrecht, March 2015

Preface

This thesis document is part of my graduation project for my master's degree in Technical Artificial Intelligence. I started on this project in March 2014, with a break of two months when I was co-teaching the master course Multi-agent Learning. Motivated by the courses Algorithms & Networks and Probabilistic Reasoning, I chose to combine the topics of graph theory and probability theory within my thesis project.

Then, when considering acknowledgements, I first want to extend my gratitude to Hans Bodlaender who supported me throughout this project by means of weekly (*Treewidth club*) meetings. He helped me by explaining a lot about treecost and having discussions about possible reduction rules. I also want to thank Silja Renooij who helped me to apply graph theory to probabilistic networks and who also gave good advice for the structure of my thesis. Furthermore, I want to thank Luuk van der Graaff and, especially, Ruvar Spauwen for sharing their ideas and helping me with the programming part of the project.

Abstract

Probabilistic inference is an important problem in probability theory and concerns the process of computing the probability distribution of variables, given the evidence of other variables. An often used algorithm for this problem is the junction-tree propagation algorithm. To minimise the time to process all the probabilities by means of the algorithm, an optimal tree decomposition is required. A good parameter that measures the optimality of a tree decomposition is treecost. Unfortunately, computing the tree decomposition with minimal treecost is NP-hard Nevertheless, it can be simplified by applying reduction rules that shrink the graph. These rules are familiar from treewidth, which is a well-studied parameter compared to treecost. In this thesis, a set of reduction rules are introduced and proven to be valid. This set includes rules that remove vertices and separate the graph by its components. Thereafter, experiments are conducted on input graphs, which are real-life probabilistic networks. Results of the experiment show that the graphs are reduced significantly, due to these reduction rules. This, in turn, decreases the time to solve probabilistic inference. Prior knowledge about graph theory and complexity theory is assumed for this thesis.

Contents

Chapter 1

Introduction

This thesis aims to combine the research fields of graph theory and probability theory by applying techniques from the first to improve the construction of certain models within the latter, namely *probabilistic networks* [22, 31]. A probabilistic network, or the equivalent *Bayesian network*, is a model of a probability distribution on a set of statistical variables. The (in)dependencies among these variables can be captured in a directed acyclic graph (DAG), by creating vertices that represent the variables and directed arcs that correspond to the relationships among them. An example of a probabilistic network is shown in Figure 1.1, borrowed from [34], originally by [18].

An important problem in the field of probabilistic reasoning is *probabilistic inference* [17, 38], which concerns the process of computing of the probability distribution over one variable given a value-assignment of zero or more variables after previous observation. The most efficient algorithm currently available for solving this problem is named the *junction-tree propagation algorithm* [23]. Before performing this algorithm, the DAG needs to be converted into an undirected graph by applying moralisation, which entails that each pair of vertices that share a common successor is made adjacent, whereupon all directions of the graph are being removed. An example of this procedure is given in Figure 1.2. Thereafter, it creates a triangulation of the graph by adding edges such that each uninterrupted cycle is of length at most 3. From this triangulation, a *tree decomposition* [30], or *junction tree*, can easily be built, which is a mapping of a graph into a tree that contains bags (cliques) of vertices. To process the evidence, the bags send each other information about the joint probability distribution and the entered evidence through the edges. Each bag uses the information received from its neighbours and its local marginal distribution to compute the marginal probability distribution on its variables. There is one bag that is designated as the root of the tree. All messages are passed from the leaves to this root, which is called *inward propagation*. Subsequently, the root sends these messages back to the leaves, called (*outward propagation*).

The Metastatic cancer case represented by a probabilistic network

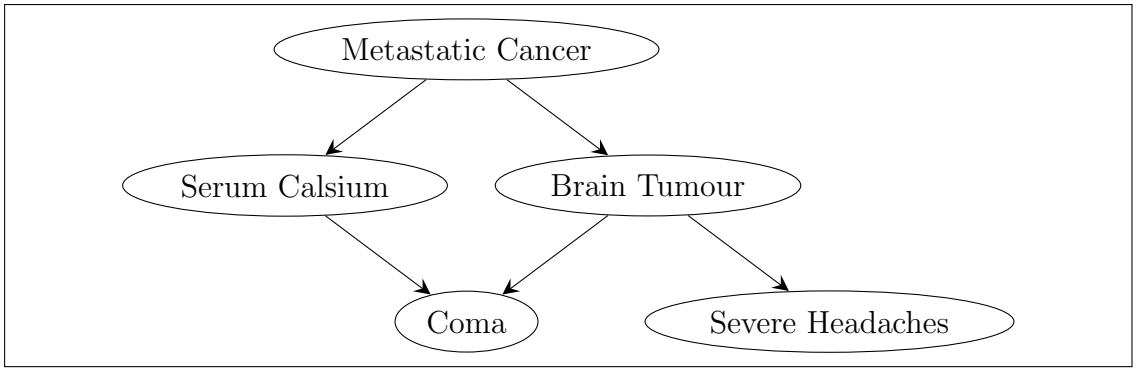

Figure 1.1: "Metastatic cancer is a possible cause of a brain tumour and is also an explanation for increased serum calcium. In turn, either of these could explain a patient falling into a coma. Sever headache is also possibly associated with a brain tumour."

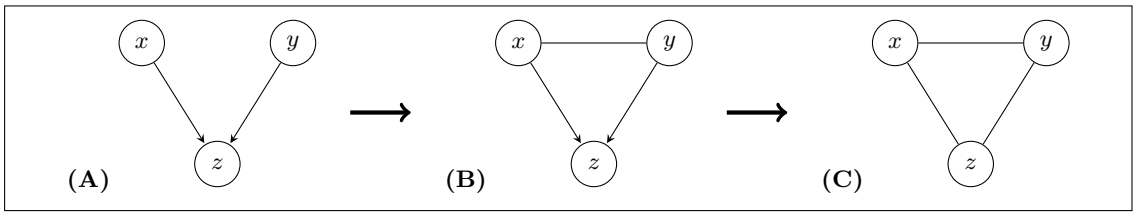

*Figure 1.2: Moralisation of a directed graph: (***A***) represents original graph where x and y share common successor z, in (***B***) an edge is placed between x and y, and (***C***) displays the final graph where all directions are dropped.*

The running time of the junction-tree propagation algorithm depends on the tree decomposition (and thus the triangulation) of a moralised graph. Therefore, it is important to take an optimal tree decomposition. An often used parameter that designates the optimality of a tree decomposition is the well-studied notion of *treewidth* [7, 9]. The treewidth of a graph equals the size of the largest bag of the tree decomposition minus one (or the maximum clique size in the graph). However, *treecost* [10], gives a more representative indication of the tree decomposition. Treecost differs from treewidth in its measurement function that takes into account the sizes of all bags. It takes the summation of the costs of all bags, where each bag has a cost equal to $2^{|bag|}$. Within the field of graph theory, treecost is a relatively novel and still little discussed graph parameter, despite its usefulness for obtaining an indication of the running time of certain graph algorithms [10].

1.1 Problem statement

Unfortunately, constructing a tree decomposition with minimal treecost (as well as minimal treewidth) of a given graph is NP-hard [24]. For the computation of treewidth successful efforts with respect to decreasing the complexity have been made, by means of preprocessing techniques [6, 8, 11, 12, 13, 21, 30]. Methods that improve the computation of treecost exist as well, although these mainly include heuristics or non-polynomial algorithms (e.g. [24, 25, 32, 40]). However, to our knowledge, no polynomial reduction rules for treecost are yet documented.

Based on the arguments above, the formal problem statement of this thesis is as follows.

There are no exact polynomial preprocessing techniques that decrease the complexity of treecost computation

1.2 Research questions

Based on the problem statement, the main purpose of this thesis is to look for exact methods to reduce the size of an input graph (e.g. a representation of a probabilistic network), to be able to perform the treecost computation on a smaller graph than the original one. Subsequently, the following main research question has been formulated:

What exact preprocessing techniques can be used best to decrease the complexity of treecost?

To answer the main research question, several steps need to be taken. These steps correspond to the following sub-questions:

SQ.1 *Which preprocessing techniques for treewidth should be considered for treecost?*

By retrieving literature about preprocessing for treewidth, we will investigate which techniques are likely to be applicable to treecost as well. Several reduction rules for treewidth and even

algorithms that create a *kernel* [21] were defined previously. Since treecost is similar to treewidth, it is possible that some of these rules and algorithms can be applied to the computation of treecost.

SQ.2 *How do our provided techniques compare to each-other based on theoretical proves and the experiment?*

We will analyse a selection (and combinations) of the candidate techniques to check whether they apply to treecost. Thereafter, experiments are performed on the techniques that are proven to be valid.

SQ.3 *What are the effects of the preprocessing algorithm on input graphs, especially graphs that represent probabilistic networks?*

Whereas preprocessing of graphs for treecost can be applied to many practical problems, this thesis focusses mainly on networks for probabilistic inference. Therefore, we will investigate the influence of the developed preprocessing techniques on probabilistic networks. In addition, we will also investigate influence on other types of networks to test the external validity of the research results.

1.3 Research objective

Based on the previous question above, we determined the following research objectives. The overall purpose of these experiments is to investigate to what extend the sizes of the input graphs will be reduced in practice, while the treecost of the graph remains unchanged. Among these techniques are mainly reduction rules that eliminate vertices and edges from the graph and separate the graph into several components. Meanwhile, a tree decomposition with minimal treecost is constructed. As was stated before, literature will be consulted to find reduction rules for the computation treecost. Another approach that will be pursued is to find rules by further investigating how graphs can be reduced without causing an increase of the treecost.

Similar to the existence of a weighted variant of treewidth [1, 41], there exists a parameter that defines the *weighted treecost*. The time to compute the probability distribution of a bag is related to the product of the variables that are included. This assumes that all variables contain two values, whereas often the number of values per variable is larger. This number is represented by the weight of a vertex. The exact time it takes to process one bag corresponds to the product of weights of the vertices in that bag. Therefore, weighted treecost gives an even better impression of the complexity of a tree decomposition. However, the difficulty of taking weights into account is that many reduction rules that can be used for treecost are no longer valid. Hence, the experiments are performed with reduction rules for "regular" treecost.

Previous to this preprocessing, the directed input graphs are moralised. This ensures that a vertex with many predecessors may induce large cliques in the graph, which is profitable for the reduction rules. We argue that by means of the rules that will be introduced in this thesis, the complexity of computing the minimal treecost will be reduced. This implies that probabilistic inference can be realised more quickly.

1.4 Thesis outline

The thesis is organised as follows. Section 2 introduces a set of basic definitions about graphs theory. Lemmata concerning tree decompositions are introduced and proven. Section 3 presents the reduction rules that were found. These rules include, amongst others, simplicial vertices and safe separators. The structure of the preprocessing algorithm is described in Section 4, together with details about the experiment and a report on the results. Finally, the thesis ends with a conclusion and ideas for further research in Section 5.

Chapter 2

Definitions and Preliminaries

The following notations are used in this thesis. A probabilistic network is a pair $\{G, P\}$ where P is a probability distribution over a set of statistical variables and $G = \{V, A\}$ is a directed, acyclic graph, where each vertex represents a statistical variable and the arcs represent the interdependencies. Since our reduction rules (as well as probabilistic inference) can only be applied to (moralised) undirected graphs, in this thesis the notation of a graph G will always refer to an undirected graph if not stated otherwise. An undirected graph $G = (V, E)$ is a pair with vertices V and (undirected) edges E. We denote the number of vertices $|V|$ by n and the number of edges $|E|$ by m. In this thesis, all graphs are assumed to be simple, which entails that the graph contain no loops or multiple edges.

Another notation for a graph is $G[V]$, which represents a graph G containing a set of vertices V. $V \setminus \{v\}$ corresponds to the set V minus vertex v. $G[V \setminus \{v\}]$ denotes graph G that contains the vertex set V minus v and its incident edges. We denote the vertex set V together with an added vertex v by $V \cup \{v\}$. The addition of edge (x, y) to graph G is denoted by $G + (x, y)$, whereas $G - (x, y)$ indicates the removal of (x, y) from G. Similarly, adding a set of vertices S to V is indicated by $V \cup S$ and removing S from V by $V \setminus S$.

A *neighbour* of v is a vertex w such that $(v, w) \in E$. This pair of vertices are said to be *adjacent*. All neighbours of v together form the *neighbourhood* of v. The *closed neighbourhood* $N[v]$ of v includes v itself, whereas the *open neighbourhood* $N(v)$ of v does not. Note that $N[v] = N(v) \cup \{v\}$. The *degree* of vertex v equals the number of neighbours it contains, and is denoted by $deg(v)$. If $deg(v) = 0$, then v is termed *isolated*. The maximum degree of graph G is denoted by $\Delta(G)$. We say that x is a *common neighbour* of vertices v and w if $(v, x) \in E$ and $(w, x) \in E$.

A *clique* is a set of vertices such that all pair are adjacent, i.e. for each pair $\{v, w\}$ it holds that $(v, w) \in E$. If all vertices in graph G are pairwise adjacent, the graph is said to be *complete*. Note that if G is complete, the entire graph forms a clique. A connected component is a set of connected vertices, i.e. each pair of vertices $\{v, w\}$ is reachable through a path in the graph. In this thesis, we abbreviate a connected component by CC. A separator $S \subset V$ is a set of vertices such that there is a pair of non-adjacent vertices $\{v, w\}$ that are separated into distinct connected components by $G \setminus S$.

Definition 2.1. *A* **minor** G' *of a graph* G *can be obtained by the removal of one or more vertices, the removal of one or more edges, or the* **contraction** *of one or more edges. The latter removes one edge* (v, w) *from the graph and merges the two incident vertices* v *and* w *into a new vertex* x*. The neighbourhood* x *consists of all previous neighbours of* v *and* w*.*

2.1 Tree decompositions and treecost

The treecost problem is the optimisation problem of finding a tree decomposition with minimal treecost of a given graph. It can be formally defined as follows:

TREECOST
Input: Graph G
Output: Smallest number k such that G has a tree decomposition with a treecost of size k.

A tree decomposition is a decomposed tree of a graph G with the following characteristics.

Definition 2.2. *A* **tree decomposition** *of a graph $G = (V, E)$ is a pair $(\{X_i | i \in I\}, T = (I, F))$, where for each i $X_i \subseteq V$ is a bag of vertices and T is a tree with a set of vertices I (s.t. each vertex represents a bag) and a set of edges F, such that:*

- $\bigcup_{i \in I} X_i = V$,
- $\forall \{u, w\} \in E, \exists_i \in I : u, w \in X_i$, *and*
- $\forall v \in V$ *the set* $I_v = \{i \in I | v \in X_i\}$ *forms a connected subtree of* T.

In other words, a valid tree decomposition satisfies the following. For a tree decomposition $TD(G)$ of G it holds that all vertices of G are included in at least one bag. Moreover, each adjacent pair of vertices are contained together in a bag. And finally, all bags that contain a vertex v are connected.

The treecost of a graph G, which is denoted by $TC(G)$, is computed by means of a tree decomposition in the following way:

Definition 2.3. *The* **treecost** *of a tree decomposition equals $\sum_{i \in I} 2^{|X_i|}$.*

Definition 2.4. *The* **weighted treecost** *of a tree decomposition equals $\sum_{i \in I} \prod_{v \in X_i} \tau(v)$, where $\tau(v)$ denotes the number of values of variable v.*

An example of a graph G together with a tree decomposition of G can be found in Figure 2.1.

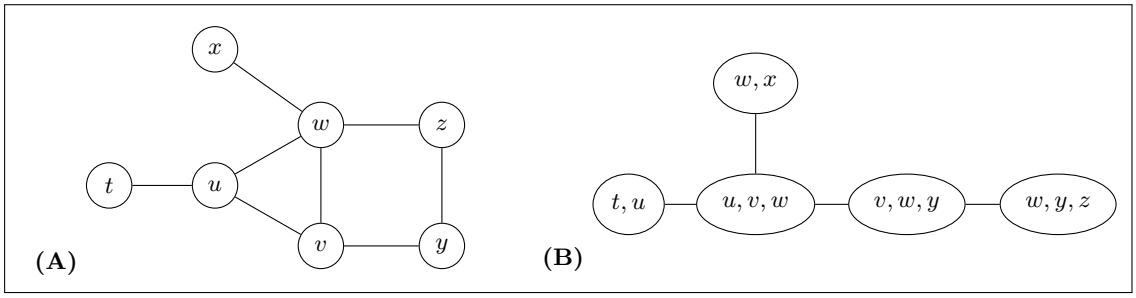

*Figure 2.1: Graph (***A***) and its tree decomposition (***B***) with (unweighted) treecost of (*2^2+2^3+2^2+2^3+2^2 =) 48.

A tree decomposition can also be characterised in terms of an *elimination ordering* [36, 37]. This is a permutation of the vertices of a graph, denoted by $\pi : V \to [n]$ (with $n = |E|$). During the constructing of such an ordering, all vertices are eliminated according to the permutation. When a vertex is removed, all its neighbours are turned into a clique. The added edges during this process are called *fill edges*. An elimination ordering that requires no fill edges, is named a *perfect elimination ordering*.

Definition 2.5. *A* **perfect elimination ordering** *of a graph G is an ordering of its vertices $v_1, ...v_n$ such that for each vertex all its higher numbered neighbours form a clique (i.e. if $(v_i, v_j) \in E$ and $(v_i, v_k) \in E$ and $i < j$ and $i < k$, then $(v_j, v_k) \in E$.*

Another way to define if a graph G perfect elimination ordering is to verify of G is *chordal*. A graph is called *chordal* if all cycles of size at least four contain a *chord*, which is an edge between two non-successive vertices in a cycle. If a graph is chordal, the graph is said to be *triangulated*.

Definition 2.6. *A* **triangulation** *H of a graph G is a triangulated supergraph of G. A* **minimal triangulation** *H of G is a triangulation such that there is G contains no triangulation that is a proper subgraph of H. Furthermore, A minimum triangulation H of G is a triangulation such that there is no triangulation of G with a lower treecost.*

2.2 Lemmata

Lemma 2.7. *Let G' be a minor of G, then the $TC(G') \leq TC(G)$.*

Proof. It is obvious that the removal of a vertex v can not increase the treecost, since a tree decomposition of the original graph G can be transformed by eliminating v from each bag X_i such that $v \in X_i$. The same holds for the removal of an edge, e.g. $G' = G - (v, w)$, where each tree decomposition of G is still valid for G'. A more complicated situation occurs when an edge contraction takes place. Let G be a graph such that $v, w \in G$ and $(v, w) \in E$ and let G' be equal to G, only with a contraction of (v, w) such that v is eliminated and w is made adjacent to all neighbours of v. Let TD be an optimal tree decomposition for G. An optimal tree decomposition TD' for G can be obtained by replacing v in each bag X_i such that $v \in X_i$ by w. Since TD' is both valid and optimal for G', we have that $TC(G') \leq TC(G)$ if G' is caused by an edge contraction of G. □

Lemma 2.8. *The upper bound on the treecost of a graph G with n vertices, equals 2^n.*

Proof. The tree decomposition that induces the greatest cost, consists of only one bag which contains all vertices. Therefore, the treecost of this tree decomposition is 2^n, with n the number of vertices. An example of a structure of a graph that would have such a large treecost is a complete graph. □

Lemma 2.9. *The lower bound on the treecost of a graph G with n vertices, equals $2n$.*

Proof. The tree decomposition that has the smallest treecost, consists of n bags which all contain only one vertices. Since each bag of size 1 has a cost of 2, the entire tree decomposition has a cost of $2n$. The only graph with n vertices that can have such a small treecost, is a graph that contains no edges. For a connected graph, the lower bound equals $4n-n$, since this is the cost of a path of length $n-1$ with two vertices per bag. □

The following lemma will be used in this thesis to compare the sizes of bags.

Lemma 2.10. *For natural numbers A, B, and C such that $A < C$ and $B < C$ it holds that $(2^A+2^B) \leq 2^C$.*

Proof. Consider a value for A and for B such that $A+1=C$ and $B+1=C$ (and thus, $A=B$). This satisfies $A<C$ and $B<C$. Then it holds that $2 \cdot 2^A = 2^C$ and $2 \cdot 2^B = 2^C$. Since $A=B$, we have that $2^A+2^B = C$ and thus $2^A+2^B \leq C$. In all other cases, either A or B (or both) would be less than $C-1$. As 2^x is an increasing function, it can be concluded that the statement $2^A+2^B \leq C$ always holds. □

The proof of the following lemmata can be found in [15, 20, 35].

Lemma 2.11. *Let $W \subseteq V$ be a clique in G and let $(\{X_i | i \in I\}, T = (I, F))$ be a tree decomposition of G. Then there is an $i \in I$ with $W \subseteq X_i$.*

Lemma 2.12. *Let G be a graph and let $(v_1, v_2, .., v_p)$ be a path in G. Let $(\{X_i | i \in I\}, T = (I, F))$ be a tree decomposition of G. Suppose $i_1, i_2, i_3 \in I$ and i_2 is on the path in T from i_1 to i_3. Suppose $v_1 \in X_1$ and $v_p \in X_3$. Then $\{v_1, v_2, .., v_p\} \cap X_2 \neq \emptyset$.*

Lemma 2.13. *Let TD be a tree decomposition of G and let X_i and X_j be two adjacent bags of TD. Then the set $\{X_i \cap X_j\}$ forms a separator of G.*

Lemma 2.14. *Let G be a graph that contains a cycle C. Let TD be a tree decomposition of G. Consider a pair of non-adjacent vertices $v, w \in C$ such that there is no bag that contains both v and w. Now let v and x break the cycle into two parts, named C_1 and C_2. Then there is a pair of vertices $x, y \in C$ with $x \in C_1$ and $y \in C_2$ such that there is a bag X_i in TD with $c, d \in X_i$.*

Chapter 3

Reduction Rules

Computation of the treecost of a graph is NP-hard, but the size of the problem can be decreased by using reduction rules. These rules allow for certain vertices and edges to be removed or for decomposing the graph into two or more connected components of which the treecost is computed separately. Since the treecost is a sum, it is important that the bags that are created during the application of the rules are being remembered. The vertices which these rules apply to are simplicial vertices, a specific type of almost simplicial vertices, inclusion minimal clique separators, and a specific type of inclusion minimal clique separators.

The first rules that are introduced remove vertices and edges from the graph. Meanwhile, a bag is created that includes the removed vertex (or vertices) and its neighbours. The rule describes which other bags this bag should eventually be connected to. Important is that such a rule is proven to be *safe*.

Definition 3.1. *A reduction rule that changes graph G into G' by removing vertices and edges, is called* **safe** *when $TC(G)$ equals $TC(G') + t$, where t is the cost of the bag that is created by that rule.*

The reduction rules are supposed to be used in the given order and can be repeated until there is no vertex left which a rule can be applied to. For each rule, the input graph is being searched for vertices which this rule concerns. Obviously, a vertex of degree 0 has a bag of its own and can after creation of this bag be removed. We assume that the graph for which the treecost needs to be computed is connected, since connected components are treated separately. We first define a rather straightforward rule for vertices of degree 1.

Rule 1. *Let vertex v have degree 1 with neighbour w.*
If *w has no other neighbours*
Then *create a bag $\{v, w\}$ and remove both v and w and their connecting edge (v, w).*
Else *create a bag $\{v, w\}$ and remove only v and edge (v, w). This bag should be connected to an arbitrary other bag containing w.*

Lemma 3.2. *Reduction rule 1 is safe.*

Proof. Since the case of w having no other neighbours is obvious, we only prove the latter case. Let v have w as its only neighbour and let $deg(w) \geq 2$. Consider any tree decomposition of the graph $G[V \setminus \{v\}]$. Since $w \in G$ this tree decomposition contains at least one bag with w in it, which is called X_i. It is now left to prove that adding a bag $\{v, w\}$ and making it adjacent to X_i is never more expensive than including v in X_i. The first option gives an extra cost of 2^2, whereas the second option multiplies the cost of bag X_i by 2, which gives an extra cost of the size of the bag. Since the size of X_i is at least 2 (because w is defined to have more neighbours than just v), the extra cost is at least 2^2. Therefore, creating a new bag $\{v, w\}$ will never lead to a higher cost than adding v to X_i. □

3.1 Simplicial vertices

In this subsection we discuss simplicial vertices and introduce a rule that can get rid of simplicial vertices to reduce the graph [2, 37]. This rule also defines which bags must be created and which edges and other vertices may be removed.

Definition 3.3. *A* **simplicial** *vertex v is a vertex such that for each pair of neighbours $\{w, x\}$ of v, it holds that $(w, x) \in E$. In other words, $N[v]$ forms a clique.*

Since all vertices of degree 1 are simplicial, Rule 1 already gave an intuitive idea of how simplicial vertices can be removed. We present some lemmata and finally come to a general reduction rule for simplicial vertices of any degree.

Lemma 3.4. *If v is a simplicial vertex, then $TC(G)$ equals $TC(G \setminus \{v\}) + 2^{|N[v]|}$. And thus it is optimal to create a bag of $N[v]$ and then remove v and its incident edges.*

Proof. Since $N[v]$ forms a clique, according to Lemma 2.11 there must be a bag containing all these vertices. It is now left to prove that it is never better to create a bag of a superset of $N[v]$. Consider any tree decomposition of $G[V \setminus \{v\}]$ that has a bag that contains all neighbours of v and at least one other vertex. If v is added to this bag, the cost (which is at least $2^{deg(v)+1}$) would be doubled. If, instead, a new bag is created of $N[v]$, the cost is increased by $2^{deg(v)+1}$. Therefore, the latter is never more expensive. And since v is already put together in a bag with all its neighbours and therefore no longer required for the construction of the tree decomposition, it may be removed from the graph. □

The following two lemmata assume that v is already removed from the graph and a bag of $N[v]$ is created.

Lemma 3.5. *For each neighbour w of simplicial vertex v such that $deg(v) = deg(w)$, i.e. w has no neighbours outside $N[v]$, w may be removed, together with its incident edges.*

Proof. Vertex w is already in a bag with all its neighbours and is therefore not required for any other bag. □

An example of the following lemma can be found in Figure 3.1.

Lemma 3.6. *Let w and x be a pair of neighbours of v. If there is no path from w to x that avoids $N(v)$, then the edge (x, w) may be removed. Otherwise, the edge remains in the graph, as any tree decomposition will contain another bag that has both w and x in it.*

Proof. Assume that the edge (w, x) forms the only path from w to x that avoids the rest of $N[v]$. This would mean that $N[v] \setminus \{w, x\}$ separates w and x. And since there already exists a bag X_i such that $w, x \in X_i$, they do not necessarily need to be put in another bag together.

If however, there is a path from w to x that avoids $N[v]$, then the edge (w, x) will not be removed. This implies that another bag will be constructed that contains both w and x, apart from the bag that consists of $N[v]$. Since the tree decomposition that will be constructed needs to be optimal, it cannot afford any redundant bags.

It is now proven that leaving the edge (w, x) in the graph is never more expensive than removing the edge, by showing that any tree decomposition will contain another bag with w and x together. Consider graph $G' = G[(V \setminus (N[v]) \cup \{w, x\}]$, i.e. the graph induced by all vertices of G outside the closed neighbourhood of v and w and w. Since w and x are adjacent, they form a cycle in G'. Lets call the vertices in the path on this cycle (from w to x) c_1, ..., c_n, where n denotes the number of vertices on this path (note that n can also be equal to 1). Let us assume that there is a tree decomposition with no bag containing both w and x and demonstrate that this leads to a contradiction. Since w is adjacent to c_1, there is a bag with these two vertices. The same holds for c_1 and c_2 etc. Eventually a bag must be created containing c_n and x. This bag must be adjacent to the bag of w and x, but so must the bag of w and c_1. This is in violation with the restrictions of tree decompositions, since a cycle is not allowed. Therefore, there will be a bag containing the vertices w and x. □

We now create a rule that concerns simplicial vertices of any degree.

Rule 2. *Let v be a simplicial vertex. Create a bag of $N[v]$ and remove v and its incident edges. For each neighbour w such that $deg(v) = deg(w)$, remove w and its edges. For each pair $w, x \in N[v]$ that has no path avoiding $N[v]$, remove the edge (w, x). All other vertices and edges from this clique must be retained.*

Lemma 3.7. *Rule 2 is safe.*

Proof. This follows from lemmata 3.4, 3.5, and 3.6. □

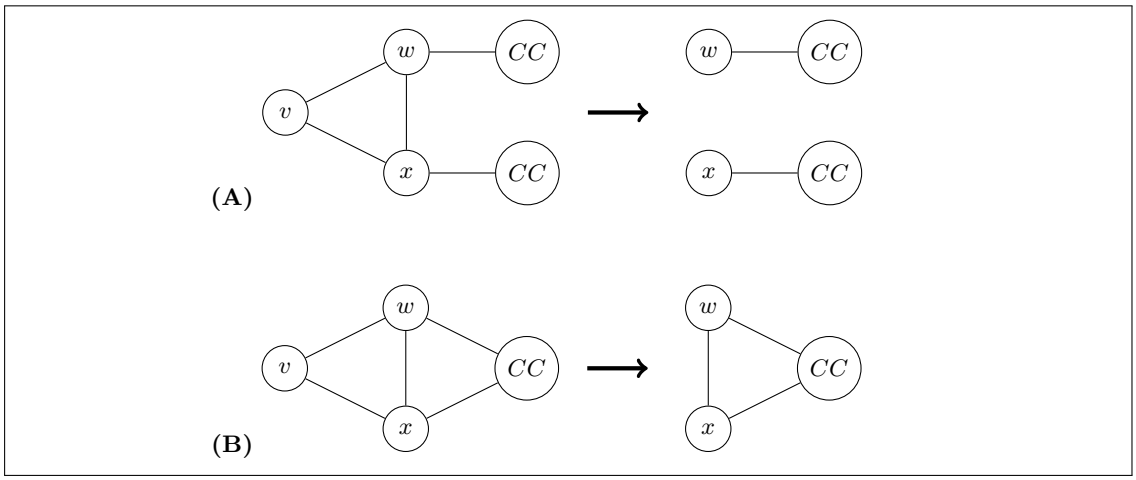

Figure 3.1: Two graph examples, before and after the application of Rule 2 are illustrated. CC represents a connected component.

3.2 Inclusion minimal clique separators

This section discusses clique separators that can split the graph into subgraphs. This enables to compute the treecost of smaller graphs, which is profitable [5, 16, 29, 39]. Figure 3.2 illustrates an example of the separation of a graph.

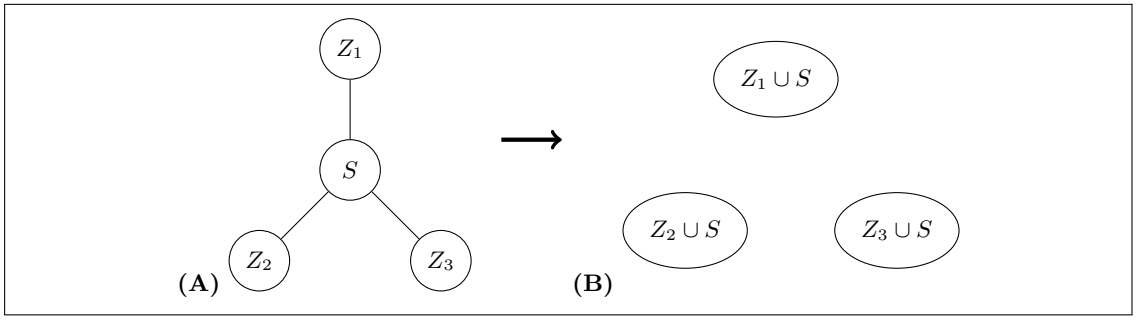

Figure 3.2: **(A)** *Separator S separates the graph into three connected components: Z_1, Z_2, and Z_3.* **(B)** *After separating, the graph consists of three subgraphs: $Z_1 \cup S$, $Z_2 \cup S$, and $Z_3 \cup S$.*

Before presenting a reduction rule that comprises inclusion minimal clique separators, we first introduce some definitions.

Definition 3.8. *A* **full connected component** *of a separator S is a connected component of $V \setminus S$ such that every vertex in S is adjacent to a vertex in the component.*

A proof of the following lemma can be found in [12].

Lemma 3.9. *A separator S of graph G is an inclusion minimal separator of G if and only if each of its connected components is full.*

To allow a separator S to split the graph into its connected components, S needs to be *safe*.

Definition 3.10. *A separator S of graph G is said to be* **safe** *for treecost when $TC(G)$ equals $\sum_Z TC(Z \cup S)$, for all connected components Z of $G[V \setminus S]$. In other words, if separator S is safe, the treecost of G can be computed by adding up all costs of the subgraphs of G, where each subgraph is created by taking a connected component of $G[V \setminus S]$ and S together.*

The next rule concerns separators of size 1. Such a separator is by definition inclusion minimal, since it is connected to all the components that are separated by such a vertex. As each set of size 1 always forms a clique, a separator of size 1 forms an inclusion minimal separator.

Lemma 3.11. *Let S be an inclusion minimal clique separator of graph G with a size equal to 1. Then S is safe for treecost, i.e. the treecost of G equals the sum of the costs of $Z \cup S$ for each connected component Z that is induced by $G[V \setminus S]$.*

Proof. Consider a separator S which consists only of vertex v and decomposes the graph into at least two connected components. Now for each component Z, consider an optimal tree decomposition of $Z \cup S$. Obviously, for each vertex $z \in Z$ such that $(v, z) \in E$ it holds that there is a bag X_i such that $v, z \in X_i$. Therefore, each tree decomposition contains at least one bag with v. Such a bag can be taken for each separate tree decomposition and then connected together in a path to form a tree decomposition for the entire graph. This is allowed, since no pair of vertices from different components are adjacent to each other.

It leaves us to prove that it is not more efficient to take two bags X_i and X_j from two different tree decompositions such that $v \in X_i$ and $v \in X_j$ and replace it by a new bag $X_k = (X_i \cup X_j)$. Since both X_i and X_j contain at least one vertex from its own component, it is known that $|X_i| < |X_k|$ and $|X_j| < |X_k|$. Lemma 2.10 implies that it is never cheaper to replace X_i and X_j by X_k. Therefore, if the individual tree decompositions are optimal, the composed tree decomposition of the entire graph is optimal as well and thus S is safe for treecost. □

The next rule holds for inclusion minimal clique separators of all degrees.

Rule 3. *If S is an inclusion minimal clique separator, then the total treecost can be computed by taking the sum of the costs of $Z \cup S$ for each component Z. The final tree decomposition can be obtained by taking a bag from every separate tree decomposition that contains S and connecting these together.*

Lemma 3.12. *Rule 3 is safe.*

Proof. The separator S is inclusion minimal, which induces that there exists a $z \in Z$ such that $(z, s) \in E$, $\forall s \in S$. Therefore, there must be a bag in this tree decomposition that contains both v and z. And since S is a clique, it holds that each tree decomposition of $Z \cup S$ contains a bag with S. If such a bag does not contain any other vertex outside S, the tree decomposition of the entire graph will contain bags that are subsets of other bags, which are redundant. Therefore, it is required to prove that each bag containing S also contains another vertex. Consider the tree decomposition created of $Z \cup S$ for a certain component Z and assume that the only bag containing S includes no other vertex. This bag is denoted by X_i. It will now be proven by contradiction that this is not possible. It is known that all vertices of S are adjacent to at least one other vertex $z \in Z$, they are contained in another bag than X_i, but not all together. But since each pair of vertices of the separator is connected by a path in Z, it is not possible to separate vertices of S in the bags linked to bag S, because this would create a cycle in the tree decomposition. Therefore, there must exist a bag that contains both S and a vertex $z \in Z$ for each Z. These bags can be connected together in a path to create the tree decomposition for the entire graph.

It is still necessary to prove that it is not only valid, but also optimal to connect these separate tree decomposition of $Z \cup S$ for each component Z to create a tree decomposition of the entire graph. It can be concluded that this is indeed optimal, since combining bags of different tree decompositions that contain S, is never cheaper due to Lemma 2.10. □

Corollary 3.13. *Let S be a clique separator that is not inclusion minimal. Then S is safe for treecost providing that bags that are subsets of other bags are being removed.*

Proof. Let S be a non-inclusion minimal separator, i.e. there exists a component Z' of $G \setminus S$ and a vertex $s \in S$ such that s is not adjacent to a vertex in Z'. Tree decompositions of $Z \cup S$ are created for each component Z. Each tree decomposition of a component Z' that is not full will have a bag that only consists of S, since vertex s that does not see Z' is simplicial and is put in a bag with just its neighbours. As S forms a clique, this is not a problem. But as there might be a bag X_i such that $S \subset X_i$, it is non-optimal to keep all bags that contain S. Therefore, all bags that consist of just S should be removed, except when there is no bag X_i such that $S \subset X_i$, which means that only one bag of just S should be retained. □

The graph is searched for inclusion minimal clique separators in increasing order of size. Since a clique separator that is not inclusion minimal is a superset of an inclusion minimal clique separator, it is assumed that the graph is already separated into its connected components.

3.3 Almost simplicial vertices

It was proven earlier that simplicial vertices can be eliminated from the graph while computing the treecost. Now we examine vertices that are almost simplicial.

Definition 3.14. *A vertex v is* **almost simplicial** *when all its neighbours are adjacent, except for one neighbour, which is called the special vertex.*

For treewidth there exists a rule that removes all almost simplicial vertices. Unfortunately, no such rule exists for treecost. However, some rules can be defined for a specific situation, namely vertices that are almost simplicial and can be made simplicial by adding one edge.

3.3.1 Almost simplicial vertices of degree 2

A reduction rule can be created for all vertices that are almost simplicial and have only two neighbours. Consider such a vertex v with neighbours w and x that are not adjacent, which makes v almost simplicial. Both vertices can be the special vertex, but for now w is denoted as special. If there is no path between w and x that avoids v, then v separates the graph into two connected components. Therefore, the bags $\{v, w\}$ and $\{v, x\}$ can be added and connected to each other and to the tree decomposition of their own component. Subsequently, v and its incident edges can be removed. A less trivial concerns the situation where a path between w and x that avoids v does exist. Before a rule is created for this situation, a new lemma needs to be introduced.

Lemma 3.15. *Let G be a graph and let v be simplicial with two non-adjacent neighbours w and x. Consider an optimal tree decomposition TD of G that has no bag that contains v, w, and x together, but where the bags X_i and X_j, such that $v, w \in X_i$ and $v, x \in X_j$, are adjacent. Then either $X_i \setminus \{w\} = X_j \setminus \{x\}$ or the tree decomposition can be transformed into a new tree decomposition TD' such that this is the case, without increasing the treecost.*

Proof. It will now be demonstrated how TD where $X_i \setminus \{w\} \neq X_j \setminus \{x\}$ can be transformed into TD', such that $X_i \setminus \{w\} = X_j \setminus \{x\}$. We replace X_i by $X_{i1} = (X_i \cap X_j) \cup \{w\}$ and X_j by $X_{j1} = (X_i \cap X_j) \cup \{x\}$. Note that $v \in X_{i1}$ and $v \in X_{j1}$. If $(X_i \setminus \{w\}) \subset (X_j \setminus \{x\}))$, then $X_i = X_{i1}$ and clearly no new bag should be added to TD. The same holds for the case where $(X_j \setminus \{x\}) \subset (X_i \setminus \{w\})$. If, however, $X_i \neq X_{i1}$, a new bag $X_{i2} = X_i \setminus \{v\}$ is created and added between X_{i1} and all former neighbours of X_{i1}, except X_{j1}. Bag X_{i1} is now only adjacent to X_{i2}

and X_{j1}. A similar addition to the graph of a bag X_{j2} should be made if $X_j \neq X_{j1}$.

First it must be determined that the new tree decomposition is still valid. Since v has only w and x as its neighbours, v cannot be adjacent to any vertex outside X_{i1} and X_{j1}. It follows from Lemma 2.13 that $\{X_i \cap X_j\}$ forms a separator, and thus X_{i1} and X_{j1} properly separate X_{i2} from X_{j2}.

Subsequently, it must be proven that $TC(TD') \leq TC(TD)$. As X_{i1} and X_{i2} are both are smaller than X_i, it can be derived from Lemma 2.10 that the cost of these new bags together is never larger than the cost of X_i. The same holds for the cost of X_{j1} and X_{j2} compared to the cost of X_j. Therefore we have that $TC(TD') \leq TC(TD)$. □

An example of the transformation described in the proof above is shown in Figure 3.3.

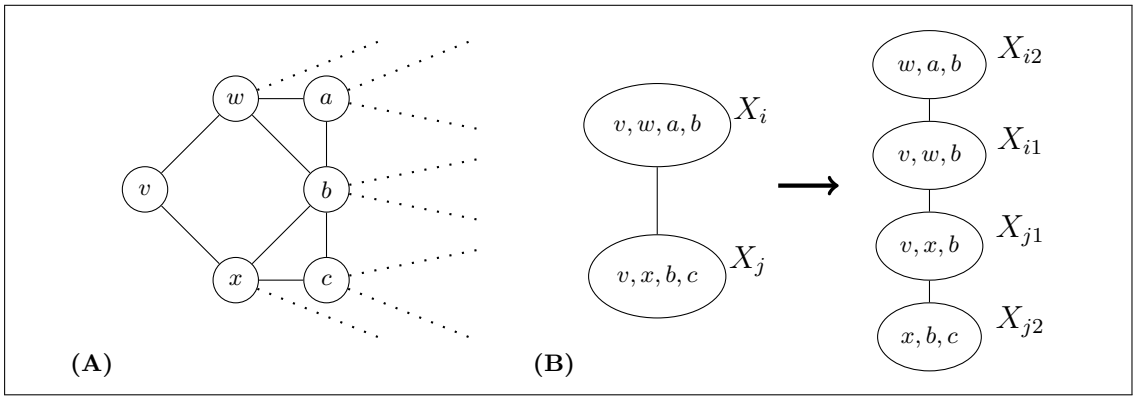

Figure 3.3: **(A)** *Graph G with simplicial vertex v,* **(B)** *Tree decomposition of G where $X_i \setminus \{w\} \neq X_j \setminus \{x\}$ that is transformed in a tree decomposition where $X_i \setminus \{w\} = X_j \setminus \{x\}$.*

Now we consider the case where bag X_i and bag X_j from Lemma 3.15 (such that $v, w \in X_i$ and $v, x \in X_j$) are non-adjacent.

Lemma 3.16. *Let G be the graph described in Lemma 3.15 with an optimal tree decomposition TD such that bags X_i and X_j are non-adjacent. Let P be the path from X_i to X_j, with $P = (X_1, ..., X_p)$ and $X_i = X_1$ and $X_j = X_p$. Then either $(X_1 \setminus \{w\}) \subseteq X_2$ and $(X_p \setminus \{x\}) \subseteq X_{p-1}$ or TD can be transformed such that this is the case, without increasing the treecost.*

Proof. Consider the case where $(X_1 \setminus \{w\}) \not\subseteq X_2$. This implies that there is at least one vertex in X_1, apart from w, that is not contained in X_2 (clearly, $v \in X_2$, as $v \in X_1$ and $v \in X_p$ and X_2 is on the path from X_1 to X_p). First we claim that it can be assumed that any optimal tree decomposition would not have bags outside P that contain v, since v is only adjacent to w and x. Let Y denote the set of vertices in X_1 that are not contained in X_2 (apart from v). Formally defined: $Y = (X_1 \setminus X_2) \setminus \{w\}$. For each vertex $y \in Y$ it holds that it is non-adjacent to w. Therefore, v and y do not have to be put in a bag together. Furthermore, Lemma 2.13 it holds that $X_1 \cap X_2$ forms a separator and so does $X_1 \setminus Y$. Hence, Y may be removed from X_1. Since vertices in Y can still be adjacent to w or vertices in $X_1 \cap X_2$, we create another bag containing these vertices, denoted by X_0. Note that X_0 is equal to $X_1 \setminus \{v\}$. The path can be changed by adding bag X_0 between X_1 and all the neighbours of X_1, except for X_2. We remove the incident edges of X_1, except those that connect X_1 to X_0 and X_2 (which makes X_0 the starting point of the path, instead of X_1). Both X_0 and the new version of X_1 are smaller than the original bag X_1, which implies (by Lemma 2.10) that the treecost is not increased. Obviously, the same method can be used to make X_{p-1} a subset of X_p. □

Figure 3.4 displays an example of a transformation of a path described in the proof above.
We now define a reduction rule that removes almost simplicial vertices of degree 2.

Rule 4. *Let v be almost simplicial with non-adjacent neighbours w and x. If there is no path between w and x that avoids v, then create two adjacent bags, namely $\{v, w\}$ and $\{v, x\}$. For bags*

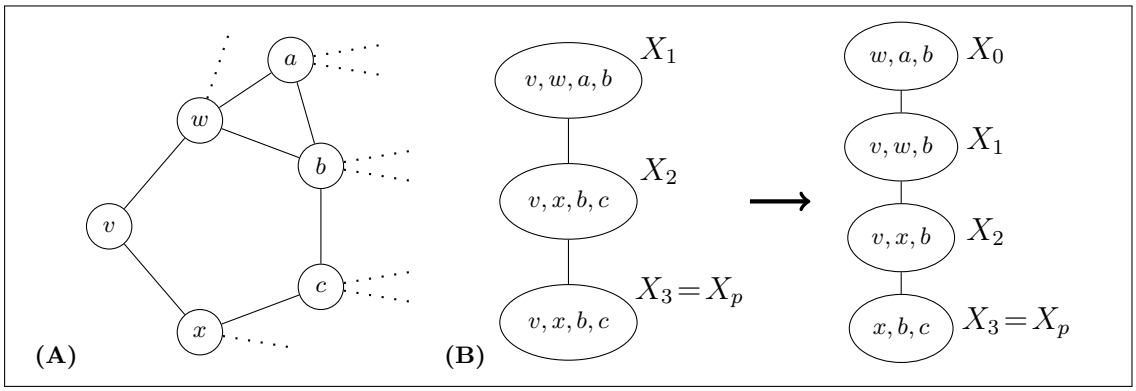

Figure 3.4: **(A)** *Graph G with simplicial vertex v,* **(B)** *Path P from tree decomposition of G where $X_i \setminus \{w\} \neq X_j \setminus \{x\}$ that is transformed in a tree decomposition where $(X_1 \setminus \{w\}) \not\subseteq X_2$.*

that are created later during the preprocessing, it holds that an arbitrary bag containing w (if one exists) should be connected to $\{v, w\}$, whereas an arbitrary bag containing x (if one exists) should be connected to $\{v, x\}$.

If there is a path between w and x that avoids v, then add the bag $\{v, w, x\}$, remove v and its incident edges, and add the edge (w, x).

Lemma 3.17. *Rule 4 is safe.*

Proof. Since v forms a clique separator in the first case (where no path between w and x avoiding v exists), this part of the rule is already verified in the proof of Reduction rule 3. We therefore continue with the case where there is a path between w and x avoiding v. To prove that this is optimal, it will be shown that any optimal tree decomposition of a graph G that does not contain a bag with $\{v, w, x\}$ can be transformed into a tree decomposition that does contain such a bag, without increasing the treecost.

Since v and w, as well as v and x, are adjacent, there must exist a bag with v and w and a bag with v and x. Consider the case where x is not contained in the bag with v and w, and w is not contained in the bag with v and x. Both bags contain a vertex unequal to v, w, and x, otherwise the bag is a subset of another bag, which makes the tree decomposition non-optimal. The bag of v and w is denoted by X_{vw} and the bag of v and x by X_{vx}. If these two bags are adjacent in the tree decomposition then it is known from Lemma 3.15 that their other included vertices (thus the vertices apart from v, w, and x) are the same. We will call this set of vertices S (i.e. $S = X_{vw} \setminus \{v, w\} = X_{vx} \setminus \{v, x\}$). These two bags can be removed and thereafter replaced by two new bags, namely: a bag of $S \cup \{w, x\}$ (which will be connected to the former neighbours of X_{vw} and X_{vx}) and a bag of $\{v, w, x\}$ (which is only adjacent to the former bag). If X_{vw} and X_{vx} are non-adjacent, it becomes more difficult. One of the two bags can be chosen, say X_{vw}, whereupon the bag $\{v, w, x\}$ can be connected to just this bag. Note that this increases the treecost by $2^3 = 8$. Then v is replaced for x in X_{vw}, which does not change the treecost. In all bags that are on the path between X_{vw} and X_{vx} the vertex v can be replaced for x as well. Now it holds, by Lemma 3.16, that the bag X_{vx} is a subset of its former bag on the path and may thus be removed, which decreases the treecost by at least $2^3 = 8$. Its former bag on the path is connected to the neighbours of X_{vx}. This new tree decomposition has a treecost that is at most the treecost of the former tree decomposition, since the size of the removed bag X_{vx} can never be less than three (which is the size of the added bag $\{v, w, x\}$). □

3.3.2 Almost simplicial vertices of degree 3

The next step is to discuss what reductions can be made for almost simplicial vertices of degree 3. Consider a vertex x with neighbours w, x, and y, where w is its special neighbour. There are

two cases: either w is not connected to any of the other neighbours of v, or w is only connected to one of them. Without loss of generality it is assumed that in the last case w is connected to x and not to y. First the case where w is not connected to x or y is discussed (shown in Figure 3.5 (A)).

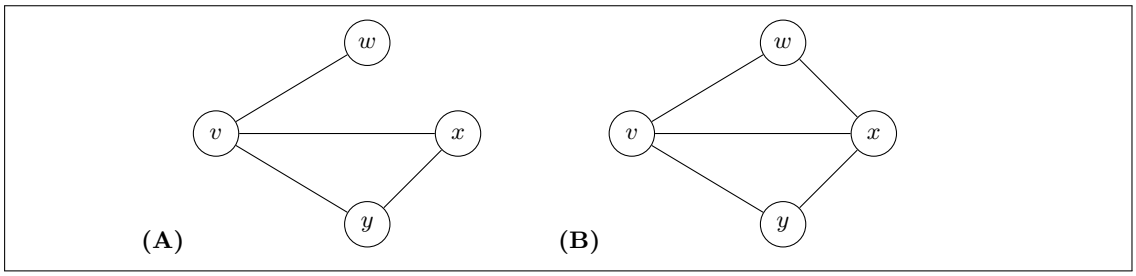

*Figure 3.5: Almost simplicial vertex v with special neighbour w. **(A)** w is not connected to x or y. **(B)** w is only connected to x.*

Case (A): w is not adjacent to x or y

The situation where the special neighbour w of the almost simplicial vertex v is not connected to either x or y, is divided into several subcases, which are distinguished by whether or not there is a path from w to x or y (or both) that avoids v. If there is no path from w to either x or y that avoids v, the situation is not interesting, since v separates w from x and y and thus two bags must be created: $\{v, w\}$ and $\{v, x, y\}$ (see Lemma 3). There are three other possibilities, namely:

- There is a path from w to x or to y (but not both) that avoids $N[v]$,
- there is a path from w to x and one to y that both avoid $N[v]$, such that both paths avoid each other (and thus form two different components),
- there is a path that connects w to both x and w, which means that w, x, and y are all connected to that same component.

These possibilities are now considered one by one. The first rule applies when there is a path from w to exactly one of the other neighbours of v. Without loss of generality, it is assumed that this is x.

Rule 5. *Consider the graph from case A, with almost simplicial vertex v and its three neighbours w, x, and y of which only x and y are adjacent. Let there be a path from w to x that avoids both v and y, but no path from w to y that avoids both v and x (see Figure 3.6). First create the bags $\{v, w, x\}$ and $\{v, x, y\}$, then remove v and its incident edges, and finally add an edge between w and x. If there is no path left between x and y apart from the edge that connects them, then this edge can be removed.*

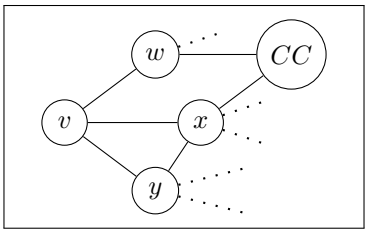

Figure 3.6

Lemma 3.18. *Reduction rule 5 is safe.*

Proof. Since there is no path between w and y that avoids v and x, it holds that v and x form a clique separator. The graph can therefore be separated into these two components, whereafter a tree decomposition of each of these components can be created, together with v and x. In one of these components, v becomes simplicial, thus the bag $\{v, x, y\}$ is required. In the other component, v is almost simplicial and has a degree of 2. Therefore, it holds that v can safely be put in a bag with w and x. □

Rule 6. *Again, consider the graph of case A, where x and y are the only adjacent neighbours of almost simplicial vertex v. Let there be a path from w to x that avoids both v and y and a path from w to y that avoids both v and x (see Figure 3.7). Let these two paths be separated in two different components by $N[v]$. Then create the bag $\{v, w, x, y\}$, remove v and connect w to x and to y. If there is no path from x to y, apart from its edge, then this edge may be removed.*

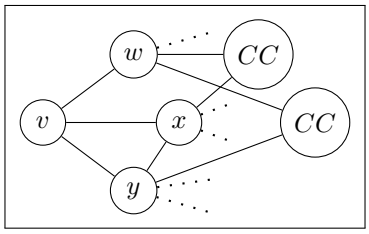

Figure 3.7

Lemma 3.19. *Reduction rule 6 is safe.*

Proof. For this prove we require the term *almost clique separator*, which is a separator such that all its vertices, except for one, form a clique. Section 3.4 will discuss these kind of separators extensively. Reduction rule 3.37 asserts that for all almost clique separators of size 2 an edge may safely be added between its two vertices, followed by a proof. The vertices w and x form such an almost clique separator, which, according to Rule 3.37, implies that the edge (w, x) is added. Therefore, it holds that these must be put in a bag together. The same holds for w and y, since they y form an almost clique separator as well. Given that v, x, and y form a clique, there must be a bag containing these three vertices. If separate bags of the three sets that were just mentioned (i.e. $\{w, x\}$, $\{w, y\}$, and $\{v, x, y\}$) are created, there would be a cycle in our tree decomposition, which is not allowed. Therefore, it is optimal to create a bag of $\{v, w, x, y\}$. □

Finally the last possibility is being considered, where w, x, and y are all connected to the same component, after removing v (see Figure 3.8). This situation is more difficult than the other ones and the optimal action depends on the rest of the graph. Therefore, a specific situation is discussed first, where w, x, and y share the same neighbourhood apart from $N[v]$, which forms a clique. The optimal bag creation depends in this case on the number of their common (and only) neighbours.

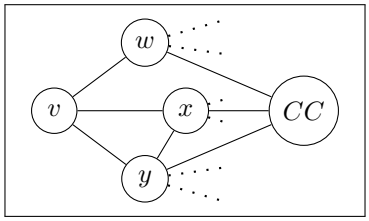

Figure 3.8

Rule 7. *Again, consider the case where only x and y are adjacent. Let the graph be structured such that after the removal of v, the vertices w, x, and y would still be connected to the same component (as is illustrated in Figure 3.9). Let w, x, and y have the same neighbours outside $N[v]$, which are all adjacent to each other. We now count the number of neighbours outside $N[v]$ of w, x, and y.*

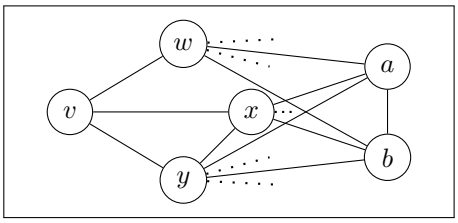

Figure 3.9

If the number of these neighbours of w, x, and y equals 1, then put v, w, and this neighbour in one bag and v, x, y, and the neighbour in another bag. If there are two common neighbours outside $N[v]$ (e.g. a and b), their are two options with equal cost: the first option, which is similar to the one with one common neighbours, is to put v, w, a, and b in one bag and v, x, y, a, and b in another one. The second option is to create one bag of v, w, x, y, and one bag of w, x, y, and the two common neighbours. If there are more than two common neighbours outside $N[v]$, then create a bag of v, w, x, and y and then a new bag of w, x, y, and the common neighbours.

Lemma 3.20. *Rule 7 is safe.*

Proof. If the common (and only) neighbours outside $N[v]$ of w, x, and y have other neighbours outside $N[v]$, then these common neighbours form a clique separator. Let S denote this clique separator. According to Rule 3, the graph can be separated in several components, of which one only consists of $N[v] \cup S$. Since this component is finite, the correctness of the rule can simply

be verified by computing the cost of each option. Clearly, if the common neighbours do not have other neighbours, the graph only consists of $N[v] \cup S$, which means that the graph itself is finite. And thus, for both cases it is easy to see that the rule is safe. □

For the last case, where w is not connected to either x or y, and the vertices w, x, and y are connected to the same component, no general rule could be found. In other words, if the neighbours of v do not share a common neighbourhood outside $N[v]$ that forms a clique, no rule can be applied. We noticed that in many cases it is not efficient to put $N[v]$ in one bag. And since no pattern could be found of what bag w should be put in, v could not be easily eliminated. Therefore, we provide no rule for this case.

Case (B): w is only adjacent to x

Now the latter and most interesting case is being discussed, where w is adjacent to x and x to y. In the proof of Rule 4 it is described how any path in an optimal tree decomposition from X_{vw} (the bag containing v and w) to X_{vx} (the bag of v and x) can be transformed such that there is a bag containing v, w, and x together. This algorithm will be used again for the current case. In fact, the reason that the algorithm works is because there is only one edge missing to cause $N[v]$ to form a clique, namely the edge (w, y). This implies that there must be a bag containing v, w, and x and a bag containing v, x, and y. Therefore, only one path is needed to convert the tree decomposition such that $N[v]$ forms a bag.

Rule 8. *If there is no path between w and y that avoids v and x (as shown in Figure 3.10), create the bags $\{v, w, x\}$ and $\{v, x, y\}$. Then remove v and its incident edges. Each neighbour of v that has no neighbours outside $N[v]$, can be removed as well. If there is no path between w and x, apart from the direct edge between them, that avoids v, the edge between w and x can be removed. These vertices are not required in another bag together. The same holds for the pair x and y.*

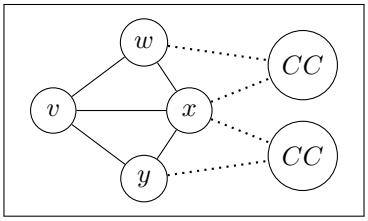

Figure 3.10

Lemma 3.21. *Reduction rule 8 is safe.*

Proof. Since there is no path between w and y that avoids v and x, we have that v and x form a clique separator of the graph. Therefore, separate tree decompositions of the two parts that they separate can be created, including v and x. As v is simplicial in both parts, it can be put in a bag with its neighbours, which are w and x in the one part and x and y in the other part. All connected components of the graph minus v and x that do not contain y, can be connected to the bag $\{v, w, x\}$ and all connected components that do not contain w can be connected to $\{v, x, y\}$. Obviously it is possible that there are no such connected components. Since the costs of creating two separate bags of size 3 (namely $2 \cdot 2^3 = 16$) is equal to the cost of only one bag of size 4 (also $2^4 = 16$), it is also optimal to create one bag of $N(v)$: $\{v, w, x, y\}$. □

Rule 9. *If there is a path between w and y that avoids v and x (as shown in Figure 3.11), then create the bag $\{v, w, x, y\}$ and remove v and its incident edges. We add an edge between w and y. If x has no neighbours outside $N(v)$, it can now be removed. If there is no path between w and x that avoids v, apart from the edge between them, then the edge between w and x can be removed.*

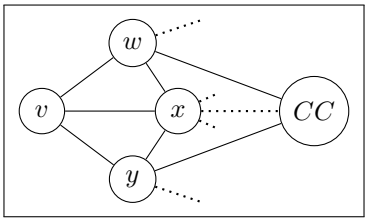

Figure 3.11

Lemma 3.22. *Reduction rule 9 is safe.*

Proof. To prove that this is optimal, it is shown that any tree decomposition can be taken in which w and y are not put together in a bag and transform it to one that contains the bag $\{v, w, x, y\}$. Since v, w, and x form a clique, as well as v, x, and y, we must have two bags that contain these cliques. Since both bags contain v and x, all bags on the path between these two bags must contain v and x as well. The other vertices in these bags are from a path between w and y. Now a bag with v, w, x and y is created and connected to the bag that contains v, w, and x. v and x can be replaced by y in this latter bag. This also holds for the following bags along this path. The last bag (which contains amongst others v, x, and y) can now be removed, since without v and x this bag is a subset of the former bag. As a bag of size 4 is added and a bag of size 4 is removed and all potential bags in the middle of this path have one vertex less, the treecost can never be higher.

An other possibility to prove this is to realise that w and y form an almost clique separator. This means that they must be contained in a bag together (see the proof of almost clique separators of size 2). Since the combinations $\{v, w, x\}$, $\{v, x, y\}$, and $\{w, y\}$ must be in a bag together, the only option is to create the bag $\{v, w, x, y\}$. □

For the case where no path exists between w and y that avoids $N[v]$ (i.e. $\{v, x\}$ forms a separator), it did not matter if the bag $\{v, w, x, y\}$ would be created, or the two bags $\{v, w, x\}$, $\{v, x, y\}$. Hence, it can safely be said that if w and x are adjacent, as well as x and y (which is the case in rules 8 and 9), it is optimal to put v in one bag with all its neighbours.

3.3.3 Almost simplicial vertices of any degree

It can be concluded that a general rule for almost simplicial vertices of any degree can be created for the case where the special neighbour is adjacent to all other neighbours except for one. We call this type of vertices *nearly simplicial*.

Definition 3.23. *A vertex v of a graph G is* **nearly simplicial** *when it has a pair of neighbours $\{w, x\}$, such that $N[v]$ forms a clique in $G + (w, x)$, but not in G. In other words, v would have been simplicial if only one edge was added. We call the non-adjacent pair the* **special pair**.

Clearly, all almost simplicial vertices of degree 2 are also nearly simplicial. For nearly simplicial vertices, the transformation algorithm of the proof of Lemma 9 can be applied.

Rule 10. *Let G be a graph with nearly simplicial vertex v and let w and x be the special pair. Then create a bag of $N[v]$ and remove v and its incident edges. Thereafter, remove all neighbours of v that have no neighbours outside $N[v]$. For each pair $y, z \in N[v]$ that is not connected by a path that avoids $N[v]$, remove the edge (y, z) as well, as this pair is not required to be together in another bag.*

Theorem 3.24. *Reduction rule 10 is safe.*

Proof. It is assumed that an optimal tree decomposition TD of the graph G described in the previous rule, does not contain a bag with $N(v)$. To prove that creating a bag of $N[v]$ is optimal, it is shown that transforming TD such that there is such a bag does not increase the treecost. Since $N[v] \setminus \{x\}$ forms a clique, as well as $N[v] \setminus \{w\}$, there exist bags containing these cliques. The path between these bags is denoted by $P = (X_1, ..., X_p)$, where $(N[v] \setminus \{x\}) \subset X_1$ and $(N[v] \setminus \{w\}) \subset X_p$. Note that all bags on this path contain $N[v] \setminus \{w, x\}$. The path is now transformed as follows. A bag with $N[v]$ is created and connected to X_1 which increases the cost by $2^{|N[v]|}$. For all bags from X_1 to X_{p-1} the vertex v is replaced by x, which induces a valid tree decomposition and keeps the treecost equal. As X_p is now a subset of X_{p-1}, it may be removed. X_p cannot be smaller than the bag with $N[v]$, which implies that the treecost has not increased. □

Corollary 3.25. *Let v be an almost simplicial vertex and let special vertex w be connected to no other neighbour of v. Let there be a path from w to a certain neighbour of v, say x, that avoids $N[v]$, but no path from w to any other neighbour. Then two bags can be created: $\{v, w, x\}$ and $N[v] \setminus \{w\}$.*

Proof. As $\{v, x\}$ forms a clique separator and it was proven earlier that the graph can be split by its connected components, v becomes an almost simplicial vertex of degree 2 in the first component and a simplicial vertex in the second. Therefore, it is optimal to remove v and create these two bags. □

3.4 Inclusion minimal almost clique separators

It was shown that inclusion minimal clique separators are safe. The treecost can simply be computed separately for the connected components. Now inclusion minimal separators that are almost a clique are being discussed.

Definition 3.26. *An* **almost clique** C *is a set of vertices that contains one vertex* v *such that* $C \setminus \{v\}$ *forms a clique.*

Definition 3.27. *An* **inclusion minimal almost clique separator** S *is a separator that forms an almost clique and separates the graph into only full connected.*

Again, no general rule for almost clique separators of any size could be found, but some interesting findings will be discussed. It was realised that the neighbourhood of an almost simplicial vertex forms an inclusion minimal almost clique separator, which makes the former a special case of the latter, namely an almost clique separator that separates the graph into at least one component of size 1.

3.4.1 Inclusion minimal almost clique separators of size 2

First separators of size 2 are considered, i.e. two vertices x and y that are not adjacent, but decompose the graph into several connected components. A proof is constructed for the statement that almost clique separators of size 2 are safe for treecost if x and y are made adjacent. In other words, it is safe to add an edge between x and y and then separate the graph into its connected components to compute for each component Z the treecost of $Z \cup S$. S is now a clique. If x and y are adjacent, there will always be a bag in every optimal tree decomposition that contains both x and y. To prove that it is safe to make x and y adjacent, it is shown that any tree decomposition that has no bag of x and y together, can be transformed into a tree decomposition that does have such a bag, without increasing the treecost.

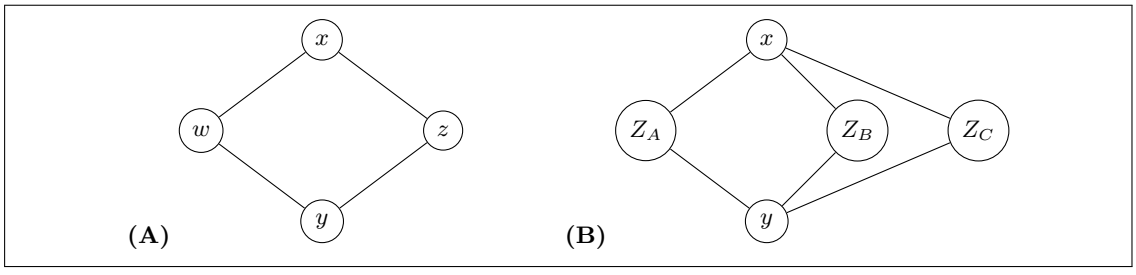

Figure 3.12: **(A)** *Simple example of a graph where* x *and* y *form an almost clique separator of size* 2. **(B)** *Graph that is separated into three components by* x *and* y.

Let G be a graph and the non-adjacent vertices x and y be a separator of the graph. For convenience, it is assumed that x and y separate the graph into only two connected components. Later a statement is made for separators that induce more than two components. The two components that are separated by x and y, are denoted by Z_A and Z_B. Assume a tree decomposition TD of G that has no bag with both x and y. There must exist exactly one path between the connected subtree of bags that contain x and bags with y. This path is designated by $P = (X_1, ..., X_p)$. The first bag, which is denoted by X_1, contains an x and is the closest to any bag with a y of all bags

that contain an x. The last bag, denoted by X_p, is the closest bag with a y to X_1. All bags in between contain neither x nor y. Note that the path between bags with x and y can be of length 2, which means that the first and last bag are adjacent. The bags of the tree decomposition that are not part of the path can consist of several connected components (it is also possible that no other bags are left). The path is converted into a path P' that has a bag with both x and y and later we add the connected components to this new path.

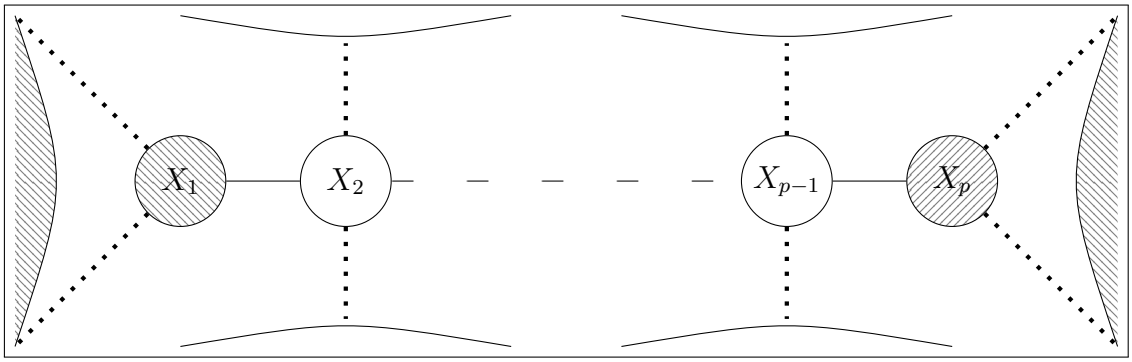

Figure 3.13: Path $P = (X1, ..., X_p)$, where $x \in X_1$ and $y \in X_p$. The dotted lines represent potential edges, which lead to potential subgraphs. The shaded part on the left may contain x and the shaded part on the right may contain y. All other nodes of the graph do not contain x or y.

Lemma 3.28. *Consider the path P described above. Each bag $X_i \in P$, such that $1 < i < p$ (i.e. all bags except X_i and X_p) contains at least one vertex of Z_A and at least one vertex of Z_B.*

Proof. From Lemma 2.12 it can be derived that for every path in $x, ..., y \in G$, it holds that if $x \in X_1$ and $y \in X_p$, then every bag between X_1 and X_p contains a vertex of the path $x, ..., y$. Since there is a path from x to y through Z_A and one through Z_B and there is no bag between X_1 and X_p containing either x or y, it holds that all these bags contain at least one vertex from Z_A and one from Z_B. □

Lemma 3.29. *For all connected components of TD that are not part of path P it is necessary to only consist of either vertices of Z_A or vertices of Z_B, given that the entire tree decomposition TD is optimal.*

Proof. It holds that the treecost of a graph G equals the treecost of its minimal triangulation H. If a triangulation H' of G is not minimal, then the $TC(G) \leq TC(H')$. It is now assumed that TD contains a path in a connected component of $TD \setminus P$ that has a vertex $a \in Z_A$ and a vertex $b \in Z_B$. Each bag X_i on this path is split into two new bags that contain $X_i \cup Z_A$ and $X_i \cup Z_B$ if $x \notin X_i$, and $\{x\} \cup X_i \cup Z_A$ and $\{x\} \cup X_i \cup Z_B$ if $x \in X_i$. The treecost cannot be increased, since the size of each bag is at most $|X_i|-1$ and $2 \cdot 2^{|X_i|-1} = 2^{|X_i|}$. It also holds that the tree decomposition is still valid, since no pair of vertices a, b such that $a \in Z_A$ and $b \in Z_B$ needs to be added in a bag together outside path P. Therefore, it can be assumed that the optimal tree decomposition TD contains no bag X_i in $TD \setminus P$ such that $\{a, b\} \in X_i$. □

The following lemma makes a statement of the case where P has a length of 2, i.e. P consists only of X_1 and X_p.

Lemma 3.30. *Let P be a path of length 2, i.e. $P = (X_1, X_p)$. Then it is safe to convert P into P' where the two bags are replaced by two new ones $Y_1 = \{x, y\} \cup ((X_1 \cup X_p) \cap Z_A)$ and $Y_2 = \{x, y\} \cup ((X_1 \cup X_p) \cap Z_B)$.*

Proof. First it is shown that by means of this transformation a new valid tree decomposition is obtained. Vertices of Z_A and vertices of Z_B are separated by x and y, which implies that no pair $\{a, b \mid a \in Z_A, b \in Z_B\}$ is adjacent. Therefore, if x and y are put together in a bag, it is not

necessary to added vertices of both Z_A and Z_B in a bag together. Since it holds by Lemma 3.29 that all bags outside path P consist of only vertices of Z_A or only vertices of Z_B (and in some cases x or y), these bags can easily be reconnected to the path P', without causing a violation.

Second, it is shown that the size of the new tree decomposition is not increased by the transformation. It may be assumed that $X_1 \setminus \{x\} = X_p \setminus \{y\}$, since this is always optimal. Since at least one vertex of X_1 is in Z_A and thus at least one vertex is removed from X_1 to get Y_1, it is known that $|Y_1| \leq |X_1|$. Similarly, for Y_2 it holds that $|Y_2| \leq |X_2|$. Therefore, the treecost has not increased by converting P into P'. □

We now introduce two new definitions that are required for the next lemmata.

Definition 3.31. *A bag is said to* **introduce** *a vertex v, if v is contained in that bag, but not in an adjacent bag.*

Definition 3.32. *A bag is said to* **eliminate** *a vertex v, if that bag does not contain v, but an adjacent bag does. Note that in this case there only exists one adjacent bag containing v, since otherwise the bags with v do not form a connected subtree.*

A vertex v is usually eliminated in a bag X_i if it is no longer required to be in X_i. The reason for this is that for each neighbour w, there already exists a bag with v and w together. The neighbour of bag X_i that contains v, denoted by v, has introduced one or more of the neighbours of v. We say that v is eliminated in bag X_i by these introduced neighbours.

From now on it is assumed that the length of path P is greater than 2. The next lemma makes a claim about tree decompositions that contain a bag which introduces two new vertices that do not share a common neighbour. If such a tree decomposition is optimal, it can be transformed into a tree decomposition which does not have such a bag.

Lemma 3.33. *Let TD be an optimal tree decomposition with adjacent bags X_i and X_j. Again, let a and b be a non-adjacent pair of vertices, such that $a \in Z_A$ and $b \in Z_B$. Furthermore, let $a, b \notin X_i$ and $a, b \in X_j$ and let $(N[a] \cap N[b]) = \emptyset$. Then TD can be converted into a tree decomposition TD' where X_j contains either a or b, but not the other one.*

Proof. It is assumed that X_i is not a superset of X_j, since this is non-optimal. Let X_k be a bag that is adjacent to X_j. There is a set of vertices A such that each vertex in A is adjacent to a and $A \subset (X_j \setminus X_k)$ (possibly $|A| = 1$). for which it holds that as soon as a is introduced after bag X_j, all vertices in A are eliminated. Moreover, there exists a set of vertices B, such that all are adjacent to b and $B \subset (X_j \setminus X_k)$, that is eliminated by b. It also holds by $(N[a] \cap N[b]) = \emptyset$ that $(A \cap B) = \emptyset$. A valid replacement for X_j would therefore consist of two new bags $X_{j1} = X_j \setminus \{b\}$ and $X_{j2} = X_j \setminus A$. Since both bags are smaller than X_j, the total treecost cannot be larger than X_j. □

An algorithm is now constructed that indicates how P can be transformed into a new path P' such that x and y are added in a bag together and the new tree decomposition is still valid. The algorithm makes use of Lemma 3.33.

Let G be a graph and let $\{x, y\}$ be an inclusion minimal almost clique separator. Assume an optimal tree decomposition where there is no bag that contains both x and y. Let $P = (X_1, ..., X_p)$ be a path such that $x \in X_1$ and $y \in X_p$ and all other bags on the path contain neither x nor y. Then this path can be transformed as follows. The first bag of the path, which is X_1, is transformed first. This bag consists of x, one or more vertices from Z_A, and one or more from Z_B. Now X_1 is replaced by two new bags A_1 and B_1, where A_1 consists of $\{x, y\} \cup (X_1 \cap Z_A)$ and B_1 consists of $\{x, y\} \cup (X_1 \cap Z_B)$. These bags are made adjacent and the path is built, starting from both A_1 and B_1, which brings these two bags in the middle of the path. All of the new bags will contain y, otherwise the bags with y will not form a connected subtree. A_1 and B_1 are the only bags with x, since x is not required for any other bags. For the rest, one side of the path will only

contain vertices from Z_A, whereas the other one will only have vertices of Z_B. Now the second bag of the original path is taken, denoted by X_2. Since it was assumed that P was longer than 1, it holds that X_2 is not the last bag of the path yet. Now all vertices $Y = X_2 \setminus X_1$ are considered. It holds by Lemma 3.33 that either $Y \subset Z_A$ or $Y \subset Z_B$. If $Y \subset Z_A$ then X_2 can be replaced by $A_2 = \{y\} \cup (X_2 \cap Z_A)$ and then be connected to A_1. And similarly, if $Y \subset Z_B$ then X_2 can be replaced by $B_2 = \{y\} \cup (X_2 \cap Z_B)$ and connected to B_1. By travelling further along the original path (X_3, X_4,...), each bag X_i is replaced by either $A = \{y\} \cup (X_i \cap Z_A)$ or $B = \{x\} \cup (X_i \cap Z_B)$. This continues until the last bag of the path is reached, denoted by X_p. This bag contains, apart from y, only vertices of Z_A and Z_B that were also contained in the former bag. This is correct, because it is always optimal to have y as the only new vertex in X_p compared to X_{p-1} and no other vertices. Since bags with y and $X_p \cap Z_A$ and with y and $X_p \cap Z_B$ are already created, there is no need for another bag, and X_p may therefore simply be removed. The path can be finished now. At this point the connected components of the original tree decomposition that were not part of path P can be taken and connected to appropriate bags of the new path P'. Algorithm 1 describes this algorithm in pseudocode. Figure 3.14 displays an example of a graph and its corresponding tree decomposition of which a path P is transformed into a path P'.

Algorithm 1 Algorithm that converts P into P' such that P' contains a bag with both x and y

INPUT: Graph $G[V]$, Separator $\{x, y\}$, Components Z_A, Z_B, Path $P = (X_1, ..., X_p)$ that contains no bag with $\{x, y\}$
OUTPUT: Transformed path P' that does contain a bag with $\{x, y\}$

$P' \leftarrow \emptyset$
$A_1 \leftarrow \{x\}$
$B_1 \leftarrow \{x\}$
for all $\{v \mid v \in (X_1 \cap Z_A)\}$ **do**
    $A_1 \leftarrow A_1 \cup \{v\}$
end for
for all $\{v \mid v \in (X_1 \cap Z_B)\}$ **do**
    $B_1 \leftarrow B_1 \cup \{v\}$
end for
connect(A_1, B_1)
$P'[V] \leftarrow P'[V \cup A_1 \cup B_1]$
for all $\{X_i \mid 2 \leq i \leq (P.length - 1)\}$ **do**
    if $X_i \setminus X_{i-1} \subset Z_A$ **then**
        $A_i \leftarrow \{y\}$
        for all $\{v \mid v \in (X_i \cap Z_A)\}$ **do**
            $A_i \leftarrow A_i \cup \{v\}$
        end for
        connect(A_i, A_{i-1})
        $P'[V] \leftarrow P'[V \cup \{A_i\}]$
    else if $X_i \setminus X_{i-1} \subset Z_B$ **then**
        $B_i \leftarrow \{y\}$
        for all $\{v \mid v \in (X_i \cap Z_B)\}$ **do**
            $B_i \leftarrow B_i \cup \{v\}$
        end for
        connect(B_i, B_{i-1})
        $P'[V] \leftarrow P'[V \cup \{B_i\}]$
    end if
end for

To prove that this algorithm creates a new valid tree decomposition with no increased treecost, a few things need to be verified.

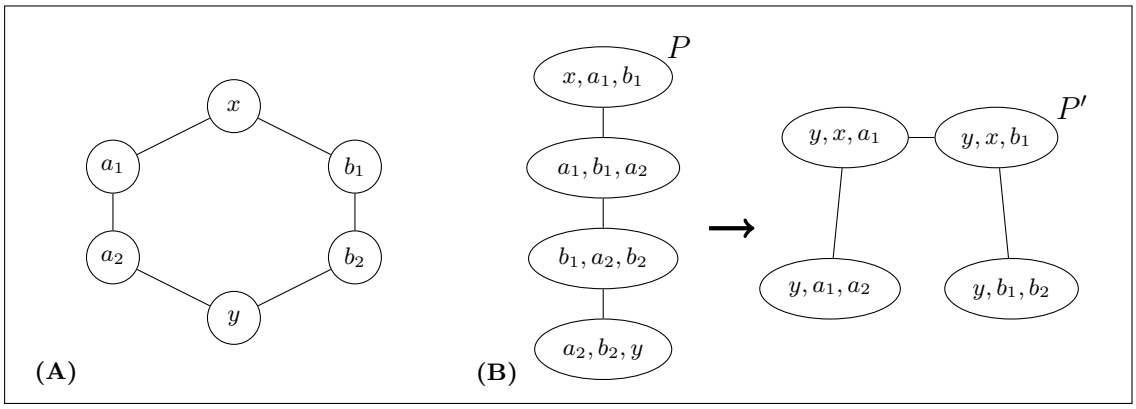

Figure 3.14: **(A)** *Graph with separator* $\{x, y\}$*. Component* Z_A *consists of* a_1 *and* a_2*, and component* Z_B *consists of* b_1 *and* b_2*.* **(B)** *Transformation of path* P *into path* P' *which both comprise the entire tree decomposition. No bag in path* P *contains both* x *and* y *(these vertices are put at the opposite ends of the path).whereas path* P *does contain a bag with both* x *and* y*.*

Lemma 3.34. *Path* P' *is still valid, i.e. for all vertices* v *such that* $v \in X_i$ *for any* $X_i \in P$ *the bags containing* v *are connected.*

Proof. Since x and y are put together in the first bag, no vertices of Z_A need to be put in a bag with vertices of Z_B. And because the order of the original path is followed (only now with Z_A and Z_B being separated), it holds that all adjacent vertices of the graph are put together in the tree decomposition and all bags containing a certain vertex form a connected tree. □

Lemma 3.35. *Path* P' *is not more expensive than* P*.*

Proof. First the cost of X_1 (the first bag of P') is being compared to the sum of costs of $A_1 = (X_1 \cup \{y\}) \setminus (X_1 \cap Z_A)$ and $B_2 = (X_1 \cup \{y\}) \setminus (X_1 \cap Z_B)$. For convenience, we define $A = |X_1 \cap Z_A|$ and $B = |X_1 \cap Z_B|$. The cost of X_1 equals 2^{1+A+B} and the cost of A_1 and B_1 equals $2^{2+A} + 2^{2+B}$. Below, it is proven that in the worst case the latter costs 8 more than the former, given that $1 \leq A$ and $1 \leq B$. More formally: $2^{1+A+B} + 8 \geq 2^{2+A} + 2^{2+B}$. To simplify the equation, both sides are divided by 2 which gives $2^{A+B} + 4 \geq 2^{1+A} + 2^{1+B}$. First this is proven for $A = B = 1$. If this is filled in, we get: $2^{1+1} + 4 \geq 2^{1+1} + 2^{1+1} \Leftrightarrow 8 \geq 8$, which is correct. Now it is proven by means of induction that if A or B is increased, the equation will still hold. If $A \leftarrow A+1$ and $B \leftarrow B$, it is found that on the left side the value of 2^{A+B} is doubled, which causes an increase of 2^{A+B}, whereas on the right side only the value of 2^{1+A} is doubled, which causes an increase of 2^{1+A}. If in this case $B = 1$, than both sides will give an equal raise. As soon as B becomes larger than 1, it is found that the increase on the left side is larger. The case where $A \leftarrow A$ and $B \leftarrow B+1$ is similar. Now it can be derived from this that since the equation holds for $A = B = 1$ and it still holds if either A or B is increased by one, it holds for any value for $1 \leq A$ and $1 \leq B$. Therefore, in the worst case the treecost is increased by 8. Since no increase is preferred, this cost increase must be compensated later.

All bags $(X_2, ..., X_{p-1})$ are now compared to their replacements. The replacement of a bag can never be more expensive, since that bag is a strict subset of the original bag but then with y added. Furthermore, X_p is not replaced by any bag. The cost of X_p is at least 8 (since its size is at least 3), thus this compensates the loss of the first comparison. □

Lemma 3.36. *All remaining components can be connected correctly to the new path, without causing any violation for the tree decomposition.*

Proof. Since the order of path P is followed, it holds that all vertices of this path that are adjacent, will again be put in a bag together. Lemma 3.29 implies that all remaining components may not contain vertices from both Z_A and Z_B. Each component will therefore have a bag in P' that it can connect to. □

Due to the previous lemmas, we can now infer the following about inclusion minimal almost clique separators of size 2 that separate the graph into two connected components.

Lemma 3.37. *Let S be an inclusion minimal almost clique separator of size 2, consisting of x and y that separate the graph into two connected components. Then it is safe to add the edge (x, y) and compute the treecost of the graph by taking the sum of the costs of $Z \cup S$ for both components.*

Proof. This follows directly from the lemmata 3.34, 3.35, 3.36. □

The following lemmas work towards a statement about almost clique separators of size 2 that separate the graph into three components.

Lemma 3.38. *Let G be a graph and let $\{x, y\}$ be an inclusion minimal almost clique separator S that decomposes the graph into three components Z_A, Z_B, and Z_C. This lemma only regards the pair of components $\{Z_B, Z_C\}$ (which is required for the following lemma), although the statement also holds for the pairs $\{Z_A, Z_B\}$ and $\{Z_A, Z_C\}$. Let TD be an optimal tree decomposition of G. Now assume that there is no bag X_i such that $x, y \in X_i$. Then there is a pair of vertices $\{b, c\}$ such that $b \in Z_B$ and $c \in Z_C$, for which an edge can be added, without increasing the treecost.*

Proof. Consider an optimal tree decomposition TD for a graph G where almost clique separator $S = \{x, y\}$ separates G into three components, namely Z_A, Z_B, and Z_C. We now prove that there are two vertices b and c (where $b \in Z_B$) and $c \in Z_C$), such that adding the edge (b, c) will not increase the treecost of G. By Lemma 2.14 we have that for each cycle C such that there is a pair of vertices $x, y \in C$ that are not in a bag together, there must be a bag containing a pair of vertices $a, b \in C$ such that if C would be broken by x and y, then a and b are in different parts of C. The vertices of graph G are part of a cycle, as both components Z_B and Z_C are connected to both x and y. Furthermore, no bag contains x and y, and the vertices b and c are in different parts of the cycle, regarding a partition by x and y. Therefore, there is a bag X_j such that $b, c \in X_j$. And since b and c are put together in a bag, it holds that TD is also a valid (and optimal) tree decomposition for $G + (b, c)$. Hence, the treecost of G equals the treecost of $G + (b, c)$. □

Lemma 3.39. *Let G be a graph with inclusion minimal separator $S = \{x, y\}$ where $G \setminus S$ induces the three components Z_A, Z_B, and Z_C. Then it is optimal to put x and y in a bag together.*

Proof. The tree decomposition of Lemma 3.38 is used to prove Lemma 3.39. By this lemma it holds that adding the edge (b, c) to G, for a certain pair $\{b, c\}$ with $b \in Z_B$ and $c \in Z_C$, does not increase the treecost. However, including this edge causes $G \setminus S$ to induce only two components, since Z_B and Z_C are connected by (b, c). And by Lemma 3.37 it is known that it is optimal to put $\{x, y\}$ in a bag if $\{x, y\}$ forms an almost clique separator that splits the graph into two components. Therefore, it can be concluded that it is also optimal to create a bag that includes x and y when there are three components. □

Although the case where x and y separate the graph into more than three connected components does not occur very often, it still needs to be proven that an almost clique separator of size 2 is always safe, regardless the number of components it seperates. Now we have come to the following theorem.

Theorem 3.40. *Let $S = \{x, y\}$ be an inclusion minimal almost clique separator. Assume that S separates the graph into at least two components. Then it is safe to connect x and y and compute the treecost for $Z \cup S$ for every component Z.*

Proof. By means of the proof of Lemma 3.39 it can be proven by induction that making x and y adjacent and splitting the graph by $G \setminus S$ also works for graphs where $G \setminus S$ induces more than three components. If a separation of z components implies that it is safe to add an edge between x and y, it is also safe for graphs where x and y separate $z + 1$ components. □

3.4.2 Inclusion minimal almost clique separators of size 3

Now almost clique separators of size 3 are being considered. Unfortunately, no general rule can be defined. However, we did create a rule that concern *nearly clique separators.*

Definition 3.41. *A* **nearly clique separator** *is a separator S such that each pair of vertices $v, w \in S$ is adjacent, except for one pair, e.g. $\{x, y\}$. In other words, there is a pair of vertices $x, y \in S$ such that S forms a clique in $G + (x, y)$, but not in G. We call this non-adjacent pair x, y the* **special pair**.

Obviously, all almost clique separators of size 2 are also nearly clique separators. Again it is assumed that S separates the graph into two connected components, Z_A and Z_B.

Lemma 3.42. *Let $S = \{x, y, z\}$ be an inclusion minimal nearly clique separator with the adjacent pairs $\{x, y\}$ and $\{y, z\}$, and with special pair $\{x, z\}$. Then it is safe to add the edge (x, z) and compute the separate costs of $Z \cup S$ for each Z.*

Proof. Again an optimal tree decomposition TD can be taken such that there is no bag with x, y, and z. It is assumed that S separates the graph into two connected components, but by means of making use of Lemma 3.39 and induction it automatically proves the case for more components. Since $(x, y), (y, z) \in E$ there must be at least one bag of x and y and one of y and z. The shortest path is taken from such bags and denoted by $P = (X_1, ..., X_p)$, where $x, y \in X_1$ and $y, z \in X_p$. By following a similar structure as is done in Algorithm 1, this path is transformed into P' such that x, y, and z are in a bag together and the treecost is not increased. Bag X_1 is replaced by adjacent bags $A_1 = \{x, y, z\} \cup (X_1 \cap Z_A)$ and $B_1 = \{x, y, z\} \cup (X_1 \cap Z_B)$. Each bag $X_i = X_2, ..., X_{p-1}$, is replaced by either $A_i = \{y, z\} \cup (X_i \cap Z_A)$ (adjacent to A_{i-1}) or $B_i = \{y, z\} \cup (X_i \cap Z_B)$ (adjacent to B_{i-1}). The following claims are now verified:

- **Path P' is valid.** Each pair $v, w \in P$ such that $(v, w) \in E$ is contained in a bag together in P'. This follows from the fact that no vertex $a \in Z_A$ needs to be put in a bag with a vertex $b \in Z_B$ and pairs of vertices from the same component are not separated.
- **The treecost is not increased.** Since $y \in X_1$ and $y \in X_p$ and the rule that states that all bags that include the same vertex form a connected subtree, it holds that all bags in P contain y. Bag X_1 is replaced by A_1 and B_1, thus these costs should be compared. Let $X_1 \cap Z_A$ be denoted by A and $X_1 \cap Z_B$ by B. The content of bag X_1 can be written as $\{x, y\} \cup A \cup B$. Furthermore, $A_1 = \{x, y, z\} \cup A$ and $B_1 = \{x, y, z\} \cup B$. The costs of X_1 and it replacements A_1 and B_1 equal: $TC(X_1) = 2^{2+|A|+|B|}$ and $TC(A_1) + TC(B_1) = 2^{3+|A|} + 2^{3+|B|}$, where $1 \leq A$ and $1 \leq B$. In the worst case, where one of A and B has a size of 1, e.g. A, then (A_1) has a cost of 16, whereas $TC(B_1) = TC(X_i)$. The replacement thus causes an increase of a cost of 16, In all other cases it is not more expensive to replace X_i by A_1 and B_1. If $|A|$ (or similar, if $|B|$), then the increase of 16 needs to be compensated. As bag X_p contains at least one vertex of Z_A and at least one of Z_B (apart from vertices y and z), its cost is at least 16. And since X_p is not replaced by a new bag, but simply removed, the cost is decreased by at least 16, which compensates for the earlier increase. All bags $(X_2, ..., X_{p-1})$ are replaced by a bag that is never larger than the former, since only one vertex is added an at least one is removed. It can now be concluded that $TC(P') \leq TC(P)$.
- **All remaining components can be connected to the path, without causing a violation.** No components of $TD \setminus P$ contain vertices of both Z_A and Z_B, they can all be safely connected to a bag in path P. □

3.4.3 Inclusion minimal almost clique separators of any size

Again, for almost clique separators of any size, it is not possible to create a rule that safely decomposes the graph into its connected components. However, we claim that every nearly clique separator S is safe for treecost We therefore introduce the following rule.

Rule 11. *Let G be a graph and let S be an inclusion minimal nearly clique, with special pair $\{x, y\}$. Then add the edge (x, y) and split G into its connected components. For each component Z, compute the treecost of $Z \cup S$ and add all these costs together.*

Theorem 3.43. *Reduction rule 11 is safe.*

Proof. Consider an optimal tree decomposition TD of G with no bag X_i such that $S \subseteq X_i$. Then TD can be converted into TD' such that TD' contains a bag with S and $TC(TD') \leq TC(TD)$. Since $S \setminus \{v\}$ forms a clique, there is at least one bag with $S \setminus \{x\}$. Similarly, there is also at least one bag that contains $S \setminus \{y\}$. The shortest path between these bags can be taken, which is denoted by $P = (X_1, ..., X_p)$, where $(S \setminus \{x\}) \subset X_1$ and $(S \setminus \{y\}) \subset X_p$. P is converted into P' in a way that is similar to Algorithm 1, which is as follows. Bag X_1 is replaced by $A_1 = S \cup (X_1 \cap Z_A)$ and $B_1 = S \cup (X_1 \cap Z_B)$. Each bag X_i of $(X_2, ..., X_{p-1})$ is replaced by either $S \cup (X_i \cap Z_A)$ or $S \cup (X_i \cap Z_B)$, depending on which component the currently introduced vertex belongs to. Bag X_p is removed.

Clearly the new tree decomposition is valid, since vertices of Z_A and Z_B that are included in bags on the path, can safely by separated as soon as S is put in a bag. Path P' cannot be more expensive, since $TC(A_1) + TC(A_2) \leq TC(X_1) + 2^{1+|S|}$, $TC(X_p) \geq 2^{1+|S|}$, and $X'_i \leq X_i$ for each $X_i \in \{X_2, ...X_{p-1}\}$. □

3.5 Interesting findings

Due to the earlier proven theorems some other conclusions can be made. Underneath can be found some interesting rules that follow from the earlier rules. These rules may contribute to the reduction of a graph. Some of the structures are easily recognisable and can safely be reduced to a smaller structure.

3.5.1 Cube rule

For treewidth it is proven that if a graph has a subgraph in the form of a cube, this subgraph may be transformed into a smaller one [14], see the Figure 3.15.

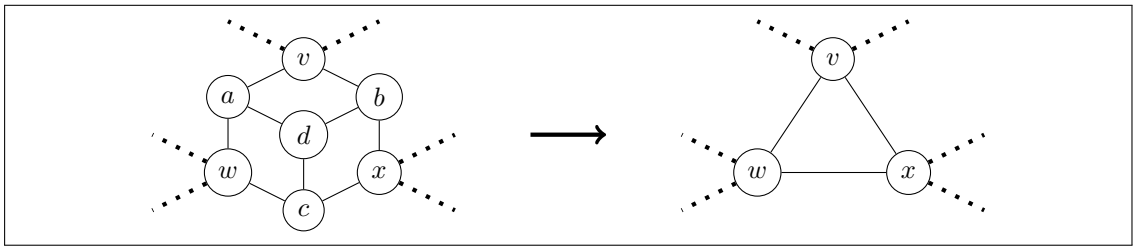

Figure 3.15: Cube rule

Unfortunately, this rule does not hold for treecost, but some other rules for some specific cases of the cube structure can be made. Let G be a graph that contains a set of vertices that forms a cube, as illustrated in Figure 3.15. Every vertex of the set $\{v, w, x\}$ that has no neighbours outside the cube, can simply be removed together with its edges, after creating a bag of its closed neighbourhood and making its two neighbours adjacent. For example, if $deg(v) = 2$ (i.e. v is only adjacent to a and b) then the edge (a, b) can be added. This can easily be verified by the fact that v is a nearly simplicial vertex and it was shown earlier that such a vertex may be removed after connecting its neighbours. For every vertex of the set $\{v, w, x\}$ that has a greater degree than 2 but no path to the other vertices of that set without avoiding the cube, it is safe to split the graph by this vertex. This is the case, since this vertex forms an inclusion minimal clique separator. If for example v has neighbours outside a and b but there is no path from v to w or x that avoids the cube, it holds that v separates the graph into two components. After this

separation, v becomes nearly simplicial in the part that contains the cube and may be treated as was described above. For every pair, constructed of vertices of $\{v, w, x\}$, such that there is a path between its two vertices, that avoids the cube, it holds that these vertices can be made adjacent, if they were not so before. This pair separates the component where the path is contained in, from the rest of the graph and thus it forms an inclusion minimal nearly clique separator of size 2. As was proven earlier, it is safe to separate the graph by this pair and connect them.

The only situation for which no change in the graph can be made is where $G \setminus \{a, b, c, d\}$ creates only one connected component. In other words, if $deg(v) > 2$, $deg(w) > 2$, and $deg(x) > 2$ and there are paths between these vertices that are connected to each other, without passing the cube, the graph cannot be reduced.

For all other options, several reduction rules, which were described above, can be applied to the graph. After the application of these rules, all these options induce (a transformation of) the structure of a cube, forming one entire connected component that contains no other vertices. Therefore, the treecost for this component can easily be computed. If $G \setminus \{a, b, c, d\}$ induces three different components, according to the former rules v, w, and x can be removed and the edges (a, b), (a, c), and (b, c) can be added. This leaves only the clique $\{a, b, c, d\}$ which clearly should be added together in a bag. Figure 3.16 depicts the optimal tree decomposition of this component, with a treecost of 40.

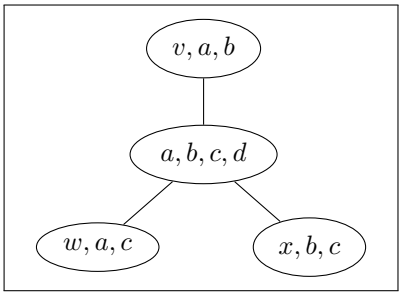

Figure 3.16

If there is no path from v to w or x that avoids the cube, but there is such a path from w to x, then the induced subgraph will have a structure as shown in Figure 3.17.

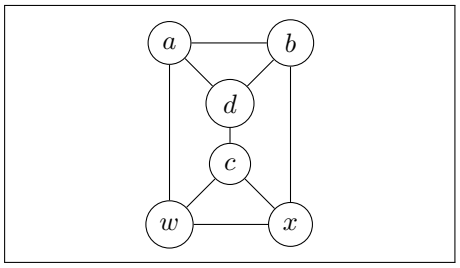

Figure 3.17

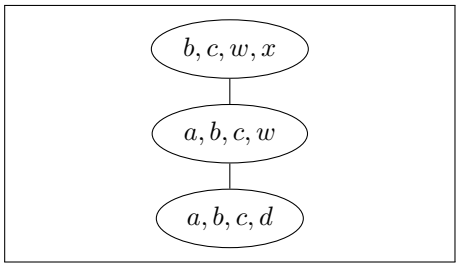

Figure 3.18

In this subgraph, all vertices are identical to each other. Therefore it does not matter which vertex comes first in the elimination order. An optimal tree decomposition can be created as shown in Figure 3.18, with a treecost equal to 48.

If there is a path between v and w that avoids the cube and a path between w and x that avoids the cube, but there exists no such path between v and x, the optimal tree decomposition of the converted subgraph equals 64. Figure 3.19 illustrates the subgraph after the transformation and one of its optimal tree decompositions.

The last situation for which an optimal tree decomposition can be created is where paths exist between v and w, v and x, w and x that avoid the cube and are not connected to each other outside the cube. Figure 3.20 displays the converted subgraph and one of its optimal tree decompositions with a treecost of 64.

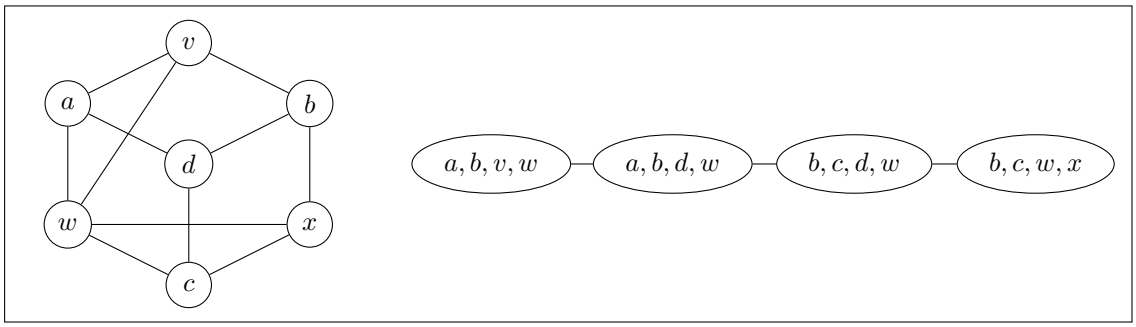

Figure 3.19

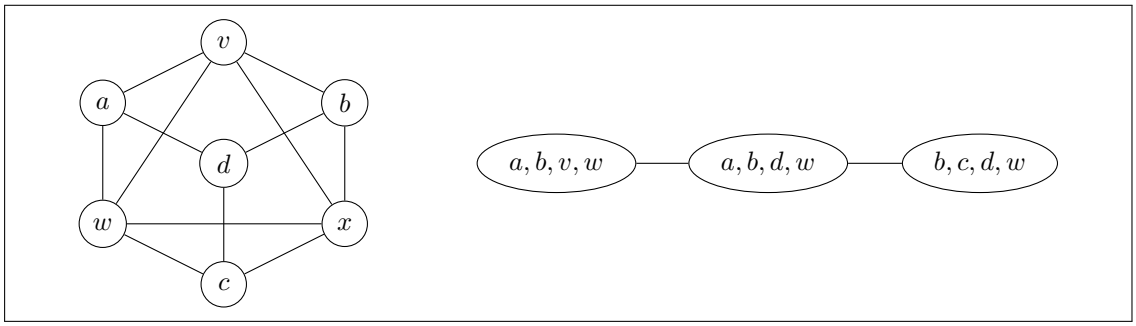

Figure 3.20

3.5.2 Common neighbours

Now the case is considered where two non-adjacent vertices, v and w, share common neighbours. There are situations where it is safe to make them adjacent. This is due to earlier made rules, namely those of safe separators. For convenience, the pair $\{v, w\}$ is considered for several numbers of common neighbours.

If v and w share one common neighbour, e.g. a, it is safe to connect v and w if and only if $\{v, w, a\}$ forms a separator. The reason for this is because $\{v, w, a\}$ forms a nearly clique.

If v and w share two common neighbours, e.g. a and b, and a and b are not adjacent, then it holds that if there is also no path from a to b that avoids v and w, it is safe to make v and w adjacent. This is the case, since v and w form an nearly clique separator of size 2.

Now let v and w share three common neighbours a, b, and c. If no pairs of a, b, and c share an edge or path avoiding v, w, and the other common neighbour, then clearly the edge (v, w) may be added, since v and w form a nearly clique separator of size 2. The same holds for the case where only one pair of their common neighbours shares such a path or edge. Furthermore, if two pairs contain such a path or edge, say a and b, and b and c, then it is also safe to connect v and w, since v, w, and b form a nearly clique separator.

For cases where v and w share more than three common neighbours, it is more complicated to define when it is safe to add the edge (v, w). Let CN be the set of common neighbours of v and v and let $|CN| = c$. The number of pairs of vertices that can be made out of a CN equals $\binom{c}{2}$. Hence, he number pairs out of CN such that its vertices are adjacent or contain a path between them that avoids v, w, and CN, is bounded by $\binom{c}{2}$. We have now arrived to the following lemma.

Lemma 3.44. *Let v and w be two non-adjacent vertices that share c common neighbours, where the set of common neighbours is denoted by CN. If the number of pairs $\{a, b\}$, where $a, b \in CN$,*

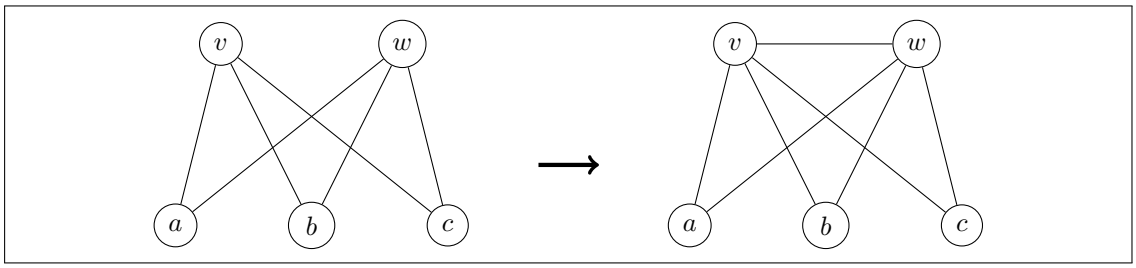

Figure 3.21: Non-adjacent vertices v and w share three common neighbours a, b, and c. Since there is no path between these neighbours that avoids v and w, the edge (v, x) can be added.

such that $(a, b) \in E$ or there is a path that connects them which avoids v, w, and CN, is less than $\binom{c}{2}$, then it is safe to add the edge (v, w).

Proof. The number of pairs that can be made out of CN equals $\binom{c}{2}$, which implies that at least one edge is missing in CN. Let $\{a, b\}$ be that non-adjacent pair that has no path avoiding v, w, and CN, then of $\{v, w\} \cup (CN \setminus \{a, b\})$ separates a and b. Moreover, since al vertices of $\{v, w\} \cup (CN \setminus \{a, b\})$ are adjacent, except for v and w, this set forms a nearly clique. Therefore, v and w can be made adjacent. □

3.6 A pseudo kernel

The reduction rules that are provided so far may cause a very effective decrease of the size of the graph. However, no reduction is guaranteed, since it is possible that none of these rules can be applied to the input graph. An example of a graph that allows no application of our reduction rules is the *Petersen graph* [28]. It would be useful if a reduction method exists, such that a reduction is guaranteed.

A powerful preprocessing technique that can be used for treewidth is *kernelisation* [11, 21], which is a polynomial-time algorithm. Hereby, an instance (G, ℓ) for the treewidth problem is taken (where G is a graph and ℓ is a value for which it is verified whether or not a tree decomposition can be created with a treecost of at most ℓ), whereafter G is reduced to a smaller size, called a *kernel*. The size of the kernel depends on the function of a parameter. Unfortunately, it is not possible to create a polynomial kernel when taking solely ℓ as a parameter, thus there must be relied on other parameter. Examples of kernelisation that are done, are based on the *Vertex Cover problem* (VC) (with a kernel of size $\mathcal{O}(|VC(G)|^3)$) and the *Feedback Vertex Set problem* (FVS) (containing a kernel of size $\mathcal{O}(FVS(G)^4)$) [11, 21].

Finding kernels for treecost is much harder than for treewidth. For instance, the kernels parametrised by VC and FVS are not valid for treewidth. However, we created an algorithm that reduces the input size into something that is similar to a kernel. We call this 'kernel' a *pseudo kernel*. The parameters that are taken are the vertex cover and the independent set (IS) of a graph. In reality, the input size will usually not be (much) smaller, but theoretically speaking, the size will be 'reduced' to a function of the vertex cover and the independent set. More precisely, by using reduction rules for simplicial vertices and pairs with many common neighbours, the size of the graph can be reduced to $\mathcal{O}(|VC|^3 + |VC|^2 \cdot \log(|IS|))$, where $|VC|$ denotes the size of the optimal vertex cover and $|IS|$ the size of the optimal independent set.

Before we continue, the meaning of the parameters are first defined.

Definition 3.45. *The* **vertex cover** *of a graph G is the smallest set of vertices $S \subset G$ such that for all vertices $v \in G$ it holds that either $v \in S$ or $s \in S$ such that $(v, s) \in E$.*

Definition 3.46. *The* **independent set** *of a graph G is the greatest set of vertices $S \subset G$ such that for each pair of vertices $v, w \in S$ it holds that $(v, w) \notin E$.*

An interesting fact is that if VC is the smallest vertex cover of graph G, then for all vertices in G that are not in VC, it holds that they are in IS and vice versa. This immediately highlights a weakness of our kernel, since if n denotes the number of vertices of the original graph, then $|VC| + |IS| = n$.

For the creation of the kernel, we take the graph G, a parameter ℓ, and a vertex cover $X \subseteq G$ as an input. The output of our kernelisation algorithm is (G', ℓ', X'), such that the treecost of G' less than ℓ' if and only if the treecost of G is less than ℓ.

3.6.1 Reduction of simplicial vertices

For every vertex v that is simplicial a bag will be created of only v and its neighbours. Thereafter, v and its incident edges can be removed. Since this bag has a cost of $2^{1+|N[v]|}$, we can subtract this number from the parameter. Since this operation can cause creation of redundant bags in the future, there are some other operations that need to be done. For each w that is a neighbour of v and has the same degree as v: remove w and its incident edges, since all its neighbours are included in the first bag. For each pair of neighbours that has no path avoiding v or its neighbours (apart from the edge connecting them): remove the edge connecting this pair. They are only connected via $N[v]$ and thus $N[v]$ forms a separator for the graphs that connect these pairs. All other vertices and edges from this clique must be retained, since each pair that does have a path avoiding $N[v]$ forms a cycle that avoids v, thus there must exist a bag containing this pair.

Before the simplicial vertices are being reduced, one can search for clique separators in the graph in $\mathcal{O}(mn)$ time. The total treecost equals the sum of the costs of each connected component and the clique separator together.

3.6.2 Many common neighbours

Let vertices v and w, where $(v, w) \notin E$ and $v \in VC$ or $w \in VC$, have x common neighbours. If the tree decomposition contains a bag with v and the common neighbours and a bag with w and the common neighbours, this would cost $2^{1+x} + 2^{1+x}$. This composition is not allowed if the cost is larger than ℓ. Therefore, bags should be created that contain both v and w. Hence, if any pair of vertices v and w where $\{v, w\} \notin E$ have more than $^2\log(\ell) - 2$ common neighbours, add an edge between v and w.

3.6.3 Trivial decomposition

A trivial tree decomposition would be if all vertices of the vertex cover are put in one bag and all other vertices are separately added in a bag together with the vertex cover and linked to the first bag. This would cost $2^k + 2^{k+1} \cdot (n-k)$, where $k = |VC|$. If ℓ is more than this number, we know for sure that the treecost of G is at most ℓ. Consequently, the answer on the question whether a tree decomposition can be found for instance (G, ℓ, X) is *yes*.

Furthermore, the answer to this question is easy to find when one of the following situations occur. If ℓ is less than the lower bound (which is at least $4n-4$ if the graph is connected), then no tree decomposition can be found with a treecost of at most ℓ. Moreover, if ℓ is larger than the upper bound (which is at most 2^n for a simple graph), the a tree decomposition with a treecost of at most ℓ can easily be found.

3.6.4 The kernel

The kernel is very similar to the kernel for treewidth, described in [21]. The remaining graph is denoted by G', the remaining vertex cover by VC', and the remaining independent set by IS'. Since all simplicial vertices are removed, we have that each vertex v of IS' contains at least one pair of vertices $\{w, x\}$ (s.t. $w, x \in VC$) in its neighbourhood that are not adjacent to each other. Now lets assign each vertex v to this pair. As w and x cannot have more than ${}^2\log(\ell)-2$ common neighbours, at most ${}^2\log(\ell)-2$ vertices can be assigned to this pair. Let $VC' \leq VC = k$. Since VC' contains at most k vertices, there can only be at most $\binom{k}{2}$ pairs of $\{w, x\}$. The size of IS' is at most $({}^2\log(\ell)-2) \cdot \binom{k}{2}$. We know that $\ell \leq 2^k + 2^{k+1} \cdot (n-k)$, hence we can fill this in as follows.

$$\begin{aligned}
|IS'| &\leq ({}^2\log(2^k + 2^{k+1} \cdot (n-k)) - 2) \cdot \tbinom{k}{2} \\
&\leq {}^2\log(2^{k+1} \cdot (n-k+1)) \cdot \tbinom{k}{2} \\
&\leq (k + 1 + {}^2\log(|IS'| + 1)) \cdot \tbinom{k}{2} \\
&\in \mathcal{O}((k + {}^2\log(|IS'|)) \cdot k^2) \\
&= \mathcal{O}((k^3 + k^2 \cdot {}^2\log(|IS'|))
\end{aligned}$$

Since the reduced graph G' consists of only vertices of VC' and IS', we have that $n' = |VC| + |IS|$. Hence $n' \in \mathcal{O}(k + k^3 + k^2 \cdot {}^2\log(|IS'|)) = \mathcal{O}(k^3 + k^2 \cdot {}^2\log(|IS'|))$. As one might sees, the size of G' will in reality be less than the outcome of the kernel formula. Therefore, it is not recommended for practical use. However, the idea can be useful for inspiration when further research is done in kernels for treecost.

3.7 Reduction rules for weighted graphs

Until now the treecost of a bag X_i has only been computed by counting the number of vertices (i.e. $|X_i|$) and then taking $2^{|X_i|}$. In probabilistic networks, vertices, which represent variables, can have more than two values. The higher the number of values that a vertex contains, the larger the complexity of the bag that contains that vertex. The number of values is indicated by the weight of a vertex v, denoted by $w(v)$. This weight value is always at least 2, since a variable that has only one value is not interesting and will be left out of the network. The weight of a bag X_i is computed by multiplying the weights of all vertices that are in X_i. In this section we will prove that for weighted graphs, simplicial vertices and inclusion minimal clique separators can be treated in the same way as unweighted graphs. Unfortunately, the rules that concern nearly simplicial vertices and inclusion minimal nearly clique separators, are not valid for weighted treecost.

3.7.1 Simplicial vertices

It was proven earlier that if a vertex v in a unweighted graph G is simplicial, a bag of $N[v]$ may be created whereafter v may be removed to create an optimal tree decomposition. It will now be shown that this also holds for weighted graphs.

Theorem 3.47. *Let G be a graph with simplicial vertex v. Then it is optimal to create a bag of $N[v]$ and remove v and its incident edges. Subsequently, each vertex $w \in N(v)$ for which $deg(v) = deg(w)$ must be removed and for each pair $x, y \in N(v)$ for which there is no path avoiding $N[v]$, except for the edge between them, the edge (x, y) must be removed.*

Proof. Consider any optimal tree decomposition TD for the graph G. Since $N[v]$ forms a clique, there is a bag X_i in TD such that $N[v] \subseteq X_i$. Now it is left to prove that $N[v] = X_i$. Consider a graph $G' = G \setminus \{v\}$. Since $N(v)$ still forms a clique, every tree decomposition of G contains a

bag X_j with $N(v)$. It must be determined that it is never more expensive to add a new bag of $N[v]$ than to add v to X_j. Let $w(v)$ be the weight of v, $w(N(v))$ the product of the weights of all neighbours of v, and W the product of the weights of $X_j \setminus N(v)$. The treecost of both options is compared, i.e. the option where $N[v]$ has its own bag and the option where v is added to X_j. More formally: we compare $TC(X_i)+TC(X_j)$, where $X_i = N[v]$, to $TC(X_k)$, where $X_k = X_j \cup \{v\}$. By filling in the variables for the weights we get $TC(X_i) + TC(X_j) = w(v) \cdot w(N(v)) + w(N(v)) \cdot W$ and $TC(X_k) = w(v) \cdot (N(v)) \cdot W)$. The following can be derived from this.

$$\begin{aligned} & w(v) \cdot w(N(v)) + w(N(v)) \cdot W \leq w(v) \cdot w(N(v)) \cdot W \\ \Leftrightarrow \quad & w(N(v)) \cdot (w(v) + W) \leq w(N(v)) \cdot (w(v) \cdot W \end{aligned}$$

Since all weight values are greater than 2, the left-hand side of the equation can never be larger than the right-hand side. The only case where both sides are equals, is when $w(v) = W = 2$, as in this case we have $w(v) + W = w(v) \cdot W$. In all other cases, the value of the right-hand side is greater. Hereby it is proven that it is never more expensive to create a new bag of $N[v]$. □

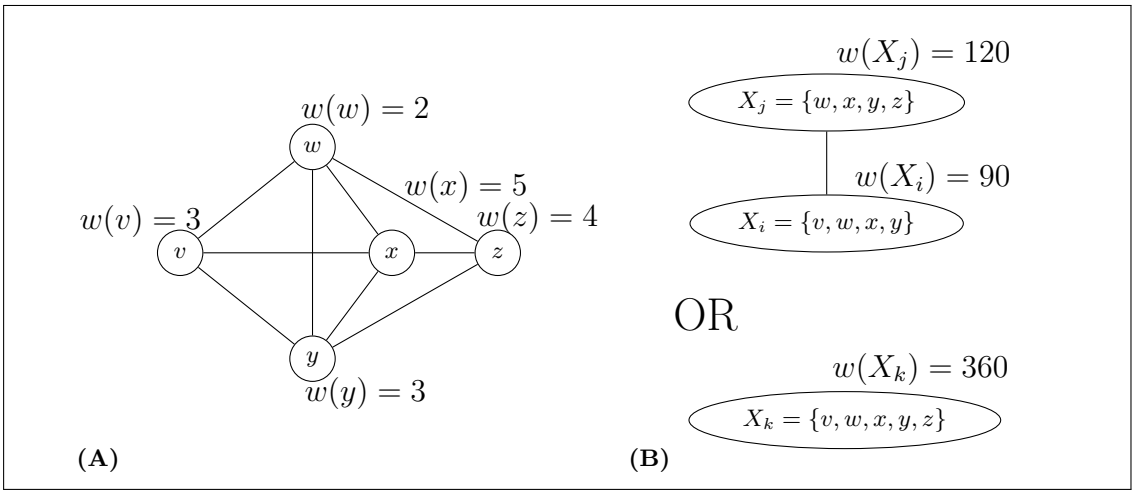

Figure 3.22: **(A)** *Weighted graph with simplicial vertex v.* **(B)** *Two possible tree decompositions with treecosts equal to* $90 + 120 = 210$ *(top) and* 360 *(bottom).*

3.7.2 Inclusion minimal clique separators

For an unweighted graph G with an inclusion minimal clique separator S it is proven that an optimal tree decomposition of G can be computed by taking the sum of the costs of $Z \cup S$ for each connected component Z of $G \setminus S$. This is the case, since no vertices from different components are required in a bag together. Moreover, it is cheaper to create bags of S and one or more vertices from only one component, than to include vertices from several components. It is now shown that clique separators are also safe for weighted graphs.

Theorem 3.48. *Let G be a weighted graph with inclusion minimal clique separator S. Then S is safe for treecost, i.e. it is optimal to compute the separate treecost of $Z \cup S$ for each connected component Z.*

Proof. Consider an optimal tree decomposition TD of graph weighted graph G. As S forms a clique, it holds that there is at least one bag containing S. S is inclusion minimal, which implies that for each component Z, there must be a bag X_i such that $(\{z\} \cup S) \subset X_i$, with $z \in Z$. It is now proven that it is optimal to add no pair of vertices from two different components in a bag with S. Let Z_1 and Z_2 be two different components and let X_1 and X_2 be two bags of the tree

decompositions of $Z_1 \cup S$ and $Z_2 \cup S$ respectively, such that $S \subset X_1$ and $S \subset X_2$. The weight of S is denoted by $w(S)$, the weight of $X_1 \setminus S$ by $w(W_1)$ and the weight of $X_2 \setminus S$ by $w(W_2)$. The total cost of the bags X_1 and X_2 together is now compared to the cost of a bag that may replace X_1 and X_2, namely $X_3 = (X_1 \cup X_2)$. The cost of X_1 and X_2 together equals $w(S) \cdot w(W_1) + w(S) \cdot w(W_2)$ and the cost the new bag X_3 equals $w(S) \cdot w(W_1) \cdot w(W_2)$. Since the equation is similar to the one defined in the proof of Theorem 3.47, we can derive from that proof that it is never cheaper to replace X_1 and X_2 by X_3. □

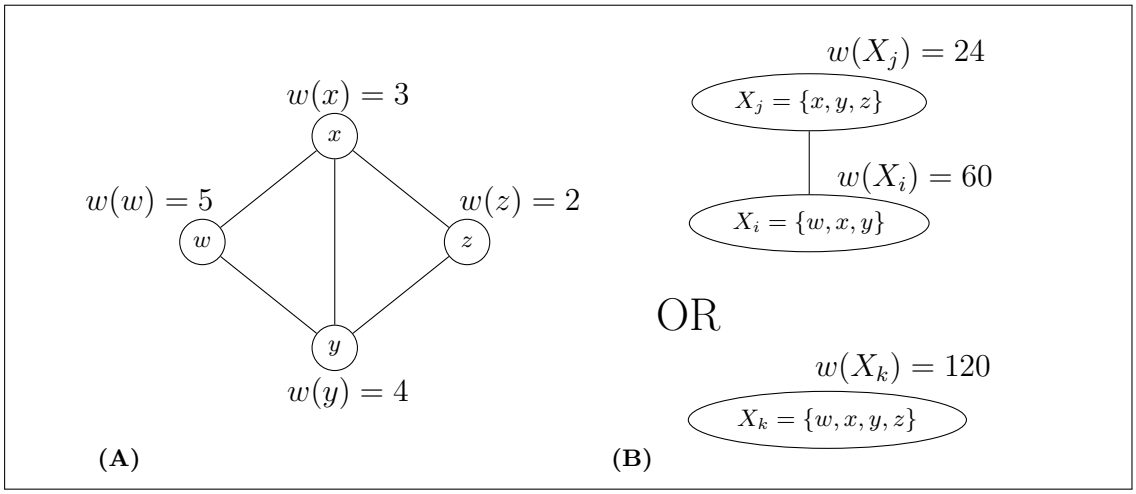

Figure 3.23: **(A)** *Weighted graph with clique separator* $\{x, y\}$. **(B)** *Two possible tree decompositions with treecosts equal to* $60 + 24 = 84$ *(top) and* 120 *(bottom).*

3.7.3 Nearly simplicial vertices and nearly clique separators

It was proven earlier that for unweighted treecost, a nearly simplicial vertex v can be removed after making its special pair adjacent. Unfortunately, no such rule can be defined for weighted graphs. The algorithm which transforms an optimal tree decomposition such that is has a bag that contains $N[v]$ cannot guarantee that the treecost will not increase, as the weight of w can be greater than the weights of the other vertices. Even though the weights of the vertices are known, the algorithm cannot be applied, since the weights of the bags in the original tree decomposition are not known.

The same holds for inclusion minimal nearly clique separators. As was proven earlier for unweighted treecost, that these separators, which only miss one edge to form a clique, are safe. For weighted graphs, no such rule can be defined. The safeness of such separators depends on the weights of the vertices that will occur in bags of the path between the two cliques of the separators of an optimal tree decomposition. Obviously, these vertices are not easily found, as this would imply that part of the tree decomposition is known.

Chapter 4

Experimental Results

In the former chapter we have proven several lemmata and theorems that concern preprocessing for treecost. Moreover, several reduction have been proven to be safe for unweighted treecost. Unfortunately, only a few rules could be found for weighted treecost. Therefore, we chose to solely investigate how the reduction rules work for unweighted treecost. To achieve this, a computational experiment has been conducted to examine the effect of our preprocessing algorithm on several input graphs. Hereby, we analyse how much is reduced from the graph and which rules have more influence than others. Since this thesis focusses on investigating the effect of preprocessing on probabilistic networks, mainly graphs that represent such networks are used as input graphs during the experiment.

4.1 Computational method

This section describes the computational method that implements the reduction rules for the preprocessing of graphs. For the experiment we chose the rules that remove simplicial and nearly simplicial vertices, and rules that split the graph by clique and nearly clique separators (i.e. rules 2, 3, 10, 11), as all other rules are special cases or combinations of these rules. An interesting outcome would be if many input graphs are reduced to the empty graph, since this would imply that computing the treecost of that graph is done in polynomial time. The goal of our experiment is to develop an algorithm that takes an input graph and reduces the number of vertices by removing (nearly) simplicial vertices and separates the graph by (nearly) clique separators. By taking several input graphs, it can be discovered how many vertices and edges remain after the preprocessing. Another interesting aspect that is investigated is how many (nearly) simplicial vertices and (nearly) clique separators are found. Since most of our input graphs are graphs of probabilistic networks, a certain structure of these graphs can be recognised. A probabilistic networks has a directed graph, thus before the reduction rules are applied, it needs to be converted into an undirected graph. This is done by *moralisation*, which generally increases the number of simplicial vertices and clique separators.

First,we have conducted our experiment on sixteen graphs, which are all from probabilistic networks that represent real-life situations. For example, Alarm (anaesthesia monitoring), *Boblo* (blood type identification), *Oesoca* (oesophageal cancer), its two variants: *Oesoca+* and *Oesoca42*, *Pathfinder* (lymphatic disease), *VSD* (prognosis of ventricular septal defect in infants), and *Wilson* (liver disease) are medical applications. Then, networks from other fields are: *Barley* (quality of barley without pesticides), *Mildew* (management of mildew in winter wheat), the three networks *OOW-bas*, *OOW-solo*, and *OOW-trad* (maritime use), *Ship-Ship* (probability of shipship collisions), and *Water* (biological processes of a water purification plant) [13, 14]. Thereafter, a small experiment is done on six graphs that are not from probabilistic networks, namely *CFS*, *Huck*, *Jean*, *Mainuk*, *Myciel3*, and *Myciel4*.

The computational method was implemented in Java and consists of the following steps (where the first step is specifically meant for probabilistic graphs and the latter three can be applied to all types of graphs).

1. The graph G is moralised.

2. The (nearly) clique separator with the minimal largest component is computed. If no such separator can be found, go to 3. Otherwise, for each created component. go to 2
3. The (sub)graph is investigated for (nearly) simplicial vertices, until no such vertex exists any more.
4. The treecost and number of remaining components are outputted, as well as the number of times each reduction rule is applied.

The details of the algorithm are described in the following three subsections.

4.1.1 Removing (nearly) simplicial vertices

If the degree of a vertex is small, the chance that this vertex is simplicial becomes greater and the time it costs to check whether all its neighbours are adjacent becomes smaller. Therefore all vertices are ordered by degree, starting with the smallest degree. This is done in $\mathcal{O}(n \log_n)$ time by using Quicksort [33]. By going through this list of vertices, we then search for the first vertex that is either simplicial or nearly simplicial. Since a bag must be created of the closed neighbourhood of this vertex the current treecost is increased by $2^{|N[v]|}$. If the vertex was nearly simplicial, the missing edge is added. Each neighbour that has no neighbours outside the neighbourhood is removed. Thereafter, it is checked for each pair of neighbours if there is a path between them that avoids the neighbourhood. If not, the edge that connects them is removed. After the removal of a (nearly) simplicial vertex, the next vertex on the list is being checked. The difficult aspect is that as soon as all vertices are checked for simpliciality, new vertices might have become simplicial. Fortunately, this is only possible for neighbours of recently removed vertices. Therefore, upon going through the list, another list is kept which remembers all neighbours of (nearly) simplicial vertices. When this list becomes empty, the graph contains no longer (nearly) simplicial vertices. Since each time a (nearly) simplicial vertex is eliminated only its neighbours are added to the list (with a maximum of $\Delta(G)$ which denotes the maximum degree of the graph), the number of times a vertex is checked for simpliciality is $\mathcal{O}(n \cdot \Delta(G))$. Verifying whether a vertex is (nearly) simplicial costs $\mathcal{O}(\Delta(G)^2)$. After a vertex v is confirmed to be (nearly) simplicial, for all its neighbours it is investigated whether they contains neighbours outside $N[v]$ (which are subsequently removed if this is not the case). This has a complexity of $\mathcal{O}(\Delta(G)^2)$. For each pair of neighbours $\{w, x\}$ we search for a path that avoids $N[v]$ and the edge $\{w, x\}$ (if not, the edge will be removed). The algorithm that computes whether such a path exists may use depth first search (DFS) or breadth first search (BFS) which both uses $\mathcal{O}(n+m)$ time. To sum up, the function which removes all (nearly) simplicial vertices has a complexity of $\mathcal{O}(n \cdot \Delta(G)) \cdot \mathcal{O}(2 \cdot \Delta(G)^2 + (n+m) \cdot \Delta(G)^2)$ or simply $\mathcal{O}((n^2+m) \cdot \Delta(G)^3$.

4.1.2 Finding inclusion minimal (nearly) clique separators

To decompose the graph into its connected components, a (nearly) clique separator needs to be computed. There are several ways to find such separators. The first algorithm that computes all minimal (nearly) clique separators that was implemented for this research lists all minimal separators and then checks which ones are (nearly) cliques [27]. Finding only one minimal separator can be done in polynomial time, where all neighbours of an arbitrary vertex v are taken of which the ones that have no neighbours outside $N[v]$ are removed. From here new separators can be found until a separator is also a (nearly) clique. This however implies that most separators that are used for the reduction of the graph, only separate very few vertices from the rest of the graph, whereas it is more efficient if the largest components of $G \setminus S$ are as small as possible. To compute which minimal nearly (clique) separator has the smallest largest component, all minimal separators of the graph need to be generated. Whereas finding a new minimal separator takes polynomial time, the number of minimal separators of a graph is exponential. For the larger input graphs, this takes too much time and memory. Therefore, a new algorithm had to be found. The point is that the number of minimal separators is exponential in the size of the graph, but the number of minimal clique separators is linear. An easy way to find all minimal clique separators

Algorithm 2 Remove (nearly) simplicial vertices

INPUT: Graph $G[V]$ and current treecost TC
OUTPUT: Reduced graph $G[V]'$ and increased treecost TC'

```
1: V_check ← orderByDegree(V)            ▷ keeps list of vertices that still need to be checked
2: while V_check ≠ ∅ do
3:     v ← removeFirst(V_check)
4:     if simplicial(v) OR nearlySimplicial(v) then
5:         TC ← TC + 2^(1+deg(v))
6:         if nearlySimplicial(v) then
7:             G ← G + (w, x) such that ((w, x) ∉ E AND w, x ∈ N(v))
8:         end if
9:         for all {w | w ∈ N(v), degree(v) == degree(w)} do
10:            V ← V \ {w}
11:            V_check ← V \ {w}
12:        end for
13:        for all {{w, x} | w, x ∈ N(v)} do
14:            if path(w, x) = FALSE then    ▷ if no path between w and x that avoids N[V]
15:                G ← G − (w, x)
16:            end if
17:        end for
18:        V ← V \ {v}
19:    end if
20: end while
```

is to create a minimal tree decomposition and take all intersections of adjacent bags. The minimal tree decomposition is created from a minimal triangulation, which is a chordal graph such that no other chordal graph that is a proper subset of the current one can be created from the graph. Such a triangulation can be computed in various ways of which one is described in [3, 19]. Another method to compute the minimal triangulation is to use the extended variant of the Maximum Cardinality Search algorithm, called MCS-M [4]. In this algorithm, a permutation is created based upon weights of the yet unnumbered vertices. These weights are updated during the process.

We now outline the complexity of the algorithm that generates the separators. The MCS-M algorithm computes a minimal triangulation of a graph in $\mathcal{O}(nm)$ time [4]. Creating the tree decomposition from this permutation takes $\mathcal{O}(n \cdot \Delta(G))$. Consequently, the intersections of adjacent bags must be computed, which represent the minimal clique separators of the graph. There are $\mathcal{O}(n)$ bags of size $\mathcal{O}(TW)$, where TW denotes the treewidth. Since the tree decomposition forms a tree, there are only $\mathcal{O}(n)$ pairs of bags of which the intersection needs to be computed (which in turn takes $\mathcal{O}(TW^2)$ time). Clearly, the number of resulting intersections is bounded by $\mathcal{O}(n)$ as well. Now, a list of minimal separators is created, which includes all minimal clique separators and some non-clique separators. At the end, the minimal separators that do not form a clique, are eliminated from the list.

All minimal clique separators of the graph are now found. However, the nearly clique separators of a graph are also required. The most efficient way that is found so far to find these nearly clique separators, is to compute the intersections of minimal tree decompositions of all alternations of the graph where one missing edge is added. A nearly clique separator will be a clique separator in one of the alternations of the graph. This multiplies the current complexity to list all clique separators by $\mathcal{O}(n^2)$. After all minimal clique separators of an alternation of the graph is found, the added edge will be removed again and it is checked whether the separator is still a (nearly) clique, taking $\mathcal{O}(TW^2)$ time.

Obviously, a separator S is only added if the current list of separators does not contain S yet. Verifying if a separator already exists, requires going through the list of size $\mathcal{O}(n)$ and checking for similarity by comparing the $\mathcal{O}(TW)$ vertices for each separator. Thereafter, all supersets of separators are removed, since adding an edge to the graph may produce minimal separators that are not minimal for the original graph. This is done in $\mathcal{O}(n^2)$ time (note that the number of minimal clique separators equals $\mathcal{O}(n)$). Checking if a separator is already contained in the list, can be done after each run of an alternation of a graph, whereas checking if a separator is a superset of another one should be done after all separators are generated. Finally, for each separator S the size of the largest component of $G \setminus S$ is computed and the separator for which this value is the smallest, is returned. Finding the components for S is done by applying BFS or DFS, both taking $\mathcal{O}(n+m)$ time, which is done for $\mathcal{O}(n)$ separators. Computing for which separator the component value is minimal, takes $\mathcal{O}(n)$ time. Altogether the best minimal (nearly) clique separator is computed in $\mathcal{O}(n^2) \cdot \mathcal{O}(n \cdot m + n \cdot \Delta(G) + 2n \cdot (TW^2)) + \mathcal{O}(n^2) + \mathcal{O}(n \cdot (n+m))$. We abbreviate this notation as $\mathcal{O}(n^3 \cdot (m + TW^2))$.

Algorithm 3 Find best (nearly) clique separator

INPUT: Graph $G[V]$
OUTPUT: Minimal (nearly) clique separator S

```
1: separators ← ∅
2: Π ← MCSC(G[V])
3: TD ← CreateTD(Π)
4: findAllSeparators(TD)
5: for all {(v, w) | (v, w ∈ V, (v, w) ∉ E} do
6:     Π ← MCS(G[V] + (v, w))
7:     TD ← CreateTD(Π)
8:     findAllSeparators(TD)
9: end for
10: for all S ∈ separators do
11:     if (clique(S) = FALSE) AND (nearlyClique(S) = FALSE) then
12:         separators ← separators \ {S}
13:     end if
14: end for
15: for all {{S1, S2} | S1, S2 ∈ separators, S1 ⊂ S2} do        ▷ Remove supersets
16:     separators ← separators \ {S2}
17: end for
18: S ← findBest(separators)
```

4.1.3 Separating by inclusion minimal (nearly) clique separators

As soon as a separator S is returned, the components of $G \setminus S$ need to be computed. This is done by DFS or BFS with a complexity of $\mathcal{O}(n+m)$. If the separator is a nearly clique, the missing edge is added to the graph. To improve the algorithm, the missing edge is memorised and passed on together with the separator.

After determining the components, each component is either searched for new separators or simplicial vertices. As splitting the graph by a separator generally makes the new graphs smaller than removing a simplicial vertex, it is more efficient to run the separator algorithm again. Remembering the list of (nearly) clique separators and assigning them to the components they belong to, would improve the complexity of the algorithm, since finding all separators consumes a significant amount of time. Unfortunately, the list of separators can change due to alternations of the graph, such as: a separator is removed (which may cause a minimal separator to be no longer minimal) or the removal/addition of edges (which may result in new separators). When no more (nearly) clique separators of a component can be found, the graph is searched for (nearly) simplicial vertices.

Algorithm 4 Find all potential clique separators of a graph

INPUT: Tree decomposition TD
OUTPUT: List of separators $S_1, ..., S_s$

```
1: bags ← getRoot(TD)                          ▷ Keeps a list of bags that are currently visited
2: for all bag ∈ TD do
3:     bag.visited = false
4: end for
5: while bags ≠ ∅ do
6:     bag ← bags[0]
7:     bag.visited ← TRUE
8:     for all {n | n ∈ N(bag), n.visited = FALSE} do
9:         bags ← bags ∪ {n}
10:        I ← {v | v ∈, v ∈ n}                 ▷ Computes the intersection of adjacent bags
11:        if I ∉ separators then
12:            separators ← separators ∪ {I}
13:        end if
14:    end for
15:    bags ← bags \ {bag}
16: end while
```

Algorithm 5 Decompose graph by (nearly) clique separators

INPUT: Graph $G[V]$ and current treecost TC
OUTPUT: Set of c components $\{C_1, .., C_c\}$ of $G[V]$, with for each the unchanged treecost of TC

```
1: S ← findCliqueSeparator(G[V])
2: if S = null then removeSimplVertices(G[V], TC)
3: else
4:     components ← ∅
5:     for all {v | v ∈ V, v ∈ S} do
6:         v.visited = TRUE
7:     end for
8:     for all {v | v ∈ V, v ∉ S} do
9:         v.visited = FALSE
10:    end for
11:    for all v ∈ V do
12:        if v.visited = FALSE then
13:            v.visited ← TRUE
14:            C ← findComponent(G[V], v, S)
15:            components ← components ∪ C
16:        end if
17:    end for
18:    for all C ∈ components do
19:        separateGraph(C,TC)
20:    end for
21: end if
```

4.2 Results

The goal of our experiment is to study the effect of preprocessing. Each input graph is first separated into its connected components by the algorithm, until further separation is no longer possible. Then the graph is searched for (nearly) simplicial vertices, which are removed together with potential present neighbours that have no other neighbours outside the (nearly) clique. During the application of these rules, the current treecost is updated and passed on. Furthermore, it is counted how often the four rules (i.e. the reduction rules 2, 3, 10, 11) are being applied. Obviously, if the search for (nearly) simplicial vertices comes before the search for (nearly) clique separators, the number of removed vertices will be larger than with the current order. This is the case since the neighbourhood of a simplicial vertex forms a clique separator and if this is also a minimal clique separator, it fails to be a separator after the removal of the simplicial vertex. If that simplicial vertex is found before its open neighbourhood is found as a separator, the number of simplicial vertices increases by 1, but the number of clique separators is not. In the case where the separator is found first, the number of clique separators is increased, instead of the number of simplicial vertices. The same holds for nearly simplicial vertices and nearly clique separators. After all, changing the order of the rules does not affect the final outcome. At the end of the process, the algorithm returns how much is left of the graph. An interesting outcome of the experiment would be if the reduction rules that we used, would suffice to reduce some input graphs that belong to probabilistic networks, to the empty graph.

Table 4.1 reveals the outcomes of the experiments for the graphs of the probabilistic networks which the reduction rules are applied to. First, the number of vertices and arcs (denoted by $|V|$ and $|A|$ respectively) before the moralisation and the number of edges $|E|$ after the moralisation are given. Thereafter, it is shown how often the rules are employed and what the sizes of the remaining graphs are. With respect to the latter, the results are expressed in terms of the number of components (denoted by $|com|$), vertices, and edges. We first observe that five out of sixteen graphs have no vertices left after the preprocessing algorithm. This implies that no other reduction rules are required for further preprocessing, since there is nothing left that can be reduced. Hence, by solely employing the four reduction rules, an optimal triangulation can be created for these graphs. The graphs that end up empty are the ones that represent *Boblo*, *Oesoca*, *Oesoca42*, *VSD*, and *Wilson*.

A graph that has very few vertices left at the end, is the one of *Alarm*. After the preprocessing, this graph only consists of one component that has six vertices and nine edges. When looking at the adjacent vertices of the graph, we find that its final structure is exactly as the graph shown in Figure 4.1. This graph is exactly the same as the graph in Figure 3.17. Since this graph is relatively small, it is possible to create the corresponding tree decomposition in little time. As is shown in Figure 3.18, the optimal tree decomposition of this graph has a treecost of 48. Therefore, *Alarm* can also be reduced to the empty graph in polynomial time. Another graph that contains few vertices after the preprocessing is the one of *OOW-bas*. At the end, it has two components left, of which one has again the exact same structure as the graph in Figure 4.1. The other component contains nine vertices and nineteen edges and forms a planar graph. Interesting is that the graph is almost chordal, apart from one edge. Thus, only one edge needs to be added to create a graph that has a perfect elimination ordering. This means that, when adding this edge, all remaining vertices can be removed by our reduction rules. The graph of *Munin1* ends up with two components, of which one would have been reduced to the empty graph if the removal of almost simplicial vertices were allowed. When recognising certain patterns as is the case with the graphs of *Alarm*, *OOW-bas*, *Munin1*, we might be able to create a new reduction rule in the future.

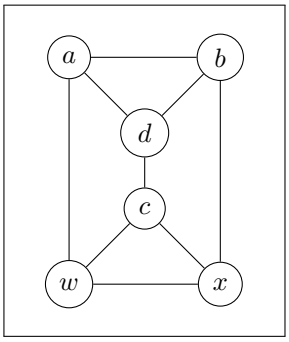

Figure 4.1

Table 4.1: Results of preprocessing for treecost

INSTANCE	Before moralisation		After moralisation		# times applied				Results after preprocessing			
	$\|V\|$	$\|A\|$	$\|V\|$	$\|E\|$	$\|SV\|$	$\|NSV\|$	$\|CS\|$	$\|NCS\|$	$\|com\|$	$\|V\|$	$\|E\|$	TC
ALARM	37	46	37	65	18	3	6	2	1	6	9	216
BARLEY	48	84	48	126	13	3	2	2	1	30	84	240
BOBLO	221	254	221	328	153	34	14	21	0	0	0	1464
MILDEW	35	46	35	80	9	6	2	5	1	14	29	284
MUNIN1	189	282	189	366	68	12	11	4	2	98	227	616
OESOCA+	67	123	67	208	31	7	1	1	1	25	123	1692
OESOCA	39	55	39	67	27	5	4	0	0	0	0	300
OESOCA42	42	59	42	72	26	7	4	0	0	0	0	324
OOW-BAS	27	36	27	54	8	3	2	4	2	15	28	144
OOW-SOLO	40	58	40	87	6	2	3	0	1	30	67	92
OOW-TRAD	33	47	33	72	4	3	2	1	1	24	55	88
PATHFINDER	109	192	109	211	75	6	3	1	1	12	43	792
SHIP-SHIP	50	75	50	114	11	2	4	2	1	33	80	152
VSD	38	52	38	62	26	7	0	1	0	0	0	308
WATER	32	66	32	123	8	1	0	0	1	23	98	192
WILSON	21	23	21	27	15	2	1	0	0	0	0	104

SV = simplicial vertex rule, (NSV) = nearly simplicial vertex rule, (CS) = clique separator rule, (NCS) = nearly clique separator rule, com = components, (TC) = treecost.

The observation that our reduction rules are able to reduce some of the graphs that represent probabilistic networks to the empty graphs is valuable when solving probabilistic inference. Knowing the optimal tree decomposition of a graph entails solving probabilistic inference in minimal time, when using the junction-tree propagation algorithm. And since computing the tree decomposition is NP-hard, it is very profitable if it can be computed in polynomial time for a certain graph. For graphs that do not end up empty, it holds that computing the tree decomposition with minimal treecost cannot be done in polynomial time. But since a large amount of the graph may be removed, it becomes simpler to compute the tree decomposition. This, in turn, makes it easier to solve probabilistic inference on the probabilistic network that the graph represents.

The last value that is displayed by Table 4.1, is the treecost of the bags that are created during the preprocessing (denoted by TC). For the graphs that were reduced to the empty graph, this value equals the treecost of the entire graph, whereas the other graphs only reveal the cost of a part of the graph. As can be observed from the table, the treecost is not always in proportion to the size of the graph. For example, the moralised graph of *Boblo*, which has 221 vertices and 328 edges, only has a treecost of 1464, whereas the smaller moralised graph of *Oesoca+*, with just 67 vertices and 208 edges, has only been reduced to about half of its size, but already has a treecost of 1692. This is caused by the fact that the structure of the graph of *Oesoca+* is more complex, whereby it requires more and larger bags. Because of this complex structure, the graph is also less likely to be reduced to the empty graph, since it contains relatively less simplicial vertices.

4.2.1 Preprocessing for weighted treecost

We further studied how much is reduced of the graph when only employing two rules, namely the simplicial vertex rule and the clique separator rule. Earlier, we argued that weighted treecost is an even more representative parameter to measure the complexity of probabilistic inference than unweighted treecost. Unfortunately, we discovered that of all identified rules for unweighted treecost, only the simplicial vertex rule and the clique separator rule were safe for weighted treecost. We conducted an experiment that only applies these two rules, which would indicate how much will be reduced from graphs while applying preprocessing for weighted treecost. Another purpose for this study is to investigate if the two additional rules (i.e. the nearly simplicial rule and the nearly clique separator rule) are effective, since these can be seen as very specific cases of the almost simplicial vertex rule and the almost clique separator rule. We compare these results to the results of the experiment that allowed the complete set of reduction rules. These results are revealed by Table 4.2, which displays the number of remaining vertices and edges for both sets of rules.

The differences between the remaining graphs after applying two rules and those after applying all four rules, were not very significant. Only the graphs of *Boblo*, *OOW-bas*, *VSD*, and *Wilson* showed significant difference in their results. This implies that for these graphs, the nearly simplicial vertex rule and the nearly clique separator rule are very important. The graphs of *Boblo*, *VSD*, and *Wilson* were even reduced to the empty graphs, due to these additional rules. However, for most graphs, the sizes of the remaining graphs when only using two rules, were only slightly larger than or even equal to the remaining sizes when using four rules. We observed that in many cases, when solely reducing the graph by its simplicial vertices and clique separators, there are only a few nearly simplicial or nearly clique separators left (usually with a size of 2 or 3). After removing these, it occurs sometimes that new simplicial vertices or clique separators emerge. However, in most occasions no more reduction can be achieved. This explains the small difference for most graphs between employing two or four rules. Nevertheless, we can argue that the nearly simplicial vertex rule and the nearly clique separator rule are effective, because in general, a larger part of the graph is reduced in comparison to when two rules are applied and, even in some cases, this difference is surprisingly large. For instance, on average the number of vertices and edges removed by applying two rules is decreased by 58% and 56%, respectively, while by applying the

combination of four rules, 65% of the vertices and 61% of the edges are decreased. In addition, despite that the first pair of percentages entail a smaller reduction, they indicate that preprocessing can also be effective for weighted treecost. To ensure that the graph is reduced significantly, more rules for weighted graphs should be determined, if possible. When counting the number of rules that were used, we observed that the simplicial vertex rule was applied most often. This was expected, since moralisation makes all predecessors of a vertex adjacent, which may induce many simplicial vertices.

Table 4.2: Comparing results of application of two different sets of rules

INSTANCE	before preprocessing		after preprocessing			
			SV and CS		SV, NSV, CS, and NCS	
	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$
ALARM	37	65	7	11	6	9
BARLEY	48	126	30	84	30	84
BOBLO	221	328	40	56	0	0
MILDEW	35	80	15	31	14	29
MUNIN1	189	366	104	237	98	227
OESOCA+	67	208	26	129	25	123
OESOCA	39	67	0	0	0	0
OESOCA42	42	72	0	0	0	0
OOW-BAS	27	54	21	39	15	28
OOW-SOLO	40	87	31	68	30	67
OOW-TRAD	33	72	26	58	24	55
PATHFINDER	109	211	14	49	12	43
SHIP-SHIP	50	114	35	84	33	80
VSD	38	62	6	12	0	0
WATER	32	123	23	98	23	98
WILSON	21	27	6	8	0	0

4.2.2 Preprocessing for graphs outside probability theory

Our main purpose was to investigate how well our reduction rules performed on graphs that represent probabilistic networks. However, we also wanted to have an impression of how effective these rules are on other graphs. Table 4.3 reports the results of our preprocessing algorithm, applied to six undirected graphs that are not from probabilistic networks (namely *CFS*, *Huck*, *Jean*, *Mainuk*, *Myciel3* and *Myciel 4*). The results that show what is reduced from the graphs are very divergent. For example, the graphs of *CFS*, *Huck*, *Jean*, and *Mainuk*, show that a reasonable number of vertices and edges are removed due to the preprocessing, e.g. between 52% and 85%, and between 60% and 84%, respectively. The relative small graphs of *Myciel3* and *Myciel 4*, on the other hand, show no reduction at all. None of the graphs are reduced to the empty graph. Therefore, we conclude that our reduction rules are more effective on graphs that correspond to probabilistic networks.

4.2.3 Comparing our results to preprocessing for treewidth

In the introduction we argued that treecost gives a better representation of the optimality of a tree decomposition than treewidth. However, we discovered that many reduction rules used for

Table 4.3: Results of reduction rules applied to graphs that are not of probabilistic networks

INSTANCE	before preprocessing		after preprocessing			
	$\|V\|$	$\|E\|$	$\|com\|$	$\|V\|$	$\|E\|$	TC
CSF	32	94	1	10	33	428
HUCK	74	301	1	11	49	5032
JEAN	80	254	2	24	94	3116
MAINUK	48	198	2	23	80	2000
MYCIEL3	11	20	1	11	20	0
MYCIEL4	23	71	1	23	71	0

treewidth do not apply for treecost, e.g. the rule that removes almost simplicial vertices. Therefore, it was expected that the rules that we defined for treecost would reduce less of a graph than the rules that are provided for treewidth. To compare the sizes of our remaining graphs to those that were reduced by rules for treewidth, we analyse the results of the experiment that is conducted in [13]. In this paper, the input graphs are reduced by the following rules: the simplicial vertex rule, the almost simplicial vertex rule, the buddy rule (which removes pairs of vertices that have three common neighbours and subsequently adds edges between the non-adjacent pairs of these neighbours), the cube rule, the extended cube rule (similar to the cube rule, with the only difference that the vertex in the middle may have neighbours outside the cube). Among these, the following rules are not valid for treecost: the almost simplicial vertex rule, the buddy rule, the cube rule (which we have only proven to be valid in specific cases), and the extended cube rule. Clearly, when using all these rules, much more of a graph may be reduced than when only using our four reduction rules. The set of rules for treewidth, which were identified in the paper, does not include safe separators. Hereby, the remaining graph will always consists of only one component. This might be a drawback of the experiment, since splitting the graph by a separator may induce new simplicial (or almost simplicial) vertices.

In Table 4.4 the results of both preprocessing algorithms are displayed. From the values in the table we can derive that, despite the fact that the number of vertices and edges are not equal for many input graphs, the differences are only small. For instance, the average number of vertices and edges that were removed by preprocessing for treecost equals 65% and 61% of the initial graph, respectively, while in the case of treewidth, these numbers equal 77% and 71%, respectively. Furthermore, the minimum decrease in vertices and edges equals 25% and 20% for treecost, and 30% and 22% for treewidth, respectively. The reduction rules that were used for treewidth reduced eight graphs to the empty graph. The rules that we provided for treecost could remove all vertices of only five of these graphs. This means that three of them did not end up empty in our experiment. One of them is the graph for *Alarm*, of which the remainder had a structure that was small enough to reduce empty in little time, as was discussed earlier. The other two are the graphs for *OOW-bas* and *Mildew*, which both contained a component with a relative simple structure and therefore could be removed further, by means of reduction rules that might be determined in the future. In [13] it is shown that these two graphs are only reduced to the empty graph if the almost simplicial vertex rule is applied, a rule that is not valid for treecost. Hence, it seems that this rule is very profitable when reducing graphs of probabilistic networks. In addition, the nearly simplicial vertex rule is also useful, but less strong than the almost simplicial vertex rule.

Table 4.4: Results compared to preprocessing of treewidth

INSTANCE	before preprocessing		after preprocessing			
			rules for TC		rules for TW	
	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$
ALARM	37	65	6	9	0	0
BARLEY	48	126	30	84	26	78
BOBLO	221	328	0	0	0	0
MILDEW	35	80	14	29	0	0
MUNIN1	189	336	98	227	66	188
OESOCA+	67	208	25	123	14	75
OESOCA	39	67	0	0	0	0
OESOCA42	42	72	0	0	0	0
OOW-BAS	27	54	15	28	0	0
OOW-SOLO	40	87	30	67	27	63
OOW-TRAD	33	72	24	55	23	54
PATHFINDER	109	211	12	43	12	43
SHIP-SHIP	50	114	33	80	24	65
VSD	38	62	0	0	0	0
WATER	32	123	23	98	22	96
WILSON	21	72	0	0	0	0

Chapter 5

Discussion and Conclusion

The primary goal of this thesis was to provide a preprocessing algorithm that reduces the size of graphs that correspond to probabilistic networks. We focus on probabilistic inference, which is a well-known and (NP-)hard problem in the field of probability theory. This problem concerns the process of computing the probability distribution of variables given the evidence of other variables. To ensure that the time to solve this problem is minimal, an optimal tree decomposition is required. We chose to measure tree decompositions by means of the parameter treecost, which can give a good indication of the time required to process the computation of all probabilities. Since our preprocessing technique decreases the complexity of computing the minimal treecost of a graph, probabilistic inference will be solved more quickly.

5.1 General findings and research contribution

We have provided and proven several reduction rules that decrease the size of graphs. To answer [SQ.1], we have studied the rules that were determined for treewidth. The most important, yet trivial, that we considered, concern simplicial vertices and inclusion minimal clique separators. Two less trivial rules are the nearly simplicial vertex rule and the nearly clique separator rule, where "nearly" entails that only one edge is missing. These four rules together comprise the set of reduction rules that we applied on a set of input graphs during our experiment. These graphs mainly contains representations of probabilistic networks. The results of this experiment show that most of the input graphs were significantly decreased in size. Some graphs were even reduced to the empty graph, which implies that an optimal tree decomposition can be computed solely by preprocessing. Since there are more reduction rules that apply for treewidth than there are for treecost, we also studied the difference between preprocessing for both these parameters. Subsequently, we noticed that the largest difference was caused by the fact that treewidth allows the removal of almost simplicial rule, whereas treecost does not. Since the nearly simplicial vertex rule is a special case of the almost simplicial vertex rule, it was no surprise that the former rule removed less vertices than the latter rule. Hereby, we have answered [SQ.2].

Other rules that were introduced concern the cube structure and the set of common neighbours of a pair of vertices. Whereas the cube rule and the common neighbour rule are valid for treewidth, these rules could only be verified for special cases. These cases were proven by making use of earlier provided rules, such as the simplicial vertex rule and the clique separator rule.

Finally, we have investigated which rules could be defined for weighted treecost. Since several variables in probabilistic networks contain more than two values, weighted treecost would be a more representative measure for the complexity than unweighted treecost. Unfortunately, we found that only two rules applied for weighted treecost. The only two rules that were proven to be valid, are the simplicial vertex rule and the clique separator rule. The results show that the input graphs were reduced to a smaller graph, despite the fact that the reduction was not as effective as for unweighted treecost. Moreover, it was discovered that some graphs were only reduced to the empty graph, when the removal of nearly simplicial vertices and the graph separating by nearly clique separators was allowed.

Subsequently, we discuss the effect of our reduction rules on a set of input graphs to answer [SQ.3]. Overall, we conclude that our preprocessing method is profitable for probabilistic inference, since many real-life probabilistic networks were reduced significantly. For the graphs that were reduced to the empty graph, an optimal tree decomposition can be computed in polynomial time. This entails that solving probabilistic inference by means of the junction-tree propagation algorithm can be realised in minimal time. Although this thesis focusses mainly on probabilistic inference, we also conducted an experiment during which our preprocessing algorithm was applied to graphs that do not correspond to probabilistic networks. We observed that our rules removed less vertices from those graphs. None of the graphs were reduced to the empty graph and even two of the six graphs had nothing removed at all. Therefore, to employ reduction rules on graphs that do not represent probabilistic networks, more rules need to be defined, since our reduction rules are not effective enough for these graphs.

5.2 Future research

Compared to treewidth, the number of reduction rules that are safe for treecost is much smaller. Many rules that are used for treewidth, are no longer valid when computing the minimal treecost. It would be an interesting research to investigate if more rules can be defined for treecost, in addition to the rules described in this thesis.

Initially, we envisioned to find a kernelisation algorithm, such that the complexity of computing the treecost of a graph is guaranteed to decrease. Unfortunately, no effective kernel was found, apart from a "pseudo-kernel" that does not decrease the input size in practice. Therefore, parametrised kernelisation can be an interesting topic for future study.

Another consideration concerned applying aspects from spanning trees with many leaves. For instance, the research by [26] provides lower bounds and upper bounds for treewidth, given the number of leaves of a spanning tree of the graph. Computing a strict upper bound for treecost, would lead to new preprocessing techniques. However, no such bounds were found during this thesis project. Nevertheless, it might be useful to consider investigating this topic in the future again.

Finally, as has been clarified before, the nearly simplicial vertex rule and nearly clique separator rule do not apply for weighted treecost. Since we have only proven two rules to be valid for weighted treecost, future research can further investigate whether some effective rules can be created such that the weights of the graph are taken into account. This might lead to a much more optimal preprocessing algorithm for the probabilistic inference problem.

Bibliography

[1] Emgad Bachoore and Hans L Bodlaender. Weighted treewidth algorithmic techniques and results. In *Algorithms and Computation*, pages 893–903. Springer, 2007. 3

[2] C Berge and P Duchet. Strongly perfect graphs. *Annals of Discrete Mathematics*, 21:57–61, 1984. 7

[3] Anne Berry. A wide-range efficient algorithm for minimal triangulation. In *SODA*, volume 99, pages 860–861. Citeseer, 1999. 35

[4] Anne Berry, Jean RS Blair, Pinar Heggernes, and Barry W Peyton. Maximum cardinality search for computing minimal triangulations of graphs. *Algorithmica*, 39(4):287–298, 2004. 35

[5] Anne Berry, Romain Pogorelcnik, and Genevieve Simonet. An introduction to clique minimal separator decomposition. *Algorithms*, 3(2):197–215, 2010. 9

[6] Hans L Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 226–234. ACM, 1993. 2

[7] Hans L Bodlaender. A tourist guide through treewidth. *Acta cybernetica*, 11(1-2):1, 1994. 2

[8] Hans L Bodlaender. *Treewidth: Algorithmic techniques and results*. Springer, 1997. 2

[9] Hans L Bodlaender. Treewidth: characterizations, applications, and computations. In *Graph-theoretic concepts in computer science*, pages 1–14. Springer, 2006. 2

[10] Hans L Bodlaender and Fedor V Fomin. Tree decompositions with small cost. In *Algorithm TheorySWAT 2002*, pages 378–387. Springer, 2002. 2

[11] Hans L Bodlaender, Bart MP Jansen, and Stefan Kratsch. Preprocessing for treewidth: A combinatorial analysis through kernelization. *SIAM Journal on Discrete Mathematics*, 27(4):2108–2142, 2013. 2, 28

[12] Hans L Bodlaender and Arie MCA Koster. Safe separators for treewidth. *Discrete Mathematics*, 306(3):337–350, 2006. 2, 10

[13] Hans L Bodlaender, Arie MCA Koster, and Frank van den Eijkhof. Preprocessing rules for triangulation of probabilistic networks*. *Computational Intelligence*, 21(3):286–305, 2005. 2, 33, 42

[14] Hans L Bodlaender, Arie MCA Koster, Frank van den Eijkhof, and Linda C van der Gaag. Pre-processing for triangulation of probabilistic networks. In *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*, pages 32–39. Morgan Kaufmann Publishers Inc., 2001. 25, 33

[15] Hans L Bodlaender and Rolf H Möhring. The pathwidth and treewidth of cographs. *SIAM Journal on Discrete Mathematics*, 6(2):181–188, 1993. 6

[16] Vincent Bouchitt and Ioan Todinca. Treewidth and minimum fill-in: Grouping the minimal separators, 1999. 9

[17] Gregory F Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial intelligence*, 42(2):393–405, 1990. 1

[18] Gregory Floyd Cooper. Nestor: A computer-based medical diagnostic aid that integrates causal and probabilistic knowledge. Technical report, DTIC Document, 1984. 1

[19] Dias Dahlhaus. Minimal elimination ordering inside a given chordal graph. In *Graph-Theoretic Concepts in Computer Science*, pages 132–143. Springer, 1997. 35

[20] Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 201–208. AUAI Press, 2004. 6

[21] BMP Jansen et al. The power of data reduction: Kernels for fundamental graph problems. 2013. 2, 3, 28, 30

[22] Finn V Jensen. Bayesian networks basics. *AISB quarterly*, pages 9–22, 1996. 1

[23] Finn V Jensen and Frank Jensen. Optimal junction trees. In *Proceedings of the Tenth international conference on Uncertainty in artificial intelligence*, pages 360–366. Morgan Kaufmann Publishers Inc., 1994. 1

[24] Uffe Kjærulff. Triangulation of graphs–algorithms giving small total state space. 1990. 2

[25] Uffe Kjærulff. Optimal decomposition of probabilistic networks by simulated annealing. *Statistics and Computing*, 2(1):7–17, 1992. 2

[26] Daniel J Kleitman and Douglas B West. Spanning trees with many leaves. *SIAM Journal on Discrete Mathematics*, 4(1):99–106, 1991. 45

[27] Ton Kloks and Dieter Kratsch. Listing all minimal separators of a graph. *SIAM Journal on Computing*, 27(3):605–613, 1998. 34

[28] S Chandra Kumar and T Nicholas. b-continuity in peterson graph and power of a cycle. *International Journal of Modern Engineering Research*, 2:2493–2496, 2012. 28

[29] Hanns-Georg Leimer. Optimal decomposition by clique separators. *Discrete mathematics*, 113(1):99–123, 1993. 9

[30] Jiří Matoušek and Robin Thomas. Algorithms finding tree-decompositions of graphs. *Journal of Algorithms*, 12(1):1–22, 1991. 1, 2

[31] Thomas Dyhre Nielsen and Finn Verner Jensen. *Bayesian networks and decision graphs*. Springer Science & Business Media, 2009. 1

[32] Thorsten J Ottosen and Jirı Vomlel. All roads lead to romenew search methods for optimal triangulation. *on Probabilistic Graphical Models*, page 209, 2010. 2

[33] Christos H. Papadimitriou. Computational complexity. In *Encyclopedia of Computer Science*, pages 260–265. John Wiley and Sons Ltd., Chichester, UK. 34

[34] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 2014. 1

[35] Neil Robertson and Paul D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of algorithms*, 7(3):309–322, 1986. 6

[36] Donald J Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32(3):597–609, 1970. 5

[37] Donald J Rose, R Endre Tarjan, and George S Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on computing*, 5(2):266–283, 1976. 5, 7

[38] Ross D Shachter. Probabilistic inference and influence diagrams. *Operations Research*, 36(4):589–604, 1988. 1

[39] Robert E Tarjan. Decomposition by clique separators. *Discrete mathematics*, 55(2):221–232, 1985. 9

[40] MER van Boxel. Improved algorithms for the computation of special junction trees. 2014. 2

[41] Frank Van Den Eijkhof and Hans L Bodlaender. Safe reduction rules for weighted treewidth. In *Graph-Theoretic Concepts in Computer Science*, pages 176–185. Springer, 2002. 3