Bachelor Thesis

Matching-based Algorithms for Computing Treewidth

Moses Ganardi

Supervisor: Prof. Dr. Arie M.C.A. Koster
Lehr- und Forschungsgebiet Diskrete Optimierung
Rheinisch-Westfälische Technische Hochschule Aachen

Aachen, October 2012

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt sowie Zitate kenntlich gemacht habe.

Aachen, den 5. Oktober 2012

(Moses Ganardi)

Contents

1 Introduction

The notions of *treewidth* and *tree decompositions* have attained a lot of attention in the past thirty years. Tree decompositions provide a good answer to how to deal with NP-complete graph problems. Many graph problems are easy to solve when one restricts the input graphs to be trees. This observation leads to the idea of regarding graphs that "resemble" trees, which can for example be measured by treewidth. It turns out that a lot of NP-complete problems become linear-time solvable when restricted to graphs of bounded treewidth.

These algorithms usually use a dynamic programming approach on a tree decomposition, which is a tree-like structure on the input graph. A graph can have several different tree decompositions and one is interested in finding tree decompositions of small *width* such that the dynamic programming can perform fast. However, there is no polynomial-time algorithm known which computes an optimal tree decomposition, i.e. having minimal width, for a given graph.

In 1996, Bodlaender presented an algorithm which for every fixed k computes in linear-time a tree decomposition of width at most k for the input graph, or outputs that no such exists. From a practical viewpoint this result is not useful since the algorithm performs very slow as experiments have indicated. In practice it makes more sense to use non-optimal tree decomposition heuristics. The advantage is their speed and also often the computed width is close to optimal. Bodlaender and Koster give a good overview of such upper bound algorithms in [BK10].

In this thesis, we want to reuse just one of the ideas in Bodlaender's algorithm and ignore the other rather complicated parts of it. At some point in the algorithm, the input graph is reduced to a smaller graph which is obtained by contracting a maximal matching. Then the algorithm is called recursively on the reduced graph and we gain a tree decomposition of it, say of width k. One can easily transform this tree decomposition into a tree decomposition of the original graph of width at most $2k+1$. We will study how this technique can be used in tree decomposition algorithms and examine its performance in combination with another tree decomposition heuristic. Instead of using mere maximal matchings, we will test several algorithms that compute matching contractions. We are especially interested in matching contractions that cause a large decrease of the treewidth of the input graph.

This thesis is structured as follows: In Chapter 2 we will start with basic defintions from graph theory and afterwards introduce algorithms that are known for computing tree decompositions: Bodlaender's algorithm and two heuristics based on so called triangulations. We also present a post-processing method to further improve computed tree decompositions. In Chapter 3 we will introduce the main idea of this thesis how to use matching contractions to compute tree decompositions and motivate why this approach is reasonable. We will also discuss problems and limitations of this approach; in particular we study the question whether there exists a polynomial-time algorithm that minimizes the treewidth by a matching contraction. In Chapter 4 we will then present various matching types for the main algorithm and show results from our experimental evaluation using these types of matchings. In Chapter 5 we will summarize the results and mention possible improvements and extensions for our approach.

2 Algorithms for Computing Tree Decompositions

2.1 Preliminaries

We start by recapitulating basic definitions and notations from graph theory. Further notions will be introduced in the particular chapters where needed.

In this thesis we consider simple graphs $G = (V, E)$, i.e. finite, undirected graphs without loops or parallel edges. The edge set E contains undirected edges $\{v, w\}$ where $v, w \in V$ and $v \neq w$. For a graph G we write $V(G)$ and $E(G)$ for the vertex and edge set of G, or simply V and E if the referred graph is arbitrary or clear from the context. Usually we denote by n or by $|G|$ the number of vertices of a graph G. We denote the neighbourhood and the degree of a vertex v by $N_G(v)$ and $\deg_G(v)$ respectively; the closed neighbourhood of v is denoted by $N_G[v] = N_G(v) \cup \{v\}$. Often we omit the subscripts in this notation if the graph is clear from the context. We write $\omega(G)$ for the maximal clique size of G.

Removing a vertex is the operation that deletes the vertex and all edges incident to it. For a set of vertices $W \subseteq V$ we denote by $G - W$ the graph obtained from G by removing all vertices in W. *Contracting an edge* $\{u, v\} \in E$ is the operation that introduces a new vertex w with neighbourhood $N(w) = (N(u) \cup N(v)) \setminus \{u, v\}$ and afterwards removes u and v. G is called *minor* of H if there is a graph isomorphic to G that can be obtained from a subgraph of H by contracting edges. For a set of edges $M \subseteq E$ we denote by G/M the *contracted graph* which is obtained from G by contracting all edges in M.

A *matching* of G is a set of edges $M \subseteq E$ such that no two edges in M share an endpoint. A matching M is called *maximal* if there is no matching $M' \supsetneq M$, i.e. every edge $e \in E$ shares an endpoint with an edge in M. A matching M is called a *maximum (cardinality) matching* if there is no matching M' of G with $|M'| > |M|$.

Finally we introduce tree decompositions and treewidth, which became famous by Robertson and Seymour in a long series of papers [RS86]. A *tree decomposition* of a graph $G = (V, E)$ is a pair $\mathcal{T} = (\{X_i\}_{i \in I}, T = (I, F))$ consisting of a family of subsets $X_i \subseteq V$ and a tree T satisfying the following conditions:

(i) for all vertices $v \in V$, there is an $i \in I$ with $v \in X_i$,

(ii) for all edges $\{v, w\} \in E$, there is an $i \in I$ with $v, w \in X_i$ and

(iii) for all $i, j, k \in I$: if j is on the path from i to k in T, then $X_i \cap X_k \subseteq X_j$.

The *width* of a tree decomposition $\mathcal{T}$ is $\max_{i \in I} |X_i| - 1$. The *treewidth* of a graph G, denoted by $\text{tw}(G)$, is the minimum width of a tree decomposition of G. The subsets X_i are often called *bags*. To distinguish vertices $i \in I$ of the tree from vertices $v \in V$ of the graph G, we will call the vertices $i \in I$ *nodes*.

2.2 Bodlaender's algorithm

Arnborg et al. proved in 1987 that the decision problem Treewidth "Given a graph G and a number k, does G have treewidth $\leq k$?" is NP-complete [ACP87]. Hence, we cannot expect that there is an polynomial-time algorithm which computes tree decompositions of minimal width. In 1996, Bodlaender showed that the problem becomes tractable if we fix the parameter k.

Theorem 1 ([Bod96]). *For every $k \in \mathbb{N}$ there is a linear-time algorithm that decides whether a given graph G has treewidth at most k, and if so, outputs a tree decomposition of G with width at most k.*

We will not present the algorithm in its entirety. The algorithm distincts two cases, depending on a certain property of the graph. In one of the cases the following procedure is executed:

1. Obtain G' from the input graph G by contracting a maximal matching.
2. Call the algorithm recursively on G'. Either we obtain a tree decomposition $\mathcal{T}'$ of G' of width k, or we get $\text{tw}(G') > k$ (then we can also output $\text{tw}(G) > k$ and terminate).
3. Convert $\mathcal{T}'$ into a tree decomposition $\mathcal{T}$ of G of width $2k + 1$.
4. Use $\mathcal{T}$ to compute a tree decomposition of G of width k, or output that $\text{tw}(G) > k$.

The last step uses another linear-time algorithm by Bodlaender and Kloks, which, for constants k and l, converts a tree decomposition of width $\leq k$ to a tree decomposition of width $\leq l$ or returns that this is not possible [BK96]. The interest of this thesis lies in the former three steps. In particular, we will present step 3 in the next chapter.

Bodlaender's theorem is an important theoretical result. Combining it with Courcelle's theorem [Cou90], we obtain linear-time algorithms for a large family of NP-hard problems on graphs with bounded treewidth. However, the “hidden constants” in the running time analysis of Bodlaender’s algorithm are very large (due to the mentioned subroutine in step 4) such that this algorithm is unusable in practice, even for small values of k [Rö98]. In a practical setting we should switch to heuristics which compute (non-optimal) tree decompositions with acceptable running times.

2.3 Greedy triangulation algorithms

We will present heuristics that compute tree decompositions based on so called triangulations of graphs. At first we need to introduce further notions.

A graph is *chordal* (also *triangulated*) if every cycle of length at least four has a *chord*, i.e. an edge connecting two non-consecutive vertices of the cycle. A graph $H = (V, F)$ is called *triangulation* (also *chordalization*) of $G = (V, E)$ if $E \subseteq F$ and H is chordal. An *elimination ordering* of a graph $G = (V, E)$ is a bijection $\pi : V \to \{1, \dots, n\}$. An elimination ordering π is *perfect* if for all $v \in V$ the set of higher numbered neighbours $\{w \in N(v) : \pi(w) > \pi(v)\}$ forms a clique.

The following theorem connects the concept of elimination orderings and chordal graphs.

Theorem 2 ([FG65]). *A graph is chordal if and only if it has a perfect elimination ordering.*

From that theorem we get a method to compute a triangulation of a graph using an (arbitrary) elimination ordering of the graph. We add edges to the graph to make the elimination ordering perfect. We call the chordal graph obtained by this procedure the *fill-in graph G_π of G with respect to π*. The edges added to the graph are called *fill-in edges*.

Algorithm 1 FillIn(Graph G, Elimination ordering π)

```
$H := G$
**for** $i = 1, \dots, n$ **do**
    let $v := \pi^{-1}(i)$
    make $\{w \in N_H(v) : \pi(w) > \pi(v)\}$ into a clique in $H$
**end for**
**return** $H$
```

We introduce further notations for the fill-in graph: We denote by $N_\pi^+(v) = \{w \in N_{G_\pi}(v) : \pi(w) > \pi(v)\}$ the set of higher numbered neighbours of v in the fill-in graph with respect to π and we define $\deg_\pi^+(v) = |N_\pi^+(v)|$.

Every elimination ordering possibly gives a different triangulation of the graph. The next theorem draws the connection to treewidth. It states that, in terms of finding good upper bounds on the treewidth, one should regard elimination orderings which induce fill-in graphs with small cliques.

Theorem 3 ([Bod98])**.** *Let G be a graph and $k \in \mathbb{N}$. The following statements are equivalent.*

(i) *G has treewidth $\le k$*

(ii) *There is a triangulation H of G such that H has maximal clique size $\le k + 1$.*

(iii) *There is an elimination ordering π of G such that the fill-in graph of G with respect to π has maximal clique size $\le k + 1$.*

Note that every (inclusion-)maximal clique in the fill-in graph of G is of the form $N_\pi^+[v] = N_\pi^+(v) \cup \{v\}$ for some vertex v. Hence, one could add a fourth equivalent statement to the theorem above: there is an elimination ordering π of G such that $\deg_\pi^+(v) \le k$ for all vertices v. If we are given an arbitrary elimination ordering π of G we can compute the fill-in graph and obtain an upper bound on the treewidth:

$$\operatorname{tw}(G) \le \max_{v \in V} \deg_\pi^+(v) = \omega(G_\pi) - 1$$

We will call this upper bound the *width of π*[1].

But where do we get the elimination ordering from? Instead of looking for an optimal elimination ordering (which minimizes the maximal clique size in the fill-in graph), we will build up an elimination ordering by successively choosing vertices greedily according to certain criteria. We pick a vertex v, make its neighbourhood into a clique and remove v from the graph. We call this operation *eliminating v* (hence the name elimination ordering). Then we pick the next vertex in the new graph, eliminate it, and so on. This procedure is called the *elimination game*.

Algorithm 2 EliminationGame(Graph G, Criterion C)

```
for $i = 1, \dots, n$ do
    choose a vertex $v$ from $G$ according to $C$
    set $\pi(v) := i$
    eliminate $v$ from $G$
end for
return $\pi$
```

Note that if we undo all deletions of vertices and edges after the elimination game, we will get exactly the fill-in graph with respect to the resulting elimination ordering. In the elimination game we can simultaneously compute the width of the resulting elimination ordering if we compute the degree of a vertex before it is eliminated and keep the maximum value.

[1]which is dependent on the graph, but usually we do not mention it

The choice of the next vertex should be such that the formation of large cliques is avoided. Here we focus on two implementations: MinimumDegreeFillIn (short: mdfi) always picks a vertex with minimum degree. This heuristic is also used as preprocessing for solving systems of sparse linear equations [Liu85]. GreedyFillIn (short: gfi) picks a vertex with the minimum number of non-adjacent neighbours such that the number of added fill-in edges is (locally) minimized.

It turns out that GreedyFillIn gives better upper bounds in practice than MinimumDegreeFillIn, in trade for a slower performance. These simple greedy algorithms often perform better than several other, more complicated upper bound algorithms [BK10].

In their basic forms these algorithms only yield an elimination ordering of the input graph. The width of the elimination ordering is an upper bound on the treewidth, but it does not give us a concrete tree decomposition as needed in a real practical setting. An elimination ordering of width k can be easily converted into a tree decomposition of width k using the fill-in graph: For every vertex v, we create a bag $X_v = N_\pi^+(v) \cup \{v\}$ and connect every node v in the tree with the lowest higher numbered neighbour of v in the fill-in graph. More details and a correctness proof can be found in [BK10].

2.4 Post-processing

After applying a tree decomposition heuristic on a graph, there are several methods to further post-process the computed tree decomposition in order to decrease its width. Most of them are based on so called *triangulation minimization*. A triangulation H of G is called *minimal* if there is no triangulation of G which is a proper subgraph of H. Given a graph $G = (V, E)$ and a triangulation $H = (V, F)$ of G, the triangulation minimization problem is to compute a minimal triangulation $G' = (V, E')$ of G such that $E' \subseteq F$.

To improve the width of a tree decomposition $\mathcal{T}$ we can first compute a triangulation of the graph $G = (V, E)$ by adding edges to all pairs of non-adjacent vertices that occur together in some bag of $\mathcal{T}$. Let F be the set of newly added edges; it can be shown that $H = (V, E \cup F)$ must be chordal [BK10]. From that we compute a minimal triangulation, which hopefully decreases the maximal clique size. This triangulation can be converted to a (perfect) elimination ordering, for example by the Maximum Cardinality Search algorithm [TY84], from which we can compute a new tree decomposition with (lower) width. References and results of using triangulation minimization can be found in [BK10].

Here we tried the following slightly different approach. We want to remove edges from the edge set F maintaining chordality, however, instead of looking for a minimal triangulation, we aim to minimize the maximal clique size. We want to use a integer linear program to achieve that, but expressing chordality directly seems to be rather complicated. Instead, we directly apply Maximum Cardinality Search on H and use the computed perfect elimination ordering π of H to solve an easier problem: Minimize the maximal clique size of H by removing edges from F such that π still is a perfect elimination ordering of the resulting graph. The resulting triangulation is not neccessarily optimal, i.e. a triangulation of G with minimum maximal clique size, because we imposed the restriction that π must be perfect.

Given a triangulation $H = (V, E \cup F)$ of $G = (V, E)$ and a perfect elimination ordering π of H, our problem can be modelled by an ILP formulation as follows:

$$\begin{array}{lll}
\text{minimize} & w & \\
\text{subject to} & X_e = 1, & \text{for all } e \in E \\
& X_{\{u,v\}} + X_{\{u,w\}} - X_{\{v,w\}} \leq 1, & \text{for all 3-cliques } \{u, v, w\} \text{ in } H \\
& & \text{with } \pi(u) < \pi(v) < \pi(w) \\
& 1 + \sum_{w \in N^+_{H,\pi}(v)} X_{\{v,w\}} \leq w, & \text{for all } v \in V \\
& w \geq 0,\ X_e \in \{0,1\}, & \text{for all } e \in E \cup F
\end{array}$$

The binary variables X_e encode the edges of the solution graph G^*. The first condition ensures that only edges from F can be removed. The second set of inequalities expresses that π is perfect for G^*, by stating that any two distinct higher numbered neighbours of a vertex must be adjacent. The integer variable w is the maximal clique size of G^*, which is the maximal number of higher numbered neighbours of a vertex. Since w is minimized, it automatically assumes this maximum value by the third condition.

In the course of this work, we realized that the usage of this ILP is actually not needed, because the fill-in graph of G with respect to π is always an optimal solution to this problem: π is obviously perfect for its fill-in graph and all fill-in edges with respect to π must occur in any triangulation of G for which π is perfect. Thus, the fill-in graph has the minimal maximum clique size of all these graphs.

Therefore, to post-process a tree decomposition we will simply triangulate the graph G by making the bags into cliques and apply Maximum Cardinality Search, which gives a perfect elimination ordering π of the triangulation. Regarding π as an elimination ordering of G, we can convert it to a new tree decomposition of G.

3 Contracting Matchings and Tree Decompositions

We adopt the simple idea used in Bodlaender's algorithm: contract a matching in the input graph and compute a tree decomposition of the contracted graph. For that we use heuristic algorithms such as MinimumDegreeFillIn or GreedyFillIn, which we will call *base algorithms* (to avoid confusion of names). Afterwards, we transform the obtained tree decomposition back into a tree decomposition of the input graph.

We start by looking at how this transformation works exactly. Later in this chapter, we also present a similar method that uses elimination orderings instead of tree decompositions.

3.1 Expanding a tree decomposition

Let us briefly look at a more general situation where we merge multiple vertices into single vertices instead of just endpoints of edges. Formally, one can describe this by a *merging function* $f : V \to V'$ from the vertex set V of the original graph G into a new vertex set V' with the interpretation that all vertices in $f^{-1}(\{v'\}) = \{v \in V : f(v) = v'\}$ are merged into a single vertex v'. For example, the contraction of a matching M can be described by the function f_M with

$$f_M(v) = \begin{cases} v, & \text{if } v \text{ is not matched by } M, \\ a_{\{v,w\}}, & \text{if } \{v,w\} \in M \text{ for some } w \in V. \end{cases}$$

Sometimes we also allow ourselves to write $f_M(\{u,v\})$ for the vertex $f_M(u) = f_M(v)$.

For such a merging function f, we define the new graph $f(G)$ with

$$\text{vertex set } f(V) \text{ and edge set } \Big\{\{f(u), f(v)\} : \{u,v\} \in E\Big\}.$$

Recall that we also introduced the notation G/M for $f_M(G)$ in the case of matchings. Now, if we are given a tree decomposition of $f(G)$, we can undo all mergings in the bags to get a tree

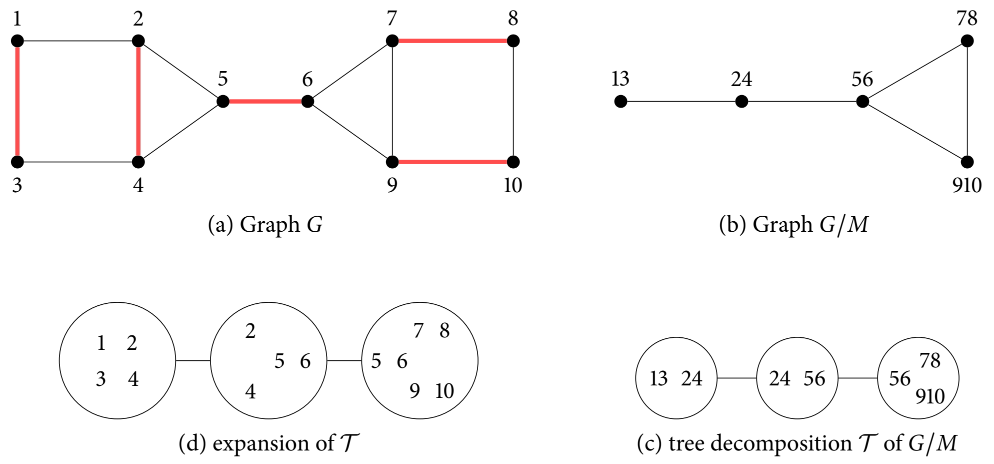


Figure 3.1: Illustration of a matching contraction and a tree decomposition expansion. Matching edges are marked red in panel (a).

decomposition of G.

Lemma 1. *Let $G = (V, E)$ be a graph and $f : V \to V'$ be a merging function. If $(\{X_i\}_{i\in I}, T)$ is a tree decomposition of $f(G)$, then $(\{f^{-1}(X_i)\}_{i\in I}, T)$ is a tree decomposition of G.*

Proof. Since every vertex $f(v)$ and every edge $\{f(u), f(v)\}$ of $f(G)$ occurs in some bag X_i, every vertex v and every edge $\{u, v\}$ of G also occurs some bag $f^{-1}(X_i)$.

Now let j be a node on the path from i to k in T. If a vertex v is contained in $f^{-1}(X_i)$ and $f^{-1}(X_k)$, then $f(v)$ is contained in X_i and X_k. Thus, X_j also contains $f(v)$, which implies that $f^{-1}(X_j)$ contains v. □

We call $(\{f^{-1}(X_i)\}_{i\in I}, T)$ the *expansion* of $(\{X_i\}_{i\in I}, T)$[1] (expansion as the opposite of contraction). When a tree decomposition is expanded, the size of a bag can increase by a factor which is at most the maximal size of a vertex subset that we have merged. This gives us an upper bound on the treewidth of G:

$$\operatorname{tw}(G) \leq r \cdot (\operatorname{tw}(f(G)) + 1) - 1, \quad \text{where } r = \max_{v' \in V'} |f^{-1}(\{v'\})|.$$

In our special case we only merge endpoints of edges (the case $r = 2$) and we get the upper bound $\operatorname{tw}(G) \leq 2 \cdot \operatorname{tw}(f(G)) + 1$ as mentioned in the previous chapter.

[1]we usually do not mention the corresponding function f and graph G

Algorithm 3 Input: Graph G, merging function f, base algorithm $\mathcal{A}$

```
compute $f(G)$
apply $\mathcal{A}$ on $f(G)$, obtain tree decomposition $(\{X_i\}_{i\in I}, T)$ of $f(G)$
**return** $(\{f^{-1}(X_i)\}_{i\in I}, T)$
```

Our main procedure is formulated in Algorithm 3. Usually the base algorithm $\mathcal{A}$ can only compute upper bounds k and k' on the treewidth of G and $f(G)$ respectively. We will compare k with the width of the expanded tree decomposition, which is at most $r \cdot (k' + 1) - 1$ with the notation above. Even if this algorithm might not to be able to compete with the pure application of the base algorithm, we want to examine how good the computed upper bound is. Usage of our algorithm might pay off if it runs considerably faster than the base algorithm and gives acceptable upper bounds.

3.2 Contracting edges vs. merging subgraphs

Now one could pose the question why we restrict ourselves to the special case of contracting only edges. If we only merge vertex sets which induce a connected subgraph, we obtain a minor of the original graph. A well-known fact states that taking minors does not increase the treewidth.

Lemma 2. *If G is a minor of H, then* $\operatorname{tw}(G) \leq \operatorname{tw}(H)$.

Proof. We transform a tree decomposition of H into a tree decomposition of G without increasing its width. If $(\{X_i\}_{i\in I}, T)$ is a tree decomposition of H and G is a subgraph of H, then $(\{X_i \cap V(G)\}_{i\in I}, T)$ is a tree decomposition of G. Now it suffices to regard the case of a single edge contraction. If G is obtained from H by contracting the edge $\{u, v\}$ to the vertex w, then we can replace any occurrence of u or v in a bag X_i by w and we get a tree decomposition for G. The conditions of a tree decomposition can be verified easily. □

If $f(G)$ is indeed a minor of G and we fix r, then from Lemma 2 it follows that

$$\operatorname{tw}(G) \leq r \cdot (\operatorname{tw}(f(G)) + 1) - 1 \leq r \cdot (\operatorname{tw}(G) + 1) - 1,$$

which can be seen as a constant-factor approximation in a certain sense. Assuming that the base algorithm computes an upper bound k' for $\operatorname{tw}(f(G))$ that is not too far from the real value, the width $r \cdot (k' + 1) - 1$ computed by Algorithm 3 will also be not much greater than $r \cdot (\operatorname{tw}(G) + 1) - 1$. Although r is not a precise approximation-ratio as in the theory of approximation algorithms, our algorithm has this property that we should expect from a good heuristic. Therefore, we are interested in merging only such vertices that do not cause the treewidth to increase.

It is easy to see that merging non-adjacent vertices can increase the treewidth of a graph. For example, if the two endpoints of a path with length ≥ 3 are merged, the treewidth increases from 1 to 2. In general, we cannot easily determine whether a given merging of vertices is "good" in our sense, i.e. does not increase the treewidth. A simple argument shows that this problem is even NP-hard.

For the hardness-proof we need a simple lemma about tree decompositions, from which we can derive that the treewidth of a graph is at least its maximal clique size minus one.

Lemma 3 ([Bod98]). *Let $(\{X_i\}_{i\in I}, T)$ be a tree decomposition of $G = (V, E)$. If $W \subseteq V$ is a clique in G, then there exists an $i \in I$ with $W \subseteq X_i$.*

Theorem 4. *The decision problem "Given a graph G and a merging function f, is $\operatorname{tw}(f(G)) \leq \operatorname{tw}(G)$?" is NP-hard.*

Proof. Observe that any graph can be built up from isolated edges by merging the endpoints properly.

We present a polynomial-time reduction from TREEWIDTH to this problem. Given a graph $G = (V, E)$ and $k \in \mathbb{N}$, let G' be the graph which consists of $|E|$ isolated edges and a $(k+1)$-clique. More formally, G' has the vertex set

$$\{(e,u),(e,v) : e = \{u,v\} \in E\} \cup \{c_1,\dots,c_{k+1}\}$$

and edges $\{(e,u),(e,v)\}$, for all $e = \{u,v\} \in E$ and edges $\{c_i, c_j\}$, for all $i \neq j$.

If we merge all vertices in $\{(e,u) : u \in e \in E\}$ to a single vertex u, for all $u \in V$ and also merge the clique $\{c_1,\dots,c_{k+1}\}$ to a vertex c, we obtain G plus an isolated vertex c. Let f be a merging function that describes this merging. The graph G' has treewidth k and the merged graph $f(G')$ has the same treewidth as G. Therefore, it holds that

$$\operatorname{tw}(G) \leq k \iff \operatorname{tw}(f(G')) \leq \operatorname{tw}(G').$$

The computation of G' and f can clearly be done in polynomial time. This completes the reduction. □

This result shows that it is hard to find out whether a merging function is useful or not in the sense that we do not want to increase the treewidth by merging vertices. One could think of an algorithm in which we repeatedly merge multiple vertices into single vertices to lower the treewidth and stop at some point. As soon as we merge non-adjacent vertices, we are unsure whether the treewidth has increased or not and it seems to be infeasible to find that out, as we have seen, making such an algorithm rather useless.

Hence, we are restricted to merging connected subgraphs. We will restrict our method further to merging only endpoints of edges and leave the more general method open. One should notice that merging large connected subgraphs to single vertices means a high loss of information in the merged graph. One the one hand the base algorithm might perform better (time and quality of the bound) if the merged graph is small, on the other hand we expect that the expanded tree decomposition will have a larger width (think of the extreme case of contracting the whole graph into one vertex). For this reason, we stick to the simplest special case of contracting only matchings.

3.3 Limitations of the approach

3.3.1 Objectives of the approach

It remains to determine according to which criteria we should choose the matching M that is to be contracted. Assume that $\mathcal{T}$ is the tree decomposition of G/M computed by the base algorithm with width k and $\mathcal{T}'$ is the expansion of $\mathcal{T}$ having width k'. Our overall goal is to minimize the width of $\mathcal{T}'$. To achieve that we want to find a matching such that firstly the width k of $\mathcal{T}$ is small and secondly the increase from k to k' is also small. An extreme case for no increase in the width, i.e. $k = k'$, can be achieved by contracting no edges at all. Hence there is no meaningful optimization possible in that direction. Also, here we usually want to contract a maximal, "dense" matching which makes it likely that at least one bag of $\mathcal{T}$ contains $k-1$ vertices which originated from an edge contraction such that the bound $k' = 2k+1$ is reached. We will look into that again in the next chapter. Here we will focus on minimizing the width k of $\mathcal{T}$.

For minimizing the width of $\mathcal{T}$ one could also think of two directions. We can try to minimize k, but if we want to make the selection of the matching independent of the underlying base algorithm, we should minimize the treewidth of G/M instead of k. We imagine that it is quite challenging to analyze the heuristics and to find out which matching M minimizes the width of the tree decomposition for G/M calculated by the base algorithm; however, this of course could yield better results. In this thesis, we decided to ignore the underlying base algorithm and define the overall objective to be the minimization of $\mathrm{tw}(G/M)$.

3.3.2 Minimizing treewidth of the contracted graph

We were not able to devise a polynomial-time algorithm that computes a matching M minimizing the treewidth of G/M and we strongly assume that there does not exist such an algorithm. Usually the non-existence of a polynomial-time algorithm $\mathcal{A}$ is "shown", assuming $P \neq NP$, by constructing another polynomial-time algorithm for an NP-hard problem under the assumption that $\mathcal{A}$ exists. In fact, we can show that the following problem is NP-hard.

Theorem 5. *The decision problem "Given a graph G and a number k, does there exist a matching M such that* $\operatorname{tw}(G/M) \leq k$*?" is NP-hard.*

Proof. The proof idea was adopted from [BGHK92]. Again we use a reduction from Treewidth, so let $G = (V, E)$ be a graph and $k \in \mathbb{N}$. Let $G' = (V', E')$ be the graph with

$$\begin{aligned} \text{vertex set } V' &= \{v_1, v_2 : v \in V\} \text{ and} \\ \text{edge set } E' &= \big\{\{v_1, v_2\} : v \in V\big\} \cup \big\{\{u_i, v_j\} : \{u, v\} \in E,\ 1 \leq i, j \leq 2\big\}. \end{aligned}$$

Clearly, G' is polynomial-time computable from G.

Note that $M^* = \big\{\{v_1, v_2\} : v \in V\big\}$ is a maximal matching in G' and G'/M^* is isomorphic to G. We will to prove the following equivalence.

$$\operatorname{tw}(G) \leq k \iff \text{there exists a matching } M \subseteq E' \text{ such that } \operatorname{tw}(G'/M) \leq k.$$

If $\operatorname{tw}(G) \leq k$ holds, then we can pick the matching $M = M^*$. For the other direction, we will use that $\operatorname{tw}(G') = 2 \cdot \operatorname{tw}(G) + 1$, which we will show later. If $\operatorname{tw}(G'/M) \leq k$ for some matching M, then we can expand a tree decomposition of G'/M and get $\operatorname{tw}(G') \leq 2k + 1$. Using the lemma we conclude $\operatorname{tw}(G) \leq k$.

It remains to show that $\operatorname{tw}(G') = 2 \cdot \operatorname{tw}(G) + 1$. Again we have $\operatorname{tw}(G') \leq 2 \cdot \operatorname{tw}(G) + 1$ by the expansion argument. Conversely, let $\mathcal{T}' = (\{Y_i\}_{i \in I}, T)$ be a tree decomposition of G' with minimal width and set

$$X_i = \{v \in V : \{v_1, v_2\} \subseteq Y_i\}.$$

One easily verifies that $\mathcal{T} = (\{X_i\}_{i \in I}, T)$ is a tree decomposition of G: Every vertex v is contained in some bag X_i, since $\{v_1, v_2\}$ is an edge in G'. If $\{u, v\}$ is an edge in G, then $\{u_1, u_2, v_1, v_2\}$ is a 4-clique in G', implying that there is a bag Y_i containing the clique and that the corresponding bag X_i contains u and v. If j is a node on the path between nodes i and k in T and $v \in X_i \cap X_k$, then $\{v_1, v_2\} \subseteq Y_i \cap Y_k$ holds. It follows that $\{v_1, v_2\} \subseteq Y_j$ and hence $v \in X_j$. The size of the bag X_i is at most half the size of Y_i and so $2 \cdot \operatorname{tw}(G) + 1 \leq \operatorname{tw}(G')$ is proved. □

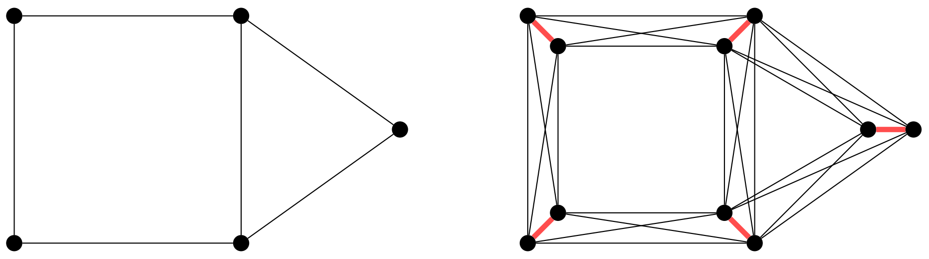

(a) Original graph G (b) Graph G' with matching M^*

Figure 3.2: Illustration of the NP-hardness proof

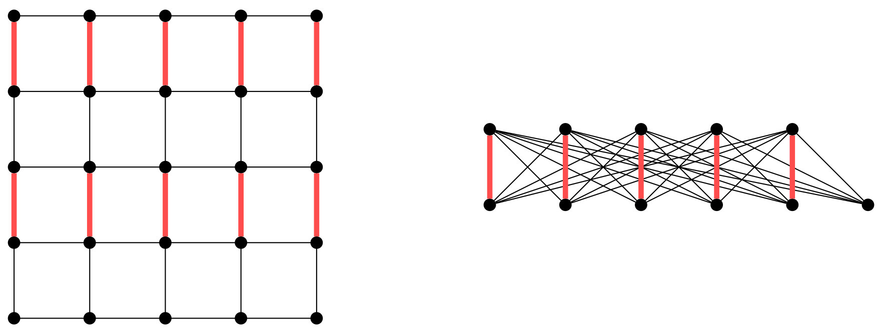
However, even with this theorem the existence of an polynomial-time algorithm which minimizes the treewidth of G/M does not directly lead to $P = NP$. The problem is that we cannot use such a hypothetical algorithm to solve the problem above in polynomial-time, as we cannot compute the treewidth of G/M in polynomial-time for an optimal matching M which we have computed.

Our second approach was to regard simpler decision problems, namely by fixing the parameter k in the above problem formulation. The NP-hardness of such a problem would prove the non-existence of the minimization algorithm, unless $P = NP$, since we can decide in linear-time whether the treewidth of a graph is at most k, for a fixed k, by Bodlaender's theorem. Unfortunately, if we fix the parameter k, the adapted problems also become polynomial-time decidable.

Theorem 6. *For all $k \in \mathbb{N}$ the decision problem "Given a graph G, does there exist a matching M such that $\operatorname{tw}(G/M) \le k$?" is polynomial-time decidable.*

Proof. We just sketch the proof idea to not go beyond the scope of this thesis and assume the reader to be familiar with the concepts used (see [Cou90], [RS04]). First of all, note that a graph with $\operatorname{tw}(G) > 2k + 1$ cannot be contracted with a matching to $\operatorname{tw}(G/M) \le k$ (again expansion argument). We can verify $\operatorname{tw}(G) \le 2k + 1$ using Bodlaender's algorithm. If it indeed returns a tree decomposition of width at most $2k + 1$, we can use Courcelle's theorem to decide the validity of a formula in monadic second-order logic expressing the desired property.

The formula can be constructed as follows. There is a formula $\varphi_{\le k}$ that expresses that a graph has treewidth at most k, as the class of graphs with treewidth at most k is characterizable by a finite set of forbidden minors and the relation "H is minor of G" for a fixed graph H can be expressed in monadic second-order logic. Since $\varphi_{\le k}$ should be evaluated in G/M, we have to adapt the formula. For example, two vertices in G are considered equal in G/M if and only if



(a) (5×5)-grid (b) Complete bipartite graph $K_{5,6}$

Figure 3.3: Contracting a matching can decrease treewidth to its half, or not at all.

they are either the same or they are matched by the same edge in M, which can be expressed in monadic second-order logic. Every occurrence of an atom $u = v$ in $\varphi_{\leq k}$ has to be replaced by such a suitable formula and similarly we proceed for the other atoms, such as $\mathrm{adj}(u, v)$ and $\mathrm{inc}(e, v)$ expressing adjacency and incidence. If $\hat{\varphi}_{\leq k}$ is the transformed formula, then $\exists M \hat{\varphi}_{\leq k}$ expresses the desired property. □

This leaves the existence of a polynomial-time algorithm which minimizes the treewidth of G/M open. Here we will use heuristics to compute the matchings, which we will present in the next chapter.

3.3.3 Lower-bound instances

Another limitation of the method is the existence of hard instances. There exist graphs whose treewidth cannot be decreased by matching contraction, independent of the matching chosen. Consider a complete bipartite graph $K_{n,n+1}$ which has vertices $\{u_1, \ldots, u_n, v_1, \ldots, v_{n+1}\}$ and edges $\{u_i, v_j\}$ for all $1 \leq i \leq n, 1 \leq j \leq n+1$. It is easy to see that the complete bipartite graph $K_{n,n+1}$ has treewidth n. Contracting any maximal matching yields the complete graph K_{n+1} on $n+1$ vertices, which also has treewidth n.

In addition to this, we also want to present a graph class on which our algorithm could, at least in theory, run optimal. The *$(m \times n)$-grid* is the graph with

$$\begin{aligned} &\text{vertex set } \{1, \ldots, m\} \times \{1, \ldots, n\} \\ &\text{and edge set } \big\{\{(i_1, i_2), (j_1, j_2)\} : |i_1 - i_2| + |j_1 - j_2| = 1\big\}. \end{aligned}$$

It is known that the $(m \times n)$-grid has treewidth $\min(m, n)$[2]. The $(m \times m)$-grid has a matching of size $m \cdot \lfloor m/2 \rfloor$ as depicted in Figure 3.3. Contracting this matching yields a $(m \times \lceil m/2 \rceil)$-grid with treewidth $\lceil m/2 \rceil$. We see that grids are an optimal example for the reduction of treewidth by contraction of a matching.

3.4 Expanding elimination orderings

As an alternative to expanding tree decompositions, we will present a different approach based on elimination orderings. In our setting this is very convenient because we work with MinimumDegreeFillIn and GreedyFillIn, which operate on elimination orderings. Let M be a matching in the graph G and let σ be an elimination ordering of G/M. Consider a vertex w in G/M which originated from contracting an edge $\{u, v\} \in M$ and replace w in σ by u and v by juxtaposing u and v in an arbitrary order. If this is done for all such vertices w, we obtain a elimination ordering π for G. This procedure is also described in Algorithm 4.

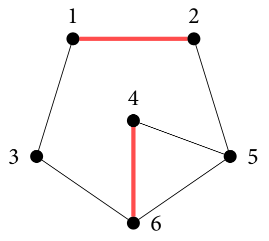
Algorithm 4 ExpandEO(elimination ordering σ, graph G, matching M)

```
set $j := 1$
for $i = 1, \dots, |G/M|$ do
    $w := \sigma^{-1}(i)$
    if $f_M^{-1}(\{w\}) = \{u, v\} \in M$ then
        set $\pi(u) := j$
        set $\pi(v) := j + 1$
        set $j := j + 2$
    else
        let $\{u\} := f_M^{-1}(\{w\})$
        set $\pi(u) := j$
        set $j := j + 1$
    end if
end for
return $\pi$
```

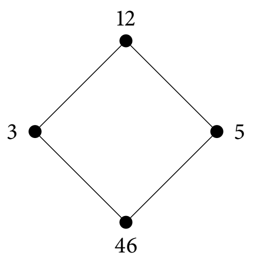
We call π an *expansion* of σ[3]. For now we leave open in which order the two endpoints of a matching edge are juxtaposed in the expansion, so an elimination ordering can have several expansions. We will show that, similar to the tree decomposition approach, the width of an expansion of σ is bounded by $2k + 1$ where k is the width of σ. The following lemmata help to calculate the width of the new elimination ordering. The notation $G \ominus v$ denotes the graph obtained from G by eliminating v. The operator $\ominus$ is left-associative.

[2]which is usually proven using the cops-and-robber game by Seymour and Thomas [ST93]

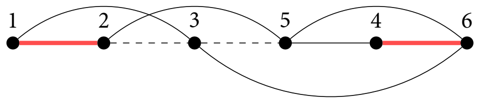
[3]the corresponding graph G and matching M are implicit



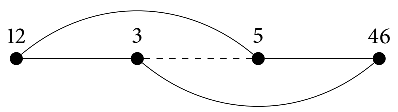
(a) Graph G



(b) Graph G/M



(d) expanded elimination ordering



(c) elimination ordering of G/M

Figure 3.4: Expansion of an elimination ordering (fill-in edges are dashed)

Lemma 4. *Let M be a matching in G and w be a vertex in G/M.*

(i) If $f_M^{-1}(\{w\}) = \{v\}$, then:

a) $(G/M) - \{w\} = (G - \{v\})/M$

b) $(G/M) \ominus w = (G \ominus v)/M$

(ii) If $f_M^{-1}(\{w\}) = \{u, v\}$, then:

a) $(G/M) - \{w\} = (G - \{u, v\})/(M \setminus \{\{u, v\}\})$

b) $(G/M) \ominus w = (G \ominus u \ominus v)/(M \setminus \{\{u, v\}\})$

Proof. All statements follow easily from the definitions. □

Lemma 5. *Let M be a matching in G and σ be an elimination ordering of G/M. If π is an expansion of σ, then $(G/M)_\sigma = G_\pi/M$.*

Proof. The two graphs have the same vertex sets, we need to show that they have exactly the same edges. We will prove that by induction on the number of vertices of G.

If G has one vertex, the statement is clear. Otherwise, let w be the first vertex in σ. We will show that $(G/M)_\sigma$ and G_π/M agree on the neighbourhood of w and on the rest (the graphs without w), which proves that the graphs must be the same. We will do a case distinction whether w originated from an edge contraction or not.

First case. Assume $f_M^{-1}(\{w\}) = \{v\}$ for some vertex v in G. Then v is the first vertex in π and by definition it holds that

$$N_{(G/M)_\sigma}(w) = N_{G/M}(w) = f_M(N_G(v)) = f_M(N_{G_\pi}(v)) = N_{G_\pi/M}(w).$$

Now let $\hat{\sigma}$ be the elimination ordering obtained from σ by removing w. Also let $\hat{\pi}$ be π without v. We can apply the induction hypothesis on $G \ominus v$, M, $\hat{\sigma}$ and $\hat{\pi}$ because $\hat{\pi}$ is an expansion of $\hat{\sigma}$. Using Lemma 4 we get

$$\begin{aligned}(G/M)_\sigma - \{w\} &= ((G/M) \ominus w)_{\hat{\sigma}} = ((G \ominus v)/M)_{\hat{\sigma}} = (G \ominus v)_{\hat{\pi}}/M \\ &= (G_\pi - \{v\})/M = (G_\pi/M) - \{w\}.\end{aligned}$$

Second case. Assume $f_M^{-1}(\{w\}) = \{u, v\} \in M$. Then u and v are the first two vertices in π, say $\pi(u) < \pi(v)$. That means that v is connected to all neighbours of u in G_π and it holds that

$$\begin{aligned}N_{(G/M)_\sigma}(w) = N_{G/M}(w) &= f_M\big((N_G(u) \cup N_G(v)) \setminus \{u, v\}\big) \\ &= f_M\big((N_{G_\pi}(u) \cup N_{G_\pi}(v)) \setminus \{u, v\}\big) = N_{G_\pi/M}(w).\end{aligned}$$

Again let $\hat{\sigma}$ be σ without w and let $\hat{\pi}$ be π without u and v. By Lemma 4 and induction hypothesis on $G \ominus u \ominus v, M \setminus \{\{u, v\}\}, \hat{\sigma}$ and $\hat{\pi}$ we get

$$\begin{aligned}(G/M)_\sigma - \{w\} &= ((G/M) \ominus w)_{\hat{\sigma}} = \big((G \ominus u \ominus v)/(M \setminus \{\{u, v\}\})\big)_{\hat{\sigma}} \\ &= (G \ominus u \ominus v)_{\hat{\pi}}/(M \setminus \{\{u, v\}\}) = (G_\pi - \{u, v\})/(M \setminus \{\{u, v\}\}) \\ &= (G_\pi/M) - \{w\}.\end{aligned}$$

This concludes the proof. □

Theorem 7. *Let M be a matching in G and σ be an elimination ordering of G/M. If σ has width k, then an expansion of σ has width at most $2k + 1$.*

Proof. The size of a maximal clique in the fill-in graph G_π can at most reduce to its half by contracting a matching M. By Lemma 5 we obtain

$$\omega(G_\pi) \leq 2 \cdot \omega(G_\pi/M) = 2 \cdot \omega((G/M)_\sigma).$$

By the definition of the width of an elimination ordering, this proves the claim. □

In terms of deriving (worst case) upper bounds on $\mathrm{tw}(G)$ from $\mathrm{tw}(G/M)$, we have shown nothing new. Later we want to compare the tree decomposition approach with the elimination ordering approach in practical experiments.

All results presented so far are independent of the order in which we position two vertices u, v from a matching edge $\{u,v\} \in M$. But the resulting width may depend on that order. Recall that the width of an elimination ordering π can be computed as the maximum of all $\deg_\pi^+(v)$. If vertex u is eliminated first, the neighbourhood of v becomes $(N_\pi^+(u) \cup N_\pi^+(v)) \setminus \{v\}$. Then, after eliminating v, this set is made into a clique.

Hence, we should put the vertex with the lower degree prior in the order. Because $G \ominus u \ominus v = G \ominus v \ominus u$, this choice does not affect the remaining fill-in graph. Thus, to gain the optimal width that we can obtain by transposing vertex pairs $\{u,v\} \in M$ in the expansion, it suffices to apply this local optimization on all edges $\{u,v\} \in M$. To implement this we have to go from left to right in the elimination ordering because implicitly we need to compute the fill-in graph. The extension of the procedure ExpandEO is shown below.

Algorithm 5 ExpandEO+(elimination ordering σ, Graph G, matching M)

```
set $H := G$
set $j := 1$
**for** $i = 1, \dots, |G/M|$ **do**
    $w := \sigma^{-1}(i)$
    **if** $f_M^{-1}(\{w\}) = \{u,v\} \in M$ **then**
        set $\pi(\operatorname{argmin}_{x \in \{u,v\}} \deg_H(x)) := j$
        set $\pi(\operatorname{argmax}_{x \in \{u,v\}} \deg_H(x)) := j+1$
        set $j := j+2$
        $H := H \ominus u \ominus v$
    **else**
        let $\{u\} := f_M^{-1}(\{w\})$
        set $\pi(u) := j$
        set $j := j+1$
        $H := H \ominus u$
    **end if**
**end for**
**return** $\pi$
```

4 Computational Evaluation

In this chapter we introduce algorithms that compute different kinds of matchings and present the results that we get by applying the main algorithm in combination with these matching algorithms. For the sake of completeness, we formulate Algorithm 3 again for the case of matchings.

Algorithm 6 Input: Graph G, base algorithm $\mathcal{A}$

```
  compute a matching $M$ of $G$
  contract all edges in $M$, obtain $G/M$
  apply $\mathcal{A}$ on $G/M$, obtain tree decomposition $(\{X_i\}_{i \in I}, T)$ of $G/M$
  return $(\{f_M^{-1}(X_i)\}_{i \in I}, T)$
```

We implemented the algorithms in C++ on the basis of the Treewidth Optimization Library (TOL), a framework for the computation of tree decompositions, developed by Arie M.C.A. Koster. The algorithms were tested on a Linux 3.0.2 machine with an Intel®Core™2 Duo CPU T5670 (1.80 GHz) and 2 GB RAM. Sometimes we use integer linear programs in the algorithms, which we solved using the integer programming solver SCIP, which is developed at the Zuse Institute Berlin (ZIB) [Ach09].

4.1 Test graphs and generation of random partial k-trees

We tested the algorithms with different kinds of graph instances. We started by running the algorithms on a collection of various graphs from the TOL repository. It contains interesting graphs from graph theory, but also graph instances which are closer to "real world" applications such as a graph from computational biology or travelling salesman problem graphs.

However, we quickly realized that for some situations it would be good, if we could generate random graphs with adjustable parameters such as the size, the edge density and of course the treewidth. Running the algorithms on a large series of graphs allows for a better analysis because we may be able to recognize how certain parameter settings can influence the performance of the algorithms. These graphs however are most likely not a good representation of graphs that would occur in "real world" applications, as they are artificially constructed instances. The algorithms may behave very differently in practice than on such random graphs.

To generate such random graphs we follow the approach from [BK10], which we want to present in the following. A *k-tree* is a graph G that has a perfect elimination ordering π such that $\deg_\pi^+(v) = \min\{k, n - \pi(k)\}$ for all vertices $v \in V(G)$. To put it differently, the last k vertices (in fact, the last $k+1$ vertices) in π form a clique and all other vertices have exactly k higher numbered neighbours. We see that a k-tree is chordal and has treewidth k. A *partial k-tree* is a subgraph of a k-tree and hence, has treewidth at most k. In fact, it can be shown that a graph is a partial k-tree if and only if it has at most treewidth k [Bod98], and that is why we are interested in generating random partial k-trees.

Let us first see how to generate a random k-tree G. For the sake of simplicity we will use the vertex set $\{1, \dots, n\}$, which also defines the (perfect) elimination ordering π of the graph. Algorithm 7 generates a random k-tree with n vertices.

Algorithm 7 RandomKTree($n \in \mathbb{N}$, $k < n$)

```
$G := (\{1, \dots, n\}, \emptyset)$
make $\{n - k, \dots, n\}$ into a clique in $G$
**for** $i = n - k - 1, \dots, 1$ **do**
    $U := N_G[j]$ for some random $j \in \{i + 1, \dots, n - k\}$
    $U := U \setminus \{u\}$ for some random $u \in U$
    connect $i$ to all vertices $i' \in U$
**end for**
**return** $G$
```

First of all, we make the last $k+1$ vertices $\{n-k, \dots, n\}$ into a clique. Now the idea is to connect every vertex v with a k-clique of higher numbered vertices so that the subgraph induced by v and together with all higher numbered vertices satisfies the condition of a k-tree, going from the back to the front. Recall that every $(k+1)$-clique in a chordal graph with perfect elimination ordering π occurs as a set $N_\pi^+[v]$ for some vertex v, and we can show that every k-clique C can be extended to a $(k+1)$-clique. If one vertex in C is connected to a vertex u that is lower numbered than all vertices in C, then $C \cup \{u\}$ forms a $(k+1)$-clique. Otherwise let $v \in C$ be the lowest numbered vertex in C. $N_\pi^+[v]$ is a superset of C and forms a $(k+1)$-clique because $\pi(v) \leq n - k$ must be true in this case. Hence, we can find a random k-clique by removing a random vertex from a random neighbourhood $N_G[j]$.

To generate a random partial k-tree, we generate a random k-tree and remove edges randomly. For that we introduce a third parameter $p \in [0, 1]$ which specifies the probability for an edge to be removed. By the defining equation, a k-tree always has $kn - k(k+1)/2$ edges and the randomly generated partial k-tree has $(1-p)(kn - k(k+1)/2$ edges on average.

Finally, to ensure that the graph still has treewidth *exact k* after removing the edges, we apply a lower bound algorithm, i.e. an algorithm that computes a lower bound on the treewidth of a graph. Here we use a heuristic called Maximum Minimum Degree [BK11]. We repeatedly generate partial k-trees until the lower bound is exactly k.

4.2 Maximal matchings

At first, we start by using arbitrary maximal matchings as in Bodlaender's algorithm. We call Algorithm 6 using maximal matchings mm. This matching type should serve as a reference point for other matching types being an "average" matching with no special properties.

Maximal matchings can be computed greedily by repeatedly adding an edge $\{v, w\} \in E$ to the matching and removing the vertices v and w, until the graph is empty. The resulting matching is dependent on the choice of the edges, of course. In our implementation we simply use the order of the edges as stored in the representation of the test graphs.

	MinimumDegreeFillIn			GreedyFillIn		
graph name	none	mm	maxcard	none	mm	maxcard
1ubq	12	19	23	12	19	21
C5	2	4 (5)	4	4	4	4
C5+	2	4 (5)	3	2	4	3
Clebsch	10	9	9	8	9	9
gr216	91	137	143	89	133	141
hamming9-2	203	203	203	203	203	203
kneser8-3	34	35	41	34	35	43
knights8_8	20	21	25	20	23	23
macaque71	21	30 (31)	29	20	28	29
petersen	4	9	9	5	9	9
pyramid3	5	7	7	5	7	7
risk	5	7	9	4	7	9
scoregraph	320	508 (511)	577	307	496	559
sheep	330	685 (689)	635	260	507	537
sudoku	51	53	59	53	53	55

Table 4.1: Comparison of computed widths (none: pure application of the base algorithm, mm: maximal matching, maxcard: maximum cardinality matching)

Evaluation

Table 4.1 shows the upper bounds for a collection of 15 graphs computed by mm using the base algorithms MinimumDegreeFillIn and GreedyFillIn. As a first observation we can state that in most cases mm does not give an improvement to the base algorithm. For some instances, the upper bounds of mm come close and for the graph "Clebsch" mm even performs better than MinimumDegreeFillIn. For the graph "sheep", which is by far the largest of the tested graphs (6418 vertices and 18474 edges), the width computed by mm is noticeably larger than twice the width computed by MinimumDegreeFillIn. In this rather odd case MinimumDegreeFillIn computes a higher width for a graph than for a half-sized minor of it.

Finally, we want to mention that if k is the width computed for the contracted graph, for most of the graphs $2k + 1$ was indeed reached as the width of the expanded tree decomposition. As an example, in the column mm for MinimumDegreeFillIn, the number $2k + 1$ is given in brackets, if it differs from the "real" width of the expansion. For the larger graphs we get a difference of 3–4, for the other smaller graphs the difference is at most one.

4.3 Maximum cardinality matchings

As a first improvement towards more sophisticated matching types, we use maximum cardinality matchings (we denote the corresponding algorithm by maxcard). This will minimize the vertex number of the contracted graph. The computation of maximum matchings (on general graphs) is more involved; the well known Blossom algorithm by Edmonds [Edm65] in its original form runs in time $O(n^4)$. Here we use an implementation from the LEMON graph library [DJK11].

Evaluation

In Table 4.1 we can compare the performance of mm and maxcard. We see that for most of the graphs maxcard does not give better bounds than mm, and especially for the medium-sized and large graphs the bounds are clearly worse. The width for the "sheep" graph using MinimumDegreeFillIn is improved; here we assume that mm "unluckily" (being a quite unfounded approach) produces a contracted graph that is bad for MinimumDegreeFillIn because we see that for GreedyFillIn the situation on the "sheep" graph is reversed. We verified this with a second experiment: We ran mm again on five random permutations of the "sheep" graph to force mm to select different maximal matchings every time. The average upper bound computed is 629 which is better than the bound computed by maxcard.

Contracting a maximum cardinality matching seems to be the wrong direction to lower the treewidth, realizing that maxcard even cannot compete with picking arbitrary, "average" maximal matchings.

4.4 Weighted matchings

Instead of trying to find a matching that is as large as possible, it seems to be more important to find the "right" collection of edges for a matching, in terms of lowering the treewidth by contraction. Our next idea is to give weights to edges according to how much an edge contraction can contribute to the decrease of treewidth. We present two approaches that such an edge weighting: in the first approach we compute a maximum weighted matching and in the second, we use a greedy algorithm to build up a matching.

4.4.1 Edge weighting functions

We believe that in general such a reasonable weighting function does not really exist because treewidth and tree decompositions are concepts which are based on large substructures in the graph, such as large cliques, and cannot be fully captured by local structures as for example vertex neighbourhoods. Despite of that, we want to try out some weighting functions which might perform decently, at least on some graph instances.

We have three ideas for weighting an edge $\{u, v\}$, which can be quickly computed locally:

1. **Common Neighbours.** The number of neighbours that u and v have in common.
2. **Symmetric Difference.** The number of neighbours that u and v do not have in common, excluding u and v themselves.
3. **Degree Sum.** The sum of the degrees of u and v.

For the first and the third criterion, we consider edges with large weights as useful for contraction; for the second criterion we prefer edges with small weights. We want to briefly discuss, why we decided to choose these criteria.

Common Neighbours

If vertices u and v have a large number of common neighbours, the contraction of $\{u, v\}$ causes a lot of edges incident to u to be merged with edges incident to v. Contracting such edges preferably leads to a contracted graph with a lower edge number, which then could mean a lower treewidth. This can be partially justified by the following theorem.

Theorem 8 ([Bod96]). *If $G = (V, E)$ has treewidth at most k, then $|E| \leq k|V| - \frac{1}{2}k(k+1)$.*

Colloquially, this theorem states that a graph must be sparse to have small treewidth, thus, it is a good idea to try to lower the number of edges.

Moreover, if the endpoints of an edge $\{u, v\}$ share many neighbours, they belong to many 3-cliques, which implies that u and v will occur together in many bags of any tree decomposition of the graph. By contracting the edge $\{u, v\}$, automatically the two vertices will always occur together in the bags of the expanded tree decomposition.

Symmetric Difference

The name of the second criterion is borrowed from set theory where the set

$$A \bigtriangleup B = (A \cup B) \setminus (A \cap B)$$

is called symmetric difference of A and B. Here we regard the weight $|N(u) \bigtriangleup N(v)| - 2$ (minus two, to exclude u and v themselves), which is in some way dual to the common neighbours criterion. In the best case according to this weighting, the endpoints u and v of an edge have the exactly same neighbourhood (again, excluding u and v). In this situation, one can easily see that there is an optimal tree decomposition such that u and v always occur together in the bags. Because of that, we make no error if we force u and v to always occur together in the expanded tree decomposition when we contract the edge $\{u, v\}$. If there exist additional neighbours which are not shared by both endpoints, a minimal-width tree decomposition might need to have bags containing only one endpoint and not the other. Contracting that edge could cause non-optimality in the expanded tree decomposition. If we contract an edge with a small weight, we expect that this error will be small, too.

Degree Sum

Lastly, we have the third criterion, the degree sum, which was chosen with no deeper theoretical foundation. One can think of some situations where this can be good. For example, we should definitely contract edges in large cliques, and such edges will have a large weight. Another illustrating example are graphs containing long induced paths. We think it does not make sense to contract edges in such induced paths, rather one should focus on more complicated structures in the graph. The degree sum of an edge in an induced path is at most four.

Mixture weightings

Instead of evaluating edges by just one weighting function, we can use a primary and a secondary weighting function. Basically we are combining two orderings on the edges into one new ordering. At first, we order two edges by the primary weighting; if they happen to have the same weight, we order them by the secondary weighting. If w_1 and w_2 are two weighting functions, this can be encoded as a new weighting function

$$w(e) := m \cdot s_1 \cdot w_1(e) + s_2 \cdot w_2(e)$$

where m is a value larger than the maximum of w_2 and $s_i \in \{-1, 1\}$ determines whether w_i is to be minimized or maximized. In our case the sign for common neighbours and degree sum should be 1 and for symmetric difference it should be −1. Using a mixture weighting may be helpful, if during the algorithm many edges have an optimal weight. The choice of the next edge will be arbitrary, if we just pick any of these. By adding a second selection criterion we get a more deterministic choice and hopefully a better result.

4.4.2 Maximum weighted matchings

In our first approach to use the presented weighting functions, we compute a maximum weighted matching, i.e. a matching M that maximizes $\sum_{e \in M} w(e)$, where w is the weighting function. Note that it does not make sense to use the symmetric difference weighting, since we want to choose edges with small weight for that specific weighting. The optimal matching would always be the trivial empty matching having weight zero.

The maximum weighted matching problem can be solved with an adapted version of the Blossom algorithm. Again we used the LEMON library to compute maximum weighted matchings. We abbreviate this matching type with maxweight.

Evaluation

For evaluating these algorithms, we restricted ourselves to just using MinimumDegreeFillIn as the base algorithm. In Table 4.2 we can see that for the large graphs the maxweight approach can yield better widths than mm; the common neighbours weighting seems to perform best. How-

			maxweight		greedy		
graph name	none	mm	cn	ds	cn	ds	sd
1ubq	12	19	19	21	15	19	17
C5	2	4	4	4	4	4	4
C5+	2	4	4	3	4	3	4
Clebsch	10	9	11	11	9	13	9
gr216	91	137	147	145	131	145	139
hamming9-2	203	203	325	327	203	329	203
kneser8-3	34	35	43	41	43	43	43
knights8_8	20	21	29	29	25	27	25
macaque71	21	30	23	29	23	23	24
petersen	4	9	7	9	7	7	7
pyramid3	5	7	7	7	6	7	7
risk	5	7	7	7	7	7	7
scoregraph	320	508	427	573	420	371	566
sheep	330	685	579	565	569	439	560
sudoku	51	53	61	63	55	61	57

Table 4.2: Using weighted matchings with different weighting functions (base algorithm: mdfi)

ever, we also get very bad results for all weighting functions, e.g. "hamming9-2" or "knights8_8". The problem that we see with maxweight is that for the weighting functions used, the addition of weights has no interpretable meaning and neither has the quantity $\sum_{e \in M} w(e)$, which we maximize. For example, it does not make sense to regard selecting many edges with few common neighbours and selecting one edge with many common neighbours in whatever sense equivalent.

4.4.3 Greedy weighted matchings

Therefore, we decided to switch to a greedy approach. Besides the reason mentioned above, this approach will also be much faster. The general procedure is to compute the weightings for all edges and pick an edge with maximal (or minimal respectively for the symmetric-difference-criterion) weight. We contract the edge, update the weights and pick the next edge of optimal weight which is not incident to an already matched vertex.

We decided not to remove the vertex that results from contracting the selected edge because during the execution of the algorithm the edge weights should give an approximate description of the local structures in the final contracted graph.

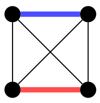
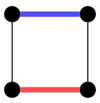
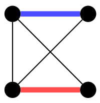
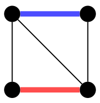
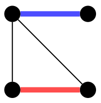
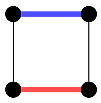
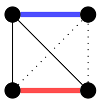
Common Neighbours	 −1	 +1		
Symmetric Difference	 −1	 −1	 −1	 −2
Degree Sum	 −1			

Table 4.3: Contracting the red edge changes the weight of the blue edge by the given value.

Implementation issues

Let us make some remarks on the implementation of this procedure. First of all, one should not recompute all edge weights in every iteration. It is clear that, for the three weighting functions which we used here, it suffices to update the weights of the edges that are "near" to the contracted edge. We can also exclude all edges that share an endpoint with the contracted edge. Since we are not allowed to add such edges to the matching, we do not need to care about their weights.

Table 4.3 shows an overview of all possible situations in which a weight is changed after an edge contraction. The values specify by how much the weight is changed. For common neighbours and symmetric difference, one has to consider all such induced subgraphs of size four as depicted, for instance by iterating over all edges incident to a neighbour of an endpoint of the matching edge and verifying whether one of the cases is fulfilled. For degree sum, one has to consider all common neighbours w of the contraction edge $\{u, v\}$ and decrease the weight of every edge incident to w by one (except for $\{u, w\}$ and $\{v, w\}$). If mixture weightings are used, one has to be more careful in the updating step, as the change of the primary weight causes a change by a certain factor in the mixture weight (factor m in the formula above).

We use a binary heap as the data structure to store the weights (also an implementation from the LEMON library). Essentially, it is a priority queue which also allows us to change the priorities of elements in the heap. The usage of a binary heap turned out to improve the speed of the algorithm drastically when we switched from using STL vectors to binary heaps.

graph name	none	cn	ds	sd	cn+sd	cn+ds	sd+cn	sd+ds	ds+cn	ds+sd
fl3795	13	19	19	20	21	19	24	22	19	19
fnl4461	37	55	49	56	59	50	50	50	55	51
pcb3038	30	46	49	45	41	45	40	40	45	49
rl5915	28	39	41	44	41	49	50	47	49	49
rl5934	26	58	49	45	39	45	42	42	52	43

Table 4.4: Greedy weighted matching algorithm on TSP instances (base algorithm: mdfi). A mixture weighting with primary and secondary weighting w_1 and w_2 is denoted by w_1+w_2.

Evaluation

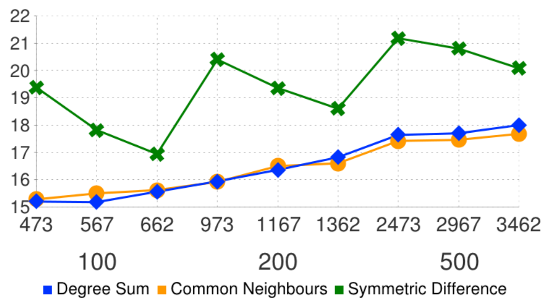
Again we only tested with the base algorithm MinimumDegreeFillIn. In Table 4.2 we can see that the greedy approach works better than using maximum weight matchings and in most of the cases there is at least one weighting function such that the greedy approach gives a better bound than mm. On smaller graphs the criteria common neighbours and symmetric difference give better bounds than degree sum; for larger graphs the degree sum criterion is clearly the winner. Still this approach is not able to outperform MinimumDegreeFillIn.

Another set of five test graphs, which are travelling salesman problem instances[1], shows that the situation is not so clear as it seems. In Table 4.4 we find that on different graphs different weighting functions can perform better and, except for “fnl4461”, all graphs have roughtly the same size of 2000 vertices.

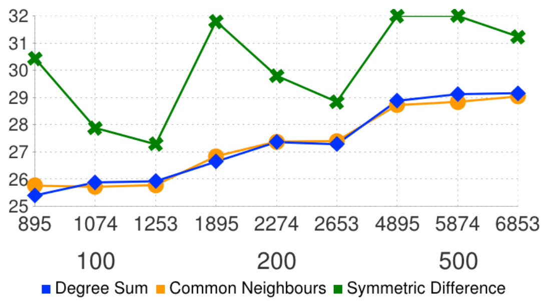
To analyze this in more detail, we ran the algorithms on random partial k-trees with different parameters. We used the parameter values $n \in \{100, 200, 500\}$, $k \in \{10, 20\}$ and $p \in \{0.3, 0.4, 0.5\}$. For every combination of parameters, we generated 50 random graphs and ran the algorithms using the three weighting functions on each of these graphs. Figure 4.1 shows the average upper bound that is computed on such a set of 50 random graphs for different weighting functions. On the x-axis the number of vertices and expected number of edges are displayed.

The plots clearly indicate that degree sum and common neighbours perform better than symmetric difference on random partial k-trees. Interestingly, symmetric difference improves with increasing edge density, whereas the two other criteria yield better bounds on sparse graphs. On random partial k-trees, degree sum and common neighbours give similar results; common neighbours seems to perform slightly better than degree sum, with increasing vertex and edge number.

[1]from `http://www2.isye.gatech.edu/~wcook/bwidth/`



(a) $k = 10$



(b) $k = 20$

Figure 4.1: Average upper bounds computed by greedy weighted matching algorithms on random partial k-trees

We also tried to generate partial k-trees for larger values of k, trying to simulate graphs such as "scoregraph" or "sheep", to analyze the quality of the algorithms on graphs with large treewidth. However, the two mentioned graphs have a relatively low edge number, considering their large treewidth; hence we need to remove a lot of edges from a randomly generated k-tree to replicate such graphs. In our experiments it turned out that the computed lower bound always dropped to a value lower than k, after removing the edges from the random k-tree. It seems to be very hard to artificially construct random sparse graphs with high treewidth (at least with the method used here).

Finally, we also tested using all six possible mixture weightings on the TSP instances, which is also presented in Table 4.4. In some cases the use of a mixture weighting can give an improvement to using only the primary weighting.

4.5 Maximum edge reduction

We have already seen that contracting edges with many common neighbours is a reasonable method to lower the treewidth of a graph. As a reason we mentioned the large decrease of the edge number, when such edges are contracted. Next, we want to study whether we can improve the widths computed, if this idea is taken further. We call $|E(G)| - |E(G/M)|$ the *edge reduction* for a given matching M. The problem that we want to solve in this section is: Find a matching M which maximizes the edge reduction.

We believe that this problem is NP-complete but due to the lack of time we were not able to work further on this. We solve the problem by formulating it as an integer linear program. In the following, we present two possible formulations for the maximum edge reduction problem.

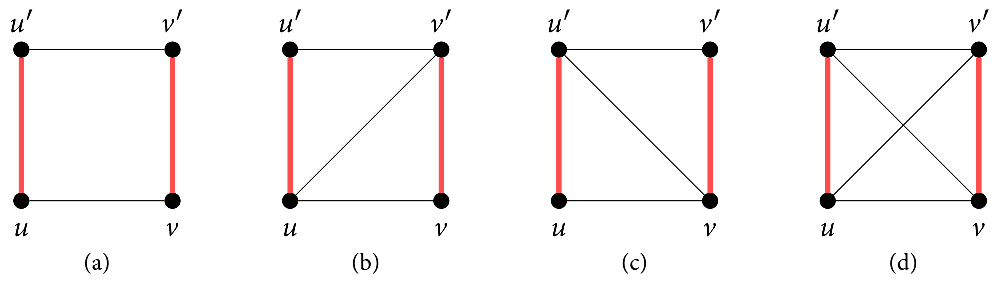


Figure 4.2: Possible cases for two matching edges contained in a 4-cycle

4.5.1 ILP Formulation A

For the first formulation (ilp-a) we have to think about how we can express the edge reduction for a given matching in a integer linear program. After the contraction of a matching M, there are three different ways how an edge $\{u, v\}$ could have "vanished":

1. The edge is a matching edge: $\{u, v\} \in M$.
2. The endpoints u and v share a common neighbour w, and either $\{u, w\} \in M$ or $\{v, w\} \in M$. Then $\{u, v\}$ is merged with $\{v, w\}$ or $\{u, w\}$ respectively.
3. There are matching edges $\{u, u'\}, \{v, v'\} \in M$ incident to u and v such that $\{u, u', v, v'\}$ forms a 4-cycle. Then $\{u, v\}$ is merged with $\{u', v'\}$.

We have to pay attention not to count edges twice because these cases may intersect.

Combining case 1 and 2 we add $1 + \text{cn}(e)$ for every $e \in M$ to the edge reduction count where $\text{cn}(\{u, v\})$ denotes the number of common neighbours of u and v. Now consider the third case; all possible subcases are shown in Figure 4.2. The edge reduction for subcases (b) and (c) is already correctly counted by our estimate (reduction of four edges), but in subcase (d) we have a reduction of five edges, whereas our estimate counts six. Hence we need to subtract one for all such 4-cliques that contain two matching edges. Lastly, in case (a) the edges $\{u, v\}$ and $\{u', v'\}$ are merged, thus we add one for all induced 4-cycles containing two matching edges.

Before we can formulate the problem as an integer linear program, we define the sets $S_{\square}$ and $S_{\boxtimes}$, which are sets of unordered pairs of edges. If $\{u, v\}$ and $\{w, x\}$ are two vertex-disjoint edges, we define

$$\{\{u, v\}, \{w, x\}\} \in S_{\square} \iff \{u, w\}, \{v, x\} \in E \text{ and } \{u, x\}, \{v, w\} \notin E$$

and

$$\{\{u,v\},\{w,x\}\} \in S_{\boxtimes} \iff \{u,w\},\{u,x\},\{v,w\},\{v,x\} \in E.$$

Summing the results so far up, we can calculate the edge reduction for a matching M by

$$\sum_{e \in M} (\mathrm{cn}(e) + 1) + |\{\{e,f\} \in S_{\square} : e,f \in M\}| - |\{\{e,f\} \in S_{\boxtimes} : e,f \in M\}|.$$

For the integer linear program, we use binary variables X_e for all edges $e \in E$ with the interpretation $X_e = 1$ if and only if e is in the matching. Also we introduce binary variables $Y_{\{e,f\}}$ for all $\{e,f\} \in S_{\square} \mathbin{\dot\cup} S_{\boxtimes}$, which shall assume the value $Y_{\{e,f\}} = \min\{X_e, X_f\}$, that means $Y_{\{e,f\}} = 1$ if and only if both edges e and f are in the matching, which are the situations of interest as discussed previously. The ILP is now formulated as follows:

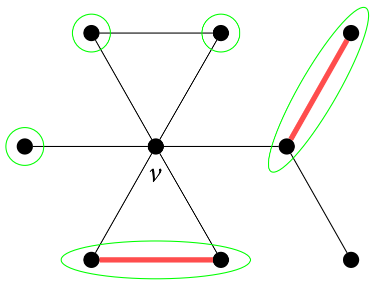
$$\begin{array}{lll}
\text{maximize} & \sum\limits_{e \in E} (\mathrm{cn}(e) + 1) \cdot X_e + \sum\limits_{\{e,f\} \in S_{\square}} Y_{\{e,f\}} - \sum\limits_{\{e,f\} \in S_{\boxtimes}} Y_{\{e,f\}} & \\
\text{subject to} & \sum_{e \ni v} X_e \le 1, & \text{for all } v \in V \\
 & Y_{\{e,f\}} \le X_e, & \\
 & & \text{for all } \{e,f\} \in S_{\square} \\
 & Y_{\{e,f\}} \le X_f, & \\
 & X_e + X_f - Y_{\{e,f\}} \le 1, & \text{for all } \{e,f\} \in S_{\boxtimes} \\
 & X_e, Y_s \in \{0,1\} & \text{for all } e \in E,\ s \in S_{\boxtimes} \cup S_{\square}
\end{array}$$

The first inequality expresses that the variables X_e encode a valid matching. Since the variables $Y_{\{e,f\}}$ are maximized for $\{e,f\} \in S_{\square}$ and minimized for $\{e,f\} \in S_{\boxtimes}$, the other inequalities ensure that $Y_{\{e,f\}}$ indeed is the minimum of X_e and X_f.

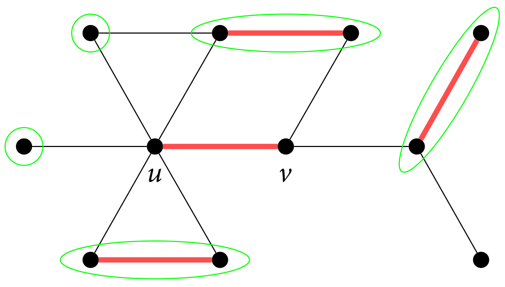
In the worst case, the set $S_{\boxtimes}$ contains all edge pairs (e.g. if the graph is a complete graph). Thus, this formulation has $O(n^2)$ variables and $O(n^2)$ inequality constraints but for sparse graphs, the size will be much lower.

4.5.2 ILP Formulation B

In our second formulation (ilp-b) we directly minimize the edge number of G/M instead of maximizing the edge reduction. To encode a matching $M \subseteq E$ we introduce a binary variable for every



(a) neighbourhood of $f_M(v)$



(b) neighbourhood of $f_M(\{u, v\})$

Figure 4.3: Vertices and edges in G (marked green) become a neighbourhood in G/M.

vertex and now also for every edge:

$$X_v = \begin{cases} 1, & \text{if } v \text{ is not matched by } M, \\ 0, & \text{otherwise,} \end{cases} \qquad \text{for all } v \in V,$$

$$X_e = \begin{cases} 1, & \text{if } e \in M, \\ 0, & \text{otherwise,} \end{cases} \qquad \text{for all } e \in E.$$

The vertex set of G/M can then be identified with $\{v \in V : X_v = 1\} \cup \{e \in E : X_e = 1\}$. We introduce further integer variables D_v and D_e for all vertices $v \in V$ and edges $e \in E$ with the meaning

$$D_v = \begin{cases} \deg(f_M(v)), & \text{if } v \text{ is not matched by } M, \\ 0, & \text{otherwise,} \end{cases} \qquad \text{for all } v \in V,$$

$$D_e = \begin{cases} \deg(f_M(e)), & \text{if } e \in M, \\ 0, & \text{otherwise,} \end{cases} \qquad \text{for all } e \in E.$$

By the handshaking lemma, instead of minimizing $|E(G/M)|$ we can minimize the following quantity:

$$2 \cdot |E(G/M)| = \sum_{v \in V} D_v + \sum_{e \in E} D_e.$$

Let us see how the neighbourhoods change after contracting a matching M, as in the examples depicted in Figure 4.3. If v is an unmatched vertex, the neighbourhood of $f_M(v)$ includes all unmatched neighbours of v, and also every matching edge incident to a neighbour of v becomes a neighbour vertex. For a matching edge $\{u, v\} \in M$ the neighbourhood of $f_M(\{u, v\})$ includes all unmatched neighbours of either u or v, and also every matching edge (except for $\{u, v\}$) incident to a neighbour of u or v becomes a neighbour vertex in the contracted graph. We define the linear terms

$$\text{degree}(v) := \sum_{u \in N(v)} X_u + \sum_{e \in \text{NI}(v)} X_e$$

and

$$\text{degree}(\{u, v\}) := \sum_{w \in N(u) \cup N(v)} X_w + \sum_{\substack{e \in \text{NI}(u) \cup \text{NI}(v), \\ e \neq \{u,v\}}} X_e,$$

where $\text{NI}(v)$ denotes the set of *neighbour-incident edges* to v

$$\text{NI}(v) = \{e \in E : e \text{ is incident to some } w \in N(v) \text{ but not to } v\}.$$

The new ILP formulation for the Maximum Edge Reduction problem is displayed below:

$$\begin{array}{lll}
\text{minimize} & \sum_{v \in V} D_v + \sum_{e \in E} D_e & \\
\text{subject to} & X_v + \sum_{e \ni v} X_e = 1, & \text{for all } v \in V \\
& \text{degree}(v) \geq D_v, & \text{for all } v \in V \\
& \text{degree}(v) \leq D_v + n \cdot (1 - X_v), & \text{for all } v \in V \\
& D_v \leq n \cdot X_v, & \text{for all } v \in V \\
& \text{degree}(\{u, v\}) \geq D_{\{u,v\}}, & \text{for all } \{u, v\} \in E \\
& \text{degree}(\{u, v\}) \leq D_{\{u,v\}} + n \cdot (1 - X_{\{u,v\}}), & \text{for all } \{u, v\} \in E \\
& D_{\{u,v\}} \leq n \cdot X_{\{u,v\}}, & \text{for all } \{u, v\} \in E \\
& X_v, X_e \in \{0, 1\}, & \text{for all } v \in V, e \in E \\
& D_v, D_e \geq 0, & \text{for all } v \in V, e \in E
\end{array}$$

The first set of equations states that the variables encode a valid matching. If $X_v = 1$, then D_v is forced to assume the value of $\text{degree}(v)$, otherwise D_v is set to zero. Similarly, if $X_e = 1$, then the inequalities express $D_e = \text{degree}(e)$, otherwise we have $D_e = 0$.

This ILP formulation has $2 \cdot (|V| + |E|)$ variables and $4 \cdot |V| + 3 \cdot |E|$ inequality constraints, which improves the size of the previous ILP formulation from quadratic to linear.

graph name	ilp-a	time	ilp-b	time
1ubq	16	4.31	-	-
C5	4	0.01	4	0.05
C5+	3	0.01	3	0.04
Clebsch	9	0.04	9	15.28
knights8_8	27	12.05	-	-
petersen	9	0.02	9	0.36
pyramid3	6	0.05	7	4.21
risk	9	0.17	-	-

Table 4.5: Upper bounds and running times (in seconds) from the two ILP formulations

4.5.3 Evaluation

Table 4.5 shows the results of ilp-a and ilp-b on the collection of 15 graphs. On the graph instances which are not listed here with or marked with "-" the ILP-solver had a running time of more than 60 seconds and we cancelled the computation. The results clearly indicate that these ILP formulations are solvable in reasonable time only for very small graphs and also do not give an improvement to the greedy algorithms using weighting functions. The smallest graph that could not be solved in under a minute (for both formulations) was "kneser8-3" with only 56 vertices and its formulation ilp-a (ilp-b) has 2800 (672) variables and 5096 (1064) constraints. It is interesting to see that ilp-b cannot be solved as fast as ilp-a despite of its linear size, which is probably due to the more complex constraints of the second formulation.

4.6 Matchings from tree decompositions

The heuristics presented so far are not able to outperform the pure application of the base algorithm. A last attempt to achieve that shall be presented in this section, which again uses an integer linear program.

At first, we want to motivate the idea informally using the analogy to trees in the real world. Suppose that we are given a graph with a certain (low) treewidth, i.e. a tree-like graph. To reduce the treewidth of the graph, we should contract many edges that stand perpendicular to the branches of the tree with the goal that all branches should be made thinner. In the language of tree decompositions the perpendicular edges are represented by the bags and the largest thickness of a branch of course stands for the width of the tree decomposition.

This leads to the following approach (which we simply call ilp-c): We run the base algorithm on the input graph G and obtain a tree decomposition $\mathcal{T}$ of G. As described in Lemma 2, $\mathcal{T}$ can be easily transformed into a tree decomposition $\mathcal{T}_H$ of any minor H of G without increasing the width. We look for a matching M such that the contracted graph G/M minimizes the width of $\mathcal{T}_{G/M}$. In this section $\mathcal{T}_{G/M}$ will be simply called the *minor tree decomposition*. This directly gives an upper bound for the treewidth of G/M, but as always in our setting of expanding tree decompositions, we run the base algorithm again on G/M and might get a better width than the width of $\mathcal{T}_{G/M}$.

This approach should be seen dual to the expansion approach: We can use the information of a (non-optimal) tree decomposition, i.e. a description of how the graph is structured like a tree, to find a matching whose contraction causes a large treewidth reduction. The expansion approach, which is done afterwards, conversely uses that matching to compute a new tree decomposition of the graph.

Consider the following ILP formulation for a given a graph $G = (V, E)$ and a tree decomposition $\mathcal{T} = (\{B_i\}_{i \in I}, T)$.

$$\begin{array}{lll}
\text{minimize} & w & \\
\text{subject to} & \sum_{e \ni v} X_e \leq 1, & \text{for all } v \in V \\
 & |B_i| - \sum_{e \subseteq B_i, e \in E} X_e \leq w, & \text{for all } i \in I \\
 & w \geq 0, X_e \in \{0,1\}, & \text{for all } e \in E
\end{array}$$

Here we use the variable names B_i for the bags to avoid confusion with the variables X_e. Again we use binary variables X_e for all edges $e \in E$, which encode the matching, and an additional integer variable w, which is the maximal bag size of the minor tree decomposition. The first set of inequalities again expresses that the variables X_e encode a valid matching. By contracting a matching M the number of vertices in a bag B_i is reduced by the number of matching edges whose endpoints are both in B_i. The second set of inequalities assures that w assumes the maximum bag size of the minor tree decomposition, since w is minimized.

Evaluation

For the experiments, we only used MinimumDegreeFillIn as the base algorithm. Table 4.6 compares the upper bounds computed by ilp-c with MinimumDegreeFillIn. We also list the widths from the intermediate steps, i.e. the width of the minor tree decomposition and of the tree decomposition that is computed for the contracted graph G/M (columns: minor and con).

graph name	none	minor	con	ilp-c
1ubq	12	7	7	13
C5	2	2	2	2
C5+	2	1	1	3
Clebsch	10	5	5	10
gr216	91	52	52	91
hamming9-2	203	203	203	203
kneser8-3	34	17	17	34
knights8_8	20	11	11	21
macaque71	21	10	11	21
petersen	4	3	3	6
pyramid3	5	3	3	5
risk	5	3	3	6
scoregraph	320	163	163	321
sheep	330	234	267	361
sudoku	51	25	25	51

Table 4.6: Widths computed in ilp-c compared to mdfi

In most of the cases the ILP solution the minor tree decomposition width is close to a half of the original width such that the final upper bound has a difference of at most by one to the width from MinimumDegreeFillIn. Hence, ilp-c cannot improve the performance of the base algorithm. For the "sheep" graph, we again have an interesting situation because MinimumDegreeFillIn computes an upper bound for the contracted graph that is larger than the width that we get from the minor tree decomposition. For the graph instance "hamming9-2" we also have an odd case, as the solution of the ILP yielded that no edges should be contracted.

Although this approach seems to have failed, we can learn something from the results. The experiment proved that many of our test graphs are actually "good" instances, in the sense that for these graphs indeed there exist matching contractions that can lower the treewidth of the graph to its half. This gives us hope that our general approach is not totally pointless but might have room for improvement. Maybe we simply have not found the right algorithms yet to compute good, treewidth reducing matching contractions.

graph name	none	time	td	time	eo	time	td+pp	time	eo+pp	time
fl3795	13	9.65	19	3.65	18	4.27	18	5.11	18	5.41
fnl4461	37	22.29	49	8.51	40	10.53	43	12.67	37	13.46
pcb3038	30	7.84	49	3.00	43	3.99	42	4.90	38	5.21
rl5915	28	7.35	41	3.04	36	3.72	34	4.66	31	4.73
rl5934	26	8.10	49	3.34	46	4.17	41	5.61	38	5.25
scoregraph	320	58.14	371	18.72	366	158.13	351	176.05	351	293.96
sheep	330	249.28	439	98.08	432	535.38	361	602.92	326	915.90

Table 4.7: Upper bounds and running times of greedy weighted matching algorithm. td: tree decomposition expansion, eo: elimination ordering expansion, pp: post-processing. (base algorithm: mdfi)

4.7 Further extensions and running time analysis

Finally we want to compare the expansion of tree decompositions with the expansion of elimination orderings. In the worse case, both approaches can give an increase from width k to $2k + 1$. Additionally, we will test the presented post-processing method and have a look at the running times of our algorithms compared to the ones of the base algorithm. We will use the greedy weighting matching algorithm with the degree sum weighting, which seemed to be the best performing algorithm of the presented ones.

We use the base algorithm MinimumDegreeFillIn. As usual we also run the pure MinimumDegreeFillIn algorithm on the test instances as a reference point, but this time we additionally convert the computed elimination ordering into a tree decomposition, as it would be needed in a practical settings. Our developed algorithms also output tree decompositions, not just the upper bounds or elimination orderings, and this allows a fair comparison regarding the running times.

Table 4.7 shows the results on the five TSP graphs, “scoregraph” and “sheep”. We see that expanding elimination orderings generally gives better bounds than expanding tree decompositions. The relative difference is larger on the TSP graphs with smaller treewidths than on “scoregraph” and “sheep” having large treewidths. Adding the post-processing to the tree decomposition approach generally (td+pp) takes slightly more time than the elimination ordering approach (eo) and for most of the graphs the algorithm td+pp computed better bounds. Both extensions combined (eo+pp) can result in a total improvement up to 25% to the standard tree decomposition approach without post-processing. This algorithm unexpectedly even outperformed MinimumDegreeFillIn on the “sheep” graph.

However, we have to admit that these results for the “sheep” graph took quite a long computation time, clearly longer than MinimumDegreeFillIn. On the five TSP graphs the greedy weighted matching algorithm always performed faster than MinimumDegreeFillIn, but for the computed bounds the base algorithm proves to be better. On average the algorithm eo+pp improved the running time of MinimumDegreeFillIn by 40% in trade for an increase of the upper bound by around 20%.

In practical applications, algorithms that use tree decompositions usually have running times which are exponential in the width of the given tree decomposition. Thus, the computation of a good tree decomposition is often worth accepting longer running times of the tree decomposition algorithm. In the light of that, we think that one would prefer the greedy triangulation algorithms, such as MinimumDegreeFillIn, over our algorithms.

5 Summary

In this thesis we have studied a general method to compute tree decompositions with the help of a matching contracted graph and an additional tree decomposition algorithm. We presented the possible approaches of expanding tree decompositions or elimination orderings and discussed several matching algorithms, which we evaluated in computational experiments. We come to the conclusion that the algorithms developed often compute widths that are far lower than the predicted upper bound of $2k+1$, where k is the upper bound computed for the contracted graph, but rarely give widths that are lower than k. Our best performing algorithm computes the matching by greedily picking edges according to a certain edge weighting. Considering both the quality of the computed upper bounds and the running times, this algorithm gives the best results when executed on larger graphs (1000 and more vertices).

Clearly, the main issue lies in designing a good matching algorithm. An experiment using a specific algorithm demonstrated that the treewidth of a graph can be lowered to its half by a matching contraction but in this thesis the algorithms proposed either could not decrease the treewidth low enough or had unacceptable long running times. Towards finding better matching algorithms, we think that one should focus on the open question of the existence of a polynomial-time algorithm that minimizes the treewidth of a graph by contracting a matching.

We believe that our approach can be improved in several directions. We have seen that using our approach the running times of the base algorithms can be reduced considerably, which leaves space for further computations to improve the widths of the resulting tree decompositions. Better post-processing methods could lower the width even more than in our experiments. Instead of using greedy triangulation algorithms, one could also try out other base algorithms which might require more time to compute better upper bounds. We also mentioned already that instead of contracting only matchings, one could study whether merging larger connected subgraphs into single vertices can improve our results.

A last idea we propose is a recursive algorithm that on every recursion level expands a computed tree decomposition (or elimination ordering) and lowers the width of the expansion using post-processing methods. This resembles the procedure of Bodlaender's algorithm but in the form of a non-exact heuristic algorithm.

Bibliography

[Ach09] Tobias Achterberg. SCIP: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[ACP87] S. Arnborg, D. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

[BGHK92] Hans L. Bodlaender, John R. Gilbert, Hjálmtýr Hafsteinsson, and Ton Kloks. Approximating treewidth, pathwidth, and minimum elimination tree height. In *Graph-Theoretic Concepts in Computer Science*, pages 1–12. Springer, 1992.

[BK96] Hans L. Bodlaender and Ton Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *Journal of Algorithms*, 21(2):358–402, 1996.

[BK10] Hans L. Bodlaender and Arie M.C.A. Koster. Treewidth computations I. Upper bounds. *Information and Computation*, 208(3):259–275, 2010.

[BK11] Hans L. Bodlaender and A.M.C.A. Koster. Treewidth computations II. Lower bounds. *Information and Computation*, 209(7):1103–1119, 2011.

[Bod96] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996.

[Bod98] Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. In *J. Algorithms*, pages 1–16. Springer, 1998.

[Cou90] B. Courcelle. The monadic second order theory of Graphs I: Recognisable sets of finite graphs. *Information and Computation*, 85:12–75, 1990.

[DJK11] Balázs Dezs, Alpár Jüttner, and Péter Kovács. LEMON – an Open Source C++ Graph Template Library. *Electron. Notes Theor. Comput. Sci.*, 264(5):23–45, July 2011.

[Edm65] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17(3):449–467, 1965.

[FG65] D.R. Fulkerson and O.A. Gross. Incidence matrices and interval graphs. *Pacific Journal of Math.*, 15:835–855, 1965.

[Liu85] Joseph W. H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. Math. Softw.*, 11(2):141–153, June 1985.

[RS86] Neil Robertson and P.D. Seymour. Graph Minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986.

[RS04] Neil Robertson and P.D. Seymour. Graph Minors. XX. Wagner's conjecture. *Journal of Combinatorial Theory, Series B*, 92(2):325–357, 2004.

[Rö98] Hein Röhrig. Tree decomposition: A feasibility study. Master's thesis, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1998.

[ST93] P. D. Seymour and Robin Thomas. Graph searching and a min-max theorem for tree-width. *J. Comb. Theory Ser. B*, 58(1):22–33, May 1993.

[TY84] R. Tarjan and M. Yannakakis. Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, 1984.