Utrecht University

Faculty of Science
Department of Information and Computing Sciences

Master's Thesis

A simulated annealing method for computing rank-width

Author
Florian Nouwt

Supervisor
Hans L. Bodlaender

13 May 2022

Abstract

In this thesis we show that simulated annealing is a very viable heuristic method for approximating rank-width [38, 47] and other branch-decomposition based width parameters. We present the various aspects of the algorithm in detail and discuss the design choices that were made with the help of practical experiments. Finally, extensive benchmarks were performed to assess the performance of the algorithm. We improved many of the currently best known rank-width upper bounds [6, 7] and show the first practical results for $\mathbb{F}_4$-rank-width [34, 33] and maximum matching-width [54].

Acknowledgements

First of all I would like to thank my supervisor Hans Bodlaender for suggesting the topic of this thesis and the support during our weekly meetings. I would also like to thank Martin Beyß for providing me with his complete thesis [6], which was very useful for comparing my results. Finally I would like to thank Michel Maasdijk for providing me with a modern computer at university I could use for running the final benchmarks.

Contents

Chapter 1

Introduction

Finding faster algorithms for NP-hard problems has always been a widely researched topic in computing science. After all, many real-life problems such as the planning of public transport, vehicle routing and other scheduling problems are NP-hard, and being able to solve them, or at least approximate them in increasingly better ways, can save companies a lot of money. When it comes to NP-hard problems on graphs, width parameters such as tree-width, branch-width and clique-width have been particularly useful. These parameters quantify the structure of a graph and come with a decomposition that makes it possible to design faster algorithms for NP-hard problems with a running time depending on the parameter (and the number of vertices). This is often done using dynamic programming techniques, making use of properties of the decomposition. A major problem with those width parameters, however, is that computing them exactly is usually NP-hard in itself. A lot of research was therefore devoted to finding faster exact and approximation algorithms for them. At the same time, instead of just trying to improve algorithms for existing width parameters, new parameters were also proposed which promised to give a low value for a different or larger variety of graphs and/or being relatively easier to compute, compared to existing parameters. One such parameter is rank-width [38, 47], which was chosen as the main topic for this thesis.

Rank-width is a width parameter for undirected graphs by Oum and Seymour [38, 47]. A rank-decomposition is a subcubic (degree at most 3) tree that has exactly one leaf for each vertex of the input graph. Each edge of the decomposition naturally represents a partition between the leaves on each side of it. The width of an edge is the rank of the binary field adjacency matrix of this partition in the input graph. The width of the decomposition is the maximum over all edges. The rank-width of a graph is then the smallest possible decomposition width. Note that for algorithms that use a (rank-)decomposition as part of the input it is not required to have an optimal decomposition. As long as the width of the given decomposition is low enough, they can still work efficiently. This makes it interesting to create faster approximation algorithms that produce low width (but not guaranteed to be optimal) decompositions. Computing the exact rank-width of a graph was proven to be NP-hard [43, 46].

Most of the research that has been done for rank-width so far has been focused on properties, applications and approximations with guarantees on the running time and the quality of the solution. Unfortunately those approximation algorithms are oftentimes not very usable in practice. The only currently existing practical result that approximates rank-width without guarantees appears to be a heuristic algorithm by Beyß [6, 7]. In this thesis we will show that by approximating rank-width using local search variant simulated annealing we can get better and faster results.

This thesis is structured as follows: First some preliminary topics will be explained in Chapter 2, and then Chapter 3 will give an introduction to rank-width, its properties, applications and directed variants. Chapters 4, 5 and 6 explain the various aspects of the algorithm in detail, combined with discussion about the results of experiments that show the effect of various design choices that had to be made. Based on the results of those experiments the best configuration was chosen to perform the final benchmarks in Chapter 7, which eventually leads to a conclusion and a discussion about possible future research directions in Chapter 8. The appendices contain a description of how to use the program that was written for this thesis (Appendix A) and a complete overview of the benchmarking results (Appendix B). Note that all experiments in Chapters 4, 5 and 6 were performed for rank-width. In the final benchmarks, the directed rank-width variant $\mathbb{F}_4$-rank-width [34, 33] and undirected width parameter maximum matching-width [54] were additionally tested, to show that the algorithm can support other width parameters that are based on branch-decompositions.

Chapter 2

Preliminaries

In this chapter some concepts are introduced that are important for understanding the rest of the thesis.

2.1 Branch-decomposition

Width parameters are usually defined through a decomposition for which a width can be computed in some way. A branch-decomposition [22, 47, 8, 54, 14] is a generalized type of decomposition that can be used as a framework to define various width parameters such as rank-width [38, 47], maximum matching-width (mm-width) [54], maximum induced matching-width (mim-width) [54], boolean-width [8, 54] and branch-width [50]. Note that the decomposition belonging to the width parameter "branch-width" is also referred to as a branch-decomposition, but this should not be confused with the generalized branch-decomposition framework. The branch-width width parameter by Robertson and Seymour [50] was the first to use this type of decomposition. The concept of that decomposition was later generalized without changing the name.

Definition 2.1.1 (Branch-decomposition) *Let M be a finite set, which is the ground-set of the decomposition. A branch-decomposition is a pair (T, L), consisting of a subcubic (degree at most 3) tree T and a bijection L from M to the leaves of T [22, 47, 8, 54, 14].*

For each element of the ground-set M there is thus exactly one corresponding leaf in T, and each leaf of T has exactly one corresponding element of M. Since branch-decompositions are usually used with graphs, the ground-set will usually be either the set of vertices or the set of edges of the graph. An example can be seen in Figure 2.1.

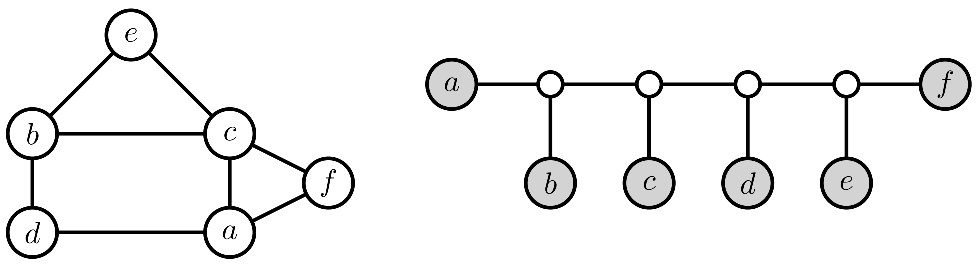


Figure 2.1 *A graph and a possible branch-decomposition with the vertices of the graph as ground-set.*

Definition 2.1.2 (Cut-function) *Let M be a finite set. A cut-function $f : 2^M \to \mathbb{R}$ computes a width value for any subset of M [47, 54]. The function f is symmetric in the sense that $f(X) = f(M \setminus X)$ for all $X \subseteq M$.*

Note that in some places [22, 47] cut-functions are defined to not only be symmetric, but also submodular such that

$$f(X) + f(Y) \geq f(X \cup Y) + f(X \cap Y) \quad \text{for all } X, Y \subseteq M.$$

However, this appears not to be strictly necessary for branch-decompositions and their corresponding width values in general.

Using these definitions it becomes possible to define various width parameters, based on the choice of ground-set and cut-function. A branch-decomposition that uses a specified cut-function f is referred to as a *branch-decomposition of f* [22, 47]. Because the decomposition uses a tree, each decomposition edge e naturally represents a partition of the leaves (A_e, B_e), and thus a partition of the elements of the ground-set M. By using the cut-function f, a width can be computed for every edge: $f(L^{-1}(A_e)) = f(L^{-1}(B_e))$. Note that the equality holds because of the symmetric property of f. The width of the decomposition itself is defined to be the maximum over all its edges. An optimal branch-decomposition of f is a decomposition that realizes the minimum width over all possible decompositions for the ground-set [50, 47, 54]. This minimum width is called the *branch-width of f* [22, 47] (not to be confused with the width parameter branch-width). A branch-decomposition is usually assumed to not have any degree 2 vertices, as each of the two edges incident to such a vertex will have the same width (equal partition), and thus one of those edges could be contracted without influencing the decomposition width [54]. As such, a branch-decomposition for a ground set of size m has m leaves, $m - 2$ internal nodes and $2m - 3$ edges [49]. In Figure 2.1 can be seen that this is indeed the case.

2.2 Maximum matching-width

One of the width parameters that is based on a branch-decomposition is the maximum matching-width (abbreviated as mm-width). It was introduced in 2012 by Vatshelle [54], together with the boolean-width and maximum induced matching-width (mim-width). As ground-set it uses the vertices of the undirected graph, and the cut-function computes the size of a maximum bipartite matching in the graph, where the two sides of the partition form the two sides of the matching. An example of this is shown in Figure 2.2. For a graph with n vertices and m

Figure 2.2 *A graph and a maximum bipartite matching corresponding to the partition $(\{a, b\}, \{c, d, e, f\})$. As there are 2 edges in the matching, the cut-function would result in the value 2 for this partition.*

edges, a maximum bipartite matching can be computed in $O(m\sqrt{n})$ time using the Hopcroft–Karp algorithm [29]. As the size of a maximum bipartite matching is at most the size of the smallest partition side, for a n vertex graph a maximum matching-width upper bound of $\lceil n/3 \rceil$ can be proven with a similar argument as for rank-width (Theorem 3.2.1). Ahn and Jeong [2] showed that computing the maximum matching-width of a graph is NP-hard.

Vatshelle [54] proved the following bounds for an undirected graph G with tree-width $tw(G)$, branch-width $brw(G)$, maximum matching-width $mmw(G)$, rank-width $rw(G)$ and boolean-width $boolw(G)$:

$$\frac{1}{3}(tw(G) + 1) \leq mmw(G) \leq \max(brw(G), 1) \leq tw(G) + 1,$$
$$rw(G) \leq mmw(G),$$
$$boolw(G) \leq mmw(G).$$

2.3 Matrix rank

Width parameter rank-width [38, 47], which is also based on a branch-decomposition, uses the rank of a binary field adjacency matrix for its cut-function. In this section we will briefly describe what the rank of a matrix is and which algorithms exist to compute it, to help in understanding Chapter 3 where rank-width will be discussed in detail.

To define the rank of a matrix, we first need to define the notion of independence of a set of vectors.

Definition 2.3.1 (Independence) *A set V of n vectors $\{v_1, v_2, \ldots, v_n\}$ is independent if there exist no non-trivial coefficients $\lambda_1, \lambda_2, \ldots, \lambda_n$ (at least one non-zero) such that $\lambda_1 v_1 + \lambda_2 v_2 + \cdots + \lambda_n v_n = 0$.*

In practice this means that none of the vectors in the set should be able to be expressed as a linear combination of the others. As such the sets

$$\left\{\begin{pmatrix}1\\3\end{pmatrix}\right\} \quad \text{and} \quad \left\{\begin{pmatrix}2\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}\right\}$$

are independent, and the sets

$$\left\{\begin{pmatrix}2\\0\end{pmatrix}, \begin{pmatrix}4\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}\right\} \quad \text{and} \quad \left\{\begin{pmatrix}0\\0\end{pmatrix}, \begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}\right\}$$

are dependent for example.

Definition 2.3.2 (Rank) *The rank of a set of vectors V is the size of the largest independent subset of V.*

As a matrix is a set of column or row vectors, it also has a rank. It holds that the column rank of a matrix is always equal to its row rank. Rank can also be computed for vectors and matrices in a finite field, as is done for rank-width.

The classic method of computing matrix rank is by using Gaussian elimination to reduce the input matrix to row echelon form (see Figure 2.3). This is a matrix where all-zero rows are at the bottom and the pivot (first non-zero element) of each row is strictly to the right of the pivot of the row above it. In such a matrix, the rank equals the number of non-zero rows. In the worst case, all rows and all columns of the matrix have to be visited, and it

$$\begin{pmatrix} \mathbf{-2} & -3 & 7 \\ 0 & \mathbf{1} & 6 \\ 0 & 0 & \mathbf{3} \end{pmatrix} \quad \begin{pmatrix} \mathbf{1} & 2 & 0 & 5 \\ 0 & \mathbf{1} & -2 & -4 \\ 0 & 0 & \mathbf{1} & 3 \end{pmatrix} \quad \begin{pmatrix} \mathbf{1} & 0 & 2 & 3 & 0 \\ 0 & \mathbf{1} & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

rank = 3 rank = 3 rank = 2

Figure 2.3 *Examples of matrices in row Echelon form and their rank. The pivot element of each row is shown in bold.*

might be needed to add a multiple of a row to many other rows. For a $m \times n$ matrix this results in a $O(mn^2)$ time algorithm. Note that especially for large matrices data locality and memory caching plays an important role in the practical performance. When matrices are stored in row-major order, SIMD instructions can be used to perform operations on rows. This is particularly efficient for fields such as $\mathbb{F}_2$ where each element can be stored as a single bit.

Better, but usually more complicated, algorithms do exist. For $\mathbb{F}_2$ this was shown for example by Albrecht et al. [3], eventually obtaining an average running time of $O(n^3/\log(n))$ for square $n \times n$ matrices. Their algorithm is supposed to be quite optimized for practical performance and is implemented as part of the M4RI library[1] for fast arithmetic on dense $\mathbb{F}_2$ matrices. They also show a simpler algorithm, which is a variant of Gaussian elimination and runs in $O(mnr)$ time for a $m \times n$ matrix with rank r. Another algorithm was proposed by Bertolazzi and Rimoldi [5]. Although no asymptotic running time is mentioned, they show with experimental results that their algorithm performs generally better than the M4RI algorithm. An implementation of their

[1] `https://bitbucket.org/malb/m4ri`

algorithm can be found on GitHub[2]. Cheung et al. [9] showed a fast randomized algorithm that works on any field and, for a given $m \times n$ matrix M and a parameter $k \leq$ min(m, n), computes min$(rank(M), k)$ in $O(|M| +$ min$(k^{\omega}, k \cdot |M|))$ field operations. $|M|$ is here the number of non-zero elements in M, and ω is the matrix multiplication exponent which depends on the used matrix multiplication algorithm. Since the running time depends on the number of non-zero entries in the matrix, this algorithm is especially interesting for sparse matrices. However, because of the randomization it is not entirely clear how usable this algorithm is in practice when an exact result is required.

[2]`https://github.com/ebertolazzi/GF2toolkit`

Chapter 3

Rank-width

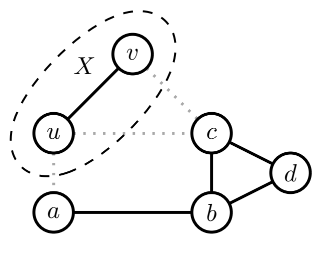
Width parameter rank-width was introduced in 2003 by Oum and Seymour [38, 47] as a replacement for clique-width [39]. Similarly to clique-width it measures the difficulty of decomposing a graph into a tree-like structure. By having a much more mathematical definition, it aims to be easier to compute and design algorithms for [39]. Unlike clique-width, rank-width is based on a branch-decomposition. Another difference is that clique-width works for both directed and undirected graphs, while rank-width only applies to simple undirected graphs. Some effort has however been done on extending rank-width to work with directed and edge-colored graphs as well [34, 33, 32, 35]. Directed rank-width will be further elaborated upon in Section 3.7. A large part of the information in this chapter was inspired by the summary paper "Rank-width: Algorithmic and Structural Results" by Oum [46].

3.1 Definition

To measure the complexity of cuts, rank-width uses the cut-rank function (ρ_G) which is based on matrix rank. An example of computing cut-rank is shown in Figure 3.1. Note that the vertices on each side of the partition do not need to be connected.

Definition 3.1.1 (Cut-rank) *Given an undirected graph $G = (V, E)$ and a subset of its vertices $X \subseteq V$, the cut-rank $\rho_G(X)$ is the rank of the binary-field $\mathbb{F}_2$ adjacency matrix A_X between X and $V \setminus X$ [47, 46].*

Cut-rank was proven to be both symmetric and submodular [47], and is as such a proper cut-function. It is symmetric, because an adjacency matrix of an undirected graph is symmetric and the row-rank of a matrix is equal to its column-rank.



$$X = \{u, v\} \qquad V \setminus X = \{a, b, c, d\}$$

$$A_X = \begin{array}{c} \\ u \\ v \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

$$\rho_G(X) = rank(A_X) = 2$$

Figure 3.1 *An example of computing cut-rank for the partition $(\{u, v\}, \{a, b, c, d\})$.*

Definition 3.1.2 (Rank-decomposition) *Let $G = (V, E)$ be an undirected graph. A rank-decomposition is a branch-decomposition with the cut-rank ρ_G as cut-function and the set of vertices V as ground-set [47, 54]. In other words, a branch-decomposition of ρ_G.*

The rank-width of an undirected graph G, written as $rw(G)$ or $rwd(G)$, is the minimum width over all possible rank-decompositions (the branch-width of ρ_G). In case a graph has no edges its rank-width is 0. Figure 3.2 shows an example of a graph and a corresponding optimal rank-decomposition.

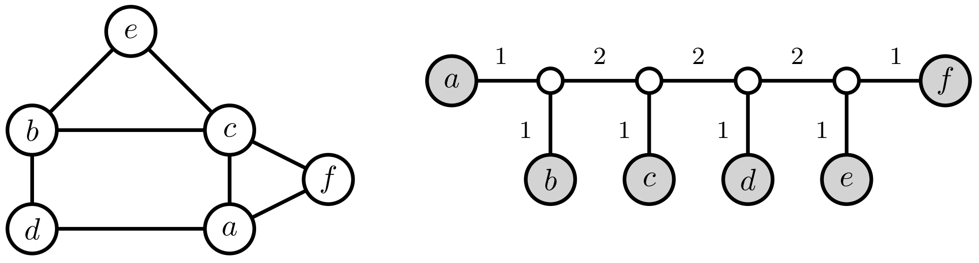


Figure 3.2 *A graph and an optimal rank-decomposition of width 2. The width of the edges is indicated.*

3.2 Bounds

It is not difficult to prove an upper bound for rank-width by using the structure of a rank-decomposition.

Theorem 3.2.1 *Let G be an undirected n-vertex graph. It holds that $rw(G) \leq \lceil n/3 \rceil$ [42, 28].*

Proof. We can show this by constructing a rank-decomposition as shown in Figure 3.3. As each vertex of the tree has degree at most 3, the best we can do is to make a center vertex c which connects to 3 branches. The leaves (gray) are evenly distributed over the three branches, such that each of them has either $\lceil n/3 \rceil$ or $\lfloor n/3 \rfloor$ leaves. From the definition of matrix rank, we know that for an $m \times n$ matrix M it holds that $rank(M) \leq \min(m, n)$. As such, we know that at least one of the three edges incident to c has a width at most $\lceil n/3 \rceil$. The other two edges cannot have a greater width by the way the construction was made. Since the width of a rank-decomposition is the maximum edge-width, the only thing left to prove now is that there is no other edge that can have a greater width. It should not be hard to see that when we move along each of the three branches that one dimension of the matrix becomes increasingly smaller, while the other dimension grows. Since the rank is at most the smallest of the two, the maximum rank and thus the maximum width is decreasing. As such, there is no edge which can have a width greater than $\lceil n/3 \rceil$ and thus the width of the decomposition is at most $\lceil n/3 \rceil$. Because the rank-width of a graph is determined by the smallest width decomposition, we can say that it will be no worse than the decomposition we constructed, and thus $rw(G) \leq \lceil n/3 \rceil$. □

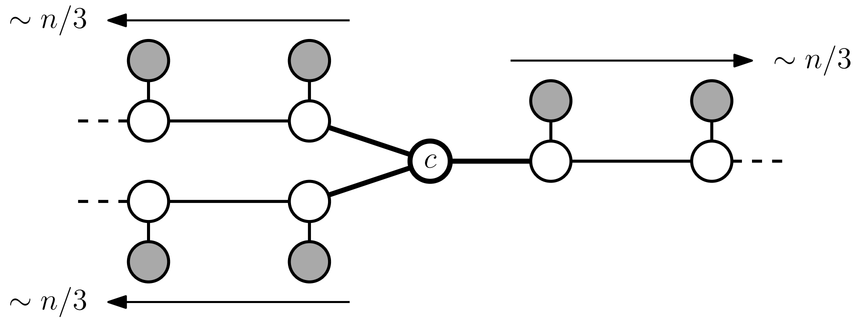


Figure 3.3 *A rank-decomposition used to prove that the rank-width of any n-vertex graph is at most $\lceil n/3 \rceil$.*

Some other width parameters also provide bounds for rank-width. Oum and Seymour [47] showed that a class of graphs has bounded rank-width if and only if it has bounded clique-width (cw). For any undirected graph G:

$$rw(G) \leq cw(G) \leq 2^{rw(G)+1} - 1.$$

A similar relation was shown by Bui-Xuan et al. [8] for boolean-width, denoted as $boolw(G)$ for an undirected graph G:

$$\log_2 rw(G) \leq boolw(G) \leq \frac{rw(G)^2}{4} + O(rw(G)).$$

Oum [44] also showed the following two bounds for an undirected graph G with branch-width $brw(G)$ and tree-width $tw(G)$:

$$rw(G) \leq \max(brw(G), 1),$$
$$rw(G) \leq tw(G) + 1.$$

It is interesting to note that for planar graphs branch-width can be computed exactly in polynomial time [51, 25, 26]. As shown before, Vatshelle [54] proved a bound for maximum matching-width (mmw). For an undirected graph G:

$$rw(G) \leq mmw(G).$$

3.3 Computational results

Computing rank-width exactly is not trivial. In [43] and [46] Oum shows that computing rank-width is NP-hard and that the decision variant, which tests given a graph G and a value k if the rank-width of G is at most k, is NP-complete. It was shown by Oum [45] that computing rank-width exactly for an arbitrary n-vertex graph can be done in $O(2^n n^3 \log^2 n \log\log n)$ time and for bipartite circle graphs in polynomial time [46]. The exponential time algorithm was implemented in SageMath [1]. Since the complexity of the algorithm is quite high, it quickly becomes too slow to be usable when the number of vertices becomes larger than approximately 20.

Because of the high cost to compute rank-width exactly for arbitrary graphs, most research has been focused on constructing approximation algorithms. The existing fixed-parameter approximations compute, for a given n-vertex graph, a rank-decomposition of width at most $f(k)$ for some function f, or confirm that the rank-width is greater than k [46]. The first of such an algorithm was shown by Oum and Seymour [47] in 2006 in the same paper where rank-width was introduced. This algorithm uses $f(k) = 3k+1$ and has an $O(8^k n^9 \log n)$ running time [46]. In 2008 it was further improved by Oum [43] to a running time of $O(8^k n^4)$, while keeping the same approximation quality. In the same paper, another algorithm which provides a slightly better approximation was presented with $f(k) = 3k - 1$ and a running time of $O(g(k) \cdot n^3)$, where $g(k)$ is some huge function.

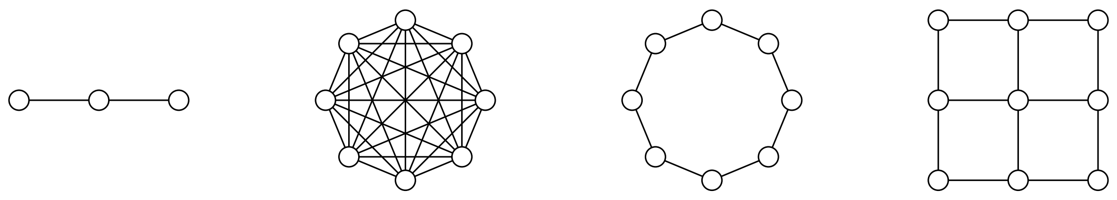
Another category of algorithms simply computes a rank-decomposition of width at most k, if it exists [46]. A generic result by Oum and Seymour [48] in 2007, which applies to functions like cut-rank, implied that this can be done with a running time of $O(n^{8k+12} \log n)$. A better result was shown by Courcelle and Oum [12], which takes $O(g(k) \cdot n^3)$ time with $g(k)$ a huge function. This algorithm was however not able to compute a decomposition directly. This issue was later solved by Hliněný and Oum [27], resulting in an algorithm with again a running time of $O(g(k) \cdot n^3)$ with $g(k)$ a huge function. This means that for any fixed k, a rank-decomposition of width at most k can be computed in cubic time, if it exists. But it still does not scale very well for larger values of k.

One particularly interesting result are the heuristic algorithms by Beyß [6, 7] which attempt to compute an upper and a lower bound for the rank-width of a given graph. This seems to be the only work that has been done on rank-width approximation algorithms without guarantees and also the first work that provides practical rank-width upper and lower bounds for a large number of graphs. The upper bound algorithm is some sort of local search algorithm (although not explicitly described as such) that attempts to improve subtrees of the decomposition tree using "*a mix of greedy and random decisions*" [6] as Beyß describes it. The main loop of the algorithm only accepts improvements of the decomposition, and the algorithm stops if it cannot find improvement for a given number of iterations, or after a time limit. The lower bound algorithm works by computing the width of all possible rank-decompositions for an induced subgraph and then growing the subgraph and making new decompositions based on the previous ones. Although some tricks are used to reduce the size of the search space, the algorithm scales quite badly and can only compute small lower bounds.

[1]`https://doc.sagemath.org/html/en/reference/graphs/sage/graphs/graph_decompositions/rankwidth.html`

3.4 Exact rank-width of graph classes

Aside from trying to compute rank-width for arbitrary graphs, there are also classes of graphs for which can be proven that the rank-width is always a certain number. This is interesting, because for graphs that belong in such a class, it becomes easy to determine the rank-width.



(a) *Path graph P_3* **(b)** *Complete graph K_8* **(c)** *Cycle graph C_8* **(d)** 3×3 *grid $G_{3,3}$*

Figure 3.4 *Examples of graphs from classes with bounded rank-width.*

Path graphs (also linear graphs), denoted by P_n with n the number of vertices, are trees with degree at most two (see Figure 3.4a). As such, they form a single line of vertices, and it should not be hard to see that this line can easily be turned into a rank-decomposition where every partition has only a single connection between the two parts. This implies that the corresponding adjacency matrix contains only a single 1, and this results in a matrix rank and edge-width of 1. This type of graphs will thus always have a rank-width equal to 1, as long as they have at least two vertices [42].

Complete graphs K_n with n the number of vertices have an edge between every pair of vertices, as shown in Figure 3.4b. This means that no matter which cut is made, the corresponding adjacency matrix will only consist of ones. This results in a rank of 1. Any complete graph with at least two vertices thus has a rank-width of 1 [42].

Another class of graphs with a known rank-width is the class of cycle graphs C_n. As shown in Figure 3.4c, these graphs have a single cycle of n vertices, such that each vertex has degree 2. According to Oum [42] and Hliněný et al. [28] it holds that

$$rw(C_n) = \begin{cases} 1 & \text{if } n = 3, 4, \\ 2 & \text{if } n \geq 5. \end{cases}$$

No proof for this was shown, however. For the case of $n = 3$ the rank-width of 1 follows directly from Theorem 3.2.1, and for $n = 4$ it is not difficult to manually construct a decomposition with a width of 1. For $n \geq 5$, the combination of having 5 or more vertices with every vertex having degree 2 seems to make it impossible for all of the edges to have a width of 1. The main reason for this appears to be that there can no longer be two vertices that connect to exactly the same other two vertices, which is required to reach a rank of 1 instead of 2.

A class of graphs for which it was not as trivial to prove the exact rank-width was the class of square $n \times n$ grids $G_{n,n}$ (Figure 3.4d). It was an open question for a while, but in 2008 it was proven by Jelínek [31] that the rank-width of $G_{n,n}$ equals $n - 1$.

It also holds that a graph has rank-width at most 1 if and only if it is a distance-hereditary graph [40]. This means that the shortest path between any two vertices in any induced subgraph is also the shortest path in the original graph.

3.5 Graph relations

For certain operations that can be performed on graphs it is known which effect they have on the rank-width. Looking at graphs that relate to each other through a series of such operations can thus be a useful tool in rank-width research. Oum [41] showed the following relations for a number of simple operations on a graph G, its vertex v, its edge e and another graph H:

- $-1 \leq rw(G \setminus v) - rw(G) \leq 0$ removing a vertex decreases rank-width by at most 1,

- $|rw(G \setminus e) - rw(G)| \leq 1$ removing an edge changes rank-width by at most 1,
- $|rw(\bar{G}) - rw(G)| \leq 1$ taking the complement changes rank-width by at most 1,
- $rw(G \oplus H) = \max(rw(G), rw(H))$ the rank-width of a disjoint union is the max of both graphs.

For two more complex operations the effect is also known [46].

Definition 3.5.1 (Local complement) *Given a graph G and its vertex v, the graph $G * v$ is the local complement at v. This is the same graph as G, but with the subgraph induced by the neighbors of v replaced by its complement [40].*

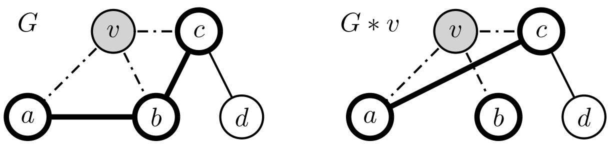


Figure 3.5 *An example of performing local complementation at v.*

Figure 3.5 shows an example of local complementation. Oum [40] showed that performing this operation preserves cut-rank, and thus rank-width. Two graphs that can be transformed into each other by a series of local complementations are said to be *locally equivalent*. When a graph H is an induced subgraph of a graph locally equivalent to a graph G, then H is a *vertex-minor* of G, and it holds that $rw(H) \leq rw(G)$. This works because as we have seen, removing vertices (taking an induced subgraph) does not increase rank-width [40].

For every rank-width at most k it can be shown that there exists a set of *forbidden vertex-minors* (also known as excluded vertex-minors) [46]. Those are graphs of at most $(6^{k+1} - 1)/5$ vertices that cannot be a vertex-minor of a graph with rank-width at most k, because otherwise the rank-width would have been higher [40].

Definition 3.5.2 (Pivoting) *Let G be a graph and uv an edge of G. Additionally, let X be the set of common neighbors of u and v, Y the set of unique neighbors of u and Z the set of unique neighbors of v (all sets exclude u and v). Pivoting on uv results in the graph $G \wedge uv$. This is the same graph as G, but with the adjacencies complemented between X and Y, X and Z, and Y and Z, and the labels of u and v exchanged [42].*

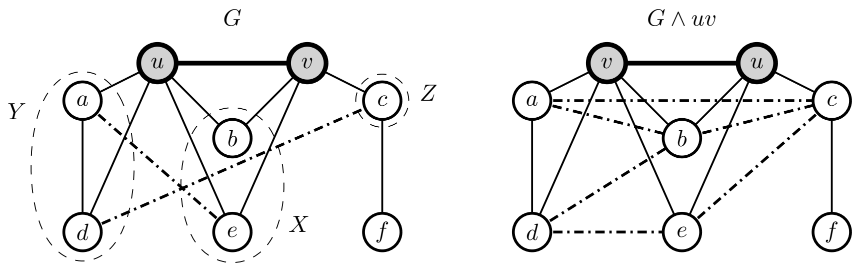


Figure 3.6 *An example of pivoting on edge uv [13].*

Figure 3.6 shows an example of how pivoting works. Just like local complementation, the operation preserves cut-rank [40]. In fact, it can be written in terms of local complementation: $G \wedge uv = G * u * v * u$ [40]. We can also say that two graphs are *pivot equivalent* if they can be changed into each other through a sequence of pivot operations. When a graph H is an induced subgraph of a graph pivot equivalent to a graph G, then H is a *pivot-minor* of G, and it holds that $rw(H) \leq rw(G)$. A pivot-minor is a restricted version of a vertex-minor, and every pivot-minor is also a vertex-minor (but not the other way around).

3.6 Applications

Rank-width is still a relatively new width parameter, and as such there are only a limited number of applications of it at the time of writing. Another reason for this is that a rank-decomposition is much more difficult to use for dynamic programming based algorithms than the decompositions belonging to other parameters [16].

To this extent, a number of alternative characterizations of rank-decompositions were invented in an attempt to make developing algorithms easier. For example by Courcelle and Kanté [10] and by Ganian and Hliněný [15, 16, 21]. Based on their work, Ganian and Hliněný showed that better pseudo-polynomial algorithms are possible for computing the chromatic number, the chromatic polynomial, deciding if a Hamiltonian path exists and a few more [17, 20]. A result that does not use those alternative characterizations is an algorithm for graph isomorphism testing by Grohe and Schweitzer [24]. For every fixed maximum rank-width, their algorithm runs in polynomial time. Aside from algorithms that are designed to work directly with rank-width, it is also possible to translate a rank-decomposition of width k into a clique-width decomposition of width at most $2^{k+1} - 1$ [47]. This means that any algorithm for bounded clique-width could also be used for bounded rank-width, but the running time might be worse than a dedicated algorithm. A result that was already shown before rank-width was even introduced is the meta theorem by Courcelle et al. [11]. It implies that for any fixed k, for every closed monadic second-order formula of the first kind (MSO_1) it can be decided in $O(n^3)$ time if a n-vertex input graph of rank-width at most k satisfies the formula [46].

3.7 Directed rank-width

When it comes to extending rank-width from undirected to directed graphs, not much literature is available. The most important work has been done by Kanté [34, 33] who introduced two rank-width based width parameters for directed graphs: bi-rank-width and $\mathbb{F}_4$-rank-width. Some later work was also done together with Rao [32]. The first parameter keeps using an $\mathbb{F}_2$ adjacency matrix (but no longer symmetric) and changes cut-rank to include both possible arc directions. The second one changes the adjacency matrix to be in $\mathbb{F}_4$ instead, without changing the actual definition of cut-rank. This is particularly interesting, because undirected graphs can be converted to directed in such a way that the width stays equal with this parameter. For a directed graph G we will write $birw(G)$ to denote its bi-rank-width and $rw_{\mathbb{F}_4}(G)$ to denote its $\mathbb{F}_4$-rank-width. It holds that [33]

$$rw_{\mathbb{F}_4}(G) \leq birw(G) \leq 4 \cdot rw_{\mathbb{F}_4}(G).$$

This means that they are equivalent parameters in the sense that one is bounded if and only if the other is bounded. For both parameters a similar equivalence exists in relation with clique-width. As such they provide a way to approximate the clique-width of directed graphs [33]. This is an interesting result as various papers [18, 19] note that clique-width is a quite powerful parameter compared to other width parameters for directed graphs, but computing clique-width directly is difficult.

3.7.1 Bi-rank-width

Bi-rank-width [34, 33] is a modification of rank-width that changes the definition of the cut-rank function. For a directed graph $G = (V, E)$ and its $\mathbb{F}_2$ adjacency matrix A, it holds that $A_{xy} = 1$ if and only if $(x, y) \in E$ (the directed arc from x to y). The adjacency matrix is no longer symmetric, since $A_{xy} \neq A_{yx}$ for unidirectional arcs. For this reason, the cut-rank is modified to incorporate both directions. We will let $A[X, Y]$ be the sub-matrix of A where the rows correspond to vertices in X and the columns to vertices in Y.

Definition 3.7.1 (Bi-cut-rank) *Given a directed graph $G = (V, E)$, a subset of its vertices $X \subseteq V$ and its $\mathbb{F}_2$ adjacency matrix A. The bi-cut-rank $bicutrk(X) = rank(A[X, V \setminus X]) + rank(A[V \setminus X, X])$. [33, 32]*

Just like the regular cut-rank, bi-cut-rank is symmetric and submodular. By replacing the cut-rank in the definition of rank-width by bi-cut-rank, we obtain bi-rank-width. It should not be difficult to see that when an undirected graph G is converted to a directed graph $\overrightarrow{G}$ by replacing every edge by a bidirectional arc, it holds that $birw(\overrightarrow{G}) = 2 \cdot rw(G)$. Kanté and Rao [32] showed that for any fixed k there is an $O(n^3)$ time algorithm that for a n-vertex graph either outputs a decomposition of width at most k or confirms that the $\mathbb{F}_4$-rank-width is greater than k. This is a similar result as for regular rank-width. In relation to clique-width, the following holds for any directed graph G:

$$\tfrac{1}{2}birw(G) \leq cw(G) \leq 2^{birw(G)+1} - 1.$$

It was also shown in [4, Lemma 9.9.15] that for a n-vertex graph, a bi-rank decomposition of width k can be converted to a clique-width expression of width at most $2^{k+1} - 1$ in $O(4^k \cdot n^3)$ time.

Some results for solving problems on graphs of bounded (bi-)rank-width in XP time are shown by Ganian et al. [20]. This includes graph coloring, chromatic polynomial, Hamiltonian path and a few more.

3.7.2 $\mathbb{F}_4$-rank-width

$\mathbb{F}_4$-rank-width (or $GF(4)$-rank-width) [34, 33] is an extension of rank-width to the four element finite field $\mathbb{F}_4$. This field consists of the elements 0, 1, α and α^2, and is a direct extension of the binary field $\mathbb{F}_2$. The field has characteristic 2, and it holds that $1 + \alpha + \alpha^2 = 0$ and $a^3 = 1$. The four elements are used to represent the directions of the arcs in the adjacency matrix. For a directed graph $G = (V, E)$, the adjacency matrix A is constructed as follows [34, 33, 32]:

$$A_{xy} = \begin{cases} 0 & \text{if } (x,y) \notin E \text{ and } (y,x) \notin E, \\ 1 & \text{if } (x,y) \in E \text{ and } (y,x) \in E, \\ \alpha & \text{if } (x,y) \in E \text{ and } (y,x) \notin E, \\ \alpha^2 & \text{if } (x,y) \notin E \text{ and } (y,x) \in E \end{cases} \quad \text{for all } x, y \in V.$$

This matrix is not symmetric like the undirected adjacency matrix, however it is σ-symmetric. This means that there is a function σ, such that for any element A_{xy} it holds that $A_{yx} = \sigma(A_{xy})$. In this case we have that

$$\begin{aligned} \sigma(0) &= 0, \\ \sigma(1) &= 1, \\ \sigma(\alpha) &= \alpha^2, \text{ and} \\ \sigma(\alpha^2) &= \alpha. \end{aligned}$$

The cut-rank function is adapted accordingly to compute the rank on sub-matrices of A in $\mathbb{F}_4$. This preserves the symmetric and submodular properties. All other aspects are identical to regular rank-width. A nice property of $\mathbb{F}_4$-rank-width is that since $\mathbb{F}_4$ is an extension of $\mathbb{F}_2$, any computation performed on purely the elements 0 and 1 is exactly the same as in $\mathbb{F}_2$. This means that a graph that is converted from undirected to directed by introducing a bidirectional arc for every edge of the original graph will have an $\mathbb{F}_4$-rank-width that equals the rank-width of the original graph [32, 35]. As computing matrix rank in a different field does not change the possible range of values, the upper bound for rank-width shown in Theorem 3.2.1 still holds for $\mathbb{F}_4$-rank-width.

Kanté and Rao [32] show that for various properties of regular rank-width similar properties also hold for $\mathbb{F}_4$-rank-width. They show for example variants of local complementation, vertex minors and pivot minors. They also show that, similar to regular rank-width, for any fixed k there is an $O(n^3)$ time algorithm that for a n-vertex graph either outputs a decomposition of width at most k or confirms that the $\mathbb{F}_4$-rank-width is greater than k. There is also a similar relation between $\mathbb{F}_4$-rank-width and clique-width as for regular rank-width [32]. For any directed graph G:

$$rw_{\mathbb{F}_4}(G) \leq cw(G) \leq 2 \cdot 4^{rw_{\mathbb{F}_4}(G)} - 1.$$

Chapter 4

Simulated annealing

In this chapter an overview will be given of the local search variant simulated annealing and an experiment is discussed that determines if using adaptive cooling improves the performance of the algorithm.

4.1 Overview

Simulated annealing is a local search variant inspired by the cooling of materials, in particular the annealing process in metallurgy [52]. The search process uses a temperature (energy) parameter that decreases over time and influences the probability of accepting a worsening of the current solution. By allowing worsening of the solution, the algorithm is able to escape from local minima. A higher temperature means more energy and as such more randomness in the search process. By slowly decreasing the temperature the search is guided towards a stable state. The way in which the temperature is decreased is called the cooling schedule. In Algorithm 4.1 a simplified overview of simulated annealing is shown. For every temperature Q iterations are performed.

Algorithm 4.1 *Simplified overview of simulated annealing*

```
input: $T_0$ = initial temperature
       $Q$ = number of iterations for each temperature
$bestScore \leftarrow score$;            // Initialize bestScore with score of initial solution
$T \leftarrow T_0$;                                                        // Initial temperature
while not end condition reached do
    $curQ \leftarrow Q$;
    while $curQ > 0$ do           // Perform Q iterations with the current temperature
        $oldScore \leftarrow score$;
        randomly select operator;                   // According to probability distribution
        perform operation and update $score$;
        if $score \leq oldScore \vee random() < e^{(oldScore - score)/T}$ then
            if $score < bestScore$ then
                $bestScore \leftarrow score$;
                store current solution as best solution;
        else
            undo operation and restore score;
        $curQ \leftarrow curQ - 1$;
    compute a new value for $T$;                               // Depends on cooling schedule
```

By increasing this value a larger number of permutations is explored before cooling down, which could lead to better results, but might also increase the time it takes to find a good solution. The end condition for the algorithm could be that the temperature becomes smaller than a certain value, but it is also possible to use a time limit, a maximum number of iterations or a combination of conditions. In the algorithm we chose to use a minimum temperature of 0.05 in combination with a configurable time limit.

The search process always starts with an initial solution which is permuted during the search process. In the algorithm we used a random construction as shown in Figure 3.3 on page 8 as initial decomposition. This ensures the initial width is at most $\lceil n/3 \rceil$ with n the number vertices of the input graph. This bound also holds for $\mathbb{F}_4$-rank-width and maximum matching-width.

4.2 Cooling

Over time various cooling schedules for simulated annealing have been proposed, such as [52, 1, 37]

- **Linear** $T_i = T_{i-1} - \beta$, with β a constant,
- **Exponential** $T_i = \alpha \cdot T_{i-1}$, with $\alpha \in [0, 1]$ a constant, and
- **Logarithmic** $T_i = T_0 / \log(i + 10)$.

In these schedules T_i denotes the temperature after cooling i times, and T_0 denotes the initial temperature. More complex schedules are also used sometimes. Theoretically the logarithmic cooling schedule can be shown to asymptotically converge to the global optimum [52, 1, 37], but as this schedule converges very slowly it is usually not practically applicable. In practice the exponential schedule is the most widely used, and therefore also the one that was chosen to be used for rank-width. With the exponential schedule the cooling rate decreases over time. For α a value of 0.95 was chosen, which appeared to work well.

4.2.1 Adaptive cooling

Additionally, various variants of adaptive cooling exist. This technique adjusts the cooling based on the progress of the search process to improve the ability to escape from local minima. The variant that was tried for rank-width is described in [1] and increases the temperature based on the difference between the score of the best solution found so far and the score of the current solution. The idea behind this is to allow the algorithm to make some larger changes to the solution when it has diverged a lot from the best found so far. Let T_i' be the temperature after applying adaptive cooling. It holds that $T_i \leq T_i' \leq 2 \cdot T_i$ [1].

$$T_i' = T_i \cdot \left(1 + \frac{score - bestScore}{score}\right)$$

4.2.1.1 Performance

An experiment was performed to see if adaptive cooling improves the performance of the algorithm. The experiment was performed with a T_0 (initial temperature) of 25 and a Q (iterations before cooling) of 25600, without the thresholding heuristic (Section 5.2), but with caching (Section 5.5), the square sum score function (Section 5.1) and all operators (Chapter 6). The same initial solution was used each time. A laptop was used with an Intel® Core™ i7-6700HQ processor with a base frequency of 2.60 GHz and 8 GB ram. For each configuration, the program was run 10 times for 60 seconds. Table 4.1 shows for each configuration the average achieved width.

From the results it appears that using adaptive cooling has little impact on the performance of the algorithm. For some graphs adaptive seemed to perform a bit better and for others it made the results slightly worse, but in both cases the differences were rather small. As such on average there does not seem to be much of an advantage of using adaptive cooling. This can also be seen in Figure 4.1 which shows for each graph a comparison between the best obtained result without (blue) and with adaptive cooling (red).

			average width	
graph	*vertices*	*edges*	*normal*	*adaptive*
1kw4	67	672	19.3	**19.1**
celar08	458	1655	14.5	**13.5**
fpsol2.i.1	496	11654	**8.8**	9.7
mulsol.i.1	197	3925	**4.8**	5.3
myciel7	191	2360	42.7	**41.7**

Table 4.1 *Results of the adaptive cooling experiment showing for each configuration the average achieved width.*

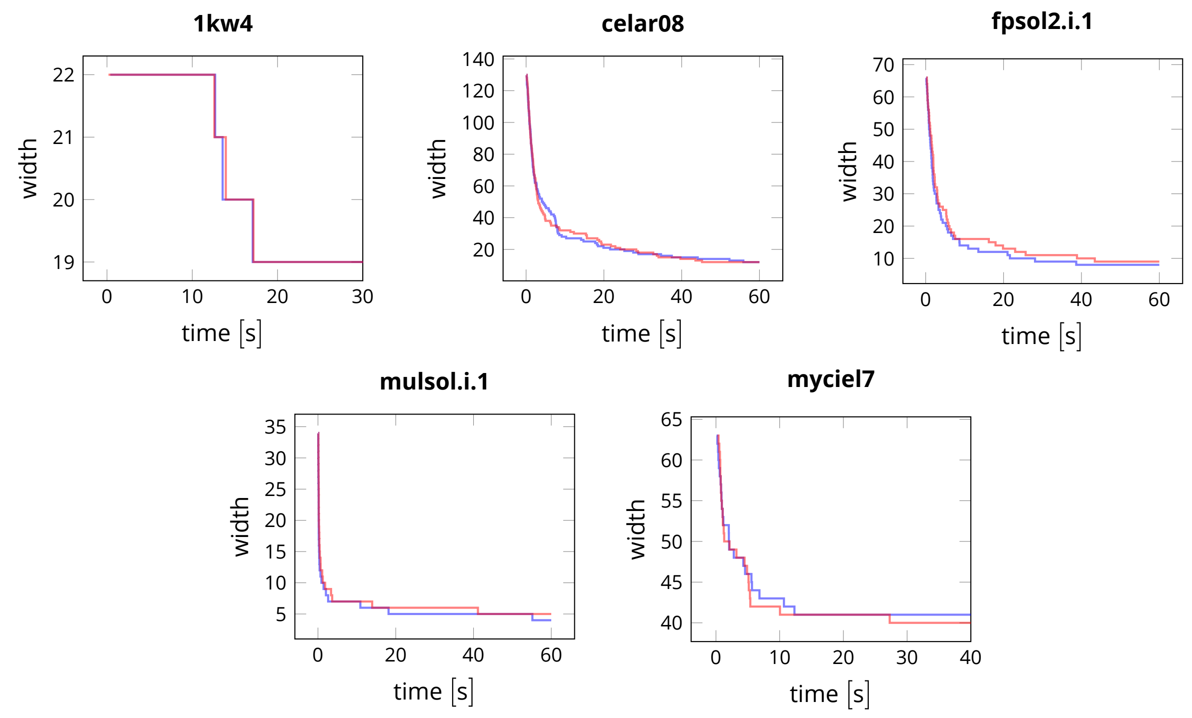


Figure 4.1 *Width improvement over time in the adaptive cooling experiment. For each of the graphs the best result without (blue) and with (red) adaptive cooling is shown.*

Chapter 5

Score

In this chapter we will look at what kind of score (objective) function performs well, which techniques are used to compute the score and how it can be done in an efficient way. This involves computing the partition of each edge and the corresponding width and combining that into a score value.

5.1 Score function

As the score function plays an important role in the performance of the algorithm an attempt was made to try different functions to see what works best. A major issue in the comparison between the different functions is that when the score values change, the simulated annealing temperature should also be changed, which makes it difficult to do an objective comparison.

The easiest observation that could be done is that using just the decomposition width as score gives very poor results, no matter the choice of temperature. The reason for this is that it causes many different decompositions to have the same score. This is also mentioned by Overwijk et al. [49] for their local search algorithm for branch-width. In practice there are usually various smaller changes that need to be made before the width can actually be decreased, but since such small changes will not change the score the algorithm will blindly accept any small change without knowing anything about its quality. In general we could say that decreasing the width of any edge is likely a good improvement, especially when it is an edge with a high width. Much better results can be obtained when this heuristic is incorporated into the score function by including edge widths.

5.1.1 Comparison

To see if penalizing high width edges more helps in the performance of the algorithm, an experiment was done that compares two score functions. Let $G = (V_G, E_G)$ be an undirected n-vertex graph and let the pair (T, L) be a rank-decomposition with $T = (V_T, E_T)$ a subcubic tree and L a bijection from V_G to the leaves of T. Furthermore let c_e be the cut-rank that corresponds to edge $e \in E_T$ of the decomposition tree T and let $w = \max_{e \in E_T} c_e$ be the width of the decomposition. The following two score functions were compared:

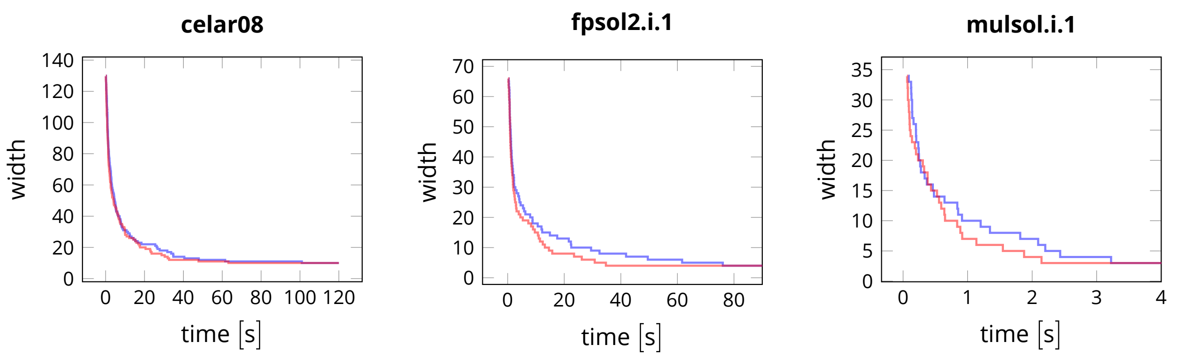
- Linear sum $\sum_{e \in E_T} c_e + w \cdot n$,
- Square sum $\sum_{e \in E_T} {c_e}^2 + w^2 \cdot n$.

As mentioned in the previous section, when adjusting the score values the initial temperature (T_0) also needs to be changed since together they determine the amount of randomness. For this reason different initial temperatures were also evaluated. The experiment was performed with a Q (iterations before cooling) of 25600, without adaptive cooling (Section 4.2) and without the thresholding heuristic (Section 5.2), but with caching (Section 5.5) and all operators (Chapter 6). The same initial solution was used each time. A laptop was used with an Intel® Core™ i7-6700HQ processor with a base frequency of 2.60 GHz and 8 GB ram. For each configuration, the program was run 5 times for 120 seconds. Note that at this point, especially with higher initial temperatures, the algorithm has not always stopped finding improvements yet. From these 5 the best result (smallest width, or if equal best time) is shown in Table 5.1. Because of the randomness in the search process the results are only an indication of a possible outcome.

	best width and corresponding time											
	celar08				**fpsol2.i.1**				**mulsol.i.1**			
	458 vertices, 1655 edges				496 vertices, 11654 edges				197 vertices, 3925 edges			
T_0	linear		square		linear		square		linear		square	
0.25	11	74.8 s	10	77.7 s	4	100.2 s	4	54.2 s	3	2.8 s	3	2.4 s
0.50	10	101.0 s	**10**	**63.2 s**	4	77.8 s	4	40.3 s	3	3.2 s	3	2.2 s
1.00	11	63.6 s	11	30.3 s	4	76.0 s	**4**	**34.7 s**	3	3.8 s	**3**	**2.1 s**
7.00	29	117.9 s	10	104.4 s	18	92.4 s	4	75.1 s	4	91.4 s	3	15.9 s
25.00	35	106.4 s	11	88.8 s	20	119.4 s	6	109.3 s	8	112.5 s	3	98.0 s
100.00	42	115.6 s	19	87.3 s	25	105.3 s	13	114.2 s	10	115.9 s	6	86.7 s
250.00	72	118.7 s	27	110.0 s	39	113.1 s	17	80.4 s	15	116.8 s	7	99.0 s

Table 5.1 *Results of the score function experiment showing for each configuration the best achieved width and the time it took respectively. T_0 is the initial temperature.*

The results seem to indicate that the square sum function has an advantage over the linear sum function, even when tuning the initial temperature. It also seems that setting the initial temperature too low makes the algorithm take more time before the best solution is found, possibly because it becomes harder to escape from local minima, but setting it too high will make the algorithm take a lot more time to find a good solution. From



(a) *linear 0.5 (blue), square 0.5 (red)* **(b)** *linear 1.0 (blue), square 1.0 (red)* **(c)** *linear 0.5 (blue), square 1.0 (red)*

Figure 5.1 *Width improvement over time for the best linear result (blue) and the best square result (red) for each of the three graphs in the score function experiment.*

the graphs in Figure 5.1 it appears that using the square sum function makes the algorithm converge quicker to a lower width than when using the linear sum function.

5.2 Thresholding heuristic

An interesting question, especially for larger graphs, is if we can manage to compute the score without requiring the cut-rank of every edge. We can make the following observations:

1. When improvements are made to the current decomposition, the width will usually not decrease by more than a few units at a time.
2. The most important improvements of a decomposition involve the edges with a high width, as this can eventually lead to a decrease in the width of the decomposition.

3. An upper bound for the cut-rank is the upper bound of the rank of the adjacency matrix: the minimum over the number of rows and the number of columns.

Based on this we can create a heuristic. We choose a threshold value t, slightly below the width of the old decomposition. Whenever the upper bound for the width of an edge is at most t, this upper bound will be used instead of the actual width. As long as the computed width is lower than the old width, or if the computed width is equal to t, the computation will be repeated with a lower value for t such that eventually the correct width is found and the score will have been computed with the threshold value corresponding to that width (for fair comparison with further changes and partial score updates). Because of point 1, the probability of having to run the computation multiple times is low, and with the cache as described in Section 5.5 previously computed cut-rank values can be reused. In the algorithm a threshold delta Δ is used, such that $t = \max(oldWidth - \Delta, 1)$. Note that even when the heuristic is technically disabled, $t = 1$ is still used because rank-width of 0 can be checked before starting the algorithm, and as such it is assumed that the computed width will be at least 1. This prevents having to compute the width of all edges that connect to a leaf, as we assume those edges will have a width of 1.

The precision of the heuristic score (compared to without the heuristic) depends on the value of t and the current width (or in the algorithm on the value of Δ). The heuristic score will always be an upper bound for the actual score. A disadvantage of this heuristic score is that for a part of the edges a change in width (below the threshold) will not be reflected in the score which might cause more undesired changes.

Note that point 3 still holds for maximum matching-width, as the upper bound of the size of the maximum matching is the size of the smallest partition side.

5.2.1 Performance

An experiment was performed to see if the heuristic is effective. A number of graphs with both lower and higher amounts of vertices were used, in combination with four values for Δ and also without the heuristic. The experiment was performed with a T_0 (initial temperature) of 1 and a Q (iterations before cooling) of 25600, without adaptive cooling (Section 4.2), but with caching (Section 5.5), the square sum score function (Section 5.1) and all operators (Chapter 6). The same initial solution was used each time. A laptop was used with an Intel® Core™ i7-6700HQ processor with a base frequency of 2.60 GHz and 8 GB ram. For each configuration, the program was run 10 times for 60 seconds. The average achieved width of the results is shown in Table 5.2. Note that because of the randomness in the search process this is only an indication of a possible outcome.

			average width				
graph	***vertices***	***edges***	$\Delta = 2$	$\Delta = 5$	$\Delta = 10$	$\Delta = 20$	*without*
celar08	458	1655	15.3	15.8	13.3	**11.5**	**11.5**
d1655.tsp	1655	4890	194.4	194.1	**193.3**	206.1	379.6
fpsol2.i.1	496	11654	15.8	16.0	11.0	4.4	**4.2**
myciel7	191	2360	**43.6**	46.0	45.7	44.2	44.4
queen8_12	96	2736	**23.3**	24.1	24.1	24.8	23.5
u1817.tsp	1817	5386	**223.0**	309.1	339.7	328.1	489.1

Table 5.2 *Results of the thresholding heuristic experiment showing for each configuration the average achieved width.*

From the results it appears that the heuristic is mainly effective for graphs with a larger number of vertices or higher rank-width. For smaller graphs the heuristic often causes a worsening of the results. Since for large graphs computing cut-rank takes much more time than for small graphs the amount of time saved by the heuristic is also larger. This allows the algorithm to get further than it would have gotten without the heuristic. The heuristic is also more effective for graphs with a high rank-width because the higher the width, the larger the amount of edges that do not have to be computed. For smaller graphs where the time advantage is smaller and the rank-width also often lower, the lower precision of the score values most likely makes the algorithm take worse decisions than it would have without the heuristic, leading to degraded results. Figure 5.2 shows a comparison of the width improvement over time with (red) and without the heuristic (blue).

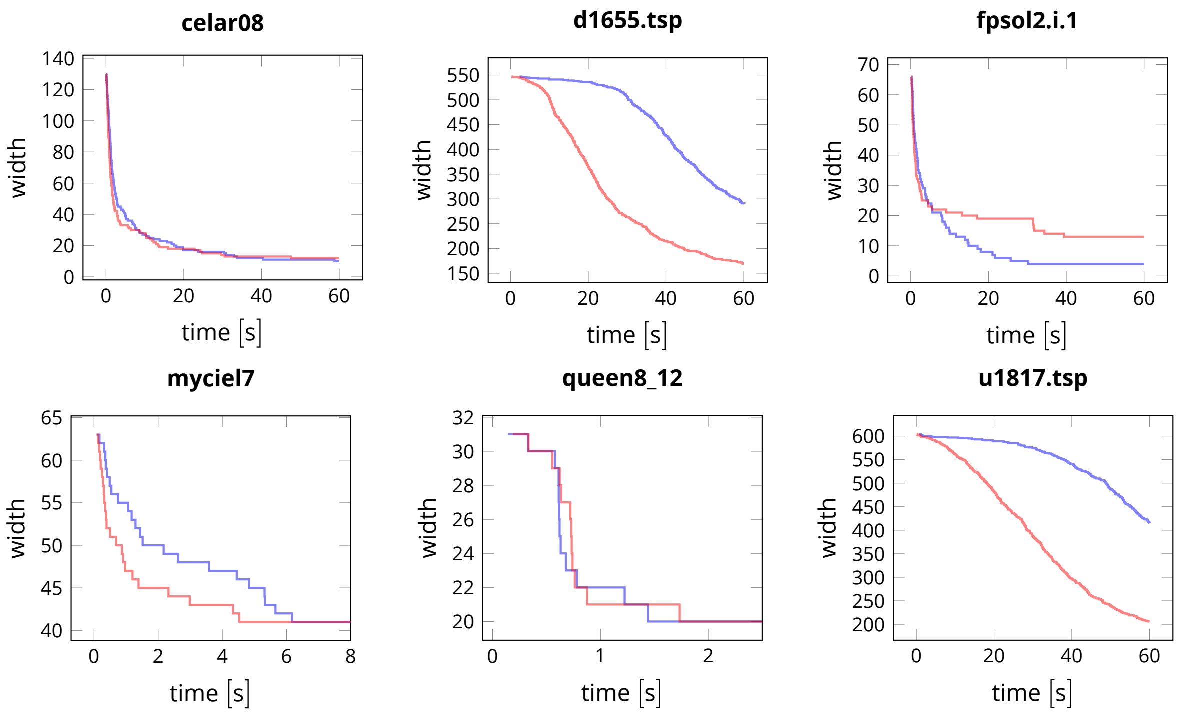


Figure 5.2 *Width improvement over time for the thresholding heuristic experiment. The graphs show the best achieved result without the heuristic (blue) and with* $\Delta = 2$ *(red).*

5.3 Computing edge partitions

In order to compute the width of a branch-decomposition the leaf partition that belongs to each edge of the decomposition needs to be computed. For this a depth-first search based algorithm can be used (see Algorithm 5.1). The depth-first order of traversal makes it possible to represent the partitions as a continuous interval in an array that stores the leaves in the order they were visited. An arbitrary leaf is used as the root of the tree, and the search is started from there. Each node is visited twice. Once before and once after exploring its children, to obtain the start and end of the interval respectively. This is done using variable $curId$, which is initialized with 0 and is used to keep track of the encountered leaves. An example of the algorithm is shown in Figure 5.3.

When a node is visited for the first time ($childrenDone = false$) the start of its partition interval is set to $curId$. Additionally if the node is a leaf, it is added to the $leaves$ array and $curId$ is increased. Then the node itself is first pushed onto the stack again, but with $childrenDone$ set to $true$ and subsequently the children of the current node are pushed.

The second time a node is visited ($childrenDone = true$) the end of its partition interval is set to $curId - 1$. Then for each of the edges from the node to a child the partition can be reported. The partition interval of a node always corresponds to the edge that connects to its parent.

For a decomposition tree with n nodes computing the partition intervals takes $O(n)$ time, since each node is visited exactly twice. If for reporting the partitions all elements of the partition need to be iterated, the complexity grows to $O(n^2)$ time. The algorithm takes $O(n)$ space to store the stack and the $leaves$ array.

Because the ids assigned to the leaves change with changes to the decomposition, in the practical implementation of the algorithm they are remapped to stable ids and then stored as bits in a bit vector. Incremental

Algorithm 5.1 *Computing edge partitions using depth-first search*

```
input  : $T$ = decomposition tree
output: All edge partitions of the decomposition
$stack \leftarrow$ makeStack();          // Empty stack for (node, parent, childrenDone) tuples
push($stack$, (leaves($T$)[0], $null$, $false$));          // Push an arbitrary leaf as root
$curId \leftarrow 0$;
while not empty($stack$) do
    ($node, parent, childrenDone$) $\leftarrow$ pop($stack$);
    if $childrenDone$ then
        $node.endId \leftarrow curId - 1$;
        for $neigh \in$ neighbors($node$) do     // Report partitions of node to child edges
            if $neigh = parent$ then
                continue
            report partition $leaves[neigh.startId \dots neigh.endId]$;
    else
        $node.startId \leftarrow curId$;
        if isLeaf($node$) then
            $leaves[curId] \leftarrow node$;
            $curId \leftarrow curId + 1$;
        push($stack$, ($node, parent, true$));       // Revisit node after children are done
        for $neigh \in$ neighbors($node$) do                              // Push children
            if $neigh = parent$ then
                continue
            push($stack$, ($neigh, node, false$));
```

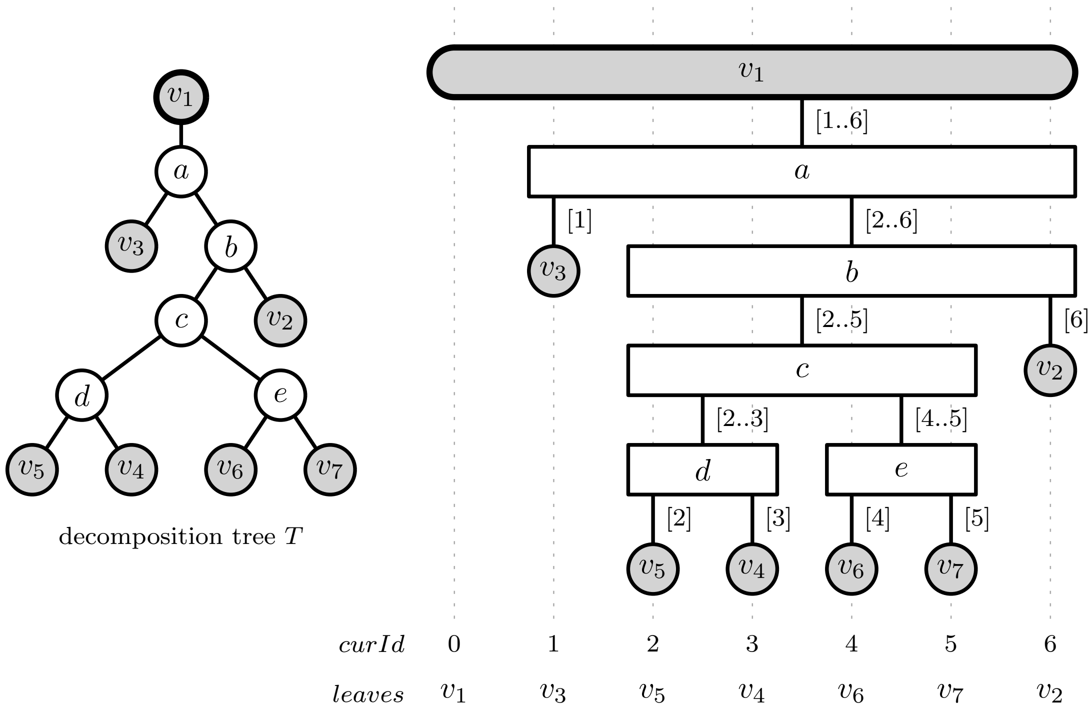


Figure 5.3 *Example of computing edge partitions using Algorithm 5.1. The root of the tree is chosen to be v_1.*

changes to the vector are made whenever possible to save time. When for example the current vector represents the partition $[5\ldots8]$ and the new partition to report is $[3\ldots8]$ only the bits that correspond to 3 and 4 have to be additionally set. Note that because of the remapping the bits belonging to 3 and 4 do not have to be adjacent.

5.4 Matrix rank

After obtaining the partition belonging to an edge the cut-rank needs to be calculated, which means computing the rank of an adjacency matrix. As described in Section 2.3, there exist various algorithms to do this. Because the size of the matrix depends on the partition, there should either be a good way to decide which algorithm to use in which case, or an algorithm should be used that on average performs well in most cases. Since the matrix rank computations are completely independent from each other, the cut-rank of multiple edges can be computed in parallel.

5.4.1 Gaussian elimination

As described in Section 2.3 Gaussian elimination is the classic way of computing the rank of a matrix. Algorithm 5.2 shows the basic algorithm for $\mathbb{F}_2$ matrices and in Algorithm 5.3 the few changes needed for $\mathbb{F}_4$ matrices are shown. This difference is because $\mathbb{F}_2$ only has two elements and therefore multiplication of pivot rows is meaningless. The rest of this section describes the techniques used for the practical implementation of the basic algorithm.

5.4.1.1 $\mathbb{F}_2$ bit vector

Since the only operation performed for Gaussian elimination in $\mathbb{F}_2$ is addition, which is identical to xor operations on the numbers 0 and 1, a natural and efficient way to represent the rows of the matrix is using bit vectors. Modern processors can perform xor operations on SIMD vectors of up to 256 or even 512 bits, which can make a large difference in performance. Since we know that all elements to the left of the pivot are zero, this can be used to speed up the adding process by skipping (almost) all those zeros.

Because operations on columns (such as searching for a 1) are much slower than operations on rows (adding them), the adjacency matrix belonging to a partition is constructed such that the number of rows is minimized. That way vectorization can be used as much as possible. Instead of completely constructing new bit vectors from scratch that omit columns that are not on that side of the partition, the algorithm instead just ignores those columns so that the bit vectors from the adjacency matrix of the entire graph can simply be used. As a result of this the vectors are at most twice as big as they would have been otherwise (when the partition represents half of the graph on each side), however because of the SIMD math and the fact that the partition is usually not balanced this does not make a big difference in practice compared to the time saved by not constructing new bit vectors.

5.4.1.2 $\mathbb{F}_4$ bit vector

Bit vectors can also be used for Gaussian elimination in $\mathbb{F}_4$ by applying some tricks. Two bits are used to represent each element in the vector (see Table 5.3).

0	$\rightarrow$	`00`
1	$\rightarrow$	`01`
α	$\rightarrow$	`10`
α^2	$\rightarrow$	`11`

Table 5.3 *Representation of $\mathbb{F}_4$ elements using bits. The left bit is the most significant bit (MSB).*

Algorithm 5.2 *Computing the rank of a $\mathbb{F}_2$ matrix using Gaussian elimination*

```
input  : $M$ = $\mathbb{F}_2$ matrix of size $m \times n$
output: $rank(M)$
$curRow \leftarrow 0$;
$curCol \leftarrow 0$;
while $curRow < m \wedge curCol < n$ do
    $success \leftarrow false$;
    for $i = curRow \dots m-1$ do          // Find a non-zero in the current column
        if $M[i][curCol] \neq 0$ then
            swap($M[i], M[curRow]$);
            $success \leftarrow true$;
            break
    if not $success$ then                   // All zeros column
        $curCol \leftarrow curCol + 1$;
        continue
    for $i = curRow + 1 \dots m-1$ do       // Clear the rest of the current column
        if $M[i][curCol] \neq 0$ then
            $M[i] \leftarrow M[i] + M[curRow]$;
    $curRow \leftarrow curRow + 1$;
    $curCol \leftarrow curCol + 1$;
return $curRow$;                            // curRow is now equal to the rank of M
```

Algorithm 5.3 *Changes needed to Algorithm 5.2 for $\mathbb{F}_4$ matrices (replaces lines 13-15)*

```
$pivot \leftarrow M[curRow][curCol]$;
for $i = curRow + 1 \dots m-1$ do       // Clear the rest of the current column
    if $M[i][curCol] \neq 0$ then
        $factor \leftarrow M[i][curCol]/pivot$;
        // Multiplying the pivot row by factor turns pivot into M[i][curCol]
        // Adding to row i then yields a 0, since x + x = 0 in F4
        $M[i] \leftarrow M[i] + M[curRow] * factor$;
```

Because there are more than two elements in $\mathbb{F}_4$ addition alone no longer suffices and multiplication and division operations need to be used as well (see Algorithm 5.3). As can be seen in Table 5.4, $\mathbb{F}_4$ addition is still an xor operation when bit notation is used. Note that subtraction is identical to addition.

$+$	0	1	α	α^2
0	0	1	α	α^2
1	1	0	α^2	α
α	α	α^2	0	1
α^2	α^2	α	1	0

$+$	00	01	10	11
00	00	01	10	11
01	01	00	11	10
10	10	11	00	01
11	11	10	01	00

Table 5.4 *Addition in $\mathbb{F}_4$ shown with mathematical and bit notation. With bits the operation is identical to xor.*

Multiplication

Multiplication in $\mathbb{F}_4$ (see Table 5.5) is not as trivial to perform using bit vectors. When multiplying each element

$\times$	0	1	α	α^2
0	0	0	0	0
1	0	1	α	α^2
α	0	α	α^2	1
α^2	0	α^2	1	α

$\times$	00	01	10	11
00	00	00	00	00
01	00	01	10	11
10	00	10	11	01
11	00	11	01	10

Table 5.5 *Multiplication in $\mathbb{F}_4$ shown with mathematical and bit notation.*

of a vector by the same constant a technique based on the Russian peasant multiplication method can be used, as was implemented in the moepgf finite field arithmetic library[1] for example. This method is based on the fact that it holds that

$$a \cdot b = 2a \cdot (b/2).$$

When using integer numbers and bit shifts (denoted by $\ll$ and $\gg$ for logical left and right shift respectively), it can be written as

$$a \cdot b = \begin{cases} (a \ll 1) \cdot (b \gg 1) & \text{if } b \text{ is even,} \\ (a \ll 1) \cdot (b \gg 1) + a & \text{if } b \text{ is odd.} \end{cases}$$

By applying this equation recursively, the multiplication can be entirely written using bit shifts and addition:

$$f(a,b) = \begin{cases} 0 & \text{if } a = 0 \vee b = 0, \\ f(a \ll 1, b \gg 1) & \text{if } b \text{ is even,} \\ f(a \ll 1, b \gg 1) + a & \text{if } b \text{ is odd.} \end{cases}$$

Since we are working in $\mathbb{F}_4$, the numbers a and b we use can have at most 2 bits. As such the recursion can be fully unrolled. To get rid of the odd/even case distinction we can use bit operations and multiply by the appropriate bit. For the $\mathbb{F}_4$ formula $y = c \cdot x$ we get

$$c_2 = \begin{cases} (c \ll 1) \text{ xor } 7 & \text{if } c \geq 2, \\ c \ll 1 & otherwise, \end{cases} \qquad \text{(polynomial reduction)}$$

$$y = ((x \,\&\, 1) \cdot c) \text{ xor } ((x \gg 1) \cdot c_2).$$

Note that the polynomial of the field $\mathbb{F}_4$ is $X^2 + X + 1$, which is equivalent to the bit representation $111 = 7$. Since the cases for $c = 0$ and $c = 1$ are trivial (all zeros and no change, respectively), these can be handled separately and the case distinction for c_2 can simply be replaced by the case that $c \geq 2$. It should not be difficult to see that this equation can easily be adapted to work on numbers packed into a vector since none of

[1] https://github.com/moepinet/libmoepgf/blob/master/src/gf4.c

the operations can cause overflows as long as proper bit masks are applied. For 16 elements of $\mathbb{F}_4$ packed into a 32 bit x the equation becomes for example

$$y = ((x \mathbin{\&} \texttt{0x}55555555) \cdot c) \text{ xor } (((x \gg 1) \mathbin{\&} \texttt{0x}55555555) \cdot c_2).$$

In this equation 0x55555555 is the hexadecimal representation of the bit mask `01010101010101010101010101010101`.

Division

For Gaussian elimination in $\mathbb{F}_4$ division (see Table 5.6) is only performed on scalars and is therefore trivial to implement, for example using a lookup table.

$\div$	1	α	α^2
0	0	0	0
1	1	α^2	α
α	α	1	α^2
α^2	α^2	α	1

$\div$	01	10	11
00	00	00	00
01	01	11	10
10	10	01	11
11	11	10	01

Table 5.6 *Division in $\mathbb{F}_4$ shown with mathematical and bit notation. Left is divided by top. Division by 0 is undefined.*

5.4.1.3 Disjoint set

For some very sparse $\mathbb{F}_2$ matrices it might be advantageous to store the indices of the ones of every row as a disjoint set instead of a bit vector to prevent storing many zeros. For this specific use case using a sorted array as data structure suffices. The operations that need to be performed consist of getting the smallest element (if it exists) and computing the symmetric difference (similar to xor for elements of a set) between two arrays. The first operation costs $O(1)$ time since the array is sorted such that index 0 is always the smallest element. The second operation is performed like a merge in the merge sort algorithm, but with the special case that an element is skipped if it is in both arrays. This costs $O(m + n)$ time with m and n the length of the first and second array respectively. With this representation it turned out to be beneficial to not clear the rest of a column (lines 13-15 in Algorithm 5.2) immediately after finding a pivot, but instead storing which row belongs to each column and only clearing the leading elements (all elements smaller than $curCol$) of the row when it is actually going to be used (between line 5 and 6).

5.4.2 GF2 toolkit

As described in Section 2.3 GF2 toolkit[2] is a library for fast arithmetic with dense matrices in $\mathbb{F}_2$, and it also supports computing the rank of a matrix. The library supports SIMD, but only 128 bit vectors.

5.4.3 Comparison

An experiment was done to see if it would be profitable to use one algorithm or the other based on the type of matrix, for example by looking at its size or density. The three algorithms, Gaussian elimination with bit vectors, Gaussian elimination with disjoint sets and GF2 toolkit, were tested with adjacency matrices that correspond to random partitions of random graphs with all combinations of the following properties:

- 8 vertex counts: 50, 250, 400, 800, 1500, 4000, 10000, 30000,
- 5 average degrees: 1, 3, 5, 10, 20,
- 5 partition fractions: 0.05, 0.125, 0.25, 0.33, 0.5.

[2]`https://github.com/ebertolazzi/GF2toolkit`

Note that for example a partition fraction of 0.25 means that one side of the partition has a quarter of the vertices and the other side the rest. The benchmark was performed on a laptop with an Intel® Core™ i7-6700HQ processor with a base frequency of 2.60 GHz and 8 GB ram. BenchmarkDotNet[3] was used to perform the benchmark, with settings `SimpleJob(RunStrategy.Throughput, launchCount: 1, warmupCount: 5, targetCount: 5)`.

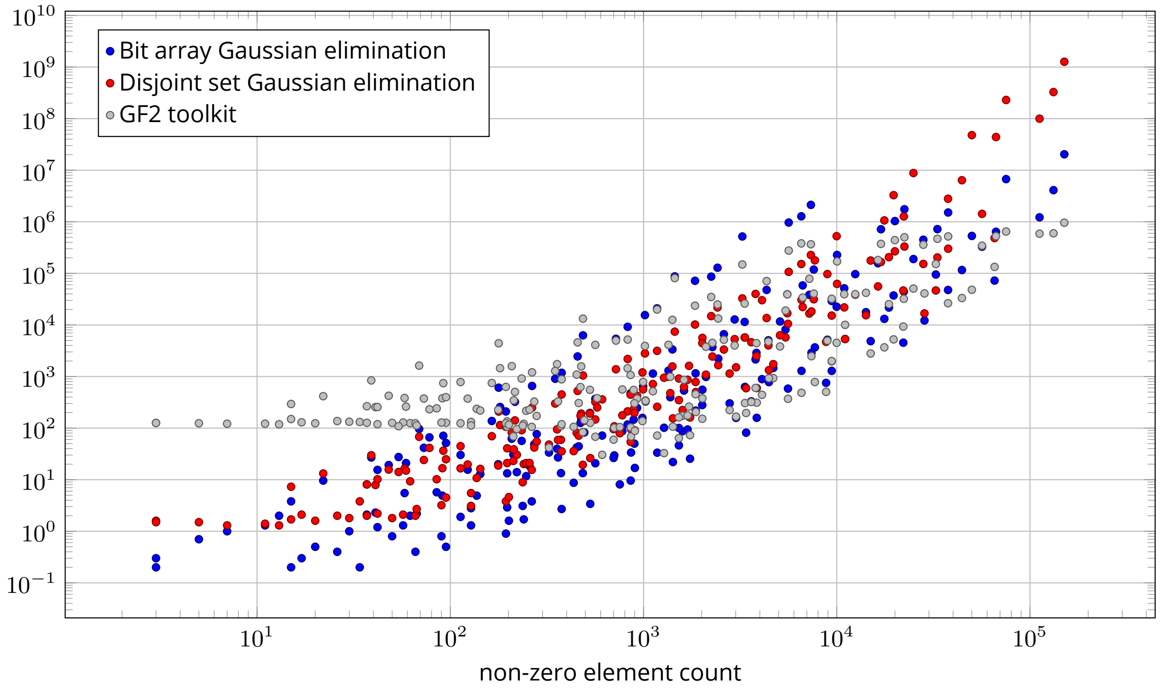


Figure 5.4 *Logarithmic scatter plot showing for the three algorithms the average computation time for the tested matrices, sorted on non-zero element count.*

In Figure 5.4 a logarithmic scatter plot is shown comparing average computation time for all three algorithms with the number of non-zero elements in the matrix. There is a clear correlation visible between non-zero element count and computation time. For lower amounts of non-zero elements GF2 toolkit performs considerably worse than the two Gaussian elimination algorithms, but for very high numbers it appears to perform better. This is most likely because GF2 toolkit is targeted at very dense matrices. When it comes to the two Gaussian elimination algorithms, the bit array variant shows a larger variation in computation time for similar non-zero element counts than the disjoint set variant, but the latter is slower on average, especially for high non-zero element counts. Overall there is not a very clear winner for specific ranges of non-zero element counts, and as such we can conclude that the non-zero element count (at least on its own) is not very usable to decide which algorithm to use.

Another comparison that could be made was with the size of the smallest side of the partition. This is shown in the logarithmic scatter plot in Figure 5.5. Again a clear correlation can be seen. The results are largely similar to the comparison with non-zero element count. GF2 toolkit has a high start up cost, but has a better performance for very large sizes. On the other hand, disjoint set Gaussian elimination performs quite poor on average as the size grows (although for some large, most likely very sparse, graphs it is also the fastest). Bit array Gaussian elimination appears to perform quite good over the whole range. Overall the size of the smallest partition side cannot predict very precisely which algorithm should be used, although GF2 toolkit could be a good choice for the very large sizes.

If one algorithm has to be chosen to be used overall, the bit array Gaussian elimination algorithm seems to be a good choice. This is also what has been used in the program.

[3] `https://benchmarkdotnet.org/`

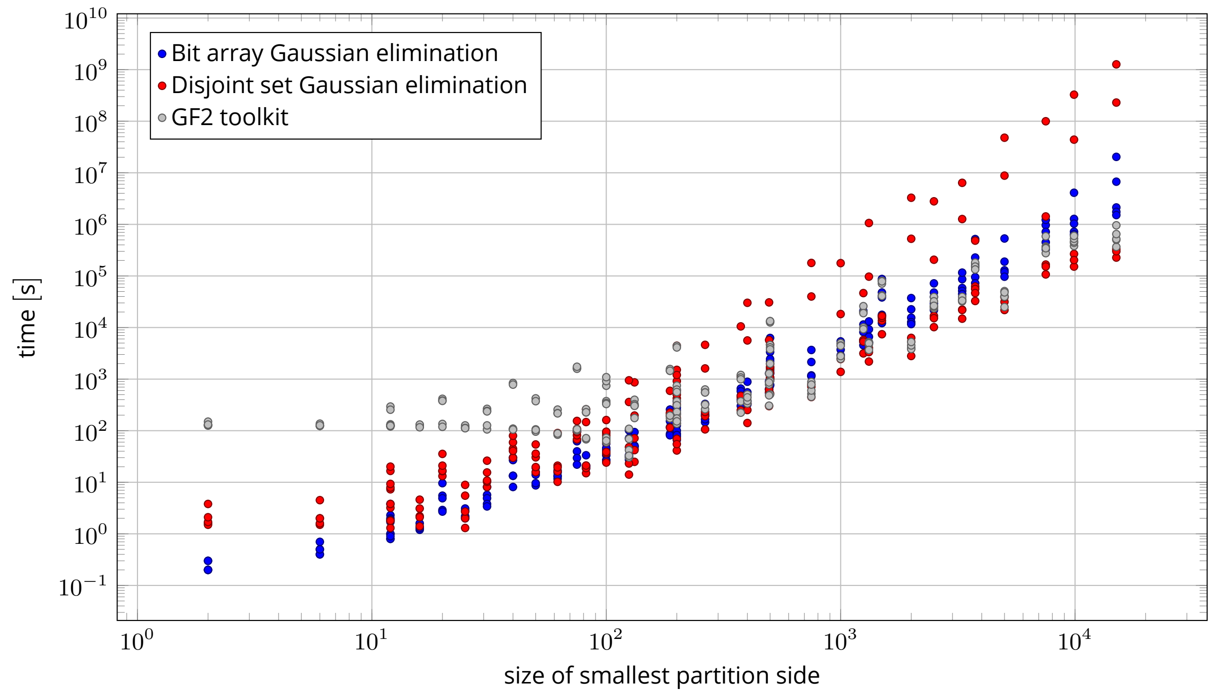


Figure 5.5 *Logarithmic scatter plot showing for the three algorithms the average computation time for the tested matrices, sorted on the size of the smallest partition side.*

5.5 Cut-rank cache

Since computing the cut-rank of a partition is quite an expensive operation, the number of such computations should preferably be minimized. As the width of an edge depends solely on the partition, a cache can be used that stores the cut-rank of the most recently encountered partitions (least recently used replacement policy). In this way recently computed values can be reused when only small changes are made to the decomposition. Because the cache uses partitions (represented by bit vectors) as keys, a cache lookup takes $O(n)$ time, with n the number of vertices in the graph. This is because a hash of the partition has to be calculated in order to perform the lookup, and when a matching hash is found a comparison is done to ensure the right key is actually found, and not a hash collision. This is still much faster than the time it would take to compute the cut-rank.

5.5.1 Performance

An experiment were performed to evaluate the performance of the cache for graphs of differing sizes and a number of cache sizes. The experiment was performed with a T_0 (initial temperature) of 1 and a Q (iterations before cooling) of 25600, without adaptive cooling (Section 4.2), without the thresholding heuristic (Section 5.2), with the square sum score function (Section 5.1) and all operators (Chapter 6). A laptop was used with an Intel® Core™ i7-6700HQ processor with a base frequency of 2.60 GHz and 8 GB ram. For each configuration, the program was run once for 60 seconds. The results are shown in Figure 5.6, 5.7 and 5.8 for cache sizes of 512, 16384 and 1048576 respectively. They show, in steps of 5%, in how many iterations of the algorithm a cache hit percentage in that range was achieved. A higher percentage means more hits, and thus a faster score computation.

From the results it becomes clear that high hit rates can be achieved, however to get good hit rates the cache should not be too small. Most likely at least as large as the number of vertices in the graph. Higher cache sizes do lead to better hit rates, but the advantage quickly becomes smaller.

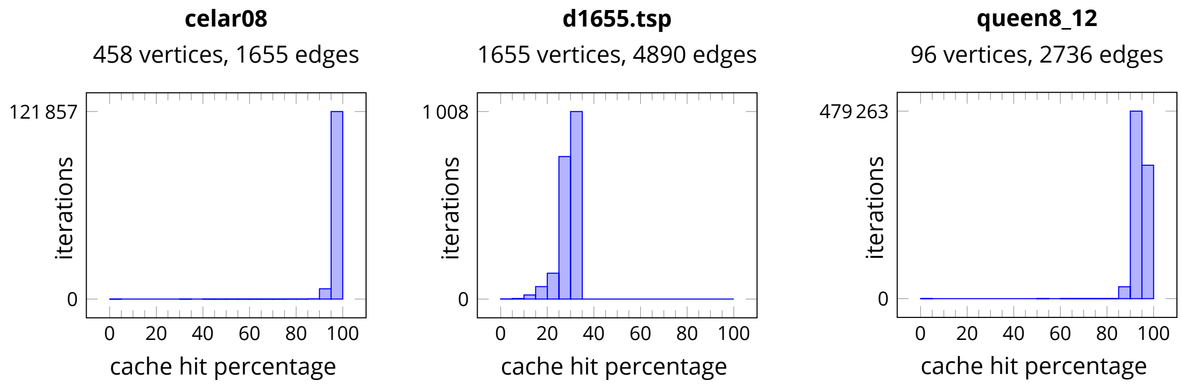


Figure 5.6 *Histograms showing for a number of graphs the cache hit rate in steps of 5% for a cache size of 512.*

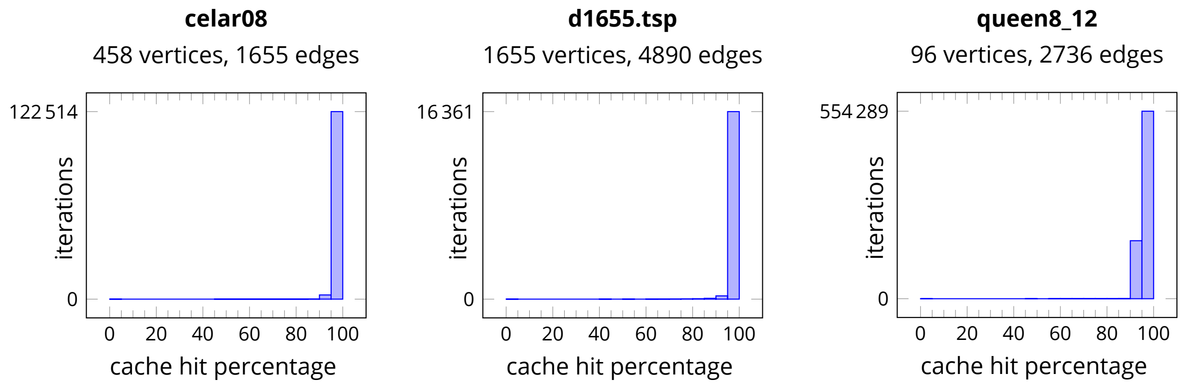


Figure 5.7 *Histograms showing for a number of graphs the cache hit rate in steps of 5% for a cache size of 16384.*

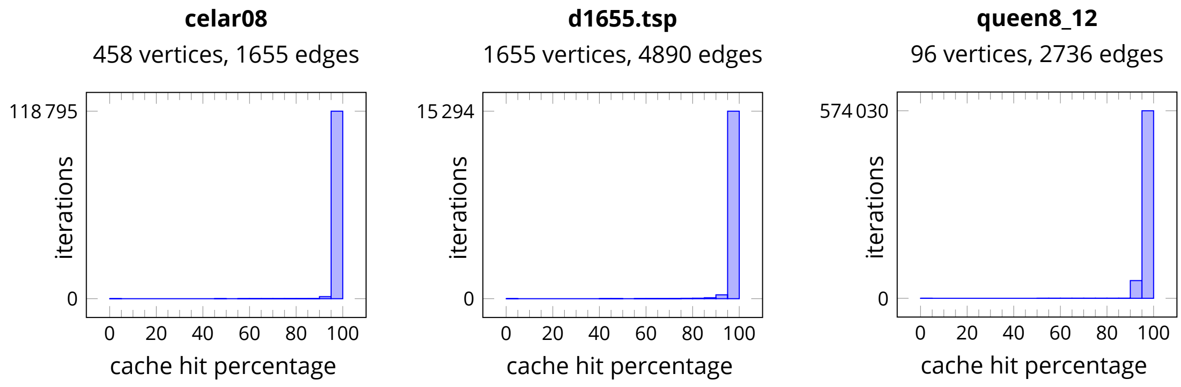


Figure 5.8 *Histograms showing for a number of graphs the cache hit rate in steps of 5% for a cache size of 1048576.*

Chapter 6

Operators

In this chapter the various operators used in the algorithm will be discussed and their contribution to the performance of the algorithm will be discussed.

6.1 Overview

In total three different operators were implemented: move subtree, leaf swap and local swap. The details of each of the operators will be explained in the following subsections. The operator probabilities were experimentally determined and were set to 0.5 for move subtree, 0.1 for leaf swap and 0.4 for local swap. Although partially updating the score is possible for all three of the operators, it was only implemented for the local swap operator, because for the move subtree and leaf swap operators the partition and width that belong to each decomposition edge would have to be stored and maintained which makes the program more complex and adds overhead. As such it was questionable if the effort of implementing it would really result in a practical improvement.

6.1.1 Move subtree

The move subtree operator is based on the operator used by Overwijk et al. [49] in their local search algorithm for branch-width. In Figure 6.1 an example of applying the operator is shown. Two random nodes a and b are chosen such that they are not adjacent and have no common neighbor. The path P between a and b is then computed to find a' and b' which are the neighbors of respectively a and b on P. Finally the two neighbors of a' other than a are connected together and the edge between b and b' is split in two parts that get connected to a'. The edges aa' and bb' (before applying the operator) correspond respectively to the edges s and t in the paper of Overwijk et al. [49]. They show that the operator is complete, as for any two branch-decompositions there exists a sequence of move operations that will transform the one into the other. Since a branch-decomposition is just a subcubic tree the proof also applies to rank-decompositions.

6.1.2 Leaf swap

The leaf swap operator (see Figure 6.2) takes two unique random leaves of the decomposition and swaps them. This can then influence the width of all edges on the path between the two leaves, since those have the two leaves on opposite sides of their partition.

6.1.3 Local swap

The local swap operator starts by taking one random internal (non leaf) node of the decomposition which we will call c (center). Then two neighbors of c are randomly chosen, but such that at least one of them is an internal node. Of these a will be any node and b will be the internal node. Finally a neighbor of b (not equal to c) is also

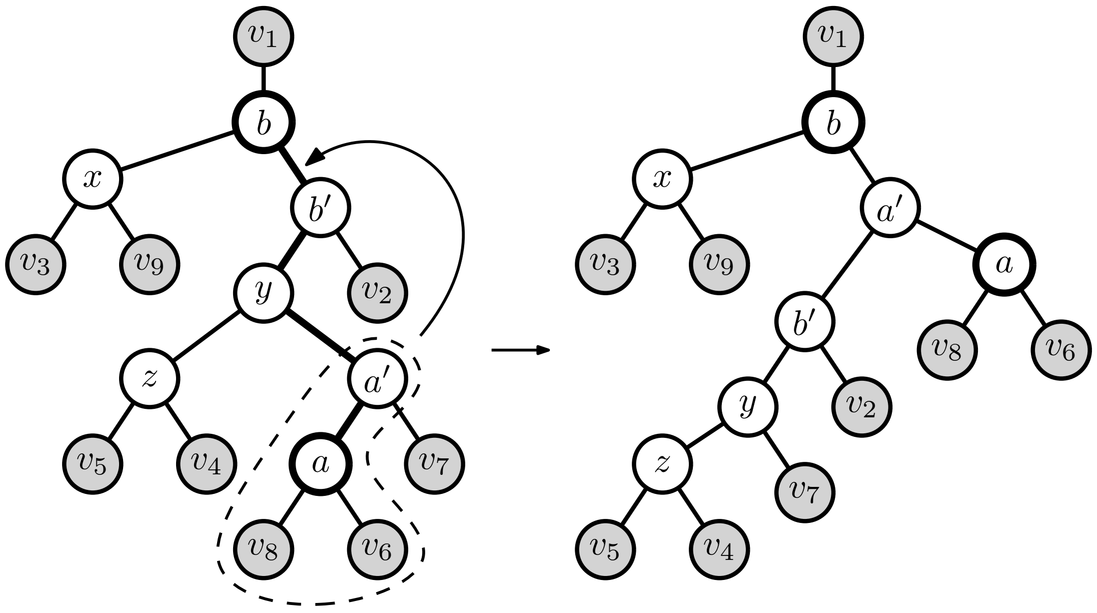


Figure 6.1 *Example of the move subtree operator applied to nodes a and b. The path P is indicated with thick edges.*

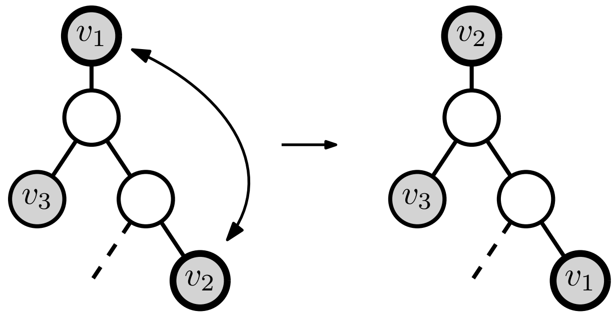


Figure 6.2 *Example of the leaf swap operator swapping v_1 and v_2.*

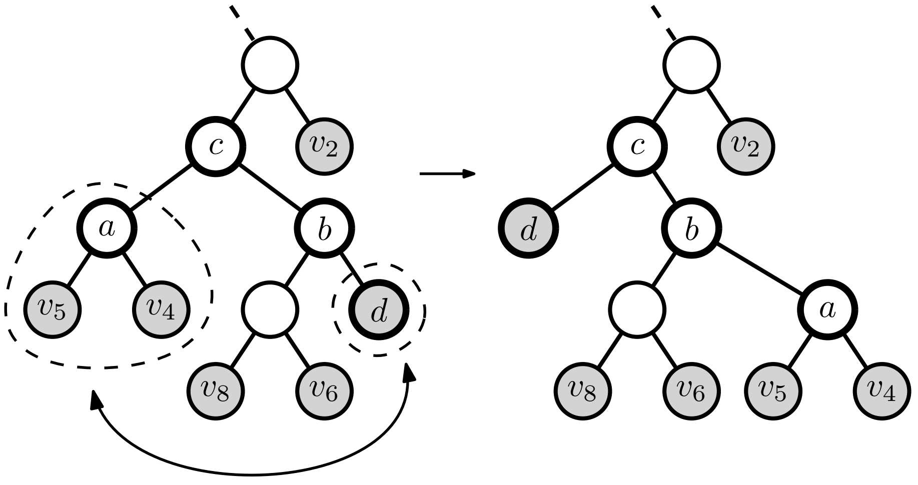


Figure 6.3 *Example of the local swap operator.*

randomly chosen and let it be d. Now the nodes a and d are swapped such that a becomes a neighbor of b and d becomes a neighbor of c. An example of this is shown in Figure 6.3. The advantage of this operator is that it makes only small changes to the decomposition. Only the width of the edge between b and c can change. This also makes this operator very suited for performing partial score updates.

6.2 Performance

Since the move subtree operator is proven to be complete, an interesting question is if the other two operators actually contribute something to the search process. For this an experiment was done with disabling the leaf swap and/or the local swap operator to see how it influences the results for a number of graphs. The operator probabilities were properly normalized to account for disabling operators. The experiment was performed with a Q (iterations before cooling) of 25600, without adaptive cooling (Section 4.2) and without the thresholding heuristic (Section 5.2), but with caching (Section 5.5) and with the square sum score function (Section 5.1). Both a T_0 (initial temperature) of 1 and 25 were tested. The same initial solution was used each time. A laptop was used with an Intel® Core™ i7-6700HQ processor with a base frequency of 2.60 GHz and 8 GB ram. For each configuration, the program was run 10 times for 60 seconds. The average achieved width of the results is shown in Table 6.1.

			average width							
			$T_0 = 1$				$T_0 = 25$			
graph	***vertices***	***edges***	*neither*	*local*	*leaf*	*both*	*neither*	*local*	*leaf*	*both*
BN_15	120	637	35.2	35.8	**30.0**	30.3	30.9	31.1	**28.8**	28.9
celar08	458	1655	13.9	12.0	15.3	**11.9**	15.7	**13.7**	16.8	14.1
d1655.tsp	1655	4890	423.8	379.3	457.5	**376.6**	438.5	**354.9**	423.8	375.9
fpsol2.i.1	496	11654	7.4	**4.4**	7.7	**4.4**	10.3	**8.5**	9.7	8.6
mulsol.i.1	197	3925	**3.0**	**3.0**	**3.0**	**3.0**	5.1	4.9	5.1	**4.8**
myciel7	191	2360	57.2	57.6	**44.1**	46.2	53.7	51.5	**41.6**	43.2

Table 6.1 *Results of the operator experiment showing for every configuration the average achieved width (neither = only move subtree, local = move subtree + local swap, leaf = move subtree + leaf swap and both = all three operators).* T_0 *is the initial temperature.*

From the results it appears that not using additional operators at all leads to the worst results. Although for some graphs the leaf swap operator seems to lead to a slight worsening of the score, for other graphs it causes a major improvement. The same holds for the local swap operator. On average both operators thus contribute to better results, but depending on the graph one or the other might be most effective. We conclude that in general it would be the best to enable both operators.

Chapter 7

Benchmarking

In this chapter the results of the final benchmarks of the algorithm will be discussed. The configuration for the benchmarks was chosen based on the results of the various experiments that were discussed in the previous chapters. To show that the algorithm can not only be used for rank-width but also for other branch-decomposition based width parameters, $\mathbb{F}_4$-rank-width [34, 33] and maximum matching-width [54] were additionally tested. Of these $\mathbb{F}_4$-rank-width is a width parameter for directed graphs, which is interesting as not much research seems to have been done for practically computing directed width parameters. This however also means that there are no results to compare to. A table with all benchmarking results can be found in Appendix B.

For the undirected parameters graphs from TreewidthLIB [53] [1] were used. Those graphs are often used in benchmarks for width parameters and as such upper bounds for some width parameters are known for a large part of them. The collection was also used by Beyß [6, 7] for benchmarking his rank-width approximation algorithm, which means we also have rank-width upper bounds to compare to. From this collection 218 small/medium (fewer than 1000 vertices) graphs were used (Table B.1) and 11 large graphs (Table B.2). Benchmarking was also done for 10 square grid graphs, which are proven [31] to have an exact rank-width of $n - 1$ for a grid of size $n \times n$.

For $\mathbb{F}_4$-rank-width 18 small/medium graphs from a collection of directed graphs on GitHub[2] were used (Table B.3). Unfortunately there were not really more sets of directed graphs of reasonable size available to be used for benchmarking. As there were no bounds available to compare our results to, the graphs were additionally converted from directed to undirected to approximate their rank-width and maximum matching-width. This makes it possible to see to some extent what kind of impact the directional information has on the width. Additionally an experiment was done with tournaments, which are complete graphs in which each edge has one of the two possible directions (not both).

In all benchmarking experiments the following settings were used: a Q (iterations before cooling) of 25600, a T_0 (initial temperature) of 5, no adaptive cooling (Section 4.2), a cache of size 16384 (Section 5.5), the square sum score function (Section 5.1) and all operators (Chapter 6). Only for the large graphs with at least 1000 vertices the thresholding heuristic (Section 5.2) was used, with a threshold delta $\Delta = 5$. Small and medium graphs (fewer than 1000 vertices) were run with a time limit of 3 minutes, large graphs with a time limit of 5 minutes. The benchmarks were performed on a Linux PC with an Intel® Core™ i5-9500 processor at 3 GHz and 16 GB of ram (different from the experiments in the previous chapters). Each graph was run 10 times, and the best achieved result was reported in Table B.1, B.2 and B.3.

[1] Note that the database is no longer available. A backup can be found here: `https://github.com/emnh/boolwidth-data`.

[2] `https://github.com/alidasdan/graph-benchmarks`

7.1 Rank-width

The results of the rank-width benchmarks can be found in Table B.1 and B.2. Overall it seems the algorithm performed quite good for the small/medium graphs, and poorer for the large graphs. The reason for this is likely the complexity of matrix rank computations and the increasing size of the search space for large graphs. In Figure 7.1 a comparison is made between the graph vertex count and the best achieved rank-width upper bound. It can be seen that even for higher vertex counts a low rank-width is possible, but some graphs also appear to have a rank-width near the upper bound (Theorem 3.2.1).

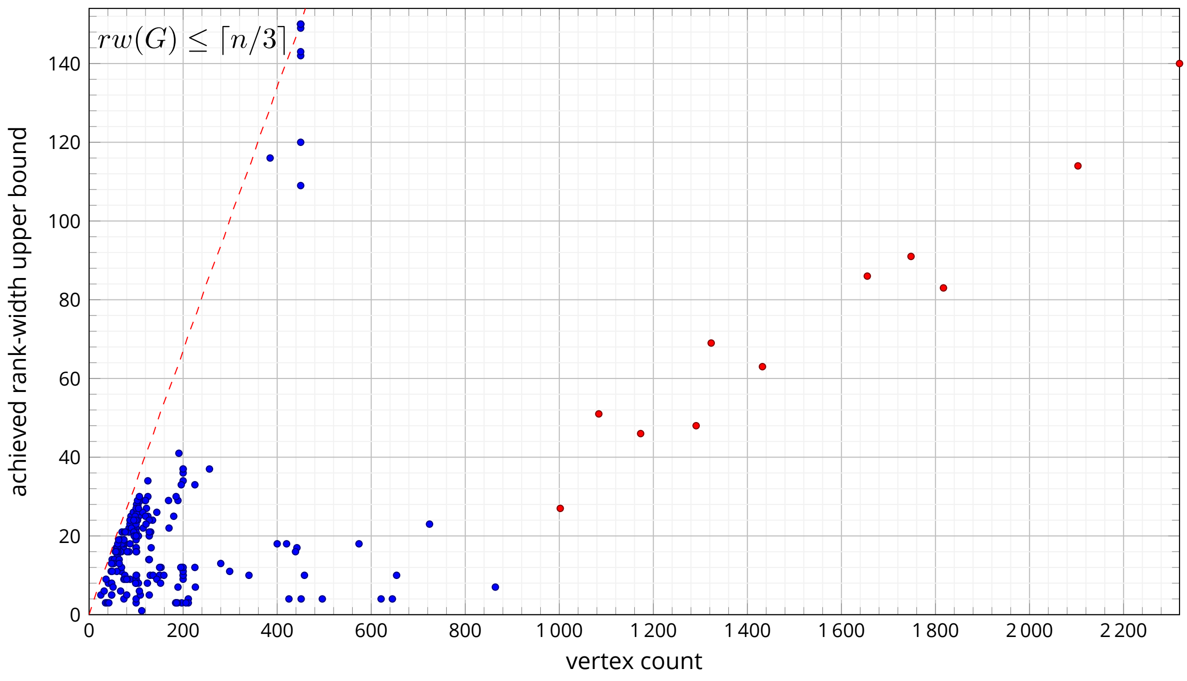


Figure 7.1 *Comparison between the graph vertex count and the best rank-width upper bound we achieved for the small/medium graphs in Table B.1 (blue) and large graphs in Table B.2 (red). The red dashed line indicates the upper bound from Theorem 3.2.1.*

7.1.1 Comparison with Beyß

The tested graphs include almost all graphs that were tested by Beyß [6, 7]. For all of those graphs we managed to find an equal or better rank-width upper bound, as can be seen in Figure 7.2. The more a point is to the right of the red dashed line, the larger the improvement compared to the result of Beyß.

An interesting result is the value 4 that could be found for graph celar06, for which a decomposition is shown in Figure 7.9. The results of Beyß [6, 7] show that a lower and an upper bound of 5 were found for this graph, concluding that the exact rank-width would be 5, but this appears to be some sort of error. Semi-manual checking of one of our obtained decompositions of width 4 did not reveal any inconsistencies, and given the results for other graphs there is no reason to suspect an issue with our program. For all other graphs for which an improved upper bound was found the newly obtained result was never lower than Beyß lower bound. Assuming that the rest of the lower bounds are correct, this means that for some more graphs the exact rank-width is known now: celar02 (rw 3), miles250 (rw 5), oesoca+ (rw 6) and pr107.tsp (rw 6). Additionally we found that bcs03 has an exact width of 1, as a width of 0 is only possible for graphs without edges. This graph was however not tested by Beyß.

Objectively comparing speed is difficult because of the vast difference in hardware used for benchmarking, however it is interesting to note that Beyß writes: *"The best run on fpsol2.i.1 however took over two hours to find the decomposition and further two hours to finish."* [6], with the best run being the one that achieved a width of 8. In our case the best of our 10 runs managed to find a decomposition of width 4 in just 31 seconds, and in most

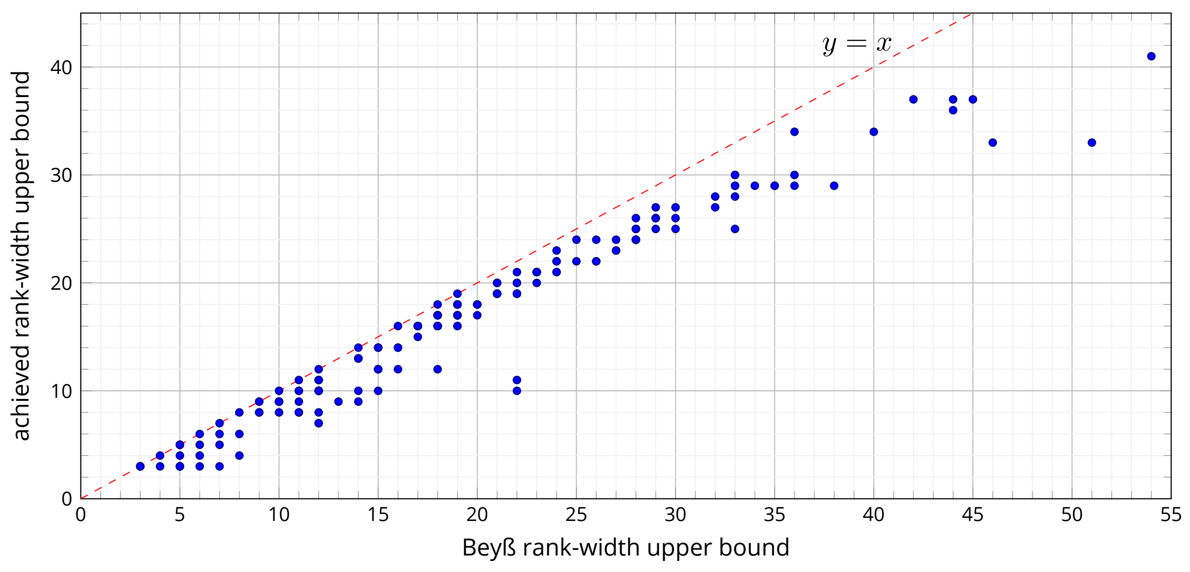


Figure 7.2 *Comparison between the best rank-width upper bound of Beyß [6, 7] and the best rank-width upper bound we achieved for the small/medium graphs in Table B.1.*

other runs at least within 1 minute. It is unlikely that the difference between 2 hours and 31 seconds can only be explained by faster hardware.

The difference in performance between the two algorithms is likely because of the better search algorithm, simulated annealing, which helps in escaping from local minima, and better operators. It seems that Beyß algorithm does not have a complete global operator like our move subtree operator, which plays an important role in the performance of our algorithm.

7.1.2 Comparison with other width parameters

In Figure 7.3 a comparison was made with the best known tree-width upper bound of graphs. For all small/medium graphs a rank-width upper bound was found less than or equal to the best known tree-width upper bound. For the large graphs the algorithm performed considerably worse. A similar thing can be seen when looking at the graphs with a known branch-width upper bound (Figure 7.4). The best rank-width upper bound we achieved is however not always less than or equal to the best known branch-width upper bound for the small/medium graphs. Nevertheless is it interesting to see that on average the small/medium graphs perform according to the theoretical bounds. Branch-width appears to be closer to rank-width than tree-width. This is not surprising as it holds [50] that for an undirected graph G with branch-width $brw(G)$ and tree-width $tw(G)$

$$\max(brw(G), 1) \leq tw(G) + 1.$$

7.1.3 Square grids

Some benchmarking was also done for $n \times n$ grids. They are useful because their exact rank-width was proven [31] to be $n - 1$. The same configuration was used as for the small/medium graphs. The results are shown in Table 7.1. For the grids of size 3x3, 4x4, 5x5 and 6x6 the exact rank-width was found, and for the grids of size 7x7, 10x10, 15x15, 20x20 and 25x25 the achieved width was one higher than the optimum. The grid of 30x30 performed considerably worse with an achieved width 9 higher than the optimum, presumably because of the large size. It is interesting that already from a size of 7x7 the optimum is not longer found. A similar thing could be seen in the results of Beyß [6].

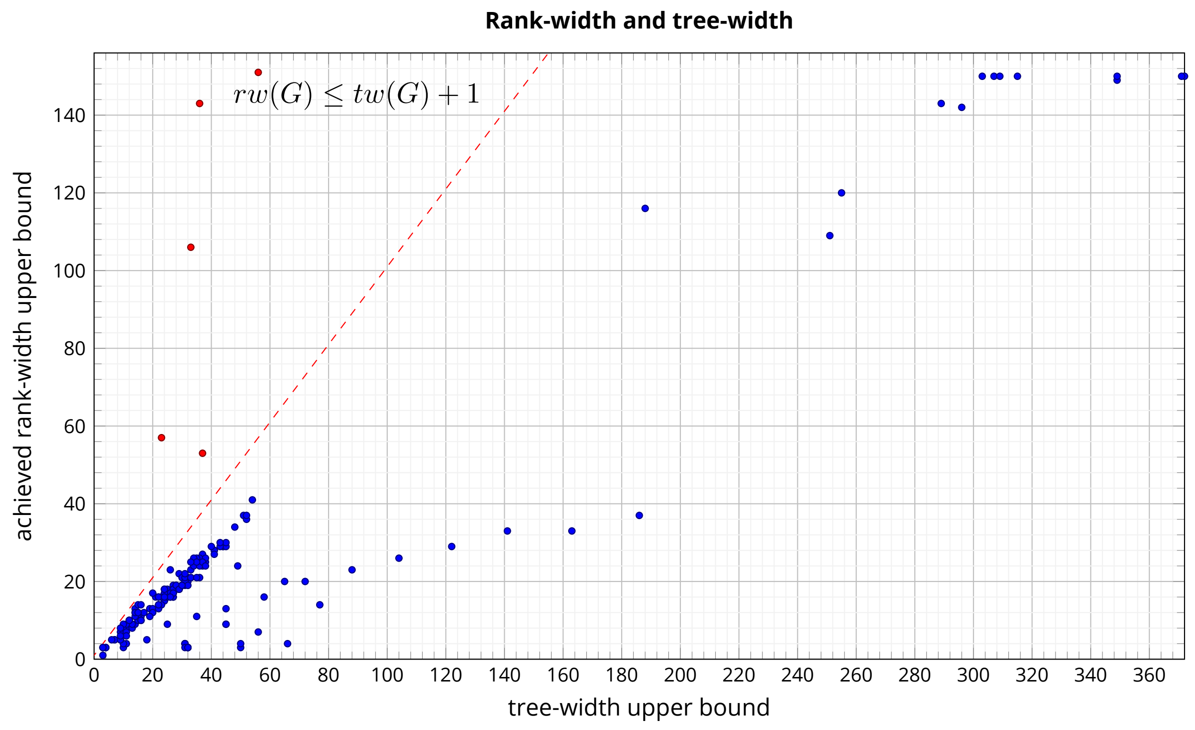


Figure 7.3 *Comparison between the best known tree-width upper bound and the best rank-width upper bound we achieved for the small/medium graphs in Table B.1 (blue) and large graphs in Table B.2 (red).*

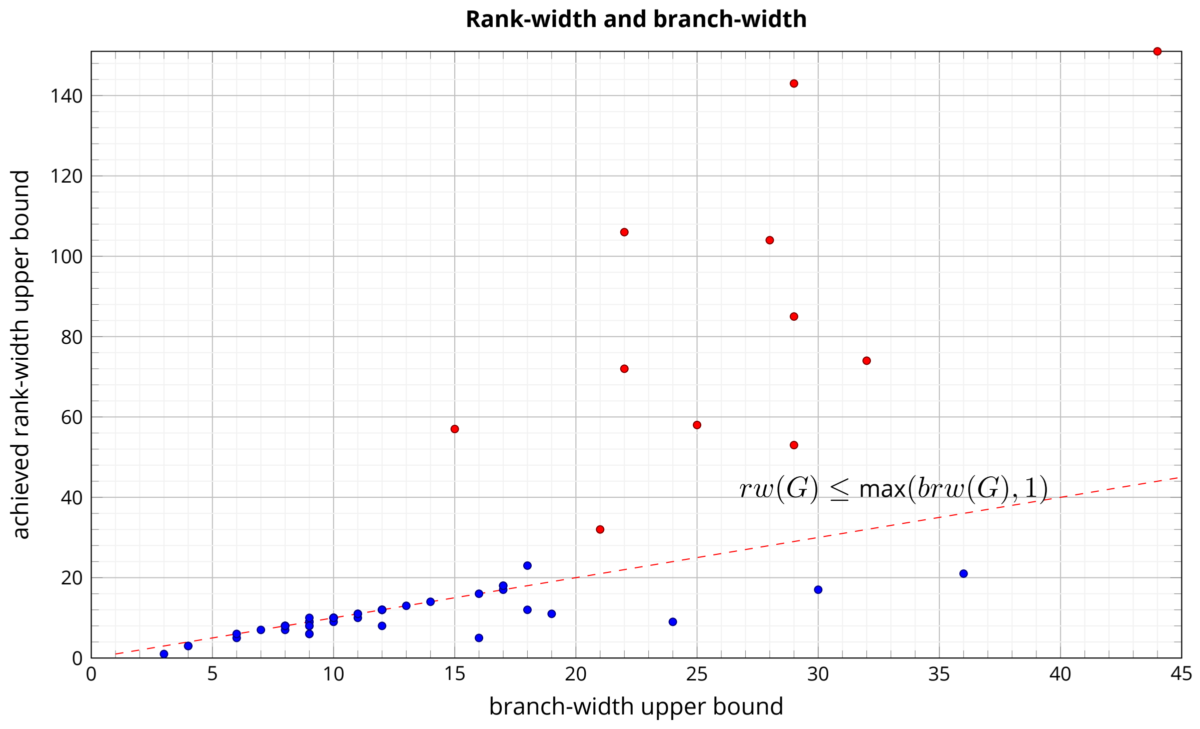


Figure 7.4 *Comparison between the best known branch-width upper bound and the best rank-width upper bound we achieved for the small/medium graphs in Table B.1 (blue) and large graphs in Table B.2 (red).*

grid	*vertices*	*edges*	*rw*	*result*	*time*
3x3	9	12	2	2	< 0.1 s
4x4	16	24	3	3	< 0.1 s
5x5	25	40	4	4	< 0.1 s
6x6	36	60	5	5	0.2 s
7x7	49	84	6	7	0.1 s
10x10	100	180	9	10	0.2 s
15x15	225	420	14	15	2.1 s
20x20	400	760	19	20	14.7 s
25x25	625	1200	24	25	60.3 s
30x30	900	1740	29	38	165.2 s

Table 7.1 *Benchmarking results for $n \times n$ grids showing the best achieved rank-width upper bound and the best time until a decomposition of that width was found.*

7.1.4 Randomness

In Figure B.1 and B.2 an overview can be seen of the achieved rank-width upper bounds during all 10 runs for each tested graph. The maximum difference between the best and worst obtained result is 21 for the small/medium graphs and 56 for the large graphs, and the average difference is 2 for the small/medium graphs and 41 for the large graphs. Especially the average of 2 seems to be a good result. This means that on average it is not needed to do a lot of runs to get a good result. Since 10 runs is a rather small amount to say something meaningful about the randomness of the algorithm, the program was additionally run 500 times for 4 different graphs to see more in detail how much the results vary between runs. The results are shown in Figure 7.5.

For celar08 and the 15x15 grid better results have a higher probability of being found, but for myciel7 and queen8_12 the results vary much more. For those latter two there was also a better result found than during the 10 main benchmarking runs. It is interesting to note that Beyß [6] performed a similar experiment which consisted of 1000 runs for the 15x15 grid. That experiment showed much more variation, with results ranging from 15 to 28, with 20 being the most common. In our case we found a value of 15 in 499 of the 500 runs. Interestingly neither Beyß nor our algorithm managed to find the optimal value of 14 for this graph despite the large number of runs.

When it comes to the time it took before the best width decomposition was found there is a lot of variation visible. For example the time to find a decomposition of width 10 for graph celar08 ranges all the way from 27 to 179 seconds (note that the time limit was 180 seconds). It is also interesting to see that the few times a value of 40 was found for myciel7 this was accomplished in less than 50 seconds, while some higher values sometimes took around 150 seconds. Overall it appears that a longer runtime does mean that better results will also be found, most likely because the decomposition was changed in such ways that it becomes difficult to get to a much better solution. It would then be better to start from the beginning again. This was something that Beyß [6] noticed as well.

7.2 $\mathbb{F}_4$-rank-width

For $\mathbb{F}_4$-rank-width there were no existing results or bounds in relation to other width parameters to compare to, and therefore a comparison was made to the results for rank-width and maximum matching-width that were obtained by converting the tested graphs to undirected. All these results are shown in Table B.3. For almost all of the tested graphs the results for the three width parameters are quite similar. Only mm9a and mm9b have a somewhat larger value for $\mathbb{F}_4$-rank-width compared to the two undirected parameters. Given the low number and variety of graphs it is difficult to draw conclusions about the relation between the parameters, however given the good results for rank-width and maximum matching-width in their own benchmarks and the similarity of the $\mathbb{F}_4$-rank-width results, we can say that the algorithm appears to perform similarly well for this width parameter.

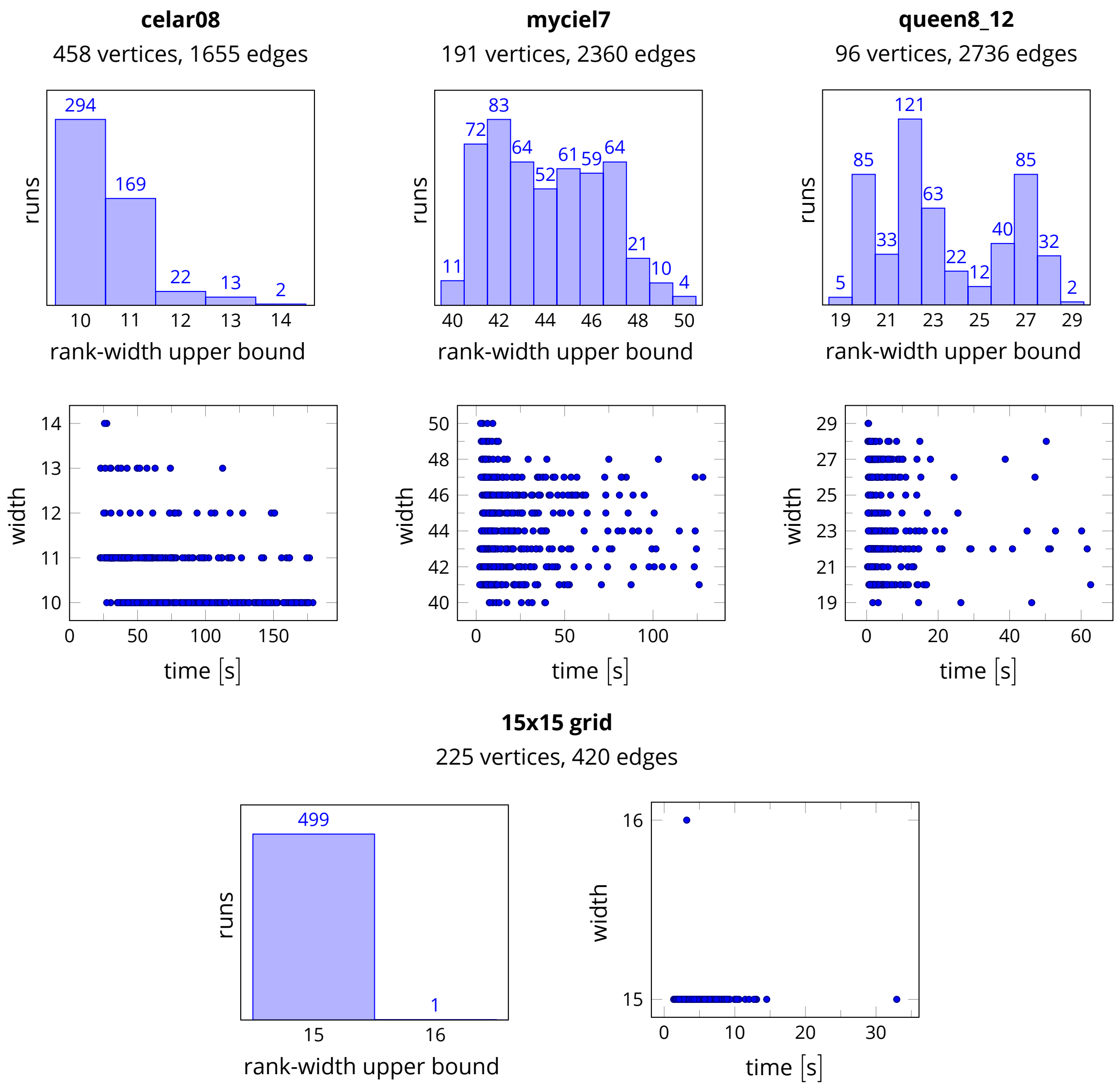


Figure 7.5 *Histograms showing for a number of graphs the achieved rank-width upper bounds during 500 runs and scatter plots showing the corresponding time for each run in seconds.*

7.2.1 Tournaments

An experiment was done with random tournaments to get an idea of their $\mathbb{F}_4$-rank-width. Tournaments are complete graphs where each edge has one of the two possible directions (no bidirectional edges). Because all the structural information is in the edge directions, using a width parameter for undirected graphs would make no sense. When using the regular rank-width on such a graph, for example, the width would always be 1, since it is just a complete graph. The experiment was performed by running the algorithm on 1000 random tournaments with 100 vertices and 1000 random tournaments with 30 vertices. The same settings were used as for the other $\mathbb{F}_4$-rank-width benchmarks, except that the time limit was set to 60 seconds, which was sufficient for these graphs, and only a single run was done for each instance.

The results showed that for all 100 vertex tournaments a width of 34 (upper bound) was found. For the 30 vertex tournaments a width of 10 (upper bound) was found 44 times and for all other instances a width of 9 was found. From this we can say that it is likely that random tournaments have a high probability of having an $\mathbb{F}_4$-rank-width near the upper bound of Theorem 3.2.1. However, tournaments in general do not need to have a high $\mathbb{F}_4$-rank-width, as some informal tests with transitive tournaments (edge $a \to b$ and $b \to c$ imply $a \to c$) showed that these likely have a width of 1. The high width for random tournaments might be because the randomness makes it unlikely that for all edges of the decomposition the matrix rows are similar enough to achieve a low rank. As smaller tournaments are less random, the probability of getting a lower width most likely becomes higher as the size of the tournament decreases.

7.3 Maximum matching-width

Maximum matching-width was implemented in the program to show that the algorithm not only works for ($\mathbb{F}_4$-)rank-width, but also for other branch-decomposition based width parameters. For computing maximum bipartite matchings the Hopcroft–Karp algorithm [29] was used.

From the results it seems that the algorithm performs similarly good for maximum matching-width as it does for rank-width. In Figure 7.6 a comparison between the best achieved maximum matching-width upper bound and the graph vertex count is shown. It can be seen that just like for rank-width even graphs with a large number of vertices can have a small maximum matching-width, and there are also graphs that have a width near the upper bound of Theorem 3.2.1. It can also be seen that compared to rank-width (Figure 7.1) maximum matching-width seems to be a bit more sensitive to the vertex count.

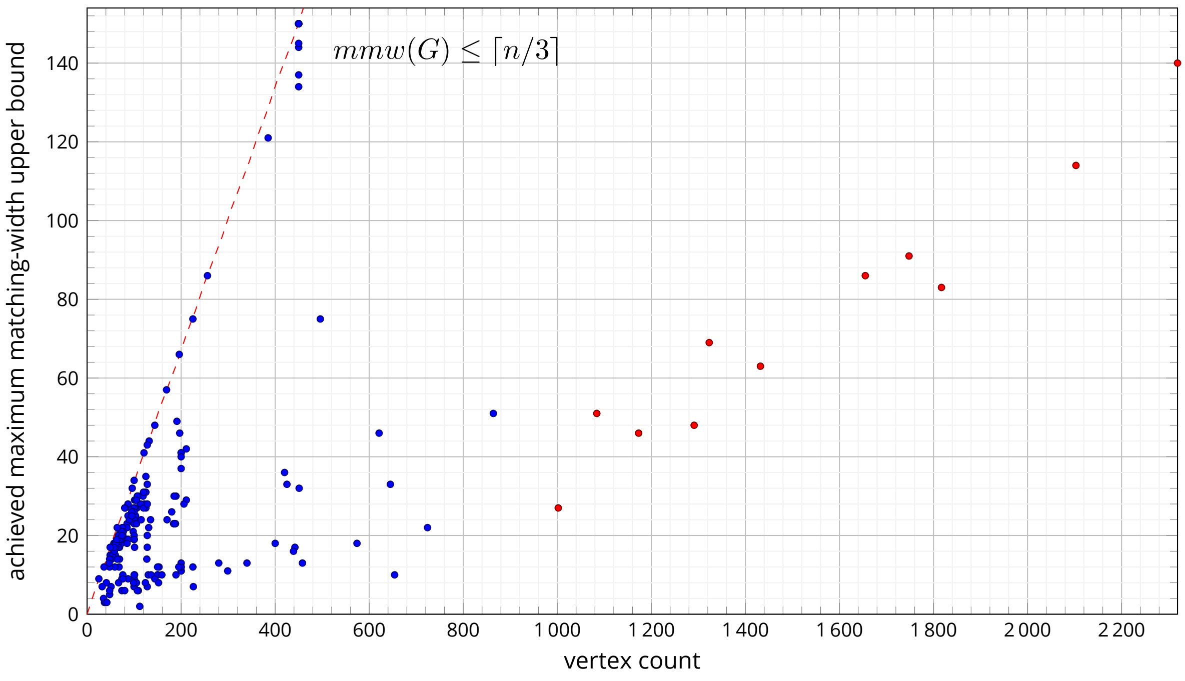


Figure 7.6 *Comparison between the graph vertex count and the best maximum matching-width upper bound we achieved for the small/medium graphs in Table B.1 (blue) and large graphs in Table B.2 (red). The red dashed line indicates the upper bound from Theorem 3.2.1.*

7.3.1 Comparison with other width parameters

In Figure 7.7 and 7.8 the best achieved maximum matching-width upper bound is compared to the best known tree-width upper bound and the best achieved rank-width upper bound respectively. Especially the comparison with tree-width is interesting as there is both an upper and a lower bound for this relation. Note that because we are comparing upper bounds the theoretical bounds are only an indication, and some results do indeed lay outside the expected area. It is interesting to see however that for most of the graphs the achieved width is somewhere in the middle between the upper and lower bound lines.

In the comparison with rank-width it can be seen that for many graphs rank-width and maximum matching-width are quite close, however for some graphs the difference is much bigger. It is likely that compared to rank-width, maximum matching-width is more sensitive to dense graphs. After all, the more edges there are in a graph, the more edges there will be between the two sides of a bipartition, and the less likely it becomes that a maximum bipartite matching has a low value. For example, rank-width has a width of 1 for complete graphs of size n, while for maximum matching-width this is $\lceil n/3 \rceil$ (upper bound) because there is an edge between every pair of vertices in any bipartition.

7.3.2 Square grids

Unlike for rank-width, there is no exact maximum matching-width proven for square grid graphs, however based on the relations with other width parameters such as rank-width and branch-width it can be shown [54] that for a $n \times n$ grid graph G

$$n - 1 \leq mmw(G) \leq n.$$

For the benchmarking the same configuration was used as for the small/medium graphs, and the results are shown in Table 7.2. From the results it seems likely that the maximum matching-width of a $n \times n$ grid is n.

grid	*vertices*	*edges*	*result*	*time*
3x3	9	12	3	< 0.1 s
4x4	16	24	4	< 0.1 s
5x5	25	4	5	< 0.1 s
6x6	36	60	6	< 0.1 s
7x7	49	84	7	0.1 s
10x10	100	180	10	0.3 s
15x15	225	420	15	2.9 s
20x20	400	760	20	20.6 s
25x25	625	1200	25	78.3 s
30x30	900	1740	30	144.0 s

Table 7.2 *Benchmarking results for $n \times n$ grids showing the best achieved maximum matching-width upper bound and the best time until a decomposition of that width was found.*

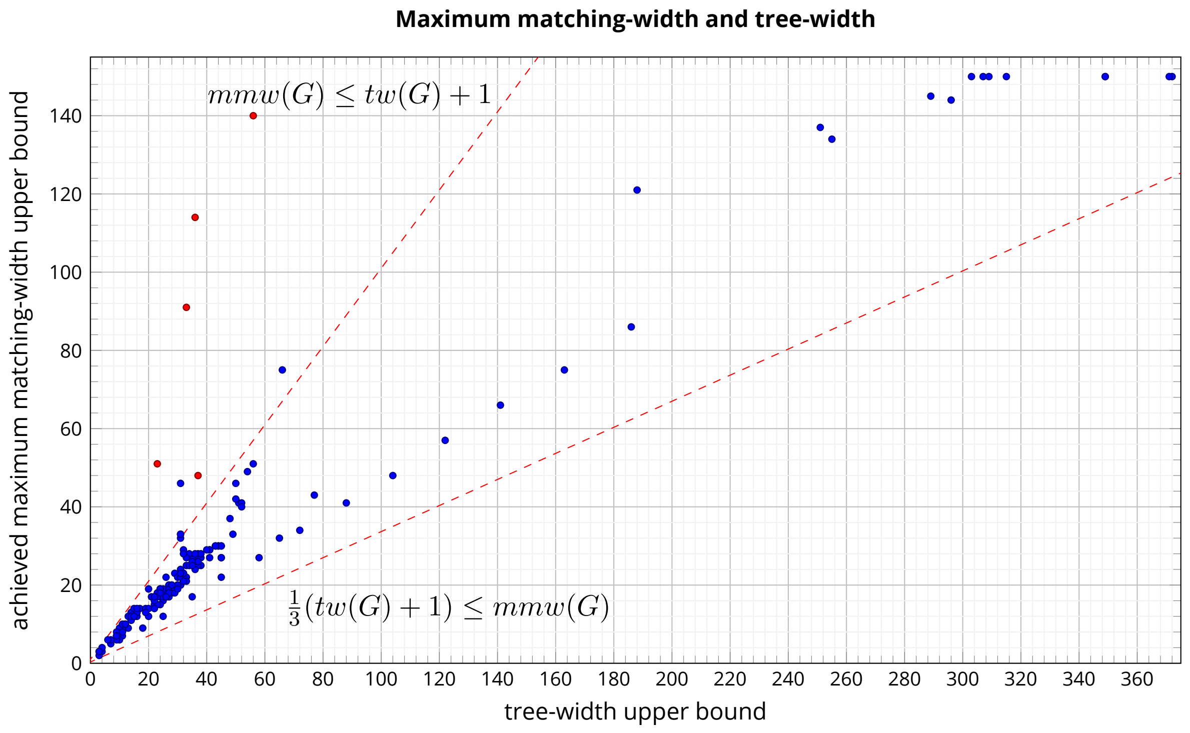


Figure 7.7 *Comparison between the best known tree-width upper bound and the best maximum matching-width upper bound we achieved for the small/medium graphs in Table B.1 (blue) and the large graphs in Table B.2 (red). The red dashed lines indicate theoretical upper and lower bounds, but as both the tree-width and maximum matching-width are upper bounds this is only an indication.*

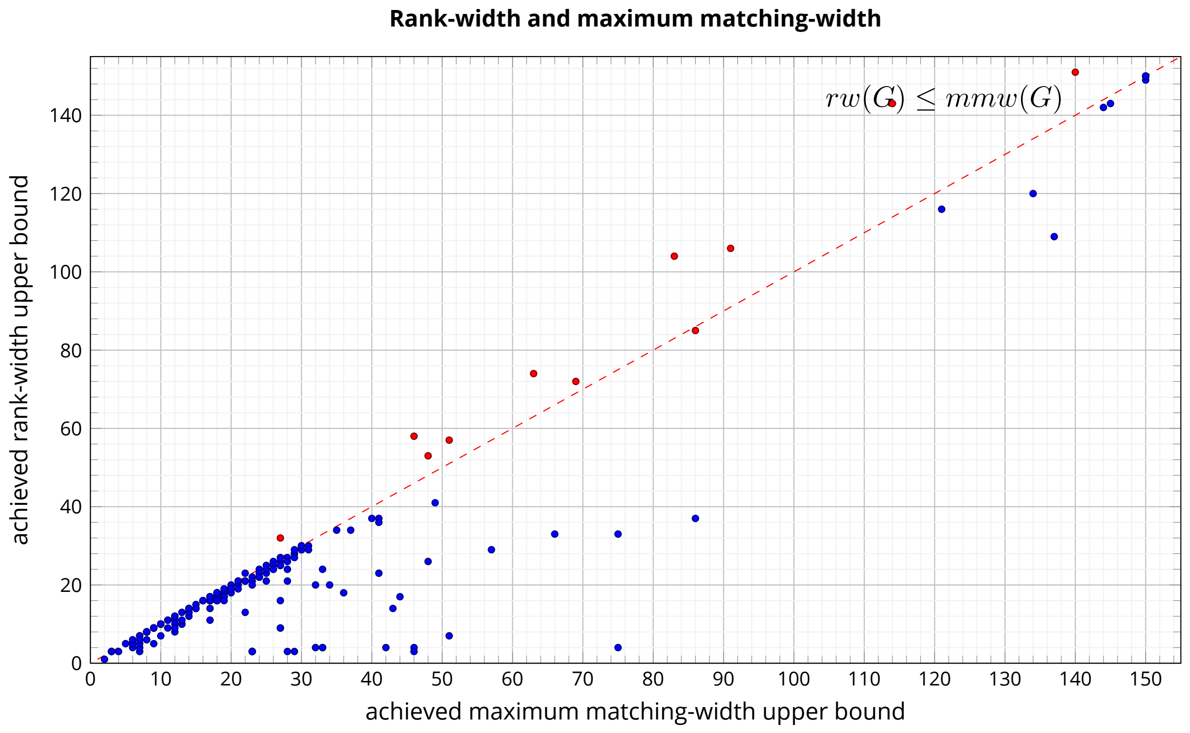


Figure 7.8 *Comparison between the best rank-width and maximum matching-width upper bound we achieved for the small/medium graphs in Table B.1 (blue) and the large graphs in Table B.2 (red).*

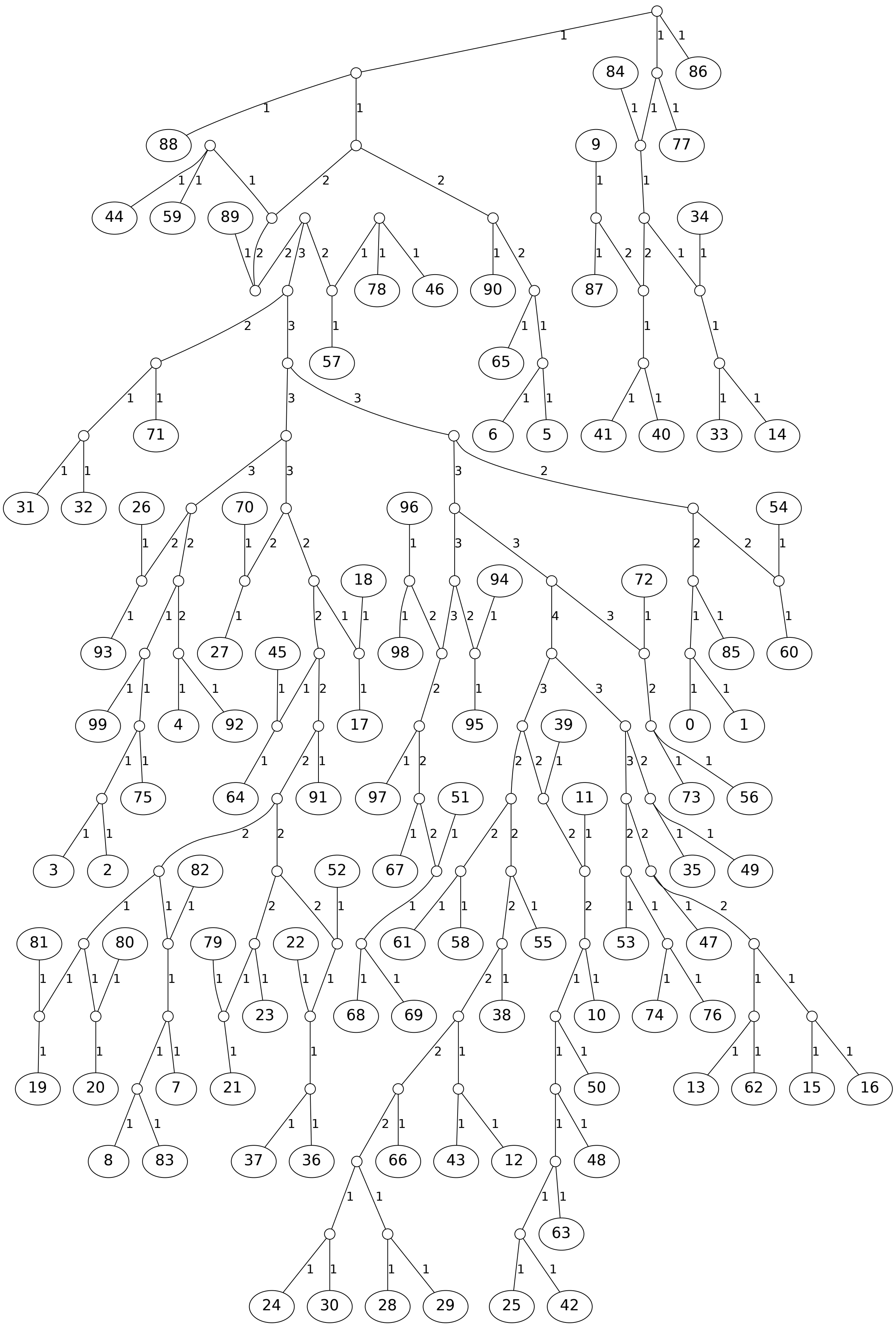


Figure 7.9 *Rank-decomposition for graph celar06 of width 4. The width of each edge is indicated.*

Chapter 8

Conclusion and future work

8.1 Conclusion

From the obtained results we can conclude that simulated annealing works very well for approximating rank-width and other branch-decomposition based width parameters, as long as the size of the graphs is not too large. Heuristics that guide the operator choices may be needed to improve the results on larger graphs. We saw that for some graphs the results can vary quite a bit between different runs, but that for all graphs we tested that were also tested by Beyß [6, 7] we were able to find an equal or better rank-width upper bound with just 10 runs, and for most graphs our best upper bound had already been found during the first 5 runs. This shows that there is no need to do a large number of runs to find a good result using our algorithm. For the graph fpsol2.i.1 we even saw that our algorithm could find a width of 4 in 31 seconds, while it took Beyß over two hours to find a width of 8. It is also interesting to note that although our algorithm performed better on $n \times n$ graphs, it could still not find the exact rank-width value for $n \geq 7$.

When it comes to the $\mathbb{F}_4$-rank-width results, we saw that the difference with rank-width and maximum matching-width is rather small, but it is not clear if this is often the case, or just a result of the small number of directed graphs we tried. More research would be needed in that regard. From the experiments on random tournaments it seems likely that these often have a high $\mathbb{F}_4$-rank-width, presumably because of the randomness.

For maximum matching-width we saw that the results are often close to those of rank-width, but that for some graphs the maximum matching-width is much larger. It is likely that maximum matching-width is more sensitive to the density of a graph than rank-width. From the results on square grids it appears likely that the maximum matching-width of a $n \times n$ graph is exactly n. It would be interesting if this could be formally proven.

8.2 Future work

During the work on this thesis some ideas and questions were raised that could not be incorporated, but that might be interesting for further research. They are presented here in no particular order.

Faster matrix rank computations

It became clear that the most important bottleneck of the algorithm is computing the rank of matrices. It might be interesting to see if it can be done faster in this specific context. An interesting observation is that the matrices the rank has to be computed for are all submatrices of the complete adjacency matrix of the graph. Perhaps this could be exploited in some way, for example with a form of preprocessing. It might also be interesting to see if it would be possible to efficiently compute the cut-rank of a partition that is very similar to a partition of which the cut-rank was already computed. For example by storing additional information and performing some sort of row/column updates.

More partial score updates

In the algorithm partial score updates were considered, but were eventually only implemented for the local swap operator because for the other two operators it would require to store extra information that needs to be updated with any changes to the decomposition. The extra complexity this would add to the algorithm made it seem not very beneficial to implement partial updates for these operators, especially with the cut-rank cache (Section 5.5). Regardless it might still be interesting to investigate in future work.

Smarter construction of the initial solution

It would be interesting to see if better initial solutions could be constructed using some heuristics. Using previously generated decompositions as initial solution (possibly with some random permutations) is also something that could be interesting to investigate.

Smarter operators

It might be interesting to see if heuristics could be used to let operators make better decisions. This might help to make the algorithm perform better on larger graphs.

Applying the algorithm to more parameters

We already showed that the algorithm works well for rank-width, $\mathbb{F}_4$-rank-width and maximum matching-width. It is likely that the algorithm, or a variant, could be applied to more different width parameters that are based on branch-decompositions.

Tuning the algorithm for its application

When the output of our algorithm is to be used as an input for a specific other algorithm it may be possible to tune the score function such that the generated decompositions are specifically efficient for that algorithm. This could be interesting in a case where it would not matter if the decomposition has just a few edges of high width, as long as the rest of the edges have a low width for example. It would then be possible to change the score function to emphasize the widths of the individual edges instead of the maximum edge-width.

Broader search using parallelism

For some graphs the results of the algorithm vary a lot between different runs. It appears that those graphs may have local minima that are difficult to escape from once certain changes are made to the decomposition. The current algorithm uses parallelism to compute the width of multiple edges at the same time to make the algorithm faster. A different application of parallelism would be to branch the algorithm so that it runs in parallel on multiple threads, starting from the same decomposition, but each making different decisions. After a number of iterations the best of the decompositions is selected and all threads would then continue again starting from that decomposition. In this way, a larger part of the search space is explored and it makes it easier to escape from local minima when they only appear on some of the threads. To some extend such an approach would take into account the observation that restarting is usually more likely to find a better solution than to run for a longer time. A downside would be that the algorithm would be slower, because multiple decomposition edges would no longer be computed in parallel (as this would not make much sense when there are already multiple threads running).

Exact maximum matching-width of square grids

From the maximum matching-width benchmarks for square grids it appears that $n \times n$ grids most likely have an exact width of n. It would be interesting if this could be formally proven.

Exact rank-width for planar graphs

For planar graphs there exists an algorithm [51, 25, 26] to compute branch-width exactly in polynomial time. Perhaps there is also a way to compute rank-width exactly in polynomial time for planar graphs.

Bibliography

[1] “A comparison of cooling schedules for simulated annealing (artificial intelligence).” [Online]. Available: http://what-when-how.com/artificial-intelligence/a-comparison-of-cooling-schedules-for-simulated-annealing-artificial-intelligence/

[2] K. Ahn and J. Jeong, “Computing the maximum matching width is NP-hard,” *ArXiv*, vol. abs/1710.05117, 2017. [Online]. Available: https://arxiv.org/abs/1710.05117

[3] M. R. Albrecht, G. V. Bard, and C. Pernet, “Efficient dense Gaussian elimination over the finite field with two elements,” 2011. [Online]. Available: https://arxiv.org/abs/1111.6549

[4] J. Bang-Jensen and G. Gutin, *Classes of Directed Graphs*. Springer International Publishing, 2018, ISBN: 978-3-319-71840-8. [Online]. Available: https://doi.org/10.1007/978-3-319-71840-8

[5] E. Bertolazzi and A. Rimoldi, “Fast matrix decomposition in $\mathbb{F}_2$,” *Journal of Computational and Applied Mathematics*, vol. 260, p. 519–532, Apr. 2014. [Online]. Available: https://doi.org/10.1016/j.cam.2013.10.026

[6] M. Beyß, “Fast Algorithms for Rank-Width,” May 2012, Diploma Thesis, RWTH Aachen University.

[7] M. Beyß, “Fast algorithm for rank-width,” in *Mathematical and Engineering Methods in Computer Science*, A. Kučera, T. A. Henzinger, J. Nešetřil, T. Vojnar, and D. Antoš, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 82–93. [Online]. Available: https://doi.org/10.1007/978-3-642-36046-6_9

[8] B.-M. Bui-Xuan, J. A. Telle, and M. Vatshelle, “Boolean-width of graphs,” *Theoretical Computer Science*, vol. 412, no. 39, pp. 5187–5204, 2011. [Online]. Available: https://doi.org/10.1016/j.tcs.2011.05.022

[9] H. Y. Cheung, T. C. Kwok, and L. C. Lau, “Fast matrix rank algorithms and applications,” *J. ACM*, vol. 60, no. 5, Oct. 2013. [Online]. Available: https://doi.org/10.1145/2528404

[10] B. Courcelle and M. M. Kanté, “Graph operations characterizing rank-width and balanced graph expressions,” in *International Workshop on Graph-Theoretic Concepts in Computer Science*, vol. 4769. Springer, 2007, pp. 66–75. [Online]. Available: https://doi.org/10.1007/978-3-540-74839-7_7

[11] B. Courcelle, J. Makowsky, and U. Rotics, “Linear time solvable optimization problems on graphs of bounded clique width,” in *Theory of Computing Systems*, vol. 33, Apr. 2000, pp. 125–150. [Online]. Available: https://doi.org/10.1007/s002249910009

[12] B. Courcelle and S. Oum, “Vertex-minors, monadic second-order logic, and a conjecture by Seese,” *Journal of Combinatorial Theory, Series B*, vol. 97, no. 1, pp. 91–126, 2007. [Online]. Available: https://doi.org/10.1016/j.jctb.2006.04.003

[13] K. K. Dabrowski, F. Dross, J. Jeong, M. M. Kanté, O.-j. Kwon, S. Oum, and D. Paulusma, “Computing small pivot-minors,” in *Graph-Theoretic Concepts in Computer Science*, A. Brandstädt, E. Köhler, and K. Meer, Eds. Cham: Springer International Publishing, 2018, pp. 125–138. [Online]. Available: https://doi.org/10.1007/978-3-030-00256-5_11

[14] E. Eiben, R. Ganian, T. Hamm, L. Jaffke, and O. Kwon, "A unifying framework for characterizing and computing width measures," *CoRR*, vol. abs/2109.14610, 2021. [Online]. Available: https://arxiv.org/abs/2109.14610

[15] R. Ganian, "Automata-formalization for graphs of bounded rank-width," Master's thesis, Masarykova univerzita, Fakulta informatiky, 2008. [Online]. Available: https://is.muni.cz/th/is66a/?lang=en

[16] R. Ganian and P. Hliněný, "Automata approach to graphs of bounded rank-width," in *Proceedings of the 19th International Workshop on Combinatorial Algorithms, IWOCA 2008, September 13-15, 2008, Nagoya, Japan*, M. Miller and K. Wada, Eds. College Publications, 2008, pp. 4–15. [Online]. Available: https://www.fi.muni.cz/~hlineny/papers/parse-rw-mso.pdf

[17] R. Ganian and P. Hliněný, "Better polynomial algorithms on graphs of bounded rank-width," in *Combinatorial Algorithms*, J. Fiala, J. Kratochvíl, and M. Miller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 266–277. [Online]. Available: https://doi.org/10.1007/978-3-642-10217-2_27

[18] R. Ganian, P. Hliněný, J. Kneis, A. Langer, J. Obdržálek, and P. Rossmanith, "Digraph width measures in parameterized algorithmics," *Discrete Applied Mathematics*, vol. 168, pp. 88–107, 2014, fifth Workshop on Graph Classes, Optimization, and Width Parameters, Daejeon, Korea, October 2011. [Online]. Available: https://doi.org/10.1016/j.dam.2013.10.038

[19] R. Ganian, P. Hliněný, J. Kneis, D. Meister, J. Obdržálek, P. Rossmanith, and S. Sikdar, "Are there any good digraph width measures?" *Journal of Combinatorial Theory, Series B*, vol. 116, pp. 250–286, 2016. [Online]. Available: https://doi.org/10.1016/j.jctb.2015.09.001

[20] R. Ganian, P. Hliněný, and J. Obdržálek, "A unified approach to polynomial algorithms on graphs of bounded (bi-)rank-width," *European Journal of Combinatorics*, vol. 34, no. 3, pp. 680–701, 2013, combinatorial Algorithms and Complexity. [Online]. Available: https://doi.org/10.1016/j.ejc.2012.07.024

[21] R. Ganian and P. Hlinný, "On parse trees and Myhill-Nerode-type tools for handling graphs of bounded rank-width," *Discrete Appl. Math.*, vol. 158, no. 7, p. 851–867, Apr. 2010. [Online]. Available: https://doi.org/10.1016/j.dam.2009.10.018

[22] J. F. Geelen, B. Gerards, and G. Whittle, "Branch-width and well-quasi-ordering in matroids and graphs," *Journal of Combinatorial Theory Series B*, vol. 84, pp. 270–290, Mar. 2002. [Online]. Available: https://doi.org/10.1006/jctb.2001.2082

[23] V. Gogate and R. Dechter, "A complete anytime algorithm for treewidth," in *UAI '04, Proceedings of the 20th Conference in Uncertainty in Artificial Intelligence, Banff, Canada, July 7-11, 2004*, ser. UAI '04, D. M. Chickering and J. Y. Halpern, Eds. Arlington, Virginia, USA: AUAI Press, 2004, p. 201–208. [Online]. Available: https://dl.acm.org/doi/10.5555/1036843.1036868

[24] M. Grohe and P. Schweitzer, "Isomorphism testing for graphs of bounded rank width," in *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, 2015, pp. 1010–1029. [Online]. Available: https://doi.org/10.1109/FOCS.2015.66

[25] I. V. Hicks, "Planar branch decompositions I: The ratcatcher," *INFORMS Journal on Computing*, vol. 17, no. 4, pp. 402–412, 2005. [Online]. Available: https://doi.org/10.1287/ijoc.1040.0075

[26] I. V. Hicks, "Planar branch decompositions II: The cycle method," *INFORMS Journal on Computing*, vol. 17, no. 4, pp. 413–421, 2005. [Online]. Available: https://doi.org/10.1287/ijoc.1040.0074

[27] P. Hliněný and S. Oum, "Finding branch-decompositions and rank-decompositions," in *Algorithms – ESA 2007*, L. Arge, M. Hoffmann, and E. Welzl, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 163–174. [Online]. Available: https://doi.org/10.1007/978-3-540-75520-3_16

[28] P. Hliněný, S. Oum, D. Seese, and G. Gottlob, “Width parameters beyond treewidth and their applications,” *The Computer Journal - CJ*, vol. 51, pp. 326–362, May 2008. [Online]. Available: https://doi.org/10.1093/comjnl/bxm052

[29] J. E. Hopcroft and R. M. Karp, “A $n^{5/2}$ algorithm for maximum matchings in bipartite graphs,” *SIAM Journal on computing*, vol. 2, no. 4, pp. 225–231, 1973. [Online]. Available: https://doi.org/10.1137/0202019

[30] E. Hvidevold, S. Sharmin, J. Telle, and M. Vatshelle, “Finding good decompositions for dynamic programming on dense graphs,” in *Proceedings of IPEC*, vol. 7112, 09 2011, pp. 219–231. [Online]. Available: https://doi.org/10.1007/978-3-642-28050-4_18

[31] V. Jelínek, “The rank-width of the square grid,” *Discrete Applied Mathematics*, vol. 158, pp. 230–239, Jan. 2008. [Online]. Available: https://doi.org/10.1016/j.dam.2009.02.007

[32] M. M. Kanté and M. Rao, “The rank-width of edge-coloured graphs,” *Theory of Computing Systems*, vol. 52, no. 4, p. 599–644, Apr. 2012. [Online]. Available: https://doi.org/10.1007/s00224-012-9399-y

[33] M. M. Kanté, “The rank-width of directed graphs,” *CoRR*, vol. abs/0709.1433, 2007. [Online]. Available: https://arxiv.org/abs/0709.1433v3

[34] M. M. Kanté, “Graph Structurings: Some Algorithmic Applications,” Ph.D. dissertation, Université Sciences et Technologies - Bordeaux I, Dec. 2008. [Online]. Available: https://tel.archives-ouvertes.fr/tel-00419301

[35] M. M. Kanté and M. Rao, “Directed rank-width and displit decomposition,” in *Graph-Theoretic Concepts in Computer Science*, C. Paul and M. Habib, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 214–225. [Online]. Available: https://doi.org/10.1007/978-3-642-11409-0_19

[36] A. M. C. A. Koster, H. L. Bodlaender, and S. P. M. van Hoesel, “Treewidth: Computational experiments,” *Electronic Notes in Discrete Mathematics*, vol. 8, pp. 54–57, May 2001. [Online]. Available: https://doi.org/10.1016/S1571-0653(05)80078-2

[37] W. Mahdi, S. A. Medjahed, and M. Ouali, “Performance analysis of simulated annealing cooling schedules in the context of dense image matching,” *Computación y Sistemas*, vol. 21, pp. 493–501, Oct. 2017. [Online]. Available: https://doi.org/10.13053/CyS-21-3-2553

[38] S. Oum, “Approximation algorithm for the clique-width,” Dec. 2003. [Online]. Available: https://mathsci.kaist.ac.kr/~sangil/pdf/rwdslide.pdf

[39] S. Oum, “Replace clique-width with rank-width,” May 2004. [Online]. Available: https://math.kaist.ac.kr/~sangil/pdf/rankwidthslide.pdf

[40] S. Oum, “Rank-width and vertex-minors,” *Journal of Combinatorial Theory, Series B*, vol. 95, no. 1, pp. 79–100, 2005. [Online]. Available: https://doi.org/10.1016/j.jctb.2005.03.003

[41] S. Oum, “Recognizing rank-width,” Jan. 2005. [Online]. Available: https://math.kaist.ac.kr/~sangil/pdf/oberwolfach2005.pdf

[42] S. Oum, “Introduction to rank-width,” Oct. 2007. [Online]. Available: https://mathsci.kaist.ac.kr/~sangil/pdf/2007eugene.pdf

[43] S. Oum, “Approximating rank-width and clique-width quickly,” *ACM Trans. Algorithms*, vol. 5, no. 1, Dec. 2008. [Online]. Available: https://doi.org/10.1145/1435375.1435385

[44] S. Oum, “Rank-width is less than or equal to branch-width,” *Journal of Graph Theory*, vol. 57, pp. 239–244, Mar. 2008. [Online]. Available: https://doi.org/10.1002/jgt.20280

[45] S. Oum, “Computing rank-width exactly,” *Inf. Process. Lett.*, vol. 109, no. 13, p. 745–748, Jun. 2009. [Online]. Available: https://doi.org/10.1016/j.ipl.2009.03.018

[46] S. Oum, “Rank-width: Algorithmic and structural results,” *Discrete Applied Mathematics*, vol. 231, p. 15–24, Nov. 2017. [Online]. Available: https://doi.org/10.1016/j.dam.2016.08.006

[47] S. Oum and P. Seymour, “Approximating clique-width and branch-width,” *Journal of Combinatorial Theory, Series B*, vol. 96, no. 4, pp. 514–528, 2006. [Online]. Available: https://doi.org/10.1016/j.jctb.2005.10.006

[48] S. Oum and P. Seymour, “Testing branch-width,” *Journal of Combinatorial Theory, Series B*, vol. 97, no. 3, pp. 385–393, 2007. [Online]. Available: https://doi.org/10.1016/j.jctb.2006.06.006

[49] A. Overwijk, E. Penninkx, and H. L. Bodlaender, “A local search algorithm for branchwidth,” in *Lecture Notes in Computer Science*, vol. 6543, Jan. 2011, pp. 444–454. [Online]. Available: https://doi.org/10.1007/978-3-642-18381-2_37

[50] N. Robertson and P. Seymour, “Graph minors. X. Obstructions to tree-decomposition,” *Journal of Combinatorial Theory, Series B*, vol. 52, no. 2, pp. 153–190, 1991. [Online]. Available: https://doi.org/10.1016/0095-8956(91)90061-N

[51] P. Seymour and R. Thomas, “Call routing and the ratcatcher,” *Combinatorica*, vol. 14, no. 2, pp. 217–241, Jun. 1994. [Online]. Available: https://doi.org/10.1007/BF01215352

[52] E. Talbi, *Metaheuristics: From Design to Implementation*. Wiley, Jul. 2009, ISBN: 978-0-470-27858-1. [Online]. Available: https://www.wiley.com/en-us/Metaheuristics%3A+From+Design+to+Implementation+-p-9780470278581

[53] J.-W. van den Broek and H. L. Bodlaender, “TreewidthLIB: A benchmark for algorithms for Treewidth and related graph problems.” [Online]. Available: https://www.cs.uu.nl/research/projects/treewidthlib/

[54] M. Vatshelle, “New width parameters of graphs,” Ph.D. dissertation, May 2012. [Online]. Available: https://hdl.handle.net/1956/6166

Appendix A

The program

The program that implements the algorithm was written in C# with .NET 5.0 and was tested on both Windows and Linux. It can approximate rank-width, $\mathbb{F}_4$-rank-width and maximum matching-width and outputs the corresponding decomposition in dot graph format. The program can make use of SIMD instructions (AVX2 recommended) and multiple cores. The source of the program can be found on GitHub [1]. It comes with a number of unit tests that verify some parts of the program.

A.1 Usage

```
RankWidthApproximate.exe [options] graph.dgf
```

The following options are available:

-ac Enables adaptive cooling (Section 4.2).

-d Approximate the $\mathbb{F}_4$-rank-width of a directed input graph (can not be used in combination with **-mm**).

-d2u Approximate the rank-width or maximum matching-width of a directed input graph by first converting it to undirected (each arc becomes an undirected edge).

-is *seed* Sets the seed for the random generator for the initial solution (by default the same random generator is used as for the search process).

-mm Approximate the maximum matching-width of an undirected input graph (or directed converted to undirected with **-d2u**).

-s *seed* Sets the seed for the random generator (random by default).

-t *temperature* Sets the initial temperature (default is 5.0).

-td *delta* Enables the thresholding heuristic (Section 5.2) with the given threshold delta.

-tl *seconds* Stops the search process if it is still running after the given number of seconds (the search process also stops when the temperature drops below 0.05).

-v Output more information during the search process.

[1] `https://github.com/Gericom/RankWidthApproximate`

A.2 Input graph format

The program supports a number of variations of the text based DIMACS graph format, for both undirected and directed graphs.

Undirected

For undirected graphs the following rules apply. Figure A.1 shows some examples of valid inputs.

- Lines starting with `c`, `n` or `x` are ignored.
- Before any edges are defined, a header is expected in the following format:

  ```
  p format vtxCount edgeCount
  ```

  The `format` field is ignored by the program, but usually contains the word `edge`. A file should only have a single header line.
- Edges can be defined as either

  ```
  vtxId1 vtxId2
  ```

  or

  ```
  e vtxId1 vtxId2
  ```

  for an edge between `vtxId1` and `vtxId2`. There are expected to be as many edge definitions as specified in the header. In case of the `e` format, any other input after `vtxId2` is ignored.
- Vertex ids can be anything without spaces. As such it does not matter if vertices are numbered from 0, from 1 or identified with any other number, letter or label.

```
p edge 4 4
1 2
2 3
3 4
4 1
```

```
c 5 vertex cycle
p edge 5 5
e 0 1
e 1 2
e 2 3
e 3 4
e 0 4
```

```
x low 19.00
p edge 3 3
n 92 1.00
n 26 1.00
n 93 1.00
e 26 93
e 26 92
e 92 93
```

```
p edge 7 11
e A B
e A C
e B D
e B E
e C D
e D F
e D E
e D G
e C G
e E F
e F G
```

Figure A.1 *Examples of supported undirected graph inputs.*

Directed

For directed graphs the format is mostly similar to the undirected format, except that the header now contains the number of arcs instead of the number of edges, and the way the arcs are defined differs slightly from the way edges were defined. They are defined as either

```
vtxId1 vtxId2
```

or

```
a vtxId1 vtxId2
```

for an arc from `vtxId1` to `vtxId2`. In case of the `a` format, any other input after `vtxId2` (such as weights) is ignored.

Appendix B

Benchmarking results

This appendix contains tables with all benchmarking results. The results are discussed in Chapter 7.

B.1 Undirected graphs

In the tables the columns mean the following:

rw result Best rank-width upper bound from 10 tries with the algorithm from this thesis.

Beyß Best rank-width upper bound achieved in the thesis of Beyß [6].

boolw ub Best boolean-width upper bound from the paper of Hvidevold et al. [30]. For an undirected graph G the following relations hold:

$$\log_2 rw(G) \leq boolw(G) \leq \tfrac{1}{4} rw(G)^2 + O(rw(G)) \quad \text{[8]},$$
$$boolw(G) \leq mmw(G) \quad \text{[54]}.$$

tw ub Best tree-width upper bound [36, 23, 6, 53]. For an undirected graph G the following relations hold:

$$rw(G) \leq tw(G) + 1 \quad \text{[44]},$$
$$\tfrac{1}{3}(tw(G) + 1) \leq mmw(G) \leq tw(G) + 1 \quad \text{[54]}.$$

brw ub Best branch-width upper bound [49, 25, 26, 53]. For an undirected graph G the following relations hold:

$$rw(G) \leq \max(brw(G), 1) \quad \text{[44]},$$
$$mmw(G) \leq \max(brw(G), 1) \quad \text{[54]}.$$

Note that for planar graphs branch-width can be computed exactly [51, 25, 26].

mmw result Best maximum matching-width upper bound from 10 tries with the algorithm from this thesis. For an undirected graph G it holds that

$$rw(G) \leq mmw(G) \quad \text{[54]}.$$

Note that the original TreewidthLIB database is no longer available. A backup can be found here: `https://github.com/emnh/boolwidth-data`.

B.1.1 TreewidthLIB small/medium

graph	*vertices*	*edges*	*rw result*	*Beyß*	*boolw ub*	*tw ub*	*brw ub*	*mmw result*
1a62	122	1516	27	29	13.62	37		28
1a8o	64	536	17	18	9.11	25		18
1aac	104	1316	28	33	12.29	41		29
1aba	85	886	22	24	10.13	29		23
1ail	69	631	16	16	8.07	24		17
1awd	89	1080	25	28	10.08	38		25
1b0n-006	98	981	20	21	10.58	32		21
1b67	68	559	11	12	6.61	16		12
1bbz	57	543	16	18	8.30	25		16
1bf4	63	658	17	19	7.90	26		18
1bkb	131	1485	21	23	14.53	30		22
1bkf	106	1264	25	28	11.69	36		27
1bkr	107	1340	29	35	14.40	44		30
1brf	49	412	14	14	7.01	22		15
1bx7	41	195	8	8	4.91	11		8
1c4q	67	756	19	22	9.45	31		20
1c5e	95	1148	26	28	11.06	36		26
1c75	69	683	19	21	9.88	30		20
1c9o	66	720	18	20	8.75	29		19
1cc8	70	813	21	23	9.35	32		21
1cka	57	605	16	18	8.55	27		18
1ctj	87	935	23	27	10.74	33		25
1cuk	189	2404	29	38		44		30
1czp	94	1195	26	29	11.47	38		27
1d3b	69	682	18	19	8.44	25		19
1d4t	102	1145	26	29	12.87	35		27
1dd3	128	1356	20	22	11.68	31		20
1dj7	73	743	17	18	9.66	27		18
1dp7	76	769	18	19	9.01	27		20
1e0b	60	518	15	17	8.13	24		15
1en2	69	463	12	12	7.24	17		14
1erv	101	1267	28	32	12.26	41		29
1ezg	66	541	16	18	8.83	23		18
1f9m	109	1349	29	35	14.27	45		30
1fjl	65	600	16	17	7.90	26		17
1fk5	85	823	21	22	10.76	31		22
1fpo	170	1840	22	25		31		24
1fr3	67	618	16	17	7.29	21		17
1fs1	114	1351	26	29	13.79	34		28
1fse	67	730	19	19	8.58	27		20
1g2b	62	649	18	19	8.72	28		18
1g2r	94	1109	24	28	12.17	37		25
1g3p	185	2221	30	36		45		30
1g6x	52	405	13	14	6.89	19		14
1gcq	68	742	19	21	9.36	30		20
1gef	119	1492	29	36	13.60	43		30
1gut	67	621	16	17	7.47	22		17
1hg7	66	705	18	20	8.81	29		19
1i07	59	397	11	12	5.52	15		12

graph	*vertices*	*edges*	*rw result*	*Beyß*	*boolw ub*	*tw ub*	*brw ub*	*mmw result*
1i0v	100	1207	27	30	12.21	41		27
1i27	73	747	18	19	8.78	27		19
1i2t	61	644	17	19	10.45	27		17
1ig5	75	816	21	23	7.75	33		22
1igd	61	630	16	18	6.89	25		17
1igq	54	503	14	16	6.89	23		15
1iib	103	1384	29	33	12.62	40		29
1iqz	77	839	21	23	10.00	33		21
1j75	56	558	16	18	8.51	27		17
1jhg	101	841	16	19	8.87	25		17
1jo8	58	608	17	19	8.46	27		18
1k61	60	581	17	18	8.32	26		17
1kq1	60	607	17	19	8.79	27		18
1kth	52	426	13	14	7.04	20		14
1ku3	61	585	16	17	7.53	23		17
1kw4	67	672	19	22	9.39	28		20
1l9l	70	697	18	20	9.26	29		19
1ldd	74	835	19	22	9.60	32		21
1ljo	74	789	19	21	8.88	30		20
1lkk	103	1162	24	28	11.89	34		25
1mgq	74	798	19	21	8.91	28		20
1oai	58	524	16	17	7.87	22		16
1on2	135	1527	24	25		36		24
1or7	180	1875	25	33		37		26
1plc	98	1167	25	28	11.28	35		26
1ptf	87	1137	24	27	11.21	38		28
1pwt	61	657	18	19	8.81	29		18
1qtn	87	788	18	18	9.15	24		19
1r69	63	692	19	21	9.12	30		19
1rb9	48	412	13	14	6.77	22		14
1rro	107	1300	30	33	15.36	43		30
1sem	57	570	16	18	8.32	26		17
1ubq	74	211	9	10	6.51	12		9
a280.tsp	280	788	13			14	13	13
alarm	37	65	3	3	2.58	4	4	3
barley	48	126	5	5	4.00	7	6	5
bier127.tsp	127	368	14	16		15	14	14
bcs01	48	176	8			13	12	12
bcs03	112	264	1			3	3	2
bcs04	132	1758	17				30	44
bcs05	153	1135	12			20	18	12
bcs06	420	3720	18					36
BN_0	100	300	17	20				19
BN_1	100	394	19	22				20
BN_2	100	494	22	26	17.24			24
BN_3	100	451	23	24	17.09			24
BN_4	100	574	24	28				26
BN_5	125	678	30	33				31
BN_6	125	948	34	36				35
BN_7	95	535	24	26				26

graph	*vertices*	*edges*	*rw result*	*Beyß*	*boolw ub*	*tw ub*	*brw ub*	*mmw result*
BN_8	100	420	21	23				23
BN_9	105	382	20	22	19.05			23
BN_10	85	304	16	17	12.72			18
BN_11	105	631	27	32				29
BN_12	90	481	22	24				24
BN_13	125	550	25	29				27
BN_14	115	456	22	26				24
BN_15	120	673	29	33				31
celar02	100	311	3	5	3.32	10		7
celar03	200	721	9	10		14		11
celar04	340	1009	10			16		13
celar05	200	681	10	12		15		12
celar06	100	350	4	5	3.81	11		7
celar07	200	817	10	12		16		13
celar08	458	1655	10	22		16		13
ch130.tsp	130	377	10	12		12	10	10
ch150.tsp	150	432	12	15		15	12	12
d198.tsp	198	571	12	15		14	12	12
david	87	812	9	10	5.32	13		9
eil51.tsp	51	140	7	7	5.78	9	8	7
eil76.tsp	76	215	10	10	7.17		10	10
eil101.tsp	101	290	10	12			10	10
fpsol2.i.1	496	11654	4	8		66		75
fpsol2.i.2	451	8691	4			31		32
fpsol2.i.3	425	8688	4			31		33
games120	120	1276	25	30		33		27
graph01	100	358	17	19	14.61	24		19
graph02	200	709	34	40		48		37
graph03	100	340	17	18	13.29	20		19
graph04	200	734	37	45		51		41
graph05	100	416	16	19	13.70	24		19
graph06	200	843	36	44		52		41
graph07	200	843	37	44		52		40
huck	74	602	4	4	2.81	10		6
inithx.i.1	864	18707	7			56		51
inithx.i.2	645	13979	4			31		33
inithx.i.3	621	13969	4			31		46
jean	80	508	5	6	3.91	9		6
knights8_8	64	168	14	15	11.06	16		14
kroA100.tsp	100	285	8	11		10	9	8
kroA150.tsp	150	432	10	12		12	11	10
kroA200.tsp	200	586	11	22		14	11	11
kroB100.tsp	100	284	9	11		11	9	9
kroB150.tsp	150	436	10	15		12	10	10
kroB200.tsp	200	580	12	18		14	12	12
kroC100.tsp	100	286	9	10		10	9	9
kroE100.tsp	100	283	8	9		9	8	8
le450_5a	450	5714	150			307		150
le450_5b	450	5734	150			309		150
le450_5c	450	9803	150			315		150

graph	*vertices*	*edges*	*rw result*	*Beyß*	*boolw ub*	*tw ub*	*brw ub*	*mmw result*
le450_5d	450	9757	150			303		150
le450_15a	450	8168	142			296		144
le450_15b	450	8169	143			289		145
le450_15c	450	16680	150			372		150
le450_15d	450	16750	150			371		150
le450_25a	450	8260	120			255		134
le450_25b	450	8263	109			251		137
le450_25c	450	17343	149			349		150
le450_25d	450	17425	150			349		150
lin105.tsp	105	292	8	10			8	8
mainuk	48	84	5	5	3.58	7		6
mildew	35	80	3	3	3.00	4	4	4
miles250	128	774	5	7	4.95	9		7
miles500	128	2340	14	15	9.42	22		17
miles750	128	4226	21	24		36		28
miles1000	128	6432	24	28		49		33
miles1500	128	10396	14	15	4.86	77		43
mulsol.i.1	197	3925	3	3	4.00	50		46
mulsol.i.2	188	3885	3	6	4.81	32		23
mulsol.i.3	184	3916	3	5	4.95	32		23
mulsol.i.4	185	3946	3	4	4.81	32		23
mulsol.i.5	186	3973	3	5	4.95	31		23
munin1	189	366	7	12		11		10
myciel5	47	236	11	11	8.12	19	19	13
myciel6	95	755	21	24	13.40	35	36	25
myciel7	191	2360	41	54		54		49
oesoca	39	67	3	3	2.32	3		3
oesoca+	67	208	6	7	4.81	11	9	8
oesoca42	42	72	3	3	2.32	3		3
p654.tsp	654	1806	10				10	10
pathfinder	109	211	5	5	3.32	6		6
pcb442.tsp	442	1286	17				17	17
pr76.tsp	76	218	9	9	7.84		10	9
pr107.tsp	107	283	6	8			6	6
pr124.tsp	124	318	8	11			8	8
pr136.tsp	136	377	10	11	6.70		10	10
pr144.tsp	144	393	9	10			9	9
pr152.tsp	152	428	8	12	6.70		8	8
pr226.tsp	226	586	7	12			7	7
pr299.tsp	299	864	11				11	11
pr439.tsp	439	1297	16				16	16
queen5_5	25	320	5	5	5.29	18	16	9
queen6_6	36	580	9	9	7.65	25	24	12
queen7_7	49	952	11	12	10.36	35		17
queen8_8	64	1456	13	14	13.16	45		22
queen8_12	96	2736	20	21	16.70	65		32
queen9_9	81	2112	16	18	17.07	58		27
queen10_10	100	2940	20	23		72		34
queen11_11	121	3960	23	27		88		41
queen12_12	144	5192	26	30		104		48

graph	*vertices*	*edges*	*rw result*	*Beyß*	*boolw ub*	*tw ub*	*brw ub*	*mmw result*
queen13_13	169	6656	29	34		122		57
queen14_14	196	8372	33	46		141		66
queen15_15	225	10360	33	51		163		75
queen16_16	256	12640	37	42		186		86
rat99.tsp	99	279	8	9	6.94		9	8
rat195.tsp	195	562	12	15			12	12
rd100.tsp	100	286	10	11			9	10
rd400.tsp	400	1183	18				17	18
school1	385	19095	116			188		121
sodoku	81	810	9	14	9.00	45		27
sodoku-elim1	80	898	9	13	9.47	45		27
tsp225.tsp	225	622	12	16		15	12	12
u159.tsp	159	431	10	14		12	10	10
u574.tsp	574	1708	18			24	17	18
u724.tsp	724	2117	23			26	18	22
water	32	123	6	6	4.39	9	9	7
zeroin.i.1	211	4100	4	6	3.70	50		42
zeroin.i.2	211	3541	3	5	5.39	32		29
zeroin.i.3	206	3540	3	7	5.39	32		28

Table B.1 *Benchmarking results for small and medium graphs from TreewithLIB.*

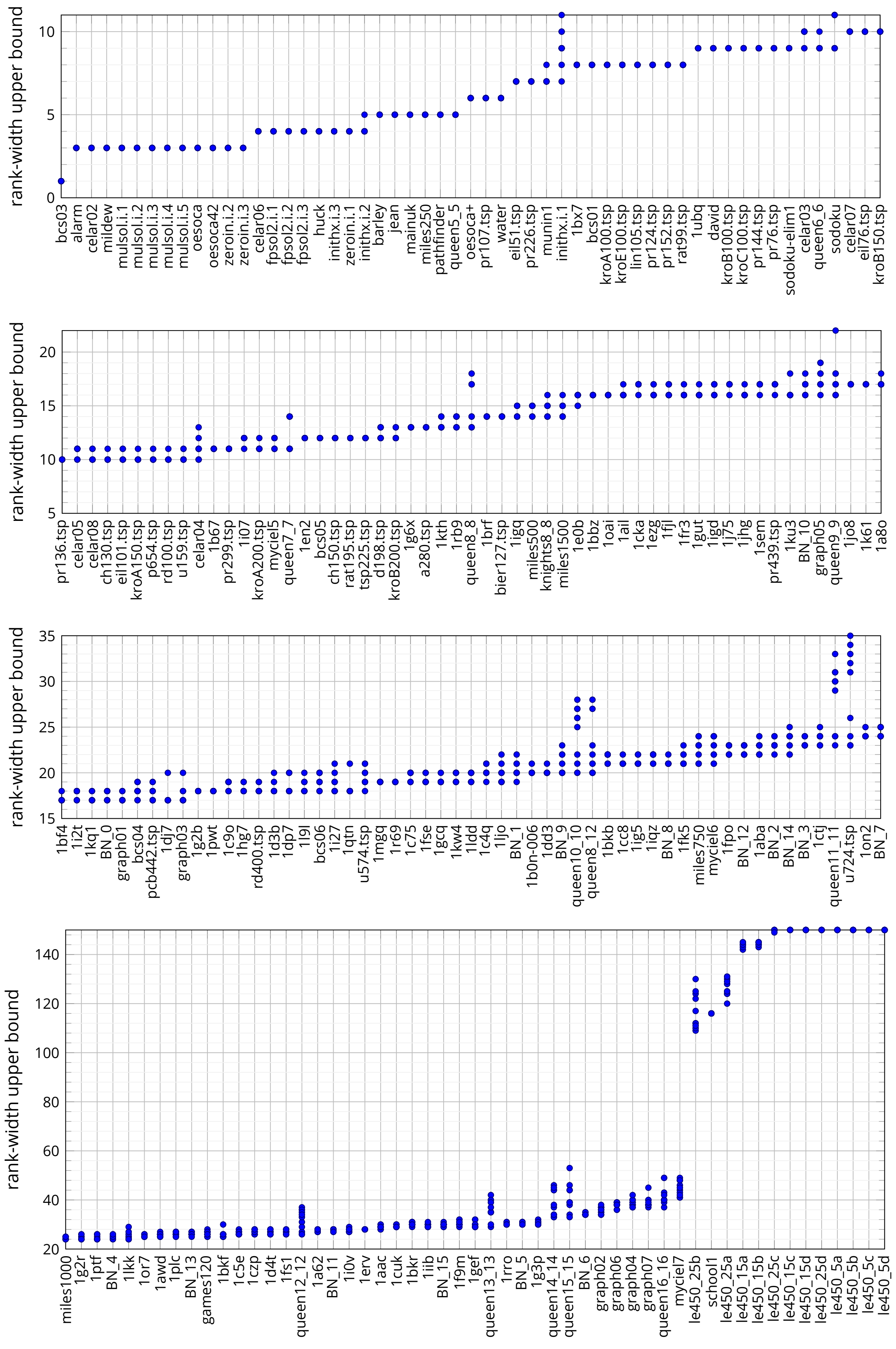


Figure B.1 *Overview of all achieved rank-width upper bounds during the 10 runs for the graphs of Table B.1.*

B.1.2 TreewidthLIB large

graph	*vertices*	*edges*	*rw result*	*tw ub*	*brw ub*	*mmw result*
d1291.tsp	1291	3845	53	37	29	48
d1655.tsp	1655	4890	85		29	86
d2103.tsp	2103	6290	143	36	29	114
pcb1173.tsp	1173	3501	58		25	46
pr1002.tsp	1002	2972	32		21	27
rl1323.tsp	1323	3950	72		22	69
u1432.tsp	1432	4204	74		32	63
u1817.tsp	1817	5386	104		28	83
u2319.tsp	2319	6869	151	56	44	140
vm1084.tsp	1084	2869	57	23	15	51
vm1748.tsp	1748	4784	106	33	22	91

Table B.2 *Benchmarking results for large graphs from TreewithLIB.*

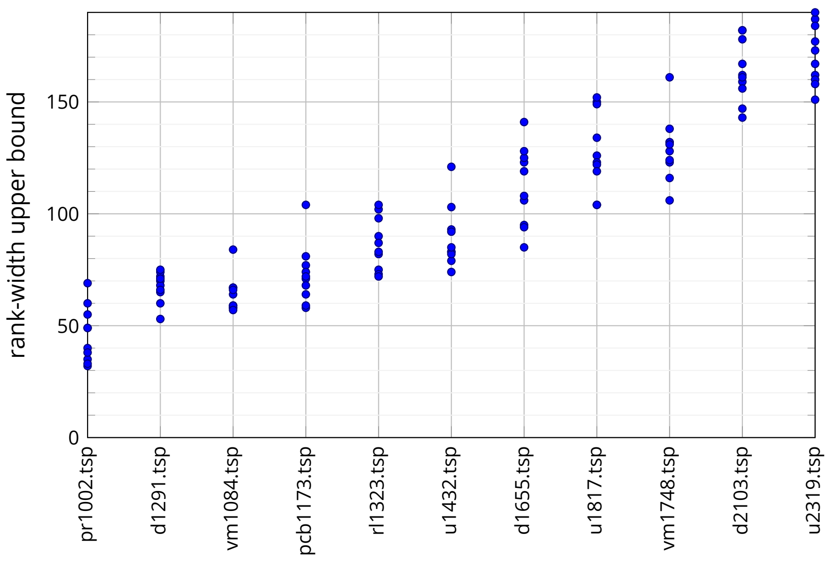


Figure B.2 *Overview of all achieved rank-width upper bounds during the 10 runs for the graphs of Table B.2.*

B.2 Directed graphs

Benchmarking results for a number of directed graphs found on GitHub [1]. For the rank-width upper bound (rw ub) and maximum matching-width upper bound (mmw ub) the graphs were converted from directed to undirected.

graph	*vertices*	*arcs*	$\mathbb{F}_4$-*rw ub*	*rw ub*	*mmw ub*
mm4a	170	454	16	16	17
mm9a	631	1182	41	33	37
mm9b	777	1452	62	56	57
mult16a	293	582	6	6	6
mult16b	333	545	5	5	5
mult32a	565	1142	7	6	6
s27	55	87	6	6	6
s208	83	119	5	5	7
s344	274	388	8	8	8
s349	278	395	8	8	8
s382	273	438	14	14	14
s400	287	462	14	14	14
s420	104	178	7	7	8
s444	315	503	14	14	14
s526	318	576	15	15	15
s526n	292	560	13	13	15
s641	477	612	12	12	14
s713	515	688	12	12	14

Table B.3 *Benchmarking results for directed graphs.*

[1] `https://github.com/alidasdan/graph-benchmarks`