Fast Algorithm for Rank-Width

Martin Beyß

RWTH Aachen University

Abstract. Inspired by the heuristic algorithm for boolean-width by Telle et. al. [1], we develop a heuristic algorithm for rank-width. We compare results on graphs of practical relevance to the established bounds of boolean-width. While the width of most graphs is lower than the known values for tree-width, it turns out that the boolean-width heuristic is able to find decompositions of significantly lower width. In a second step we therefore present a further algorithm that can decide if for a graph G and a value k exists a rank-decomposition of width lower than k. This enables to show that boolean-width is in fact lower than or equal to rank-width on many of the investigated graphs.

1 Introduction

Many interesting problems on graphs like TSP or Hamiltonian path are *NP*-complete. Thus, there is no fast way to solve them in general unless of course $P = NP$. Width parameters of graphs like tree-width, clique-width or rank-width can be used to construct fixed parameter tractable (*FPT*) algorithms for many of these problems. This means if the input graphs are restricted to have width at most k, a solution can be calculated in time at most $f(k) \cdot p(n)$ where f is a computable function, n the size of the graph and p a polynomial independent of k. For small k this provides a realistic chance of solving these problems even on large graphs. Many of those algorithms use dynamic programming and need a decomposition of the graph. Hence, practical methods of calculating decompositions of low width are of crucial importance.

A major result is Courcelles theorem [2] which states that every graph problem that is expressible as a MSO_2 formula can be decided in linear time for graphs of fixed tree-width. A similar theorem exists for rank-width and clique-width and a formula in MSO_1 [3]. Recent research shows that these theorems are not just of theoretical interest but also of practical applicability [4]. Accordingly, an algorithm that is able to calculate rank-decompositions gives a possibility to solve many hard problems on graphs. As [5] shows, there are several algorithms that can compute reasonably small tree-decompositions, but tree-width is only low for sparse graphs. Consequently, for dense graphs other width-measures have to be considered. An overview of different width-measures other than tree-width and their algorithmic applications can be found, e.g. in [6].

Although rank-width has been extensively researched over the last years, there are only few practical results. For example in [7], Oum proposed an $\mathcal{O}(|V|^3)$ algorithm that for a fixed k either returns a rank-decomposition of width at

A. Kučera et al. (Eds.): MEMICS 2012, LNCS 7721, pp. 82–93, 2013.

most $3k-1$ or confirms that the rank-width is larger than k. Nevertheless, no implementation is known and actual rank-decompositions only exist for very few graphs.

In [1] Hvidevold et. al. presented a novel approach to find decompositions of low boolean-width even on dense graphs. We can reuse their work and develop a heuristic algorithm to calculate rank-decompositions. Moreover, our algorithm is able to build rank-decompositions on graphs for which no bounds of other width-measures are known in particular many graphs where the boolean-width heuristic failed. This reveals the main weakness of boolean-width for practical application. Calculating the boolean-width of a given decomposition is costly especially in comparison with rank-width.

As an extension, an additional algorithm is implemented that decides for a given k if a graph has rank-width less than k. This allows the calculation of lower bounds for rank-width. These lower bounds can provide a quality estimation for the heuristic algorithm. Furthermore, we are able to compare results for boolean-width and rank-width on graphs from real life application, i.e. treewidthLIB [8].

2 Preliminaries

Graphs in this work are simple, undirected and loop free. A graph G is a tuple (V, E) where V is a set and $E \subset [V]^2$ and $[V]^2$ is the set of all two-element subsets of V. An element $v \in V$ is called a *vertex*, $e \in E$ is called *edge*. Instead of writing $v \in V$, we may write $v \in G$; the same goes for $e \in E$. As a shorthand we write $|G|$ for $|V|$ and $\|G\|$ for $|E|$. For $G = (V, E)$ we define functions $V(G) = V$ and $E(G) = E$, which assign a graph to its vertex and edge sets.

For a graph $G = (V, E)$ and a set $V' \subset V$ we define the *induced subgraph* as $G[V'] = (V', E')$ where $E' \subset E$ so that $\{v_1, v_2\} \in E'$ if $v_1, v_2 \in V'$. For a Tree $T = (V_T, E_T)$, $n \in V_T$ is called a *node*.

Tree-width[9] is most likely the best studied and understood width measure for graphs. It measures how similar a graph is to a tree. Hence, a tree has tree-width 1 while a complete graph with n vertices has tree-width $n-1$.

Definition 1. *Let $G = (V, E)$ be a graph, $T = (V_T, E_T)$ a tree and $\mathcal{V} : V_T \to 2^V,\ t \mapsto V_t$ a function. We call $(T, \mathcal{V})$ a* tree-decomposition *of G if it fulfils the following three conditions*

1. $V = \bigcup_{t \in V_T} \mathcal{V}(t)$
2. *For each $e = \{u, v\},\ e \in E$ there is a node $t \in V_T$ so that $u, v \in \mathcal{V}(t)$*
3. *For nodes $t_1, t_2, t_3 \in V_T$ where t_2 lies on a path from t_1 to t_3 there is always $\mathcal{V}(t_1) \cap \mathcal{V}(t_3) \subseteq \mathcal{V}(t_2)$*

The width *of a tree-decomposition $(T, \mathcal{V})$ is $max\{|\mathcal{V}(t)| - 1\ ;\ t \in V_T\}$. The lowest width of all possible tree-decompositions is the* tree-width *of G or $tw(G)$.*

Branch-width is a width measure that was introduced in [10]. Since its invention it has been generalized, and here the definition of [11] is used and extended to suit our needs.

Definition 2. *Let V be a finite set and $f : 2^V \to \mathbb{R}$ a function. If for any set $X \subseteq V$ f has the property that $f(X) = f(V \setminus X)$, it is* symmetric.

A subcubic *tree is a tree in which every node has degree at most three. If T is a subcubic tree and $L : V \to \{v \ ; \ v \text{ is a leaf in } T\}$ a surjective function, then (T, L) is called a* partial branch-decomposition *of f. If f is bijective, (T, L) is a* total branch-decomposition *of f.*

For an edge $e \in T$, $T \setminus \{e\}$ induces a partition (X, Y) of the leaves of T. The value of $f(L^{-1}(X))$ is the width *of the edge e of the partial branch-decomposition (T, L). The maximum width over all edges of T is the* width *of the partial branch-decomposition (T, L). The minimum width over all possible total branch-decompositions of f is called the* branch-width *of f, denoted by $bw(f)$.*

If $f(X) > f(Y)$, we say X is wider *than Y or analogously, Y is* narrower *than Y. Consequently, that term can be applied to branch-decompositions.*

Rank-width[11] is a recent width measure which is built on branch-width.

Definition 3. *Let $G = (V, E)$ be a simple, connected, undirected graph without loops and M its adjacency matrix over $GF(2)$. By M_Y^X we denote the submatrix of M with rows in X and columns in Y, for $X, Y \subseteq V$. For $X \subset V$ we define the* cut-rank *as $cutrk(X) = rk(M_{\overline{X}}^X)$ where rk is the standard matrix rank and $\overline{X}$ is the set complement of X in V.*

Obviously, the cut-rank function is symmetric, for that reason we can define the branch-width of the cut-rank function of a graph G as the rank-width *of G, denoted as $rw(G)$. Analogously, the term* (partial) rank-decomposition *can be defined.*

Like rank-width, boolean-width[12] is the branch-width of a special function, in this case the *cut-bool* function.

Definition 4. *Let $G = (V, E)$ be a graph and let $N(a)$ be the set of vertices adjacent to $a \in G$. For $X \subset V$ we define $UN(X)$ to be the* union of neighbourhoods *of subsets of X in $\overline{X}$.*

$$UN(X) = \{S \subseteq \overline{X} \ ; \ \exists \ A \subseteq X \wedge S = \overline{X} \cap \bigcup_{a \in A} N(a)\} \tag{1}$$

With this the cut-bool *function can be defined.*

$$cutbool : 2^V \to \mathbb{R}, \ X \mapsto \log_2 |UN(X)| \tag{2}$$

The branch-width of the cut-bool function of a graph G is called boolean-width *of G, denoted by $boolw(G)$.*

Remark 1. The set of $GF(2)$-sums of neighbourhoods of subsets of X in $\overline{X}$ is called $SN(X)$.

$$SN(X) = \{S \subseteq \overline{X} \ ; \ \exists \ A \subseteq X \wedge S = \overline{X} \cap \bigwedge_{a \in A} N(a)\} \tag{3}$$

We can easily see ([12])that $cutrk(X) = \log_2 |SN(X)|$, which shows that boolean-width and rank-width are quite closely related.

3 An Upper Bound Algorithm

This Section describes a heuristic algorithm for rank-width, which is an improved version of the boolean-width algorithm presented in [1]. Adapting their algorithm for rank-width is easily possible as both width measures are closely related branch-widths (c.f. Remark 1). However, this algorithm does not carry some of the limitations of its boolean-width variant because the cut-rank function is much easier and faster to calculate than the cut-bool function.

3.1 Overview

The goal of this algorithm is to heuristically find a narrow total rank-decomposition for a given graph. At the beginning an initial rank-decomposition is calculated, and then attempts to improve it are made. In a first step we assume that there already exists a total rank-decomposition $R = (T, L)$.

Algorithm 1. Main loop

Input: a graph $G = (V, E)$
Output: a total decomposition $R = (T, L)$ of G
1: Let T be an empty tree and $L : \emptyset \to \emptyset$
2: $best \leftarrow \infty$
3: $R \leftarrow (T, L)$
4: **while** algorithm should keep running **do**
5: $e = (n_1, n_2) \leftarrow$ first edge of R
6: ImproveSubtree(n_1)
7: ImproveSubtree(n_2)
8: **if** R is total **then**
9: $R_{opt} \leftarrow R$ ▷ new best decomposition
10: $best \leftarrow$ Width(R)
11: **else**
12: $R \leftarrow R_{opt}$ ▷ reset decomp.
13: **end if**
14: **end while**
15: **return** R_{opt}

Consequently, we skip the initialisation and start at line 4 of Algorithm 1. Inside the while loop it is tried to improve the given rank-decomposition. ImproveSubtree returns a total decomposition if and only if an improvement could be made. Accordingly, the next steps depend on whether R is total or not. If a better decomposition is found, it is saved along with its width (line 9). Otherwise R is reset. This is repeated for a certain amount of time.

For initialisation an empty decomposition is created. This has to be considered for the ImproveSubtree routine. The first rank-decomposition is calculated greedily while later calculations use a mix of greedy and random decisions.

Algorithm 2. ImproveSubtree

Input: a subtree rooted at n
Output:
1: **if** n is a leaf **then**
2: $(X, Y) \leftarrow$ Split(n)
3: **else**
4: $(X, Y) \leftarrow$ RandomSwap(n)
5: **end if**
6: **if** $max(cutrk(X), cutrk(Y)) < best$ **then**
7: remove subtrees rooted at n (if any)
8: add children n_1, n_2 to n
9: $\forall x \in X : L(x) \leftarrow n_1, \forall y \in Y : L(y) \leftarrow n_2$
10: **end if**
11: **if** n has children n_1 and n_2 of width $< best$ **then**
12: ImproveSubtree(n_1), ImproveSubtree(n_2)
13: **end if**

Algorithm 2 shows the functionality of the ImproveSubtree routine. If the root of the subtree is an internal node, a good split is already known and we try to randomly improve it. Otherwise a good split has to be calculated greedily. Assume that the node n has some children, and thus the RandomSwap routine is called. In Algorithm 2.6 it is now checked if the new cuts are narrow enough. If this is the case, the new split is assigned to the children n_1 and n_2. Thereby the subtrees rooted at them become meaningless and are removed. Then in line 12 the ImproveSubtree routine is called for the new split. In case no sufficiently narrow split could be found, the old one is reused, provided that it is narrow enough.

If the subtree consists just of the node n, a whole new partition has to be calculated with the call to the Split function in line 2. In case a sufficiently narrow one is found, two new leaves are added to n and labelled with the split. Then for these nodes ImproveSubtree is called. If the ImproveSubtree function is called with the initial rank-decomposition, there is always a new split calculated because it does not exist an old one that could be altered. As already mentioned, the first run always returns a total rank-decomposition. Later this may not always be the case. If Split is not able to find a good partition, or the old one is not good enough and RandomSwap does not find a better one, the node n does not have any children. Hence, ImproveSubtree stops there and returns a partial rank-decomposition.

Splitting a leaf (Split) is done in a greedy way. We start with the partition $(X, Y) = (\emptyset, L^{-1}(n))$ and then greedily move on element at a time from Y to X. To be precise we exchange that $y \in Y$ that leads to a minimal $max\{cutrk(X \cup \{y\}), cutrk(Y \setminus \{y\})\}$. If there are several such y we chose one randomly. In the end, we pick that pair (X, Y) so that $max\{cutrk(X), cutrk(Y)\}$ becomes minimal. Again, there may be more than one partition with equally low width and we have to chose one randomly

There are some constraints for the partition (X, Y), namely X and Y are not allowed to be too small, i.e. $\geq c \cdot \left|L^{-1}(n)\right|$ (and ≥ 1 of course). Later in this work c is referred to as the split factor.

Swapping labels (RandomSwap) is done if the node n has children n_1, n_2. Basically, there is an exchange of elements, i.e. n_1 gets labels from n_2 and n_2 gets labels from n_1. Let X be $L^{-1}(n_1)$ and $Y = L^{-1}(n_2)$. Then the random subsets $X' \subset X$ and $Y' \subset Y$ are moved to the other set. The subsets have some size constraints so that the partition (X, Y) still fulfils the size constraints of the Split function.

Pseudo code implementations of Split and RandomSwap are not shown here for conciseness. Moreover, they are extremely similar to those in [1].

3.2 Results and Discussion

To get real life results, graphs from TreewidthLIB [8] are investigated. TreewidthLIB is a collection of 710 graphs from many different fields like computanional biology, probabilistic networks, TSP instances and more. Taking these graphs also enables to compare results for boolean-width calculated in [1], tree-width and rank-width on the same graph. We only use graphs that have between 25 and 256 nodes. The algorithm is able to handle much bigger graphs, but only some are tried, mainly because the algorithm that is presented in Section 4 only works on graphs up to that size. Moreover, results for boolean-width were only found for smaller graphs, so a comparison would not be possible.

For many graphs there exist preprocessed versions for tree-width. These are only used if the original graph is too big. That reduction leaves 193 graphs. For 114 of them results for boolean-width exist, for 167 an upper-bound for tree-width is known.

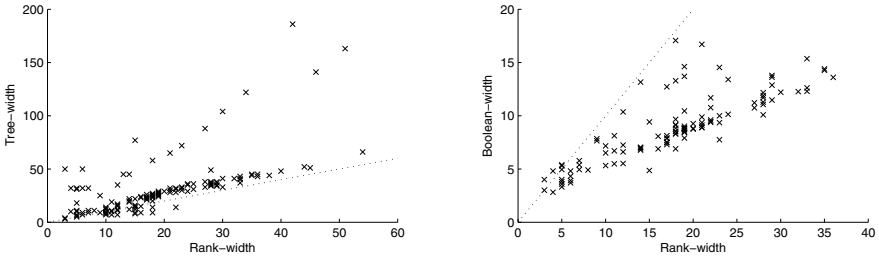
For every graph several runs with different configurations were made and we use the best result that any of this runs could produce. In general, a low split-factor and a high amount of greedy choices lead to better results.

Figure 1 shows a comparison of our results with known upper bounds for tree-width and boolean-width. The dotted line in both Figures marks the equality of both parameters.

In theory, rank-width can be as high as 2^k on a graph with boolean-width k [12]. While rank-width is sometimes almost equal to or better than boolean-width, in most of the cases boolean-width is significantly lower. The ratio between boolean-width and rank-width is between 1.33 and 0.32 with an average of 0.57.

In [13] a tight bound connecting tree-width and rank-width is established: $rw(G) \leq tw(G) - 1$. This bound is beaten by most of the results the algorithm could produce. On average the known upper bound for $tw(G) - 1$ is 70% higher than our bound for rank-width. We are also able to find rank-decompositions for graphs on which no bounds for tree-width are known.

Some results are shown in Table 1. The first graph `1bkf` is from the field of computational biology. The graph `miles1500` is converted from the stanford graph base [14]. The `myciel6` graph is a Mycielskian graph from the second



(a) Tree-width (b) Boolean-width

Fig. 1. Comparison of the results with known upper bounds for tree-width and boolean-width

Table 1. Selected results comparing upper bounds for rank-width, boolean-width and tree-width

Graph G	$\|G\|$	$\|\|G\|\|$	$rw(G)$	$boolw(G)$	$tw(G)$
`1bkf`	106	1264	28	11.69	36
`celar08`	458	1655	22	N/A	16
`eil51.tsp`	51	140	7	5.78	9
`fpsol2.i.1`	496	11654	8	N/A	66
`miles1500`	128	5198	15	4.86	77
`mulsol.i.1`	197	3925	3	4.00	50
`myciel6`	95	755	24	13.40	35
`queen7_7`	49	952	12	10.36	35

DIMACS implementation challange [15]. This is also the origin of `fpsol2.i.1` and `mulsol.i.1` which are based on register allocation of real code. A Delaunay triangulation of a travelling salesperson problem is the basis for the `eil51.tsp` graph. The `queen7_7` graph is the graph for a n-queens problem [16]. Finally, `celar08` is a frequency assignment instance [17].

The two examples for a bigger graph, `celar08` and `fpsol2.i.1`, show that the algorithm can also work successfully on bigger graphs.

We used a standard Desktop Computer with a 2.5 GHz Phenom II processor and 16GB of Ram. The algorithm ran until it could not make an improvement for 500 iterations in a row but at most 300 seconds. The actual runtime is in most cases significantly lower. Hvidevold et al. obviously allowed much longer runtimes (c.f. Table 1 in [1]). For `queen8_12` they showed a boolean-width bound of 16.7 in 3055s, whereas we could find a rank-decomposition of width 21 in 69s on that graph. Although they state that they did not aim for "fast benchmark results", this difference is definitely noteworthy.

4 A Lower Bound Algorithm

In order to evaluate the quality of the results in Section 3, one has to know the actual rank-width of the investigated graphs. As it is not known, the algorithm

that is presented here tries to calculate it, or to be more precise, it decides if for a given k, there exists a rank-decomposition narrower than k.

The next Section provides an overview on how the algorithm works in principle, before the results are presented in Section 4.2.

4.1 Overview

The algorithm tries to calculate a lower bound by enumerating all possible rank-decompositions for an induced subgraph, then growing the graph and recalculating the new rank-decompositions on the basis of the ones built in the last step. As it is given a maximal width k, which no rank-decomposition may reach, those with a higher width can be excluded. Thereby the search space can be drastically reduced. If at some point no decomposition has a low enough width, the algorithm is stopped because the bound k is a lower bound for the rank-width of the given graph. This is possible because the rank-width of a graph is at most as high as the rank-width of any of its induced subgraphs.

A problem that arises is the very quickly growing number of different rank-decompositions. As there have to be at least $2u$ vertices in a rank-decomposition before any cut possibly reaches the width of the upper bound u, no rank-decomposition can be discarded before they have $2u$ vertices. To drastically reduce the number of rank-decompositions that have to be considered, we only use partial rank-decompositions with exactly two leaves.

Algorithm 3 depicts this in pseudo code. A rank-decomposition that contains only two leaves defines exactly one cut. Therefore, we only store one set per rank-decomposition. When we grow the graph by a vertex v in line 4, two new possible rank-decompositions R_{new} and R are created. One with v in the set and one without it. The set R stays unchanged, for that reason it is important to adapt the WIDTH function in line 7.

Algorithm 3. Main loop

Input: set of vertices V

1: $v_1 \in V$, $V \leftarrow V \setminus \{v_1\}$
2: $R_{init} \leftarrow \{v_1\}$
3: $\mathcal{R} \leftarrow \{R_{init}\}$, $\mathcal{R}' \leftarrow \emptyset$
4: **for all** $v \in V$ **do**
5: **while** $\mathcal{R} \neq \emptyset$ **do**
6: $R \in \mathcal{R}$
7: **if** WIDTH$(R) < k$ **then** ▷ WIDTH function applies to current subgraph
8: $R_{new} \leftarrow R \cup \{v\}$
9: $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{R_{new}, R\}$
10: **end if**
11: $\mathcal{R} \leftarrow \mathcal{R} \setminus \{R\}$
12: **end while**
13: $\mathcal{R} \leftarrow \mathcal{R}'$, $\mathcal{R}' \leftarrow \emptyset$
14: **end for**

The graph is grown until either all vertices in V have been inserted or $\mathcal{R}$ is empty. If $\mathcal{R}$ is empty, no rank-decomposition has a width lower than k. Otherwise, the algorithm can not make a decision.

Obviously, there are always decompositions narrower than k in particular all that have a set of size smaller than k. We can show that if a graph has n vertices only those two-leafed rank-decompositions with at least $\frac{1}{3}n$ labels in each leaf represent some unique total rank-decompositions. We can therefore safely ignore them but we have to keep them for the next step of the algorithm, because by adding labels to the right leaf it may fulfil the size constraint.

4.2 Results and Discussion

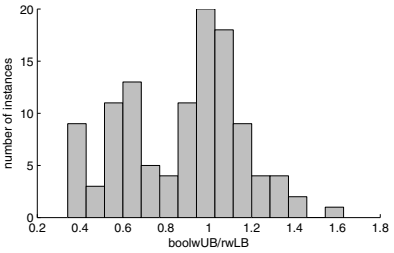
As in Section 3.2, the algorithm was checked against graphs from the treewidth-LIB. For some graphs no upper bound could be found in the time we provided. Contrary to that, we were capable of finding very good lower bounds in only a few minutes on other graphs. In some rare cases, we could even show that the known rank-decomposition is optimal. Surprisingly, we were able to find many lower bounds for rank-width which are above the upper bound for boolean-width.

In total, a lower bound could be found for 179 of the 194 graphs, 10 of which match the upper bound. For 104 graphs there also exists an upper bound for the boolean-width. In 49 cases the upper bound for boolean-width is at most as high as the lower bound for rank-width. This is a remarkable result because it provides a practical evidence that the boolean-width of a graph is in many cases lower than its rank-width.

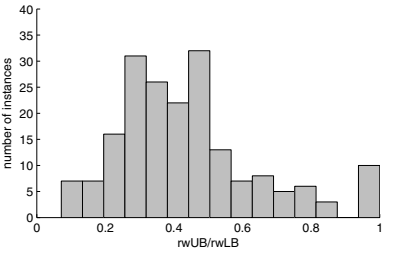
Theoretical results have already indicated this by claiming that $rw \leq 2^{boolw}$. While this bound is known to be tight, it remains unanswered how practically relevant it is especially as on the other hand $boolw \leq \frac{1}{4}rw^2$. Accordingly, it would be possible that for many instances rank-width is the lower width measure. The results in this work allow to say that at least for a part of the investigated graphs boolean-width is lower than rank-width.

Figure 2a shows the relation of boolean-width to the upper bound for rank-with in a histogram. For 104 graphs there exist an upper bound for boolean-width and a lower bound for rank-width. The median of their ratio is 0.95. So for 50% of the graphs we can safely say that its boolean-width is at worst 5% higher than its rank-width. The lower bound algorithm still suffers from limitations. Due to memory restriction we were for example not able to prove a lower bound above 10. Thus, for many graphs the optimal rank-width is significantly higher than the lower bound. Contrariwise, the values for boolean-width are the results of the first attempt to find boolean-decompositions.

The comparison to the upper bounds for rank-width is more difficult, mainly because the calculation of high lower bounds demands more resources than we are able to provide. Nevertheless, we can show the optimality of 10 rank-decompositions. The median of the ratio of lower bound and upper bound is 0.41. Moreover, for 67% of the graphs we can guarantee a 3-approximation. The algorithm by Oum [7] is able to find a 3-approximation for every graph, but there does not exist an implementation until now.



(a) Ratio of boolean-width to the lower bound for rank-width



(b) Ratio of the rank-width upper bound to the lower bound

Fig. 2. Histograms for the lower bound

Table 2. Selected results for the lower bound algorithm, UB: upper bound, LB: lower bound

graph G	$\|G\|$	$\|\|G\|\|$	rw UBs	rw LBs	$boolw$ UBs
`1bx7`	41	195	8	8	4.91
`1kw4`	67	672	22	10	9.39
`BN_12`	90	481	24	8	N/A
`celar03`	200	721	10	5	N/A
`celar06`	100	350	5	5	3.81
`graph01`	100	358	19	5	14.61
`miles750`	128	2113	24	4	N/A
`mulsol.i.2`	188	3885	6	N/A	4.81
`myciel7`	191	2360	54	7	N/A

Besides the mentioned limitations, we are able to calculate many good lower bounds and to find some astonishing results. The results should however not be interpreted as the best achievable. Rather, they merely show a small part of the possibilities.

Table 2 shows a selection of graphs with very different results for the lower bound algorithm. The first two graphs, `1bx7` and `1kw4`, are from the field of computanional biology. `BN_12` is a Bayesian network from evaluation of probabilistic inference systems [18]. Both `celar` graphs as well as `graph01` are frequency assignment instances from the EUCLID CALMA project [17]. Finally, the `mulsol.i.2` graph is a colouring problem generated from a register allocation problem based on real code from the second DIMACS challenge [15].

It was not tried to check if 2 is a lower bound of any graph, as this is only the case if a graph is distance hereditary [19]. This could be checked much easier and does most likely not apply to any of the available graphs. Thus, we can safely assume a lower bound of 2.

5 Conclusion and Outlook

A heuristic algorithm for the calculation of rank-decompositions was developed and tested against multiple graphs of TreewidthLIB. The algorithm is able to find rank-decompositions in a fast way. Unfortunately, their width is only low for a few graphs and in most cases significantly worse than the known bounds for boolean-width. As it was not clear if this is caused by the high rank-width of these graphs or by a bad result of the algorithm, a second algorithm was developed, which is able to decide if a graph has a rank-decomposition of width lower than k. Runtime and memory usage increase significantly with k, therefore tight bounds could not be found in most cases. However, we were often able to push it near or even above the known boolean-width. This evidence suggests that boolean-width is on graphs from real life application in fact a better, i.e. lower parameter.

Both algorithms use a rather simple approach and are presumably the first practical algorithms for upper and lower bounds of rank-width. Nevertheless, the results are to some extent exceptional, e.g. a rank-decomposition of width 8 on the large `fpsol2.i.1` graph or rank-width exactly 5 on the graph `celar06` .

The results presented in this work encourage further research on boolean-width. It would be of particular interest to find a fast and memory-efficient way to calculate $|UN(X)|$ (see equation 1). Both algorithms which we developed could then be adapted for boolean-width. Concerning the heuristic algorithm, even better values would be reachable, assuming that changing the parameters of the algorithm has a similar effect as for rank-width. Furthermore, a possibility to calculate lower bounds would be at hand.

Apart from that, this results also suggests further theoretical work as boolean-width seems to outperform the known width-measures. A strong theoretical background and access to good decompositions will enable to practically solve many hard problems on graphs.

References

1. Hvidevold, E.M., Sharmin, S., Telle, J.A., Vatshelle, M.: Finding Good Decompositions for Dynamic Programming on Dense Graphs. In: Marx, D., Rossmanith, P. (eds.) IPEC 2011. LNCS, vol. 7112, pp. 219–231. Springer, Heidelberg (2012)
2. Courcelle, B.: The monadic second-order logic of graphs i. recognizable sets of finite graphs. Information and Computation, 12–75 (1990)
3. Courcelle, B., Makowsky, J., Rotics, U.: Linear time solvable optimization problems on graphs of bounded clique width. Theory of Computing Systems 33, 125–150 (1999)
4. Langer, A., Reidl, F., Rossmanith, P., Sikdar, S.: Recent progress in practical aspects of mso model-checking (in preparation, 2012)
5. Bodlaender, H.L., Koster, A.M.: Treewidth computations i. upper bounds. Information and Computation 208(3), 259–275 (2010)
6. Hliněny, P., Oum, S.I., Seese, D., Gottlob, G.: Width parameters beyond tree-width and their applications. Computer Journal, 10–1093 (2007)

7. Oum, S.I.: Approximating rank-width and clique-width quickly. ACM Trans. Algorithms 5(1), 10:1–10:20 (2008)
8. Bodlaender, H., van den Broek, J.W.: Treewidthlib: A benchmark for algorithms for treewidth and related graph problems (2004), http://www.cs.uu.nl/research/projects/treewidthlib/
9. Robertson, N., Seymour, P.: Graph minors. iii. planar tree-width. Journal of Combinatorial Theory, Series B 36(1), 49–64 (1984)
10. Robertson, N., Seymour, P.: Graph minors. x. obstructions to tree-decomposition. Journal of Combinatorial Theory, Series B 52(2), 153–190 (1991)
11. Oum, S.I., Seymour, P.: Approximating clique-width and branch-width. J. Comb. Theory Ser. B 96, 514–528 (2006)
12. Bui-Xuan, B.M., Telle, J.A., Vatshelle, M.: Boolean-width of graphs. Theoretical Computer Science 412(39), 5187–5204 (2011)
13. Oum, S.I.: Rank-width is less than or equal to branch-width. Journal of Graph Theory 57(3), 239–244 (2008)
14. Knuth, D.E.: The Stanford GraphBase: a platform for combinatorial computing. ACM, New York (1993)
15. Johnson, D.J., Trick, M.A. (eds.): Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993. American Mathematical Society, Boston (1996)
16. Bell, J., Stevens, B.: A survey of known results and research areas for n-queens. Discrete Mathematics 309(1), 1–31 (2009)
17. Rlfap, E., Eindhoven, T.U., Group, R.: Euclid calma radio link frequency assignment project technical annex t-2.3.3: Local search (1995)
18. Bilmes, J.: Uai 2006 inference evaluation results. Technical report, University of Washington, Seattle (2006)
19. Oum, S.I.: Rank-width and vertex-minors. J. Comb. Theory Ser. B 95(1), 79–100 (2005)