

# Finding Good Decompositions for Dynamic Programming on Dense Graphs

Eivind Magnus Hvidevold, Sadia Sharmin, Jan Arne Telle, and Martin Vatshelle

Department of Informatics, University of Bergen, Norway \*

**Abstract.** It is well-known that for graphs with high edge density the tree-width is always high while the clique-width can be low. Boolean-width is a new parameter that is never higher than tree-width or clique-width and can in fact be as small as logarithmic in clique-width. Boolean-width is defined using a decomposition tree by evaluating the number of neighborhoods across the resulting cuts of the graph. Several NP-hard problems can be solved efficiently by dynamic programming when given a decomposition of boolean-width  $k$ , e.g. Max Weight Independent Set in time  $O(n^2 k 2^{2k})$  and Min Weight Dominating Set in time  $O(n^2 + nk 2^{3k})$ . Finding decompositions of low boolean-width is therefore of practical interest. There is evidence that computing boolean-width is hard, while the existence of a useful approximation algorithm is still open. In this paper we introduce and study a heuristic algorithm that finds a reasonably good decomposition to be used for dynamic programming based on boolean-width. On a set of graphs of practical relevance, specifically graphs in TreewidthLIB, the best known upper bound on their tree-width is compared to the upper bound on their boolean-width given by our heuristic. For the large majority of the graphs on which we made the tests, the tree-width bound is at least twice as big as the boolean-width bound, and boolean-width compares better the higher the edge density. This means that, for problems like Dominating Set, using boolean-width should outperform dynamic programming by tree-width, at least for graphs of edge density above a certain bound. In view of the amount of previous work on heuristics for tree-width these results indicate that boolean-width could in the future outperform tree-width in practice for a large class of graphs and problems.

## 1 Introduction

Many NP-hard graph problems become polynomial-time solvable when restricted to graphs of bounded tree-width or bounded clique-width. These algorithms usually have two stages, a first stage finding a decomposition of width  $k$  of the input graph, and a second stage of dynamic programming along the decomposition. The dynamic programming is typically exponential in  $k$ , e.g. given a decomposition of tree-width  $k$  it solves Maximum Weight Independent set in time  $O(n 2^k)$  and Minimum Weight Dominating set in time  $O(n 3^k k^2)$  [23]. It is therefore important to have fast algorithms for the first stage, i.e. to find decompositions of small width. For clique-width such algorithms are not known, apart from the  $2^{OPT}$  approximation achieved through rank-width [18]. For tree-width there is an  $O(nc^{k^3})$  algorithm for finding a decomposition of tree-width  $k$ , if it exists, for some constant  $c$  [4]. This algorithm is not practical [20], but much work has been done on finding decompositions of low tree-width in practical settings, see the overviews [6,5]. The web site TreewidthLIB [22] has been established to provide a benchmark and to join the efforts of people working in experimental

---

\* Supported by the Norwegian Research Council, project PARALGO.

settings to solve graph problems using tree-width and branch-width [15,19]. This includes problems from computational biology [21,24,25], constraint satisfaction [11,14], and probabilistic networks [17]. However, tree-width and branch-width are unsuitable for non-sparse graphs, as a decomposition of tree-width or branch-width  $k$  means the graph has  $O(k^2n)$  edges. Clique-width, on the other hand, can be low for dense graphs, but so far no experimental study has been done for clique-width or similar notions. To our knowledge this paper is the first case of an experimental study on computing a notion of width that works also for non-sparse graphs.

Boolean-width is a recently introduced graph parameter motivated by algorithms [10]. It is defined by a decomposition tree that minimizes the number of different unions of neighbourhoods across resulting cuts of the graph. This decomposition is natural to solve problems where vertex sets having the same neighborhoods across the cuts can be treated as equivalent. This includes problems related to Independent Set, Dominating Set, Perfect Code, Induced  $k$ -Bounded Degree Subgraph, H-Homomorphism, H-Covering, H-Role Assignment etc [1]. Similarly to treewidth, dynamic programming algorithms to solve these problems using boolean-width employ a table at each node of the decomposition tree, to store solutions to partial problems. In contrast to treewidth, the dynamic programming for boolean-width involves a non-negligible pre-processing phase computing indices of the tables, the so-called 'representatives'. Regardless, the total runtimes are in many cases close to those for treewidth, e.g. given a decomposition of boolean-width  $k$  Max Weight Independent Set is solved in time  $O(n^2 k 2^{2k})$  and Min Weight Dominating Set in time  $O(n^2 + nk 2^{3k})$  [10]. These boolean-width-based algorithms are straightforward and have been implemented in Java, without much effort, using only the description in [10]. Let us compare dynamic programming based on tree-width versus boolean-width, to solve Independent Set and Dominating Set, with focus on exponential factors. For Independent Set the exponential factor in the runtimes are  $2^{tw}$  versus  $2^{2boolw}$ , given decompositions of treewidth  $tw$  or boolean-width  $boolw$ , and boolean-width becomes preferable when  $tw > 2boolw$ . For Dominating Set the exponential factor in the runtime is  $3^{tw}$  versus  $2^{3boolw}$  and the cutoff is a bit lower, i.e. when  $tw \geq 1.9boolw$ . Oftentimes it is memory which is the bottleneck in these dynamic programming algorithms [2], and it is worth to note that for Dominating Set the treewidth algorithms use more memory, i.e. the tables will have size  $3^{tw}$  versus  $2^{boolw+1}$ .

It is known that boolean-width is never higher than tree-width or clique-width and it can be as low as logarithmic in clique-width [10]. For example, any interval graph or permutation graph has boolean-width  $O(\log n)$  while there exist such graphs of clique-width  $\Omega(\sqrt{n})$  [3,8,13]. Also, a random graph with constant edge probability will almost surely have boolean-width  $\Theta(\log^2 n)$  [1] but linear clique-width and tree-width. While these theoretical results favor boolean-width over tree-width, the cutoff  $tw \geq 2boolw$  that we arrived at above applies when we are given a decomposition of treewidth  $tw$  or boolean-width  $boolw$ , as the output of a first stage algorithm. In this paper we give a heuristic for the first stage, taking as input a graph  $G$  and finding a decomposition of  $G$  having reasonably low boolean-width. We tried various heuristics and present the one with best performance, which is a local search algorithm where the search for new solutions is based on interweaving between greedy choices and random choices. Theoretical evidence that random choices are useful for boolean-width, at least for random graphs, comes from the analysis of [1] showing that any decomposition of a random graph is expected to be a decomposition of relatively low boolean-width. On a set of graphs of practical relevance, specifically graphs in TreewidthLIB, the best known upper bound on their tree-width is compared to the upper bound on their boolean-width given by our

heuristic. For 78% of those graphs in TreewidthLIB where both tree-width and boolean-width upper bounds were encountered, the tree-width bound is at least twice the boolean-width bound, thus meeting the  $tw \geq 2boolw$  bound mentioned above. A drawback of tree-width is that it is always high when edge density is high. In contrast, boolean-width is typically low for dense graphs and our experiments show that within reasonable time we can find decompositions witnessing this. Our results indicate that, for problems like Dominating Set, using boolean-width will outperform dynamic programming by tree-width, at least for graphs of edge density above a certain bound. In view of the amount of previous work on heuristics for tree-width we expect that further work on boolean-width heuristics will substantially increase the class of graphs for which boolean-width outperforms tree-width, also for other problems besides Independent Set and Dominating Set.

The rest of the paper is organized as follows. In Section 2 we define partial and full decomposition trees and boolean-width. In Section 3 we describe the heuristic finding a decomposition of low boolean-width. In Section 4 we describe the experimental results on graphs in TreewidthLIB, and also on small grid graphs. In Section 5 we draw some conclusions.

## 2 Boolean-width

We consider undirected graphs  $G = (V, E)$  without loops. We denote the neighborhood of a vertex  $v$  by  $N(v)$  and the union of neighborhoods of a vertex subset  $A$  by  $N(A) = \cup_{v \in A} N(v)$ . The complement of  $A \subseteq V$  is denoted by  $\bar{A} = V \setminus A$  and we call  $(A, \bar{A})$  a cut of  $G$ . A partition of a set  $S$  consists of non-empty and disjoint subsets of  $S$  whose union is  $S$ . We follow custom by referring to vertices of a graph and nodes of a tree.

**Definition 1 (Full and partial decomposition trees).** A partial decomposition tree of a graph  $G = (V, E)$  is a pair  $(T, \delta)$ , where  $T$  is a full binary tree and  $\delta$  is a mapping from the nodes of  $T$  to non-empty subsets of  $V$ , satisfying the following: if  $x$  is the root of  $T$  then  $\delta(x) = V$  and if nodes  $y$  and  $z$  of  $T$  are children of a node  $x$  then  $(\delta(y), \delta(z))$  is a partition of  $\delta(x)$ . If a subtree of  $T$  rooted at  $x$  has  $|\delta(x)|$  leaves then it is called a full decomposition subtree. If  $T$  has  $|V|$  leaves then  $(T, \delta)$  is called a full decomposition tree.

Note that in a partial decomposition tree  $(T, \delta)$  of a graph  $G$ , if  $L$  is the set of leaves of  $T$  then  $\{\delta(x) : x \in L\}$  is a partition of  $V$ . Hence in a full decomposition tree there will for each vertex  $v$  of  $G$  be a unique leaf  $x$  of  $T$  with  $\delta(x) = \{v\}$ . Likewise for each vertex  $v$  of  $G$  in a full decomposition subtree rooted at  $x$ .

**Definition 2 (Unions of neighborhoods and boolean-width).** Let  $(T, \delta)$  be a partial decomposition tree of a graph  $G$ . Let  $V(T)$  be the nodes of  $T$ . Every node  $x \in V(T)$  defines a cut  $(\delta(x), \bar{\delta(x)})$  of  $G$ . The set of unions of neighborhoods of subsets of  $A$  across the cut  $(A, \bar{A})$  is  $UN(A) = \{N(X) \cap \bar{A} : X \subseteq A\}$ . The boolean-width of  $(T, \delta)$  is

$$boolw(T, \delta) = \max_{x \in V(T)} \{\log_2 |UN(\delta(x))|\}$$

The boolean-width of a graph  $G$  is the minimum boolean-width over all its full decomposition trees  $boolw(G) = \min_{\text{full } (T, \delta) \text{ of } G} \{boolw(T, \delta)\}$ .

Note that  $UN(A)$  are the subsets of  $\bar{A}$  for which there exists an  $X \subseteq A$  with  $N(X) \cap \bar{A}$  being that subset, so we always have  $\emptyset \in UN(A)$ . It is known from boolean matrix theory

[16] that  $|UN(A)| = |UN(\overline{A})|$  and this is sometimes used by our code. Let us consider some examples. If  $|UN(A)| = 2$  then the set of edges crossing the cut  $(A, \overline{A})$  induce a complete bipartite graph. If the set of edges crossing the cut  $(A, \overline{A})$  induce a perfect matching of  $G$  then  $|UN(A)| = 2^{\lfloor V/2 \rfloor}$ . In the definition of boolean-width we take the logarithm base 2 of  $|UN(A)|$  which ensures that  $0 \leq \text{boolw}(G) \leq |V|$ . If a graph has boolean-width one then it has a full decomposition tree such that, for every cut defined by a node of the tree, the edges crossing the cut, if any, induce a complete bipartite graph. From this it follows that the graphs of boolean-width one are exactly the distance-hereditary graphs [9].

**Definition 3 (Split).** A split of a set  $P$  is a partition into two subsets  $A$  and  $B$ , with the constraint that  $\min\{|A|, |B|\} \geq \frac{1}{3}|P|$ .

### 3 Heuristic Algorithm

We present a local search heuristic that given a graph  $G$  computes a full decomposition tree of  $G$ . The search for new solutions in the space of candidate solutions is based on a fine balance between greedy choices and random choices. The heuristic, given in Algorithm 1, runs for a pre-defined length of time and then returns the best full decomposition found. Each heuristic pass iterates over all decomposition nodes of the current partial decomposition tree, including the children created by this heuristic pass. A newly created tree node always starts out as a leaf node, which  $\delta$  maps to a set of vertices of  $G$  that may be larger than one. We keep track of the best full decomposition subtrees found for each  $P \subseteq V$  encountered so far and call it  $Best(P)$ .

---

**Algorithm 1 :** Generate a full decomposition of a given graph

---

**Input:** a graph  $G$   
**Output:** a full decomposition tree  $(T, \delta)$  of  $G$   
**Step 1:** /\*Greedily generate initial full decomposition tree\*/  
Initialize  $T$  with  $V(T) = \{root\}$ ,  $\delta(root) = V$   
**while**  $\exists$  leaf  $x$  of  $T$  with  $|\delta(x)| > 1$   
     $(A, B) = \mathbf{Split}(\delta(x))$ ;  
    Add leaves  $y$  and  $z$  as children of  $x$  with  $\delta(y) = A$  and  $\delta(z) = B$   
    **for** all  $x \in V(T)$  store  $Best(\delta(x))$ , the subtree rooted at  $x$   
**Step 2:** /\*Local Search for better trees\*/  
**for** fixed amount of time **do**  
    **TryToImproveSubtree**( $root$ )  
    **if**  $(T, \delta)$  is a full decomposition tree **then**  $Best(V) = (T, \delta)$   
**return**  $Best(V)$

---

#### 3.1 Greedy Initialization

Step 1 of Algorithm 1 greedily generates a full decomposition tree, to serve as the starting tree for the local search in Step 2. The greedy initialization starts with  $T$  containing a single node  $x$  (as both root and leaf) with  $\delta(x) = V$  and repeatedly calls the **Split** subroutine until we get a full decomposition tree. The **Split**( $P$ ) subroutine returns a split  $(A, B)$  of  $P$  and

is given in Algorithm 2. Starting with  $A$  being a random half of the vertices of  $P$  (unless  $P=V$ ), it adds new vertices to  $A$  one by one in a greedy fashion while minimizing  $|UN(A)|$  and  $|UN(P \setminus A)|$ , and returns the best split found along the way complying with the split constraint. The call of **Split**( $V$ ) at the root sets the initial conditions for the later splits and for this root-case we start with  $A = \emptyset$ , rather than a random half of the vertices, to allow the full benefit of the greedy choices. The local search in **TryToImproveSubtree** will for leaves of the current tree make calls to **Split**( $P$ ) but not for  $P = V$ , since the root of  $T$  will never become a leaf and instead the **RandomSwap** subroutine described in the next subsection will be applied to the root.

---

**Algorithm 2 : Split**( $P$ )

---

**Input:** Set of vertices  $P \subseteq V$ .  
**Output:** a partition  $(A,B)$  of  $P$  s.t.  $\min\{|A|, |B|\} \geq \frac{1}{3}|P|$ .  
**if**  $P = V$  **then**  $A_1 \leftarrow \emptyset$   
**else**  $A_1 \leftarrow$  random half of the vertices in  $P$   
 $i = 1$   
**while**  $|P \setminus A_i| \geq \frac{1}{3}|P|$  **do**  
    find  $x \in P \setminus A_i$  s.t.  $\max\{\mathbf{UN}(A_i \cup \{x\}), \mathbf{UN}((P \setminus A_i) \setminus \{x\})\}$  is minimized.  
     $A_{i+1} = A_i \cup \{x\}$ .  
     $i = i + 1$ .  
**end while**  
find  $i$  such that  $\max\{\mathbf{UN}(A_i), \mathbf{UN}(P \setminus A_i)\}$  is minimized and  $|A_i| \geq \frac{1}{3}|P|$ .  
**return**  $(A_i, P \setminus A_i)$ .

---

The objective function optimized locally in **Split** is  $|UN(A)|$ , the number of unions of neighborhoods of  $A$ , which directly relates to boolean-width, see Definition 2. The computation of  $|UN(A)|$  is done in a separate subroutine called **UN**( $A$ ) given in Algorithm 3. This subroutine starts by restricting from the cut  $(A, \bar{A})$  to the subsets of vertices  $(S_1, S_2)$  having an edge going across the cut  $(A, \bar{A})$ . The list  $LN$  is used to accumulate the set  $UN(A)$  in a straightforward way. Correctness is easy to show by induction on  $|S_1|$ . Early termination of the **UN**( $A$ ) subroutine is not shown in Algorithm 3 but is done if it is determined that  $|LN|$  is too large for the cut  $(A, \bar{A})$  to be interesting.

---

**Algorithm 3 : UN**( $A$ )

---

**Input:** Set of vertices  $A \subseteq V$ .  
**Output:**  $|UN(A)|$ , the number of unions of neighborhoods of the cut  $(A, \bar{A})$   
**if**  $|UN(A)|$  has already been computed **return** the stored value  
 $S_1 = \{v \in A : \exists u \in \bar{A} \wedge (u, v) \in E\}$   
 $S_2 = \{v \in \bar{A} : \exists u \in A \wedge (u, v) \in E\}$   
 $LN \leftarrow \{\emptyset\}$  /\*neighborhood set accumulator\*/  
**for** all  $u \in S_1$  **do**  
    **for** all  $Y \in LN$  **do**  
         $X \leftarrow (N(u) \cap S_2) \cup Y$   
        **if**  $X \notin LN$  **then** add  $X$  to  $LN$   
**return** The number of elements in  $LN$

---

### 3.2 Local Search

The local search used to improve the current decomposition tree is initiated at the root of the tree  $T$ , in Step 2 of Algorithm 1. In the subroutine **TryToImproveSubtree**( $x$ ), given in Algorithm 4,  $x$  is a node of the current partial decomposition tree  $(T, \delta)$  and the goal is to improve the subtree of  $T$  rooted at  $x$ . That subroutine has four main parts.

- (1) if  $x$  leaf then find candidate for split of its subset
- (2) if  $x$  non-leaf then find candidate for swap of its two children subsets
- (3) conditionally update  $(T, \delta)$
- (4) for each child of  $x$  either use stored subtree or recurse

For (1) we use the **Split** subroutine described earlier. For (2) we use the **RandomSwap**( $A, B$ ) subroutine given in Algorithm 5 that randomly swaps vertices between  $A$  and  $B$  while complying with the split constraint. At the very onset of the local search, the current  $(T, \delta)$  is the full decomposition tree found by the greedy initialization. However, the current decomposition tree ceases to be full as soon as the split given by **RandomSwap**( $\delta(y), \delta(z)$ ) in (2) is a good one and (3) updates  $(T, \delta)$  so that  $y$  and  $z$  become leaves. If the new  $\delta(y)$  is a subset of vertices for which a full decomposition subtree has never been stored, or the stored one is not good enough, then in (4) a recursive call is made to **TryToImproveSubtree**( $y$ ), with  $y$  a leaf of the current tree. If in that recursive call the split found in (1) is not good then in (3) we will return with  $y$  a leaf of the current  $(T, \delta)$  having  $|\delta(y)| > 1$ , which explains the if-statement at the very end of Algorithm 1.

---

#### Algorithm 4 : TryToImproveSubtree( $x$ )

---

**Input:** a node  $x$  of  $T$  with  $|\delta(x)| > 1$

- (1) if  $x$  is a leaf then  $(A, B) = \text{Split}(\delta(x))$
- (2) else
  - Let  $y$  and  $z$  be the children of the node  $x$ .
  - $(A, B) = \text{RandomSwap}(\delta(y), \delta(z))$
- (3) if  $\max\{\text{UN}(A), \text{UN}(B)\} < \text{boolw}(\text{Best}(V))$ 
  - then Set  $y$  and  $z$  as new leaf children of  $x$  with  $\delta(y) = A$  and  $\delta(z) = B$
  - else if  $x$  is still a leaf then return /\* in case we came from (1) \*/
- (4) if  $\max\{\text{UN}(\delta(y)), \text{UN}(\delta(z))\} < \text{boolw}(\text{Best}(V))$  then
  - for  $w \in \{y, z\}$ 
    - if subtree for  $\delta(w)$  is stored and  $\text{boolw}(\text{Best}(V)) > \text{boolw}(\text{Best}(\delta(w)))$
    - then use root of  $\text{Best}(\delta(w))$  as  $w$ .
    - else if  $|\delta(w)| > 1$  call **TryToImproveSubtree**( $w$ )
  - if the subtree  $T_x$  rooted at  $x$  is a full subtree of  $\delta(x)$
  - then update  $\text{Best}(\delta(x))$  to  $T_x$

---

Note that the local improvements made in the local search are based on randomly swapping vertices between  $\delta(y)$  and  $\delta(z)$  for two nodes  $y$  and  $z$  with the same parent. As usual in local search, there is a fine balance to trying new splits versus sticking with old splits. The goal is to neither get stuck in local minima nor to swap so many nodes that we re-randomize completely and don't get a hill-climbing effect. Note in (4) that we store for each subset  $P$  of

vertices encountered so far the best found full decomposition subtree  $Best(P)$ . The decision of when to try new splits and when to use the old splits is tied to the boolean-width of the best subtrees, and to the upper bound on boolean-width of  $G$  given by  $Best(V)$ .

---

**Algorithm 5 : RandomSwap**( $\delta(y), \delta(z)$ )

---

**Input:**  $\delta(y), \delta(z) \subseteq V$  for sibling nodes  $y$  and  $z$  of  $T$ .  
**Output:** split  $(A, B)$  of  $\delta(y) \cup \delta(z)$ .  
Let  $x$  be the parent of  $y$  and  $z$ .  
**choose** randomly  $i$  in  $0..(|\delta(y)| - \frac{|\delta(x)|}{3})$  and  $j$  in  $0..(|\delta(z)| - \frac{|\delta(x)|}{3})$ .  
**choose** randomly  $M_i \subset \delta(y)$  and  $M_j \subset \delta(z)$  with  $|M_i| = i$  and  $|M_j| = j$ .  
 $A = (\delta(y) \setminus M_i) \cup M_j$   
 $B = (\delta(z) \setminus M_j) \cup M_i$   
**return**  $(A, B)$ .

---

### 3.3 Discussion and Implementation Details

We made our implementations in Java. Subsets of vertices are stored as bitvectors of length  $n$ , i.e. the number of vertices in the graph. We expect most of the subsets we store to be of size at least  $\frac{n}{2}$  so this is an efficient way to store subsets. We also limited the boolean-width to 31, i.e.  $|UN(A)| \leq 2^{31}$ , but none of the graphs tested reached this limit. The bottleneck is rather the memory available on our machines. Let us explain. Our implementation of Algorithm ?? uses memory proportional to  $n * |UN(A)|$  bits. Since  $|UN(A)| \leq 2^{\min(|A|, |\bar{A}|)}$  the 'boolean-width 31' becomes a bottleneck only if the graph has at least 64 vertices. In that case the implementation is handling a list of neighborhoods of size  $64 * 2^{31}$  bits which is 16 GB of memory and that is more memory than our desktop had. It is part of future research to find memory efficient methods to compute the boolean-width of a cut.

As described, we are currently storing the best full decompositions of subtrees. Since bitvectors are easy to compare they are stored in a binary search tree for quick look-up. Storing all these solutions eats up memory, and for some big graphs this is the limiting factor. In the future we will consider more advanced schemes for storing the partial solutions encountered. In particular one should throw out elements that are no longer below the upper bound.

The search for new solutions in the space of candidate solutions is based on a fine balance between greedy choices and random choices, a balance that was arrived at mainly through experimentation. This appears e.g. in the choice of letting the **Split** subroutine start with a random half of the nodes on one side before trying vertices one-by-one in the more costly greedy stage. Similarly for the fully random choice of swapping in subroutine **RandomSwap**, and in the conditional tests in (3) and (4) of **TryToImproveSubtree**.

Although not specified in the pseudocode, for small subtrees we just return an arbitrary one, since if  $|\delta(x)| \leq boolw(Best(V))$  then any full subtree at  $x$  will have boolean-width at most  $boolw(Best(V))$ . The **Split**( $P$ ) subroutine given in Algorithm 2 could be stopped as soon as a subset  $A_i$  with low  $|UN(A_i)|$  and  $|UN(P \setminus A_i)|$  values has been found. It is not clear that this is always better and currently it is not done. There are many calls of **UN**( $A$ ) for many subsets  $A$  that only differ in a few vertices. A possible improvement is to store the sets of unions of neighborhoods  $UN(A)$  and use these e.g. when computing

$UN(A \cup \{v\})$  for a single added vertex  $v$ . The  $UN(A)$  subroutine given in Algorithm 3 does not recompute known values, but otherwise it may seem naive. It forms the inner loop of the heuristic and it is the bottleneck for running on graphs with many vertices. We tried different approaches such as randomly sampling subsets to approximate  $|UN(A)|$  and exploiting a correlation between the degree of a vertex and its contribution to  $|UN(A)|$ . These tests led to only insignificant improvements so for the moment we kept the naive algorithm. There are other, similar, improvements to  $UN(A)$  that can be attempted, and although they may not asymptotically improve the running-time of the heuristic they could potentially be of big help.

The balance between trying new splits and sticking to old splits is guided by the conditional test in (3) of Algorithm 4. We did try imposing stronger conditions in order to arrive at better splits sooner, but only minor improvements were seen, and only in some cases.

The heuristic ran for a predefined amount of time for each graph but there are several ways of experimenting with the stopping criteria, for example based on the size of the input graph, or on the fraction of time since an improved tree was last found.

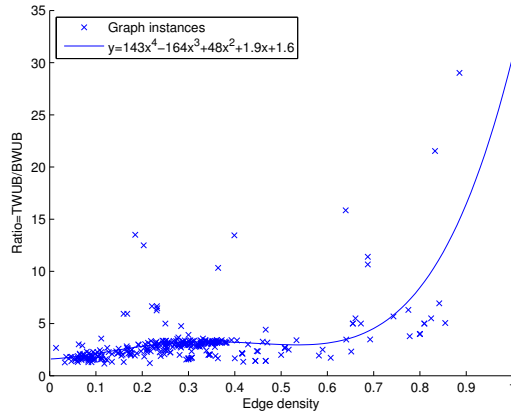
## 4 Experimental Results

All presented results have been carried out on a Linux machine with 2.33 GHz Intel Core 2Duo CPU E6550 and 2 GB RAM. Our aim was not fast benchmark results, but to explore heuristics for finding decompositions of low boolean-width. TreewidthLIB is an online repository containing a collection of 710 graphs, to be used as a benchmark for the comparison of algorithms computing treewidth. TreewidthLIB provides selected instance graphs, for which computing the treewidth is relevant, originating from applications like probabilistic networks, vertex coloring, frequency assignment and protein structures [6]. We ran our heuristic on the graphs in TreewidthLIB.

TreewidthLIB contains 710 graphs. For 482 graphs a tree-width bound is given in TreewidthLIB, and for 426 graphs we give a boolean-width bound using our heuristic. For the comparison we concentrate on the 300 graphs for which we have a bound on both tree-width and boolean-width, but let us first discuss the remaining 410 graphs. Among these 410 graphs, there are 126 having only a boolean-width bound, 182 having only a tree-width bound, and 102 having neither. The 126 graphs having only a boolean-width bound are listed in the appendix. Among the 182 graphs having only a tree-width bound there are some where the input format is flawed, containing e.g. edges between non-existing vertices, but for the majority of these graphs our heuristic simply timed out already at the greedy initialization stage. Note that for these 182 graphs, if we were given the decomposition of low tree-width  $k$ , we could easily have produced a decomposition of boolean-width at most  $k$ , using the algorithm in [1].

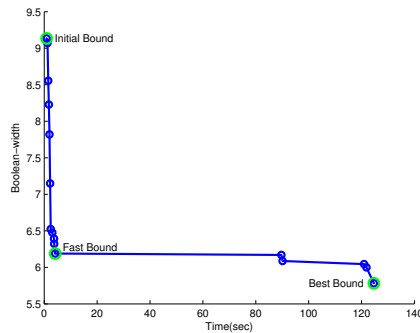
We now summarize our findings for the 300 graphs having both a tree-width bound and a boolean-width bound. Firstly, the boolean-width bound is always better than the tree-width bound, with the ratio of the tree-width bound divided by the boolean-width bound ranging from 1.15 to 29, with an average of 3.13. Not surprisingly, the ratio increased with higher edge density. In Fig.1 we have plotted this ratio against the edge density of the graphs for a total of 300 graphs. The trend line shows the growth of ratio with edge density.



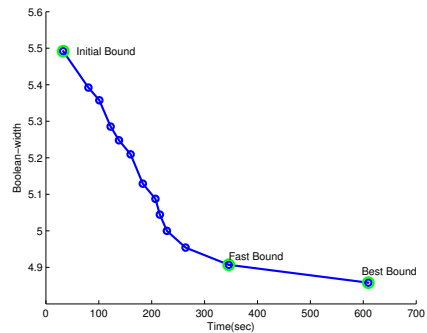


**Fig. 1.** Ratio (treewidth divided by boolean-width) versus edge density in all the 300 graphs for which heuristically computed upper bounds are known.

Our heuristic algorithm starts with greedily finding a full decomposition tree giving an Initial Bound on boolean-width and then improves this bound iteratively. In the experiments we kept track of the decrease in the boolean-width over time. In Fig. 2 and Fig. 3 the upper bounds on boolean-width, i.e. the values of  $boolw(BEST(V))$ , are shown as they decrease over time, for the two graphs called *eil51.tsp* ( $V=51$  and  $E=140$ ) and *miles1500* ( $V=128, E=5198$ ). For the graph *eil51.tsp* the Initial Bound was 9.1 after less than a second, then at the 'knee' of the curve before the improvement decays we found a Fast Bound of 6.2 after 4 seconds, and finally the Best Bound of 5.8 was found after 124 seconds. For each graph, we can likewise speak of three bounds: i) the Initial Bound given by the greedy initialization, ii) a Fast Bound found at the 'knee' of the curve, and iii) the Best Bound found possibly after a long runtime.



**Fig. 2.** Improvement of boolean-width upper bound as the local search progresses over time, for the graph *eil51.tsp* ( $V=51, E=140$ )



**Fig. 3.** Improvement of boolean-width upper bound as the local search progresses over time, for the graph *miles1500* ( $V=128, E=5198$ )

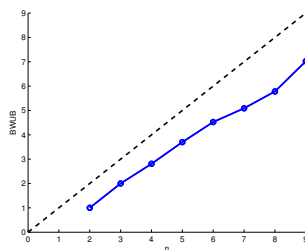
In Table 1 we summarize results for 8 selected graphs having a good variety of number of vertices  $V$ , edge density  $density$ , Time in seconds to find Initial Bound, Fast Bound, and Best Bound on boolean-width, its best known treewidth upper bound TWUB, and Ratio=TWUB/BWUB(Best Bound). The graphs are sorted by this Ratio. The *miles1500* graph is translated from the Stanford GraphBase. The *zeroin.i.1* and *mulsol.i.5* graphs originate from the 2nd DIMACS implementation challenge [12] and are generated from a register allocation problem based on real code. The *queen8\_12* also comes from the DIMACS[12] graph coloring problems and is an example of  $n$ -queens puzzle. The graph *lawd* is from the field of computational biology with each vertex representing a single side chain and each edge representing the existence of a pairwise interaction between the two side chains. The graph *celar06-wpp* is a frequency assignment instance. The graph *BN\_28* originates from Bayesian Network from evaluation of probabilistic inference systems at UAI 2006. The graph *eil51.tsp* is a Delauney triangulation of a traveling salesman problem.

**Table 1.** Results for selected graphs

Graph name	$V$	Edge $density$	Initial Bound		Fast Bound		Best Bound		TWUB	Ratio
			BWUB	Time(s)	BWUB	Time(s)	BWUB	Time(s)		
miles1500	128	0.64	5.5	32.6	4.9	345.7	4.8	609.6	77	15.85
zeroin.i.1	211	0.19	4.0	74.1	3.8	116.2	3.7	168.0	50	13.51
mulsol.i.5	186	0.23	6.4	55.3	5.4	130.0	4.9	365.2	31	6.25
queen8_12	96	0.30	16.7	3055	16.7	3055	16.7	3055	65	3.91
lawd	89	0.27	13.3	67.5	11.1	521.1	10.8	702.9	38	3.52
celar06-wpp	34	0.28	4.5	0.1	3.2	0.8	3.0	4.8	11	3.37
BN_28	24	0.18	3.3	0.02	2.3	0.05	2.0	0.3	5	2.50
eil51.tsp	51	0.11	9.1	0.9	6.2	4.1	5.8	124.6	9	1.55

#### 4.1 Grid graphs

We also ran our heuristic on graphs corresponding to the  $n \times n$  grid. These are sparse graphs having tree-width  $n$  and the upper bound we find on boolean-width is below this. See Figure 4. The boolean-width of square  $n \times n$  grids is a topic we are investigating and our current guess is that the optimal upper bound, holding for all  $n$ , is about  $0.8 * n$ . If this is correct, the value computed by the heuristic is close to optimal, which is somewhat interesting as it is our understanding that the heuristics for finding decompositions of low tree-width do not perform well on grid graphs.



**Fig. 4.** Upper-bound on boolean-width, as computed by our heuristic, for the  $n \times n$  grid, with  $n$  ranging from 2 to 9. Tree-width is given by the dotted line  $x = y$ .

## 5 Conclusion

We presented the first experimental study for finding decompositions useful also for non-sparse graphs, based on the boolean-width parameter. Experiments with the graphs in Treewidth-LIB show the strength of boolean-width versus tree-width, in a practical setting, in particular for graphs of edge density above a certain value. For more examples of real-world graphs of high edge density and high tree-width we could also look beyond the TreewidthLIB library. There are a number of open problems related to boolean-width heuristics. Firstly, we need a fast heuristic that directly constructs a reasonable upper bound on the boolean-width for any graph, regardless of how big the graph is or what its edge density is. The main issue will be to give a fast heuristic for the computation of a good upper bound on  $|UN(A)|$ . Secondly, we need to consider heuristics for computing lower bounds on boolean-width, just as it has been done for tree-width [7]. Thirdly, we should explore pre-processing to simplify the graph instances, again this has been done extensively for tree-width [5]. On the theoretical side, a prime open problem is to find an algorithm computing boolean-width exactly. These problems are of interest since our results indicate that using boolean-width could in the future outperform the use of tree-width in practice for a large class of graphs and problems.

## References

1. I. Adler, B. M. Bui-Xuan, Y. Rabinovich, G. Renault, J. A. Telle, and M. Vatshelle. On the boolean-width of a graph: Structure and applications. *Proceedings of the 36th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2010*, pages 159–170, 2010.
2. B. Aspvall, A. Proskurowski, and J. A. Telle. Memory requirements for table computations in partial k-tree algorithms. *Algorithmica*, 27(3):382 – 394, 2000.
3. R. Belmonte and M. Vatshelle. Graph classes with structured neighborhoods and algorithmic applications. *Proceedings of the 37th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2011*, 2011. see full version [www.iu.uib.no/~martinv/Papers/LogBoolw.pdf](http://www.iu.uib.no/~martinv/Papers/LogBoolw.pdf).
4. Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25:1305–1317, 1996.
5. Hans L. Bodlaender. Treewidth: Characterizations, applications, and computations. In Fedor V. Fomin, editor, *Proceedings of the 32nd International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2006*, pages 1 – 14. Springer Verlag, LNCS, vol. 4271, 2006.
6. Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations I. Upper bounds. *Information and Computation*, 208:259–275, 2010.

7. Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations II. lower bounds. Technical Report UU-CS-2010-022, Department of Information and Computing Sciences, Utrecht University, Utrecht, the Netherlands, 2010. Accepted for publication in *Information and Computation*.
8. Andreas Brandsädt and Vadim Lozin. On the linear structure and clique-width of bipartite permutation graphs. *Ars Combinatoria*, 67:273–281, 2003.
9. A. Brandstadt. personal communication.
10. B. M. Bui-Xuan, J. A. Telle, and M. Vatshelle. Boolean-width of graphs. *Theoretical Computer Science*, 2011. (to appear) see full version [www.i.uib.no/telle/bib/listofpub/BTV11.pdf](http://www.i.uib.no/telle/bib/listofpub/BTV11.pdf).
11. Hubie Chen. Quantified constraint satisfaction and bounded treewidth. In Ramon López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 17th European Conference on Artificial Intelligence, ECAI 2004*, pages 161–165, 2004.
12. The second DIMACS implementation challenge: NP-Hard Problems: Maximum Clique, Graph Coloring, and Satisfiability. See <http://dimacs.rutgers.edu/Challenges/>, 1992–1993.
13. Martin Charles Golumbic and Udi Rotics. On the clique-width of perfect graph classes. *Proceedings of the 25th International Workshop on Graph-Theoretic Concepts in Computer Science, WG'99*, pages 135–147, 1999.
14. Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. *Acta Informatica*, 124:243–282, 2000.
15. Illya V. Hicks, Arie M. C. A. Koster, and Elif Kolotoğlu. Branch and tree decomposition techniques for discrete optimization. In J. Cole Smith, editor, *Tutorials 2005*, INFORMS Tutorials in Operations Research Series, chapter 1, pages 1–29. INFORMS Annual Meeting, 2005.
16. K. H. Kim. Boolean matrix theory and its applications. 1982. Marcel Dekker.
17. S. J. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *The Journal of the Royal Statistical Society. Series B (Methodological)*, 50:157–224, 1988.
18. Sang-il Oum and Paul Seymour. Approximating clique-width and branch-width. *Journal of Combinatorial Theory, Series B*, 96:514–528, 2006.
19. Arnold Overwijk, Eelko Penninx, and Hans L. Bodlaender. A local search algorithm for branch-width. *Proceedings of the 37th Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2011*, pages 444–454, 2011.
20. Hein Röhrig. Tree decomposition: A feasibility study. Master’s thesis, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1998.
21. Y. Song, C. Liu, R. Malmberg, F. Pan, and L. Cai. Tree decomposition based fast search of RNA structures including pseudoknots in genomes. In *Proceedings of the 2005 IEEE Computational Systems Bioinformatics Conference, CSB'05*, pages 223–234, 2005.
22. Treewidthlib. <http://www.cs.uu.nl/people/hansb/treewidthlib>, 2004– . . . .
23. Johan M. M. van Rooij, Hans L. Bodlaender, and Peter Rossmanith. Dynamic programming on tree decompositions using generalised fast subset convolution. In Amos Fiat and Peter Sanders, editors, *Proceedings of the 17th Annual European Symposium on Algorithms, ESA 2009*, pages 566–577. Springer Verlag, Lecture Notes in Computer Science, vol. 5757, 2009.
24. Jizhen Zhao, Dongsheng Che, and Liming Cai. Comparative pathway annotation with protein-DNA interaction and operon information via graph tree decomposition. In *Proceedings of Pacific Symposium on Biocomputing, PSB 2007*, volume 12, pages 496–507, 2007.
25. Jizhen Zhao, Russell L. Malmberg, and Liming Cai. Rapid ab initio prediction of RNA pseudoknots via graph tree decomposition. *Journal of Mathematical Biology*, 56(1–2):145–159, 2008.

## A Appendix

In the following table we list the graphs with both treewidth and boolean-width upper bounds. Graphs are sorted by ratio of tree-width divided by boolean-width.

Graph name	Vertices	Edge density	BWUB	TWUB	Ratio
zeroin.i.1-pp	54	0.89	1.58	46	29.02
mulsol.i.1-pp	73	0.83	2.32	50	21.53
miles1500	128	0.64	4.86	77	15.85
zeroin.i.1	211	0.19	3.70	50	13.51
fpsol2.i.1-pp	233	0.40	4.91	66	13.45
mulsol.i.1	197	0.20	4.00	50	12.50
zeroin.i.2-pp	57	0.69	2.81	32	11.40
zeroin.i.3-pp	57	0.69	3.00	32	10.67
mulsol.i.5-pp	119	0.36	3.00	31	10.33
celar06-pp	16	0.84	1.58	11	6.94
mulsol.i.4	185	0.23	4.81	32	6.65
mulsol.i.2	188	0.22	4.81	32	6.65
mulsol.i.3	184	0.23	4.95	32	6.46
celar04-pp-001	16	0.78	1.58	10	6.31
mulsol.i.5	186	0.23	4.95	31	6.26
zeroin.i.3	206	0.17	5.39	32	5.93
zeroin.i.2	211	0.16	5.39	32	5.93
celar04-pp-001-000	15	0.74	1.58	9	5.68
celar05-pp-001	19	0.66	2.00	11	5.50
oesoca+-pp	14	0.82	2.00	11	5.50
1fjl-pp-003	11	0.85	1.58	8	5.05
celar02-pp	19	0.67	2.00	10	5.00
sudoku	81	0.25	9.00	45	5.00
celar01-pp-002	19	0.65	2.00	10	5.00
celar11-pp-001	19	0.65	2.00	10	5.00
munin3-pp-002	7	0.81	1.00	5	5.00
munin3-pp-001	7	0.81	1.00	5	5.00
sudoku-elim1	80	0.28	9.47	45	4.75
weeduk	15	0.47	1.58	7	4.42
munin-pp-001	6	0.80	1.00	4	4.00
munin-pp-002	6	0.80	1.00	4	4.00
diabetes-pp-001	6	0.80	1.00	4	4.00
queen8_12	96	0.30	16.7	65	3.91
mainuk-pp	9	0.78	1.58	6	3.79
celar06-wpp	34	0.28	3.00	11	3.67
huck	74	0.11	2.81	10	3.56
celar02-wpp	30	0.33	2.81	10	3.56
1awd	89	0.28	10.80	38	3.52
games120-pp-001	14	0.69	2.58	9	3.48
anna-pp	22	0.64	3.46	12	3.47
1cc8	70	0.34	9.35	32	3.42
queen8_8	64	0.36	13.16	45	3.42
1igq-pp	52	0.37	6.74	23	3.41
1bf4-pp	57	0.39	7.63	26	3.41
queen5_5	25	0.53	5.29	18	3.41
queen9_9	81	0.33	17.07	58	3.40
1ptf	87	0.30	11.21	38	3.39
queen7_7	49	0.40	10.36	35	3.38
1ljo	74	0.29	8.88	30	3.38
1ljo-pp	71	0.31	8.92	30	3.36
1i0v	100	0.24	12.21	41	3.36
1gcq-pp	64	0.36	8.95	30	3.35
1erv	101	0.25	12.26	41	3.35
1iib-pp	102	0.27	11.98	40	3.34
1igq	54	0.35	6.89	23	3.34
1aac	204	0.25	12.29	41	3.34
1l1dd	74	0.31	9.60	32	3.33
1gef-pp	117	0.22	12.93	43	3.33
1dj7-pp	70	0.30	8.12	27	3.33
1c9o	66	0.34	8.75	29	3.31
1czp	94	0.27	11.47	38	3.31
1iqz	77	0.29	10.00	33	3.30

Table 2 – Continued

Graph name	Vertices	Edge density	BWUB	TWUB	Ratio
lg2b-pp	59	0.37	8.50	28	3.29
lr69	63	0.35	9.12	30	3.29
lfjl	65	0.29	7.90	26	3.29
lbf4	63	0.34	7.90	26	3.29
lpwt	61	0.36	8.81	29	3.29
lhg7	66	0.33	8.81	29	3.29
lc4q	67	0.34	9.45	31	3.28
lpwt-pp	59	0.38	8.85	29	3.28
queen6_6	36	0.46	7.65	25	3.27
ligd-pp	59	0.36	7.66	25	3.26
lc5e	95	0.26	11.06	36	3.26
lr69	48	0.37	6.77	22	3.25
lbfk-pp	105	0.23	11.10	36	3.24
lk61-pp	56	0.37	8.02	26	3.24
lfjl-pp-004	57	0.35	8.03	26	3.24
ligd	61	0.34	7.75	25	3.23
li2t	61	0.35	8.38	27	3.22
lsem-pp	56	0.37	8.09	26	3.21
lg2b	62	0.34	8.72	28	3.21
lgcq	68	0.33	9.36	30	3.21
celar08pp-pp-033	39	0.38	5.00	16	3.20
ljo8	58	0.37	8.46	27	3.19
david-pp	29	0.47	4.09	13	3.18
lj75	56	0.36	8.51	27	3.17
liib	103	0.26	12.62	40	3.17
lfjl-pp	63	0.30	8.20	26	3.17
lgef	119	0.21	13.60	43	3.16
lig5	75	0.29	10.45	33	3.16
lcka	57	0.38	8.55	27	3.16
lf9m	109	0.23	14.27	45	3.15
lfse	67	0.33	8.58	27	3.15
celar08-pp-001	39	0.38	5.09	16	3.14
lmgq	74	0.30	8.91	28	3.14
lbrf-pp	48	0.36	7.01	22	3.14
lbrf	49	0.35	7.01	22	3.14
li9l	70	0.29	9.26	29	3.13
lkq1-pp	59	0.35	8.63	27	3.13
lk61	60	0.33	8.32	26	3.12
lsem	57	0.36	8.32	26	3.12
le0b-pp	55	0.33	7.69	24	3.12
lmgq-pp	72	0.31	8.98	28	3.12
lg2r-pp	93	0.26	11.87	37	3.12
li27-pp	70	0.30	8.67	27	3.11
lplc	98	0.25	11.28	35	3.10
lkk-pp	99	0.24	11.00	34	3.09
lku3-pp	60	0.33	7.46	23	3.08
lbfk	106	0.23	11.69	36	3.08
li27	73	0.28	8.78	27	3.08
lctj	87	0.25	10.74	33	3.07
lkq1	60	0.34	8.79	27	3.07
BN_16-pp-015	34	0.28	3.58	11	3.07
lctj-pp	86	0.25	10.78	33	3.06
lgut-pp	61	0.33	7.19	22	3.06
lbrk	107	0.24	14.40	44	3.06
lbbz-pp	56	0.35	8.18	25	3.05
lku3	61	0.32	7.53	23	3.05
lg2r	94	0.25	12.17	37	3.04
lc75	69	0.29	9.88	30	3.04
lb0n-006	98	0.21	10.58	32	3.03
lbbz	57	0.34	8.30	25	3.01
celar02	100	0.06	3.32	10	3.01
celar06-pp-000	8	0.43	1.00	3	3.00
ljhg-pp	91	0.19	8.34	25	3.00
ldp7	76	0.27	9.01	27	3.00
lkw4	67	0.30	9.39	28	2.98
lail	69	0.27	8.07	24	2.98
ld3b	69	0.29	8.44	25	2.96
lb0n	103	0.19	10.81	32	2.96

Table 2 – Continued

Graph name	Vertices	Edge density	BWUB	TWUB	Ratio
lai1-pp	68	0.28	8.11	24	2.96
1e0b	60	0.29	8.13	24	2.95
1gut	67	0.28	7.47	22	2.95
1d4t-pp	99	0.23	11.88	35	2.95
1fr3-pp	62	0.32	7.16	21	2.93
1dp7-pp	71	0.30	9.21	27	2.93
celar07-pp-001	45	0.32	5.46	16	2.93
1d3b-pp	68	0.30	8.54	25	2.93
celar06	100	0.07	3.81	11	2.89
BN_16-pp-014	34	0.28	3.81	11	2.89
1fr3	67	0.28	7.29	21	2.88
1fk5	85	0.23	10.76	31	2.88
1aba	85	0.25	10.13	29	2.86
1lkk	103	0.22	11.89	34	2.86
celar07pp-pp-012	45	0.32	5.61	16	2.85
1kth	52	0.32	7.04	20	2.84
1kth-pp	51	0.33	7.06	20	2.83
1jhg	101	0.17	8.87	25	2.82
1qtn-pp	77	0.25	8.56	24	2.80
1rro	107	0.23	15.36	43	2.80
1oai	58	0.32	7.87	22	2.80
1dj7	73	0.28	9.66	27	2.79
1oai-pp	57	0.32	7.94	22	2.77
1ezg	66	0.25	8.33	23	2.76
1g6x	52	0.31	6.89	19	2.76
oesoca+-wpp	28	0.36	4.00	11	2.75
1a8o	64	0.27	9.11	25	2.75
1d4t	102	0.22	12.87	35	2.72
1a62	122	0.21	13.62	37	2.72
1i07	59	0.23	5.52	15	2.72
1i07-pp	51	0.28	5.55	15	2.70
celar07-wpp	97	0.01	6.00	16	2.67
celar07-pp	92	0.12	6.00	16	2.67
celar10-pp-001	16	0.51	3.00	8	2.67
celar08pp-pp-032	16	0.51	3.00	8	2.67
celar09-pp-001	16	0.51	3.00	8	2.67
celar08-pp-002	16	0.51	3.00	8	2.67
1dd3	128	0.17	11.68	31	2.66
1qtn	87	0.21	9.15	24	2.62
celar11-pp-003	48	0.23	5.73	15	2.62
myciel6	95	0.17	13.40	35	2.61
celar07pp-pp	75	0.15	6.17	16	2.59
celar01-pp-001	47	0.25	5.88	15	2.55
1a62-pp	120	0.21	14.70	37	2.52
BN_29	24	0.18	2.00	5	2.50
BN_28	24	0.18	2.00	5	2.50
celar07pp-pp-011	14	0.52	2.81	7	2.49
celar11-pp-002	13	0.59	2.81	7	2.49
celar05-pp-002	47	0.25	6.07	15	2.47
1fs1	114	0.21	13.79	34	2.47
david	87	0.11	5.32	13	2.44
1b67	68	0.25	6.61	16	2.42
1b67-pp	67	0.25	6.61	16	2.42
celar03-pp-001	38	0.34	5.81	14	2.41
water-pp-001	21	0.45	3.81	9	2.36
1en2	69	0.20	7.24	17	2.35
1bx7-pp	34	0.31	4.70	11	2.34
myciel5	47	0.22	8.12	19	2.34
miles500	128	0.14	9.42	22	2.34
celar07-pp-002	16	0.45	3.00	7	2.33
celar04-pp-002	62	0.17	6.86	16	2.33
pathfinder-pp	12	0.65	2.58	6	2.32
jean	80	0.08	3.91	9	2.30
oesoca+	67	0.09	4.81	11	2.29
1en2-pp	66	0.21	7.46	17	2.28
celar03-wpp	89	0.11	6.17	14	2.27
celar03-pp	81	0.13	6.19	14	2.26
celar11-pp	96	0.10	6.64	15	2.26

Table 2 – Continued

Graph name	Vertices	Edge density	BWUB	TWUB	Ratio
lhx7	41	0.24	4.91	11	2.24
munin4-pp-003	38	0.16	3.58	8	2.23
munin4-pp-001	23	0.26	3.58	8	2.23
munin4-pp-002	23	0.26	3.58	8	2.23
lon2-pp	133	0.17	16.26	36	2.21
celar10-pp-002	76	0.15	7.25	16	2.21
celar04-pp	110	0.09	7.29	16	2.19
celar08-pp-003	76	0.15	7.41	16	2.16
water-pp	22	0.42	4.17	9	2.16
water-wpp	22	0.42	4.17	9	2.16
celar01-pp-003	58	0.19	6.97	15	2.15
celar09-pp-002	76	0.15	7.46	16	2.14
ladd3-pp	124	0.17	14.60	31	2.12
homer-pp	95	0.17	14.61	31	2.12
celar01-wpp	158	0.06	7.09	15	2.12
barley-pp-001	16	0.42	3.32	7	2.11
celar11-wpp	99	0.10	7.17	15	2.09
celar05-pp	80	0.13	7.20	15	2.08
lbbk	131	0.17	14.53	30	2.06
water	32	0.25	4.39	9	2.05
celar01-pp	157	0.07	7.39	15	2.03
munin2-pp-007	17	0.35	3.46	7	2.02
munin2-pp-011	17	0.35	3.46	7	2.02
munin2-pp-010	17	0.35	3.46	7	2.02
munin2-pp-008	17	0.35	3.46	7	2.02
lubq-pp	47	0.16	5.95	12	2.02
celar04-wpp	116	0.07	7.95	16	2.05
myciel4	23	0.28	5.00	10	2.00
fungiuk	15	0.34	2.00	4	2.00
celar06-pp-003	4	0.50	1.00	2	2.00
celar05-wpp	89	0.11	7.52	15	1.99
munin1-pp	66	0.09	5.58	11	1.99
munin1-pp-001	63	0.09	5.58	11	1.97
mainuk	48	0.18	3.58	7	1.95
pathfinder-pp-001	11	0.58	2.58	5	1.93
lbbk-pp	127	0.18	15.55	30	1.93
munin4-pp-004	38	0.16	4.17	8	1.92
graph05-pp-001	87	0.10	12.68	24	1.89
celar11-pp-004	16	0.36	3.17	6	1.89
barley-pp	26	0.24	3.70	7	1.89
celar09-wpp	142	0.06	8.49	16	1.88
munin4-pp-006	55	0.11	4.32	8	1.85
lubq	74	0.08	6.51	12	1.84
barley-wpp	29	0.20	3.81	7	1.84
munin4-pp-005	55	0.11	4.39	8	1.82
miles250	128	0.05	4.95	9	1.82
lubq	74	0.08	6.61	12	1.81
pathfinder	109	0.04	3.32	6	1.81
anna	138	0.04	6.67	12	1.78
graph01-pp-001	85	0.09	13.40	24	1.79
celar10-pp	133	0.07	9.08	16	1.76
graph05	100	0.08	13.70	24	1.75
barley	48	0.11	4.00	7	1.75
munin2-pp-009	18	0.31	3.46	6	1.73
munin2-pp-012	18	0.31	3.46	6	1.73
graph05-pp	91	0.10	13.84	24	1.73
diabetes-pp-002	8	0.61	2.32	4	1.72
a280.tsp-pp	92	0.06	8.23	14	1.70
munin3-pp-003	79	0.09	4.17	7	1.68
graph05-wpp	94	0.09	14.38	24	1.67
munin2-pp-005	16	0.30	3.00	5	1.67
munin2-pp-006	16	0.30	3.00	5	1.67
pcb3038-pp-002	15	0.30	3.00	5	1.67
myciel3	11	0.36	3.00	5	1.67
pcb3038-pp-001	11	0.40	3.00	5	1.67
celar08-wpp	190	0.05	9.64	16	1.66
graph01	100	0.07	14.61	24	1.64
graph01-pp	89	0.08	14.62	24	1.64



Table 2 – Continued

Graph name	Vertices	Edge density	BWUB	TWUB	Ratio
graph01-wpp	93	0.07	14.69	24	1.63
munin3-pp	96	0.07	4.32	7	1.62
graph03-pp-001	71	0.11	12.53	20	1.60
pigs-pp	48	0.12	5.70	9	1.56
munin2-pp-003	16	0.30	3.17	5	1.58
munin2-pp-004	16	0.30	3.17	5	1.58
graph03-wpp	84	0.09	12.74	20	1.57
eil51.tsp	51	0.11	5.78	9	1.56
mildew-wpp	15	0.30	2.58	4	1.55
alarm	37	0.10	2.58	4	1.55
kroE100.tsp-pp	92	0.06	6.48	10	1.54
graph03-pp	79	0.10	12.99	20	1.54
munin1-wpp	90	0.05	7.23	11	1.52
pigs-pp-001	47	0.12	5.95	9	1.51
graph03	100	0.07	13.29	20	1.50
knights8_8-pp	60	0.09	10.77	16	1.49
knights8_8	64	0.08	11.06	16	1.45
kroC100.tsp-pp	97	0.06	6.94	10	1.44
rl5934-pp-001	10	0.44	2.81	4	1.42
fl3795-pp-001	10	0.44	2.81	4	1.42
fl3795-pp-003	10	0.44	2.81	4	1.42
fl3795-pp-002	10	0.44	2.81	4	1.42
ch130.tsp-pp	125	0.05	8.67	12	1.38
mildew	35	0.13	3.00	4	1.33
fl3795-pp-004	11	0.42	3.00	4	1.33
fl3795-pp-006	26	0.16	3.81	5	1.31
kroA100.tsp-pp	95	0.06	7.61	10	1.31
oesoca	39	0.09	2.32	3	1.29
oesoca42	42	0.08	2.32	3	1.29
munin2-pp	167	0.03	5.49	7	1.27
fl3795-pp-005	19	0.22	3.32	4	1.20
pcb3038-pp-003	29	0.12	4.32	5	1.16

In the following table we present the graphs where boolean-width upper bound is known but no treewidth bound is reported.

Graph name	Vertices	Edge density	BWUB
BN_14-pp	87	0.10	19.27
BN_9	105	0.07	19.05
BN_3-pp	81	0.12	17.31
BN_2	100	0.10	17.24
BN_4-pp	81	0.16	17.22
BN_3	100	0.09	17.09
BN_7-pp	79	0.16	17.03
BN_2-pp	81	0.14	16.24
BN_8-pp	72	0.14	15.25
BN_12-pp	73	0.16	14.71
1aac-pp	103	0.25	14.53
BN_102	76	0.13	14.47
BN_103	76	0.13	14.47
BN_0-pp	67	0.11	13.93
BN_1-pp	74	0.13	13.85
kneser8-3	56	0.18	13.77
BN_102-pp	64	0.16	12.94
BN_103-pp	64	0.16	12.85
BN_10	85	0.09	12.72
BN_10-pp	57	0.16	11.84
lpsr-pp	94	0.22	10.75
BN_99	57	0.17	10.40
BN_98	57	0.17	10.31
lpsr	98	0.20	10.14
BN_100	58	0.17	10.06
BN_101	58	0.17	10.06

Table 3 – Continued

Graph name	Vertices	Edge density	BWUB
BN_99-pp	47	0.22	9.77
BN_98-pp	47	0.22	9.63
BN_100-pp	51	0.20	9.61
BN_101-pp	51	0.20	9.54
BN_96-pp	48	0.21	9.18
BN_97-pp	48	0.21	9.15
BN_97	54	0.17	9.10
BN_96	54	0.17	9.10
BN_94	53	0.18	8.85
BN_95	53	0.18	8.84
BN_94-pp	44	0.23	8.26
BN_95-pp	44	0.23	8.26
pr76.tsp	76	0.08	7.84
pr76.tsp-pp	73	0.08	7.71
DSJC125.9	125	0.90	7.34
macaque71	71	0.18	7.27
eil76.tsp-pp	75	0.08	7.17
pr299.tsp-pp	80	0.07	7.17
rd100.tsp-pp	97	0.06	7.17
lin105.tsp-pp	96	0.06	7.17
eil76.tsp	76	0.08	7.17
macaque71-pp	58	0.25	7.13
rat99.tsp	99	0.06	6.94
BN_78	54	0.16	6.83
rat99.tsp-pp	96	0.06	6.78
pr152.tsp	152	0.04	6.70
BN_79	54	0.16	6.39
BN_78-pp	33	0.33	6.07
BN_79-pp	33	0.33	6.07
pr124.tsp-pp	85	0.07	5.78
pr107.tsp-pp	93	0.06	5.78
BN_123	50	0.60	5.61
BN_122	50	0.60	5.52
BN_115	50	0.60	5.52
BN_114	50	0.60	5.52
BN_112	50	0.59	5.46
BN_121	50	0.60	5.43
BN_113	50	0.59	5.43
BN_120	50	0.60	5.43
BN_124	50	0.60	5.43
BN_125	50	0.60	5.43
BN_104	50	0.60	5.32
ship-ship	50	0.09	5.32
ship-ship-wpp	38	0.13	5.29
BN_107	50	0.62	5.29
BN_106	50	0.62	5.25
BN_105	50	0.60	5.21
BN_119	50	0.60	5.17
BN_90-pp	38	0.56	5.13
BN_116	50	0.61	5.13
BN_110	50	0.60	5.13
BN_118	50	0.60	5.13
BN_117	50	0.61	5.09
BN_111	50	0.60	5.09
BN_108	50	0.60	5.09
BN_86-pp	38	0.56	5.04
BN_91-pp	38	0.56	5.04
BN_109	50	0.60	5.04
BN_89-pp	38	0.56	5.00
BN_93-pp	38	0.56	5.00
BN_88-pp	38	0.56	4.95
BN_87-pp	38	0.56	4.95
BN_92-pp	38	0.56	4.91
ship-ship-pp	30	0.18	4.70
oesoca+-hugin	67	0.09	4.64
oow_solo	40	0.11	4.58
oow_solo-wpp	29	0.16	4.17

Table 3 – Continued

Graph name	Vertices	Edge density	BWUB
oow_solo-pp	27	0.18	4.17
Clebsch	16	0.33	4.00
oesoca+-hugin-wpp	28	0.36	4.00
oow-trad	33	0.14	4.00
BN_80-pp	29	0.71	3.81
oow-trad-pp	23	0.21	3.81
oow-trad-wpp	25	0.19	3.81
BN_85-pp	29	0.71	3.81
BN_81-pp	29	0.71	3.70
BN_84-pp	29	0.71	3.70
BN_83-pp	29	0.71	3.70
BN_82-pp	29	0.71	3.70
risk-pp	18	0.26	3.32
oow_bas	27	0.15	3.32
risk	42	0.10	3.32
petersen	10	0.33	3.00
munin_kgo_complete-wpp	95	0.06	3.00
oow_bas-wpp	17	0.26	2.81
vsd-hugin	38	0.09	2.58
myciel2	5	0.50	2.00
wilson-hugin	21	0.13	2.00
munin_kgo_complete-pp	16	0.34	2.00
oesoca+-hugin-pp	14	0.82	2.00
MCSSTestGraph	7	0.52	2.00
MCSSTestGraph2	9	0.36	2.00
a280.tsp-pp-001	3	1.00	1.00
1fjl-pp-002	8	1.00	1.00
1b0n-003	8	1.00	1.00
1b0n-005	8	1.00	1.00
1b0n-004	8	1.00	1.00
1b0n-001	7	1.00	1.00
1b0n-002	8	1.00	1.00
1fjl-pp-001	6	1.00	1.00