MM-Width: Observations, Algorithms and Approximations

Sebastiaan F.A.M. Brouwer

August 30, 2017

Abstract

In this thesis we investigate maximum matching-width (MM-width) further. MM-width is a graph width parameter similar to treewidth, related to the number of maximum matchings made in an induced bipartite graph made from partitions over the vertices of a graph. We improve the link between the value of maximum matching-width and the value of treewidth of a graph to $MM(G) \leq tw(G)$. We also give a bounded dynamic programming algorithm BMMDP to calculate the MM-width of a graph exactly. In addition to the exact algorithm we look into approximating the MM-width of graphs from above by using optimization algorithms based on local search and evolutionary algorithms.

In the thesis we also make general observations about maximum matching-width, investigate the MM-width of standard graphs and come up with a set of safe kernelization rules to improve the performance of our algorithms. We also use the link between maximum matching-width and the other width parameters, the MM-widths of standard graphs and widths found during run time to add upper and lower bounds to the exact algorithm.

Keywords: maximum matching-width, graph decomposition, width parameter, graph algorithm, treewidth, approximation

1 Introduction

In 2012 Martin Vatshelle introduced three new graph parameters: boolean-width, maximum matching-width (MM-width) and maximum induced matching-width (MIM-width); all three based on the notion of branch decompositions of specific set function as defined by Robertson and Seymour. These new graph parameters were both introduced and compared to other, earlier defined, graph width parameters in Vatshelle's work in [1].

MM-width has already been used in the literature to give a faster algorithm for dominating set than treewidth gives [18] and it and the related graph width parameters may bring even more such improvements.

In Vatshelle's thesis MM-width is mainly used to simplify the comparison between treewidth and parameters defined via branch decomposition of a set function. In this thesis however, we let MM-width take a more central position and we will be examining MM-width itself closer.

We will first look at the definition of MM-width to make initial analyses, after which we will find a more intuitive definition with which we will do a another set of analyses. With that information we can look at a set of standard graphs and reason about the MM-width of such graphs, such that we do not have to calculate these widths and can just easily write them out.

We also find a few kernelization rules, though we were not able to find a set of rules that gives us a polynomial kernel or prove that we have such a kernel. These rules are still useful for reducing the size of input to an algorithm though and therefore reducing the running time of an algorithm while still allowing the algorithm to output the correct MM-width of a graph.

In addition to the kernelization rules to improve on the performance of an algorithm to find the MM-width of a graph, we also use upper and lower bounds. These bounds are not only derived directly from the values we know from our earlier analyses of the values of the graph, but the lower bound also updates as we make decisions during the algorithm and can be based on the values of other graph width parameters.

Using the analyses of the MM-width values of standard graphs, we were also able to improve the link between the value of the treewidth and the value of the maximum matching-width of a graph.

We also have developed a set of approximation algorithms based on optimization algorithms to find the MM-width of graphs too large to be easily handled by the exact algorithms. To be precise, we based these on local search algorithms and evolutionary algorithms. As these approximation algorithms approximate the MM-width of a solution from above by finding non-optimal decompositions, these approximation algorithms can be used to find upper bounds on the MM-widths of larger graphs.

Finally, we have also provided a small list of known MM-widths of graphs.

We have tried writing this thesis somewhat closer to an introductory work to MM-width, by having it introduce MM-width itself and presenting examples in some situations to help illustrate both MM-width and relevant situations. We did this such that any reader with knowledge of sets and graphs can follow the contents without having to also read relating work.

2 Definitions

Definition 1. *(Graph)*
A graph $G = (V, E)$ *is a pair of a set of vertices and a set of edges, where an edge is a subset of vertices of* G*.* $V(G)$ *is the denotion of the set of vertices of* G *and* $E(G)$ *is the denotion of the set of edges of* G*.* [*In this thesis, we will only consider undirected edges between exactly two distinct vertices and graphs without duplicate edges.*]

Definition 2. *(Degree)*
Let G *be a graph. The* degree *of a vertex* $v \in V(G)$ *is the number of edges with* v *as an endpoint.*

Definition 3. *(Subgraph)*
Let G *and* H *be graphs.* H *is a subgraph of* G *if and only if* $V(H) \subseteq V(G)$ *and* $E(H) \subseteq E(G)$*. This is denoted* $H \subseteq G$*. Given* $S \subseteq V(G)$*,* $G[S]$ *is the* induced subgraph*, if and only if* $G[S] \subseteq G$*,* $V(G[S]) = S$ *and* $E(G[S]) = \{(x, y) \in E(G) \mid x, y \in S\}$*.*

Definition 4. *(Bipartite graph)*
A graph G *is* bipartite *if and only if there is a subset* $S \subseteq V(G)$*, such that* $\forall (x, y) \in E(G) : (x \in S \,\text{AND}\, y \notin S) \,\text{OR}\, (x \notin S \,\text{AND}\, y \in S)$*, thus every edge has an endpoint in* S *and one outside of it.*

Definition 5. *(Induced bipartite subgraph)*
Let G *be a graph and* S *and* R *be two subsets of* $V(G)$*, such that* $S \cap R = \emptyset$*. Then graph* H *is the* induced bipartite subgraph *of* G*,* S *and* R *if and only if* $V(H) = S \cup R$ *and* $E(H) = \{(x, y) \in E(G) \mid (x \in S \,\text{AND}\, y \in R) \,\text{OR}\, (x \in R \,\text{AND}\, y \in S)\}$*. This is denoted* $H = G[S, R]$*.*

Definition 6. *(Neighbourhood)*
For G *a graph and* $v \in V(G)$ *a vertex of that graph, the* neighbourhood $N(v) = \{x \mid \exists y \in V(G) : (x, y) \in E(G)\}$ *is the set of vertices connected directly to* v *by an edge. The* closed neighbourhood *of* v *also includes* v *itself and is denoted as* $N[v] = N(v) \cup v$*.*

Definition 7. *(Vertex complement)*
Let G *be a graph and* $S \subseteq V(G)$*. The* complement *of* S *is denoted* $\bar{S} = V(G) \setminus S$ *and indicates the set of vertices in* $V(G)$ *not in* S*.*

Definition 8. *(Connected component)*
A connected component *is an induced subgraph* H *of a graph* G*, such that for each pair of vertices* $v, w \in V(H)$*, there is a path connecting* v *with* w *through* H *and through* G*, for any vertex* $x \in V(H)$ *and* $y \in V(\bar{H})$ *there no path connecting* x *with* y *through* G*. The function* $\text{CC}(G)$ *returns a set containing each connected component of* G *exactly once.*

Definition 9. *(Connected graph)*
A graph G *is a* connected graph *if and only if* G *contains at most one connected component.*

Definition 10. *(Tree)*
A graph G is considered a tree *if and only if G is a single connected component without loops. Thus for every pair of vertices in the graph, there is exactly one unique path connecting them. Vertices of trees are called* nodes*. Nodes of degree at most 1 are called leaves. The set $L(G) = \{v \in V(G) \mid degree(v) \leq 1\}$ contains all leaves of the tree. Nodes of degree at least 2 are called internal nodes.*

Definition 11. *(Rooted tree)*
A tree T is considered a rooted tree *if there is a node r that has been designated as the* root *of the tree. In a rooted tree, the* parent *of a node v is the neighbour of v closest to the root (the root does not have a parent). A node v is considered a* child *of node w if and only w is a parent of v. Each pair of nodes in the rooted tree with the same parent is considered* siblings*. The* ancestors *of a node v is a set containing all nodes on the path to the root, including the root, starting from the parent of v, and is denoted $a(v)$. The* descendants *of a node v is the set containing all nodes that have v as an ancestor.*

Definition 12. *(Binary tree)*
A (rooted) tree T is a binary tree *if and only if each node is either a leaf or has exactly two children.*

Definition 13. *(Binary decomposition tree)*
Let G be a graph. A binary decomposition tree *of G is a pair (T, δ), where T is a binary tree and $\delta : V(G) \to L(T)$ a bijection. For every node $a \in V(T)$, $L_a = \{L(T) \mid a \in a(L(T))\}$ (the set of leaves having a as ancestor) and $V_a = \{\delta^{-1}(x) \mid x \in L_a\}$ (the set of vertices mapped to the leaves having a as ancestor).*

Definition 14. *(f-width)*
Let G be a graph, $f : 2^{V(G)} \to \mathbb{R}$ a set function on $V(G)$ and (T, δ) a binary decomposition tree of G. The f-width *of (T, δ) is $\max_{\forall a \in V(T)} f(V_a)$. The* f-width *of G is the minimum f-width over all binary decomposition trees of G.*

Definition 15. *(Matching)*
Let G be a graph. A matching *is a set of edges $M \subseteq E(G)$, such that for every vertex $v \in V(E)$ there is at most one edge in M with v as an endpoint.*

Definition 16. *(Maximal matching)*
Let G be a graph. A matching is a maximal matching *if and only if there is no edge $e \in E(G) \setminus M$ such that $M \cup \{e\}$ is a matching.*

Definition 17. *(Maximum matching)*
For all possible matchings M over a graph G, a matching is considered a maximum matching *if and only if there is no matching M' over G with $|M'| > |M|$.*

3 Naive MM-Width

Using the definitions in the previous section, we can define MM-width as such, following the definition from Vatshelle from [1]:

Definition 18. *(MM-width)*
Let G be a graph and $mm : 2^{V(G)} \to \mathbb{N}$ be a symmetric set function, where $mm(A)$ equals the size of a maximum matching in $G[A, \bar{A}]$ for $A \subseteq V(G)$. Using the definition for f-width with $f = mm$, we define the MM-width *of a G to be the minimum mm-width over all binary decomposition trees of G. The mm-width of a binary decomposition tree (T, δ) is $\max_{\forall v \in V(T)} mm(V_v)$. We denote the MM-width of a graph G as $MM(G)$ and the mm-width of a decomposition tree (T, δ) as $mm((T, \delta))$.*

Thus, to calculate the MM-width of a graph, we calculate the mm-width of each possible binary decomposition tree of the graph. To calculate the mm-width of a binary decomposition tree, we need to evaluate the function mm for each of the nodes in the binary decomposition tree. To evaluate the function for a node, we need to calculate the size of a maximum matching in the induced bipartite subgraph with on one side all the vertices in our graph that have a corresponding leaf in the binary decomposition tree and all the other vertices on the other side.

To calculate the size of a maximum matching in a bipartite graph, we can use Hopcroft Karp [8], an $O(|E(G)|\sqrt{|V(G)|})$ algorithm that finds a maximum matching of a bipartite graph G.

Using this list of steps directly, gives us our first algorithm: *Naive MM-width*. (The referred functions in the algorithm can be found in the Algorithm Addendum on page 76.)

Algorithm 1: Naive MM-width

```
Data: a graph g
Result: MM-width of the graph and the binary decomposition tree of that
        score
unlabeledtrees ⟵ AllBinaryTrees(g.vertices.Length);
allnodeorders ⟵ AllOrdersOfLength(g.vertices.Length);
bestscore ⟵ int.MaxValue;
besttree ⟵ null;
foreach unlabeledtree in unlabeledtrees do
    foreach nodeorder in allnodeorders do
        labeledtree ⟵ LabelTree(unlabeledtree, nodeorder);
        mmscore ⟵ TreeScorer(g, labeledtree);
        if mmscore < bestscore then
            bestscore ⟵ mmscore;
            besttree ⟵ labeledtree;
return bestscore, besttree
```

Naive MM-width is guaranteed to find the MM-width of any graph you input and one of the optimal binary decomposition trees. As the number of binary trees of n leaves follows the formula $\frac{(2n-2)!}{n! \cdot (n-1)!}$ (the Catalan numbers [9]), there are $n!$ ways of of ordering the n labels and we have to execute Hopcroft Karp for each

of the $2n-1$ nodes in each binary tree, our Naive MM-width algorithm runs in $O(\frac{(2n-2)!}{n!\cdot(n-1)!} \cdot n! \cdot (2n-1) \cdot (m\sqrt{n}))$, where n is the number of vertices in our graph and m the number of edges. We can roughly reduce this to $O(n! \cdot m \cdot n \cdot \sqrt{n})$ and thus $O^*(n!)$.

The performance of this algorithm is disappointing, as can be seen in the Results section on page 66 and most clearly in figure 58. The algorithm starts taking almost half an hour to calculate the MM-width of graphs consisting of 8 vertices, over 12 hours to complete the calculation of the MM-width for graphs consisting of 9 vertices and even crashes on execution on graphs of size 10 and beyond.

3.1 Naive Asynchronous MM-Width

The performance of Naive MM-width can be improved upon by utilizing the independence of each execution of the body of the inner foreach-loop. Because the calculation of the MM-width of one of the binary decomposition trees does not depend on any earlier calculation, we can calculate the MM-width of multiple binary decomposition trees simultaneously. The difficulty of calculating MM-width naively grows with a factorial factor, however. Even assuming the exponential processor transistor growth implied by Moore's law [10] translates directly into exponential computation speed growth and we assume sufficient process memory, future attempts at using either version of Naive MM-width will remain stalling at small graph sizes as factorials grow faster than exponentials.

4 Initial Analysis of MM-Width

In order to understand MM-width better and to figure out how to improve the calculation of the MM-width of a graph, we will first look at how MM-width can be calculated, which may help in creating an intuition for the MM-width of graphs. Then we will look at how we can estimate the mm scores for a part of the decomposition, further helping in creating an intuition.

4.1 Example of MM-Width

Say we have the graph G as seen in figure 1. To calculate the MM-width of G, we need to find the minimum mm-width over all the binary tree decompositions of G. There are many different binary tree decompositions possible for graph G, as any binary tree with the same amount of leaves as the graph it is a decomposition tree of, taken together with any function associating the leaves of the binary tree to the vertices of the graph. is a binary tree decomposition of that graph. (T, δ), as seen in figure 2, is one of these possible decompositions and, as we need to find the minimum mm-width over all of the decompositions, we need to calculate the mm-width of this binary decomposition tree as well.

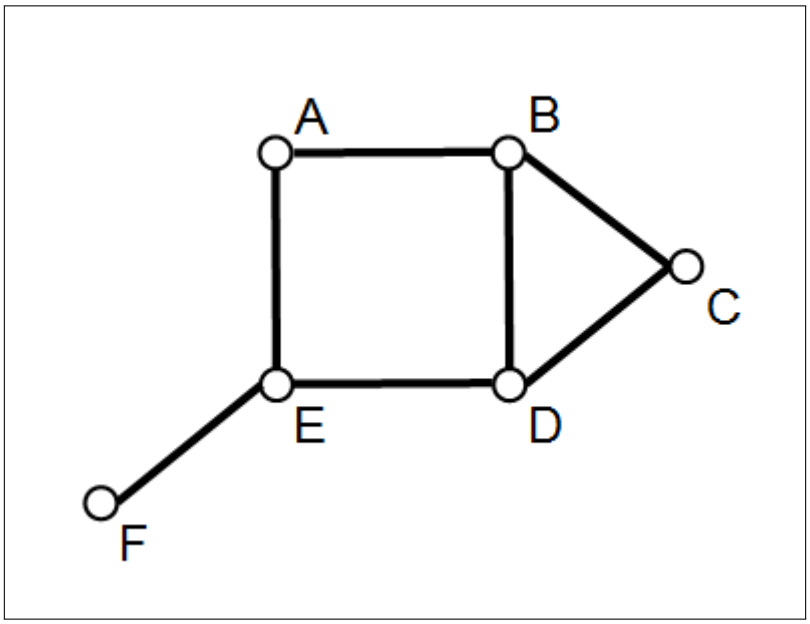

Figure 1: Graph G we are working with for our example.

The mm-width of a binary tree decomposition is the maximum over the result of the function mm over all the possible V_n of the tree, where n is a node in the tree, as we defined earlier. We demonstrate this by calculating the function mm for one of the nodes in the tree. Let's denote the parent node of $\delta(A)$ in this binary decomposition tree as v. As $mm(V_v) = mm(\{A, E, D\})$, we will construct $G[\{A, E, D\}, \{B, C, F\}]$ as seen in figure 3 and figure out the size of a maximum matching in this graph.

We can make exactly three matchings in $G[\{A, E, D\}, \{B, C, F\}]$ at most at the same time, meaning that $mm(V_v) = 3$. Doing this for each node of (T, δ), we can see that the mm-width of (T, δ) is also three, as demonstrated in figure 4.

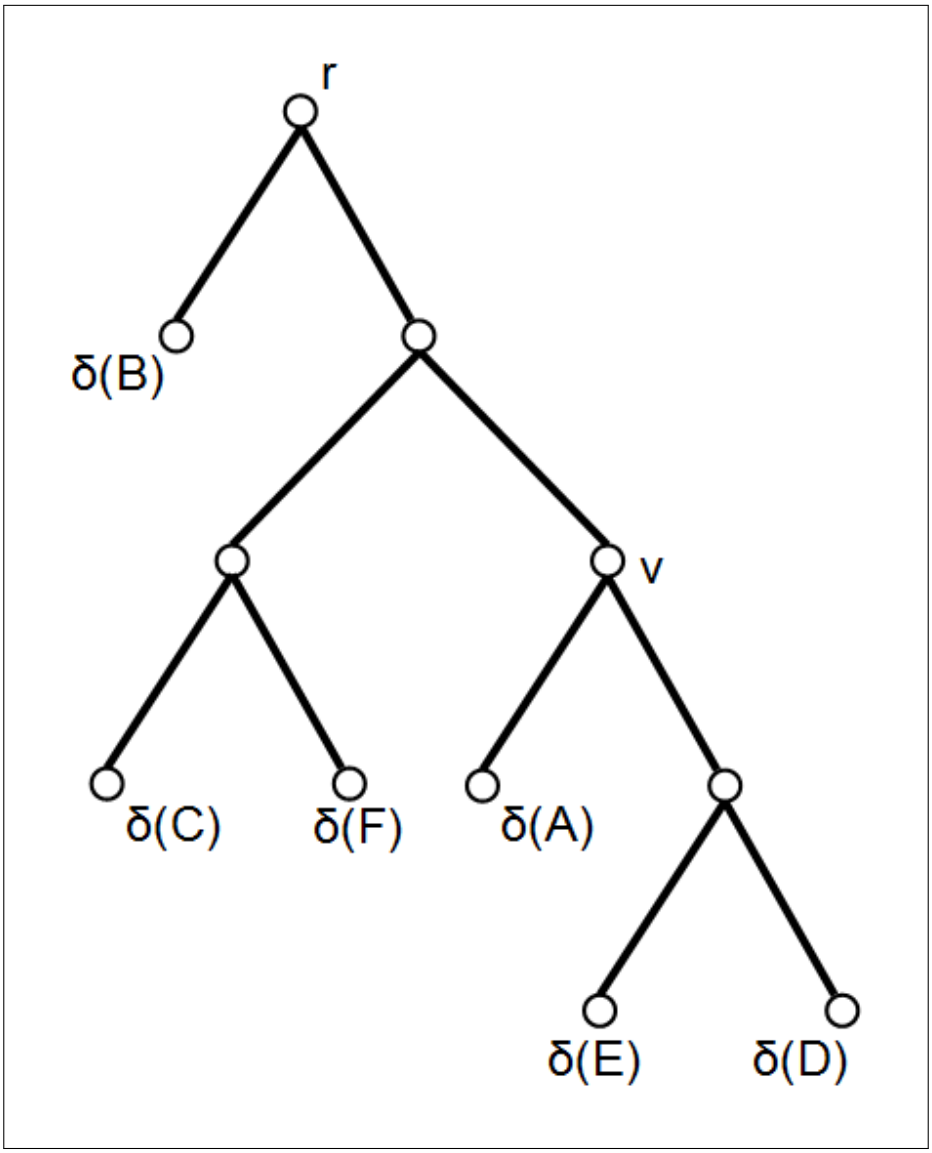

Figure 2: Binary decomposition tree (T, δ).

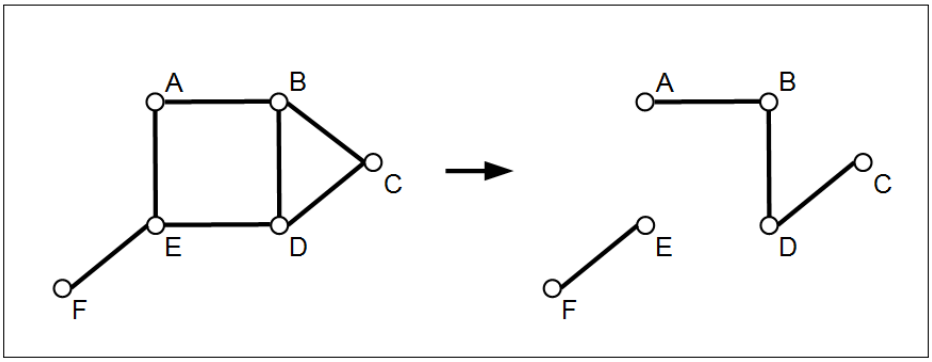

Figure 3: Graph G and $G[\{A, E, D\}, \{B, C, F\}]$.

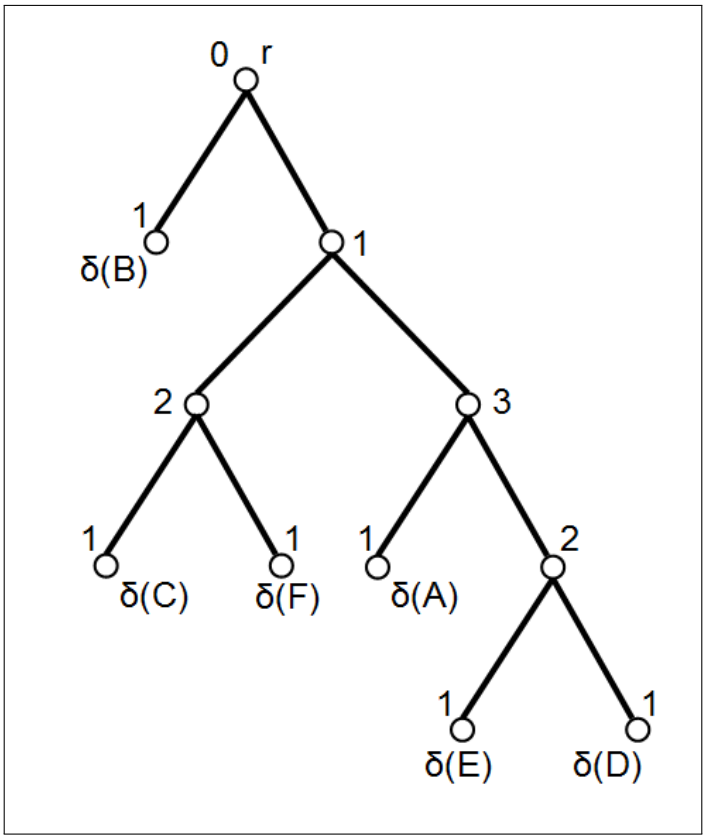

Figure 4: (T, δ) with all nodes marked with their mm score.

Looking around some more, we can actually find a better binary decomposition tree however, $(T, \delta)'$ for example, as seen in figure 5. As the highest mm score found for any possible V_n of $(T, \delta)'$ is two and we cannot find any better binary decomposition trees for G, we know now that the MM-width of G two.

To understand why $(T, \delta)'$ has a lower mm-width than (T, δ), we first look at vertex F of the graph. The only vertex that vertex can match with is E. If we find F descending the same vertices as E as much as possibly, we basically prevent F from increasing the MM-width of the graph. Similarly, C can only match with B and D, so if C finds itself either descending the same vertices as B and D, we prevent it from increasing the MM-width beyond what B and D add together. The same happens if C finds itself descending the same vertices as B and A, or D and E.

4.2 Estimation of MM Scores of Nodes

When looking at a binary decomposition tree of a graph, some scores in the tree can be easily estimated or given an upper bound.

For all nodes n in the binary decomposition tree (T, δ) of a graph G, the vertices in V_n are placed on one side of the induced bipartite subgraph when we apply the function mm, while the complement is placed on the other side. As vertices on the same side of an induced bipartite subgraph can only match with vertices on the other side, the maximum amount of matchings that can be made in the induced bipartite subgraph is at most the amount of vertices on one side and at most the amount of vertices on the other side.

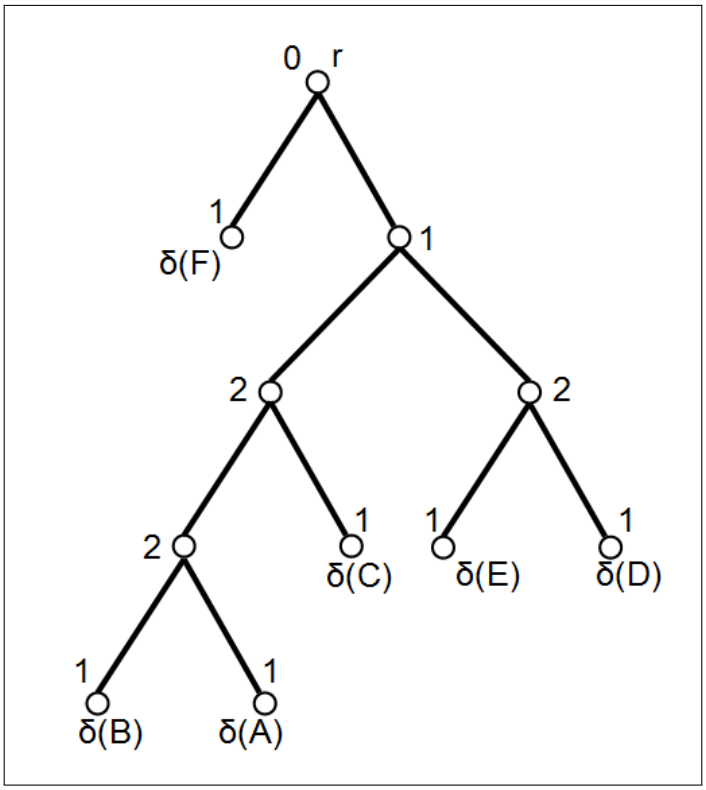

Figure 5: $(T, \delta)'$ with all nodes marked with their mm score.

Thus, we know that for all binary decomposition trees (T, δ) of graph G we know that $\forall n \in V(T) : mm(n) \leq \min(|V_n|, |\bar{V_n}|)$.

In addition, for all binary decomposition trees (T, δ) of graph G we find that the following also holds:

For the root node r, as $V_r = V(G)$ and $\bar{V_r} = \emptyset$, $\min(|V(G)|, 0) \geq mm(r) = 0$.

For a leaf node n, $|V_n| = 1$. Thus we know that $1 \geq mm(n)$. To be more precise, if $\delta(n)$ has degree at least 1, it will always be matched in $mm(\{\delta(n)\}, \bar{V_n})$, thus if G is non-trivial, for a leaf node n, $mm(n) = 1$.

5 Better Analysis of MM-Width

(Asynchronous) Naive MM-Width has a multitude of issues. The first of which is clearly the running time of the algorithm on even small graphs. The second issue is the fact that Naive MM-Width requires too much process memory, making it crash at slightly larger, but still small graphs. We can fix these issues however by changing our algorithm to not try every single possible decomposition tree in order. Before we will introduce the other method in the next section however, we shall take a look at a small improvement and analyze the properties of MM-width a bit further.

5.1 Calculation over Edges

First off, we take a look at the definition of MM-width as we gave it earlier. In this definition we defined the mm-width of a binary decomposition tree (T, δ) as $\max_{\forall v \in V(T)} mm(V_v)$, with $mm(A)$ the size of a maximum matching in $G[A, \bar{A}]$ for $A \subseteq V(G)$.

Because the size of a maximum matching in a bipartite graph with no vertices on one of the sides is always 0 and we are maximizing, we will never have to look at the root of a binary decomposition for calculating the mm-width of it. Because of this, we can redefine the mm-width of a binary decomposition tree as such:

Definition 19. *(Alternative mm-width)*
The mm-width of a binary decomposition tree (T, δ) is $\max_{\forall e \in E(T)} mm(\mathcal{P}_e)$, where $\mathcal{P}_e$ is a partition over the vertices of G, which we get by removing edge e from T, then looking at only one of the two connected components that we get and taking the the vertices corresponding for the leaves in that connected component as one side of the partition and the complement as the other side.

In this definition, instead of looking at nodes of the tree and looking at what leaves have it as ancestor, we look at the edge connecting this node to its parent and skip looking at the root node. After we remove the edge above node n, the connected component containing n has exactly those nodes in it that have n as an ancestor. Thus this alternative definition of mm-width is equivalent to the original definition. We also find that this definition is usually a bit more helpful to reason about the MM-width of graphs by hand.

All the analyses we made in the previous section also hold for this definition as they work over partitions and the function mm. If an analysis works on a node, the analysis will also work on the edge between that node and its parent (except for the parent node, which we already reasoned to always have the mm score 0).

5.2 Equivalence of Edges to Root

Say we calculate the mm score of the left child node l of the root node of a binary decomposition tree. The leaves with l as an ancestor are L_l and the MM-width of this node is $mm(L_l, \bar{L}_l)$. As L_l are exactly those leaves that are located in the left subtree of the root, $\bar{L}_l$ must be the leaves located in the right subtree of the root. This means that, for the two children of the root, the mm score is always the same.

It follows that the exact location of where the root is in a binary decomposition tree is not important to the mm-width of a binary decomposition tree, and the root

node could technically be left out entirely. However, for many applications a tree structure is very helpful and the definitions of MM-width in the literature and that we give work on trees.

In addition, as the root is not important to the score, we can choose to remove the root from the tree and then split any edge in the resulting graph to make a new root node. Any tree constructed in this way from the original tree has the same mm-width and is therefore equivalent to the tree it was constructed from.

5.3 Ordering of Siblings

As neither definition of MM-width uses anything related to whether a child is a left or right child, siblings in binary decomposition trees are interchangeable. This means that for each internal node of a decomposition tree, the tree resulting from switching the left and right subtree is also equivalent to the original tree.

Together with the *equivalence of the edges to the root node*, the *equivalence of the different ordering of siblings* implies that many decomposition trees are equivalent. Therefore, one important way of improving the efficiency of the calculation of f-widths of graphs is identifying and preventing calculation of equivalent decomposition trees.

6 Dynamically Programmed MM-Width

Using the insights of the better analysis of MM-width that we did in the previous section, we can now improve upon the complexity of Naive MM-width by using dynamic programming. Dynamic programming is a method of saving on computation time by solving subproblems and combining the results of these subproblems. More can be read about dynamic programming in [11]. In our case we will be identifying the subproblems we need to solve during the run of the algorithm and solving them the first time we find them, while retrieving the solution from a table on subsequent discoveries.

For our dynamic programming approach, we will be looking at subsets of the vertices of the graph. Each of these subsets of the vertices directly implies a partition as in our alternative mm-width definition. As there are 2^n possible subsets of a set of n vertices and calculating the mm score for one of these subsets for n vertices and m edges with Hopcroft-Karp takes $O(m \cdot \sqrt{n})$, we can calculate the mm score of all possible partitions of a graph in $O(m \cdot \sqrt{n} \cdot 2^n)$, where n is the number of vertices in the graph and m the number of edges.

Simply calculating the mm scores of all partitions is not enough though, we need to know the mm-width of the best binary decomposition tree of the graph. To do both these things in our algorithm, we will decide for every node, starting from the root node, what subset of vertices will be placed in the left subtree of the node and with that also what subset will be placed in the right subtree. This decision can be made by calculating the mm scores of the resulting partitions, however we will also be storing both the mm scores of the partitions and the optimal subtrees we have already calculated. This means that the mm score of every partition is calculated only once.

As there are $\frac{(2n-2)!}{n!\cdot(n-1)!}$ different binary trees of n leaves, a dynamic programming approach runs in $O(\frac{(2n-2)!}{n!\cdot(n-1)!} + m \cdot \sqrt{n} \cdot 2^n)$ which can be roughly reduced to $O(n! + m \cdot \sqrt{n} \cdot 2^n)$ and thus $O^*(n!)$.

[We can analyze this differently as well: each step, except for the root, we are technically trying to decide what leaves will be placed in the parent set, the left set and the right set. This means that there are 3^n possible choices for a node. This implies this is a $O^*(3^n)$ algorithm.]

In the following algorithm, the global variable MMdict is a dictionary storing the calculated optimal subtrees and the mm-width of these.

Algorithm 2: Dynamically Programmed MM-width

Data: a graph g
Result: MM-width of the graph and a binary decomposition tree of that score
$result \longleftarrow MMdynamicRecurse(g, \emptyset)$;
return *result*

Algorithm 3: MMdynamicRecurse

Data: a graph g
the set of vertices not in the subtree *parentSet*
$SetToSplit \longleftarrow parentSet.Complement()$;
if $SetToSplit.Count = 1$ **then**
  return *new Tuple(1, SetToSplit[0].Associated)*
if $SetToSplit.Count = 2$ **then**
  return *new Tuple(1,*
  "(" + SetToSplit[0].Associated + ", " + SetToSplit[1].Associated + ")")
$BestScore \longleftarrow int.MaxValue$;
$BestTree \longleftarrow$ " ";
foreach *subset S of SetToSplit* **do**
  if $S.Count = 0 \, or \, (S \cup parentSet).Count = g.vertices.Length$ **then**
    continue;
  $scoreSubL \longleftarrow -1$;
  if *MMdict.ContainsKey(S)* **then**
    $scoreSubL \longleftarrow MMdict[S].Item1$;
    $tree \longleftarrow$ "("+$MMdict[S].Item2 +$ ",";
  else
    $tupleL \longleftarrow MMdynamicRecurse(g, S.Complement())$;
    $scoreSubL \longleftarrow tupleL.Item1$;
    $tree \longleftarrow$ "("+$tupleL.Item2 +$ ",";
  $leftEdge \longleftarrow HopcroftKarp(G[S, \bar{S}])$;
  $scoreL \longleftarrow leftEdge$;
  if $scoreL < scoreSubL$ **then**
    $scoreL \longleftarrow scoreSubL$;
  $scoreSubR \longleftarrow -1$;
  $rightSubSet \longleftarrow (S \cup parentSet).Complement()$;
  if $MMdict.ContainsKey(rightSubSet)$ **then**
    $scoreSubR \longleftarrow MMdict[rightSubSet].Item1$;
    $tree \longleftarrow tree + MMdict[rightSubSet].Item2 +$ ")";
  else
    $tupleR \longleftarrow MMdynamicRecurse(g, S \cup parentSet)$;
    $scoreSubR \longleftarrow tupleR.Item1$;
    $tree \longleftarrow tree + tupleR.Item2 +$ ")";
  $rightEdge \longleftarrow HopcroftKarp(G[rightSubSet, right\bar{S}ubSet])$;
  $scoreR \longleftarrow rightEdge$;
  if $scorer < scoreSubR$ **then**
    $scoreR \longleftarrow scoreSubR$;
  $score \longleftarrow scoreL$;
  if $score < scoreR$ **then**
    $score \longleftarrow scoreR$;
  if $score < BestScore$ **then**
    $BestScore \longleftarrow score$;
    $BestTree \longleftarrow tree$;
$result \longleftarrow newTuple(BestScore, BestTree)$;
$MMdict[parentSet.Complement()] \longleftarrow result$;
return *result*

7 MM-Width of Standard Graphs

Now that we have an algorithm with which we can more reliably calculate the MM-width of graphs, we can start calculating the MM-widths of different graphs. If we can identify what kind of graph we are working with however, we may be able to forgo having to calculate the MM-width altogether. To be able to do this, we will analyze different types of graphs and reason about those graphs to figure out how to construct an optimal binary decomposition tree directly, without having to construct a large set of decompositions and choosing the minimum from those.

Earlier work towards quickly finding MM-width was done by Jeong in [17] by using a minor obstruction set, where they characterized graphs of MM-width ≤ 2. We will approach finding the MM-width of these standard graphs differently.

7.1 Single Vertex / Trivial Graph

A graph containing a single vertex is also known as a trivial graph. A trivial graph has only one way of being decompositioned into a binary decomposition tree as there is only one binary tree with a single leaf. In addition, using the original definition of MM-width, when calculating the mm score of the single node of this binary decomposition tree, the resulting induced bipartite graph will have a single vertex on one of the sides and nothing on the other. Because of this, the mm-width of the single possible binary decomposition tree for the trivial graph is 0.

Thus, for a trivial graph G, $MM(G) = 0$.

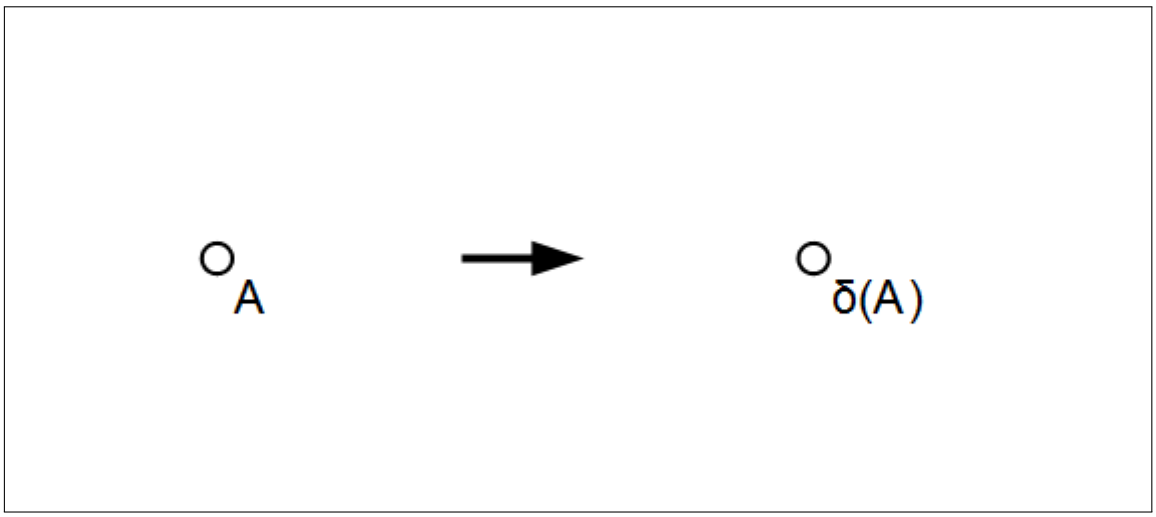

Figure 6: A trivial graph and the only possible tree decomposition of it.

7.2 Single Edge

A graph consisting of a pair of connected vertices has exactly two ways of being decompositioned into a binary decomposition tree, though these two trees are actually equivalent trees with the order of siblings swapped. Both decomposition trees have two edges, which both create the same partition when using the alternative definition of MM-width: $\mathcal{P}_{(\delta(A),r)} = \mathcal{P}_{(\delta(B),r)} = \{A : B\}$. The induced bipartite graph for

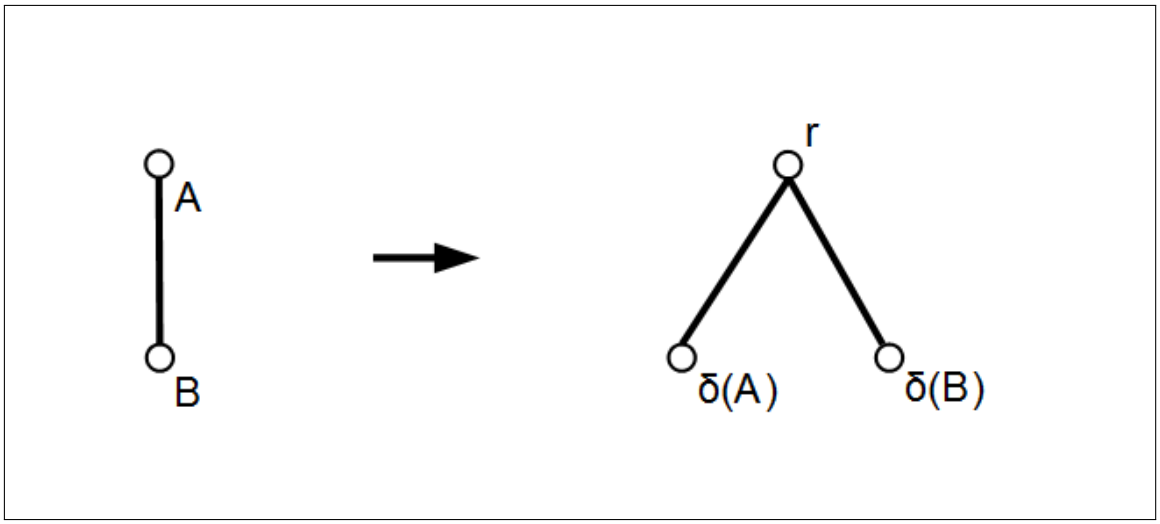

Figure 7: A graph containing a single edge and one of the two possible tree decompositions of the graph.

this partition is the exact same as the original graph and has a maximum matching of 1.

Thus, for a single edge graph G, $MM(G) = 1$. In addition, it is easy to see that any graph containing at least one edge must have MM-width at least 1.

7.3 Path Graph / Linear Graph

A path graph, also known as a linear graph, is a graph with a single connected component and in which all but two vertices have a degree of 2. Those two remaining vertices are known as the terminal vertices and have a degree of 1.

Theorem 1. *Let graph G be a path graph, then $MM(G) = 1$.*

Proof. We can construct an optimal decomposition $(T, \delta)^*$ for a path graph G as such: In $(T, \delta)^*$ we place a leaf node that is associated to one of the terminal vertices v of G. We add an internal node to T and make it the parent of the node associated to v. Then we add a leaf node $\delta(w)$ associated to the neighbor of v, w, as a left child of the inner node.

As long as $(T, \delta)^*$ does not contain an associated node for each vertex of G, we add internal nodes and leaf nodes to $(T, \delta)^*$ and to it we associate the vertex we have not associated yet that neighbors the previous vertex that we added. The new internal node becomes the parent node of the previously added internal node, while the new leaf node associated to the vertex we are handling becomes the left child of the new internal node. We finish the construction when we have added the terminal vertex on the other side of the path graph. The internal node without a parent, and thus which left child is the leaf node associated to the terminal vertex, becomes the root of the decomposition tree.

As each partition we can make over the vertices of G by removing one edge from $(T, \delta)^*$ either separates a single vertex from the rest of the vertices, or creates a partition over $V(G)$ where both sets of the partition contain a terminal vertex and for each vertex in the same set as a terminal vertex all vertices on the path between that vertex and the terminal vertex are in the same set, the MM-width of $(T, \delta)^*$

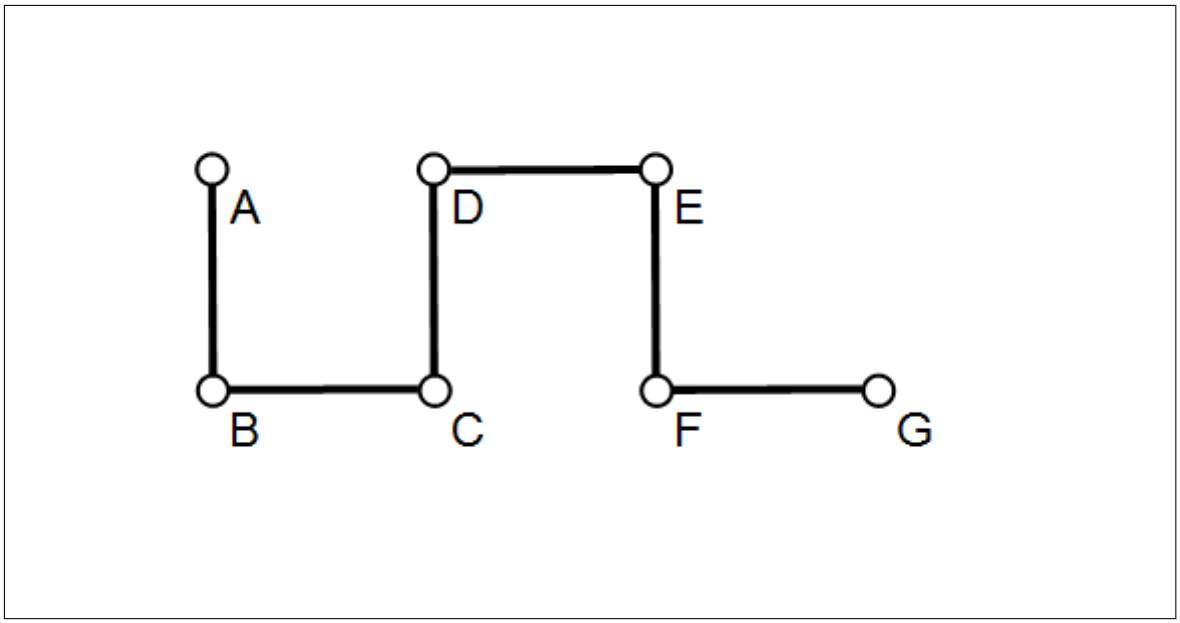

Figure 8: A path graph of length 7.

is 1. In addition, any graph containing at least one edge must have MM-width at least 1, meaning that for a path graph G, $MM(G) = 1$. □

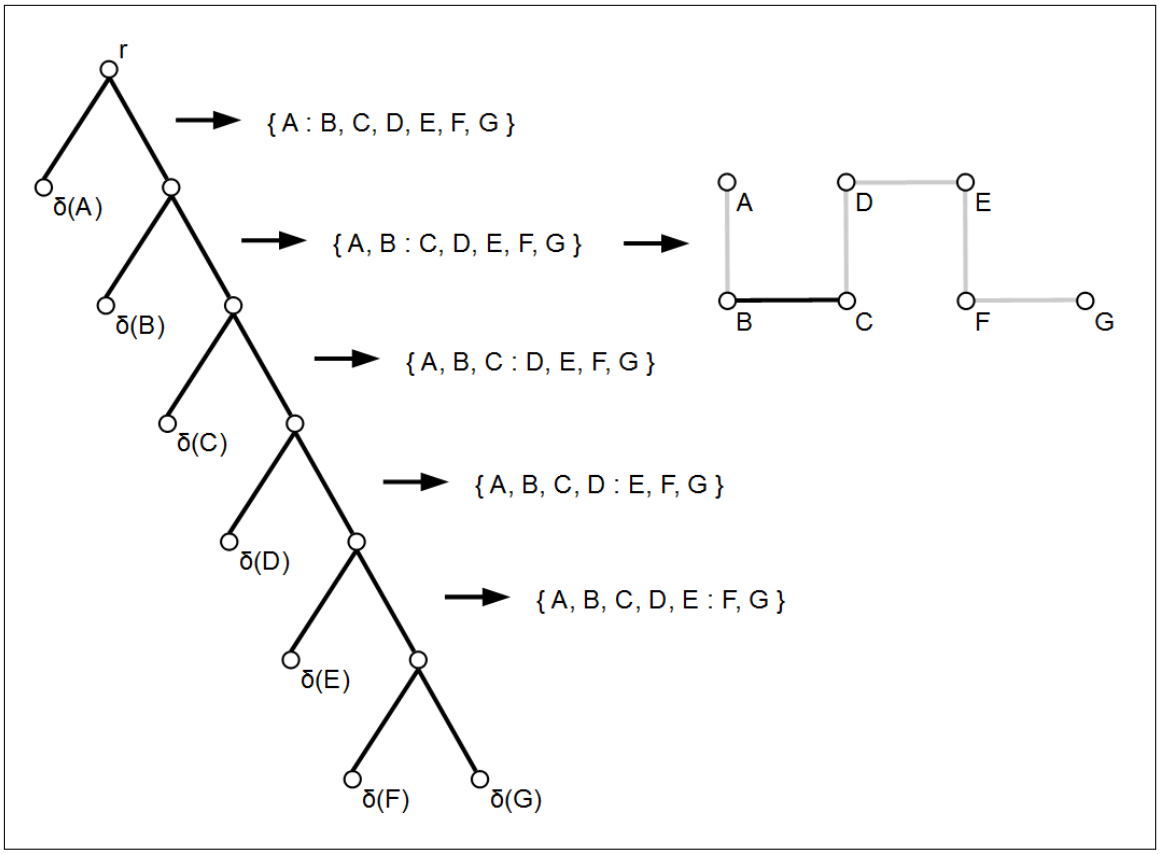

Figure 9: A linear decomposition of the path graph in figure 8, the induced partitions of some of the edges of the decomposition tree and one of the induced bipartite graphs of the partitions.

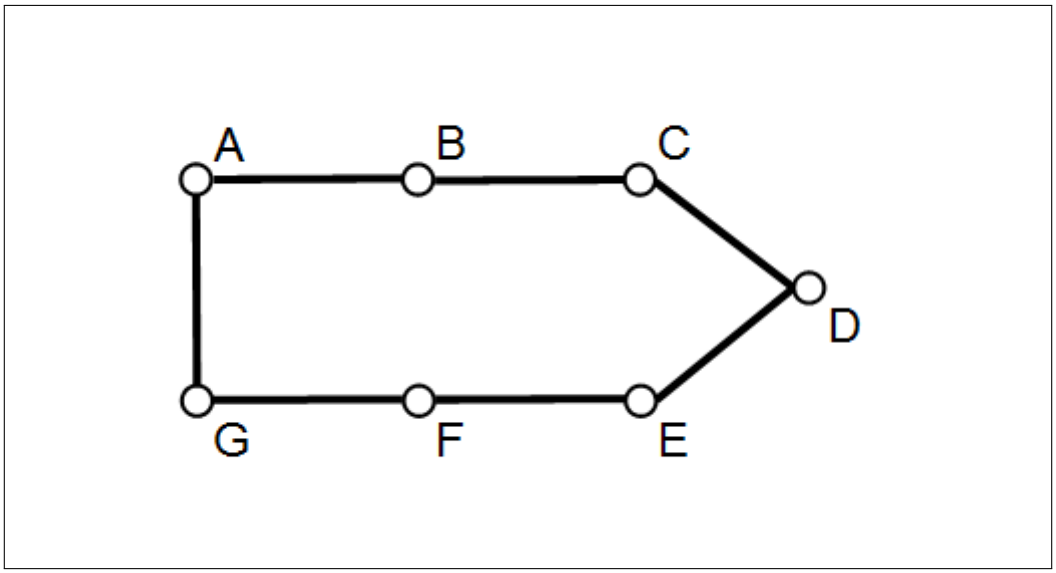

Figure 10: A cycle graph consisting of 7 vertices.

7.3.1 Caterpillar Decomposition

A binary tree decomposition like $(T, \delta)^*$ we constructed above is known as a caterpillar decomposition (or linear decomposition).

Definition 20. *(Caterpillar Decomposition)*
A binary tree decomposition is a caterpillar decomposition *if and only if each internal node of the tree, including the root, has at least one leaf node as a child.*

Because of the equivalence of ordering of siblings, we will assume that for each of the internal nodes of a caterpillar decomposition, the left child is always a leaf node, while the right node is either another internal node or the final leaf node. In addition, when making this assumption, the leaf nodes have a natural order over them from left to right.

7.4 Cycle Graph / Circular Graph

A graph consisting of a single cycle is known as a cycle graph. Such a graph consists of a single connected component in which all vertices have a degree of 2. A cycle graph consisting of n vertices is denoted with C_n. The smallest cycle graph following this definition is C_3.

Theorem 2. $MM(C_3) = 1$.

Proof. C_3 consists out of three vertices, thus $|V(G)| = 3$. For this graph, there are two distinct ways of partitioning the vertices: either all three vertices are on the same side of the partitioning, or there are two vertices on one of the sides and one vertex on the other. As the latter of these will always occur in any decomposition tree we can build for C_3, the mm-width will always be 1 for any of these trees. Therefore, $MM(C_3) = 1$. □

The same argument does not hold for cycle graphs with 4 or more vertices however. In fact, these graphs have a MM-width of 2.

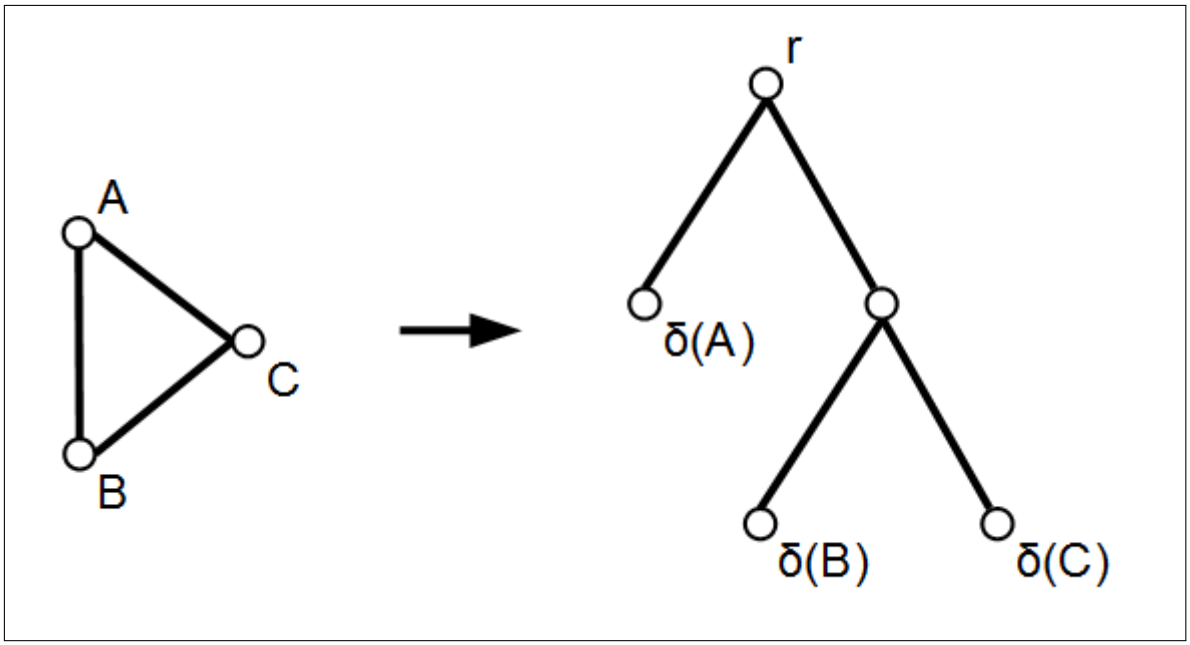

Figure 11: C_3 and an optimal decomposition tree of it.

Theorem 3. *Let graph G be a cycle graph, where $|V(G)| \geq 4$, $MM(G) = 2$.*

Proof. To construct an optimal binary tree decomposition $(T, \delta)^*$ for G, we will use a caterpillar decomposition. For the ordering of the leaf-vertex association, we will start at a random vertex $v \in V(G)$ and associate it to the first leaf node. Then we take either of the neighbors of v and associate it to the next leaf node. We go around the cycle until we reach the other neighbor of v, while associating each vertex to the following leaf node.

To find the mm-width of $(T, \delta)^*$, we will look at the partitions induced by the edges of this tree: For the partitions with a single vertex in one of the sets, the results of the function mm are 1. For the other partitions, there will be at least two vertices in both sets. Because of the construction of $(T, \delta)^*$, we know that for both sets the induced subgraphs of the set must be connected, which in turn means that there are exactly two edges with unique endpoints connecting vertices from one set to the other, making the mm score of these edges equal to 2. Thus we know that $mm((T, \delta)^*) \leq 2$ and $MM(G) \leq 2$.

In addition, for cycle graphs containing at least 4 vertices, we know that when making any decompositioning of the graph, there must be at least one partition containing two vertices in one of the sets. Both of these vertices must have a unique neighbor in the other set, meaning that there are two edges with unique endpoints connecting vertices from one set to the other. Because of this, $MM(G) \geq 2$. □

In conclusion, for cycle graph G, if $|V(G)| \geq 4$, $MM(G) = 2$. Otherwise, $MM(G) = 1$.

7.5 Tree

We defined a tree earlier in the definitions section on page 4. We will be assuming G is a rooted tree. If it is not a rooted tree, we will temporarily choose a vertex of

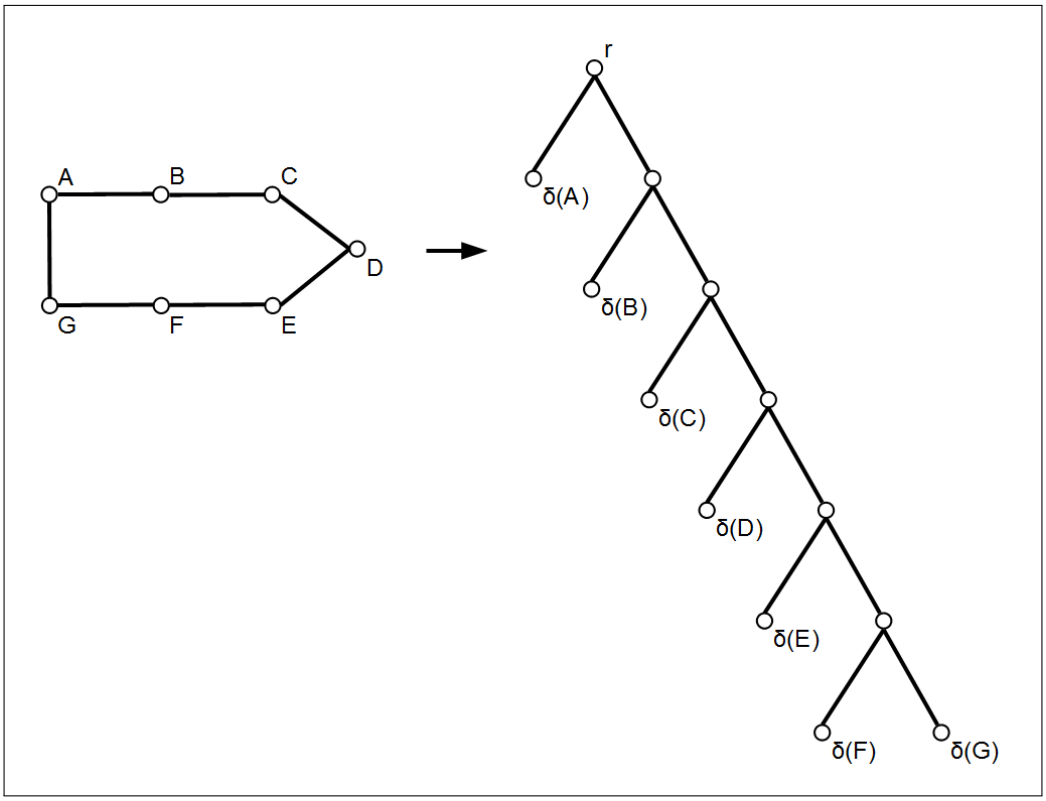

Figure 12: The cycle graph of 7 vertices and a caterpillar decomposition of it.

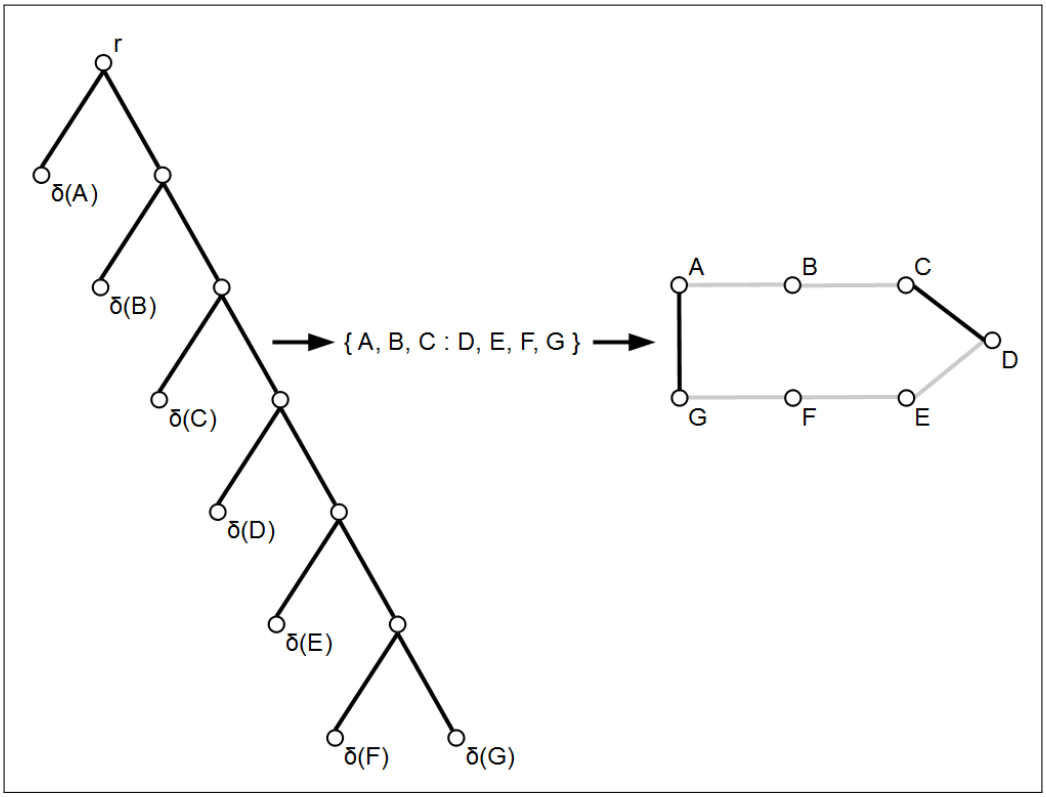

Figure 13: A linear decomposition of the cycle graph in figure 12 and the partition and induced bipartite graph of one of the edges of the decomposition tree.

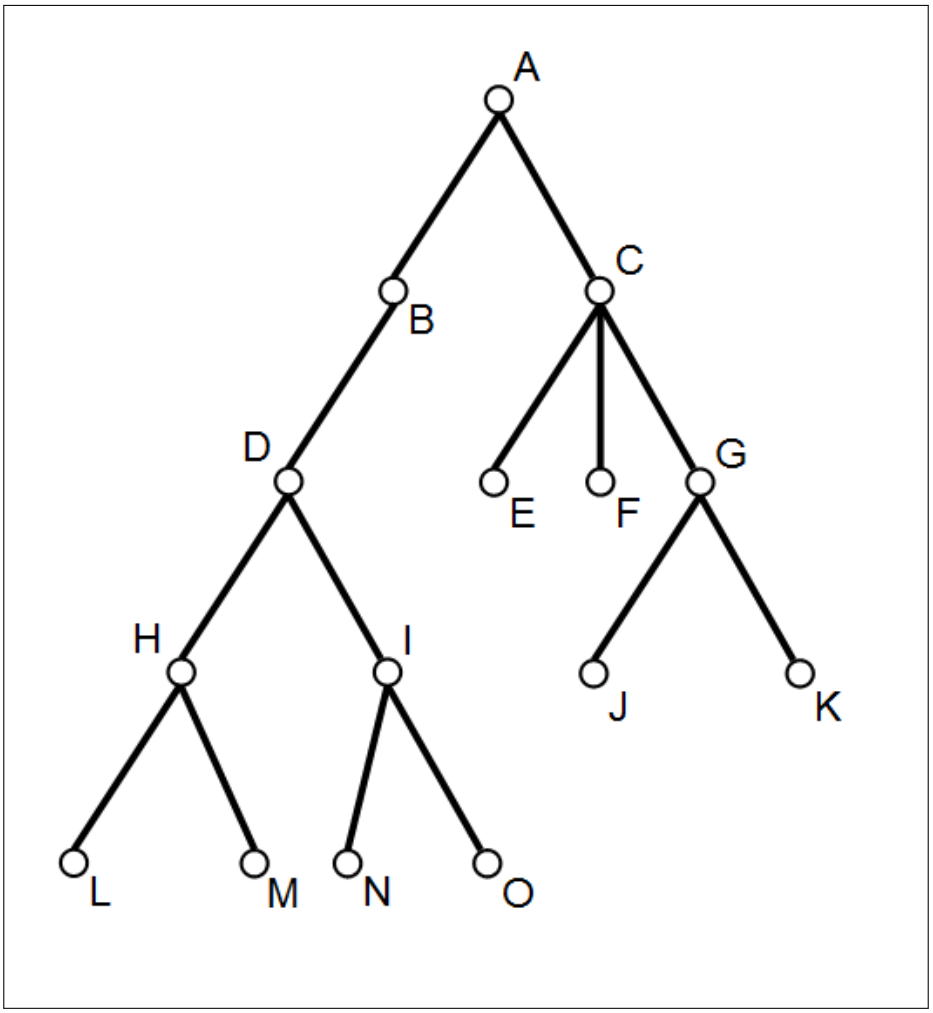

Figure 14: An example of a tree.

G to be the root node and establish parent and child links throughout the tree to make it a rooted tree.

To find an optimal binary tree decomposition $(T, \delta)^*$ of a tree G, we will be using a gadget g. This gadget will replace each of the vertices $v_n \in V(G)$, where gadget g_n replaces v_n. If v_n is a leaf of G, then g_n is a single leaf node associated to v_n. Otherwise, g_n consists of two parts: a base part and a linking part. The base part consists of an internal node and a leaf node associated to v_n. The linking part is a binary tree with a number of leaves equal to the amount of children of v_n.

When linking the gadgets together, the leaves of the linking part are merged with the highest nodes of the child gadgets, as can be seen in figure 16.

Theorem 4. *$(T, \delta)^*$ constructed from a tree G by replacing each vertex v_n with gadget g_n has mm-width 1 .*

Proof. As the mm-width of a decomposition tree equals the maximum mm score over all the edges, we will need to analyze all the edges we have introduced to construct $(T, \delta)^*$. Each of the edges that we have introduced were from the gadget we used, thus we only need to look at the edges we find in the gadget.

As stated earlier, the gadget consists out of two parts: a base part and a linking part. The base part consists out of two edges, of which one is incident to the leaf

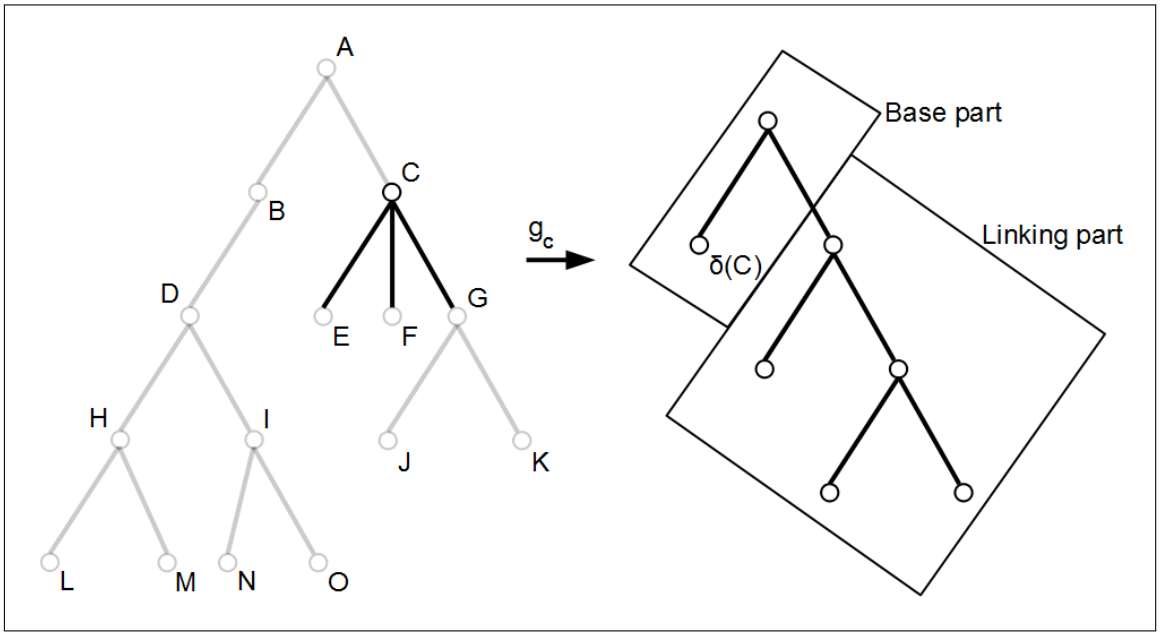

Figure 15: An example of one of the gadgets that we use to construct the tree decomposition of the tree in figure 14. We have indicated the base part and linking part of the gadget.

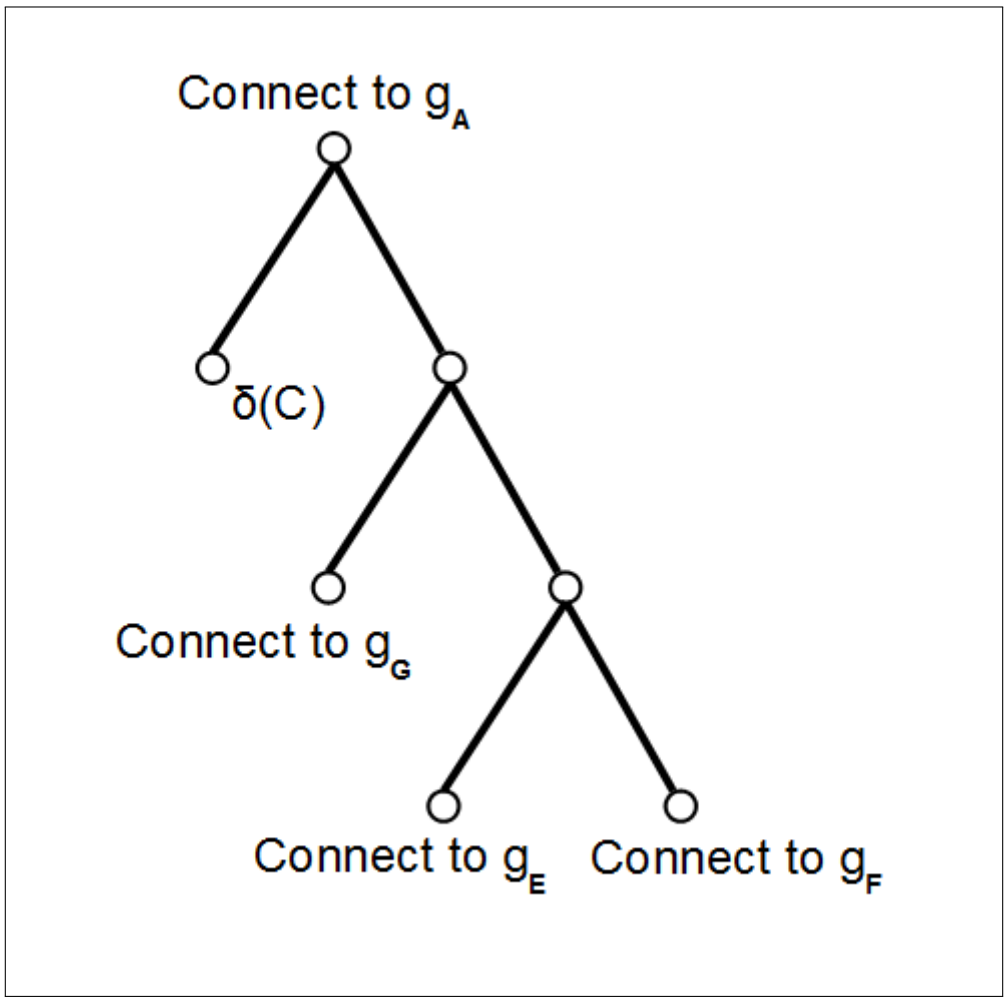

Figure 16: How the gadget in figure 15 links up to other gadgets.

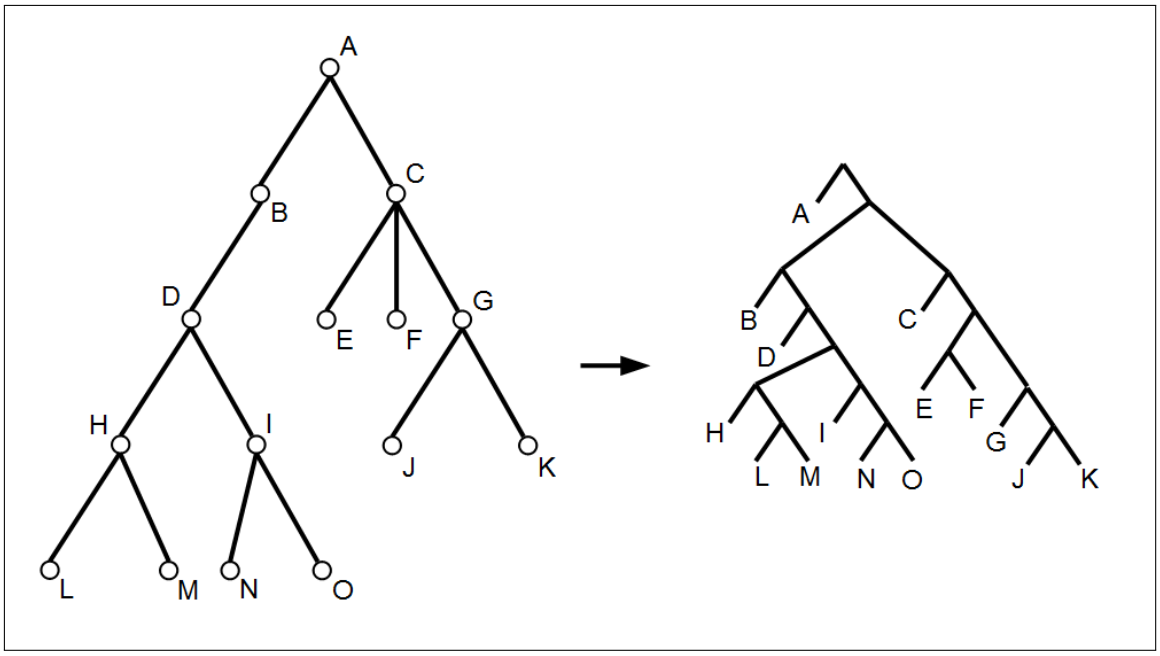

Figure 17: One of the possible tree decompositions of the tree in figure 14.

node and one connects the leaf node's parent with the leaf node's sibling. We will assume that we are looking at gadget g_n. The edge incident to the leaf node has a mm score of 1 as the induced partition only has v_n in one of the sets.

The other edge induces a partition with in one set all vertices with v_n as an ancestor and in the other set all other vertices of the graph (including v_n). This edge must also have a mm score of 1 as v_n, the only vertex of its set in the partition that can match, can only match with one of its children.

In the linking part, we get a similar situation to the second edge of the base part. Each edge induces a partition between some of the subtrees of the children of v_n and the rest of the graph. These edges must also have a mm score of 1, again because it is the only vertex of its set in the partition that can match.

The gadgets that replace a leaf of G do not introduce any edges and therefore do not have to be analyzed here.

In conclusion, all edges result in mm scores of 1, meaning that the MM-width of $(T, \delta)^*$ is 1. □

7.6 Cactus Graph

A cactus graph can be seen as a tree where some of the edges have been replaced with cycles. Any graph in which any two simple cycles have at most one vertex in common is a cactus graph.

You can "grow" you own cactus graph G as such: Start with either a tree or a cycle graph. Then, choose any vertex $v \in V(G)$ and to it we attach either a path or a cycle of any size. The paths and cycles are the *features* of the cactus graph. Repeat choosing vertices and attaching parts to the graph. G remains a cactus graph between adding features.

Seeing how cactus graphs consist out of trees and cycles, we would expect the MM-width of these graphs to be at most 2 and this is exactly what we find. However,

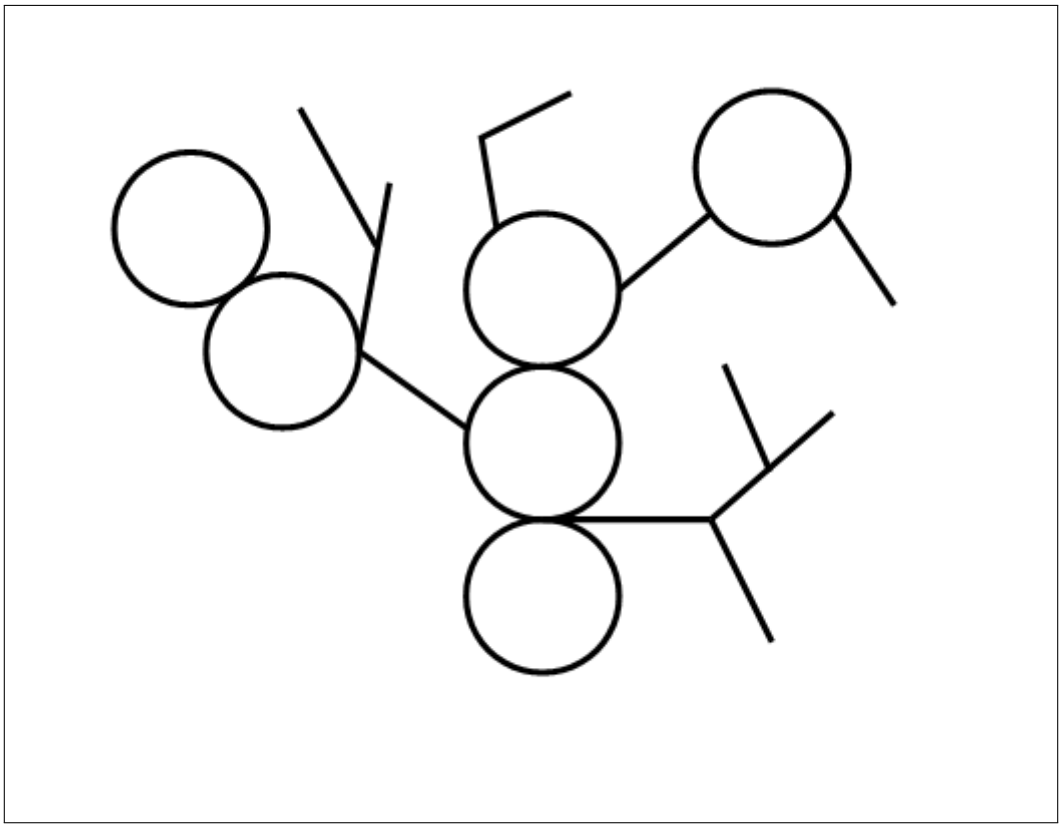

Figure 18: Rough example of a cactus graph

we will assume there is at least one cycle feature in the graph, as the graph is a tree otherwise, meaning that the MM-width would be 1.

Theorem 5. *Let graph G be a cactus graph containing at least one simple cycle, then if at least one cycle feature of G contains at least 4 vertices $MM(G) = 2$, else $MM(G) = 1$.*

Proof. To construct $(T, \delta)^*$ for G, we will be choosing a random vertex v on a cycle in G. Starting at v, we will create a caterpillar decomposition of this cycle similar to how we built a decomposition tree for a cycle graph. However, for each vertex w in the cycle that has other features attached to it, we will do the following: we will split the edge incident to the associated leaf of w to create a new internal node. If w is the common vertex of more than two features, we will create a binary tree to with a number of leaves equal to one less than the number of features w is the common vertex of. Then, to each of the leaves of this binary tree, or the internal node in case that w is the common vertex of exactly two features, we attach caterpillar decompositions starting from w of the corresponding branching feature.

In each vertex of the additional caterpillar decompositions that is the common vertex of multiple features, we will also split the edge and also attach additional caterpillar decompositions. We continue doing this splitting and adding in new caterpillar decompositions until all vertices of G have associated leaves in $(T, \delta)^*$. Finally, we contract all vertices of degree 2, except for the root, to make it a valid binary decomposition tree.

We show a rough example of the construction of a cactus graph decomposition in figures 19 through 23.

To find the mm-width of $(T, \delta)^*$, we will again look at all the edges we have created: For each edge incident to a leaf, the mm score is 1, as for each of these there is only a single vertex in one of the sets of the induced partition. For each edge we have created to connect the caterpillar decompositions together, the mm score is also 1, as any two features in the graph have at most one vertex in common, meaning that it can make at most 1 matching in the induced bipartite graph.

Finally, for the edges along the length of the caterpillar decompositions, the mm score is 1 for the caterpillar decompositions that were made for path features, while the mm score is ≤ 2 for the caterpillar decompositions that were made for cycle features. This is so, because the partitions made on these edges divide the vertices of G by removing a single edge of G for the path decompositions, while the partitions divide G by removing two edges of G for the cycle decompositions. [If such a cycle found in G has 3 vertices though, the two edges that the graph is partitioned are directly adjacent, meaning only one matching can be made in the induced bipartite graph and giving those edges of the tree decomposition mm score 1.

Thus, all edges of $(T, delta)^*$ have mm score ≤ 2, except when there is no cycle of four vertices or more, in which case all edges of $(T, \delta)^*$ have mm score 1. □

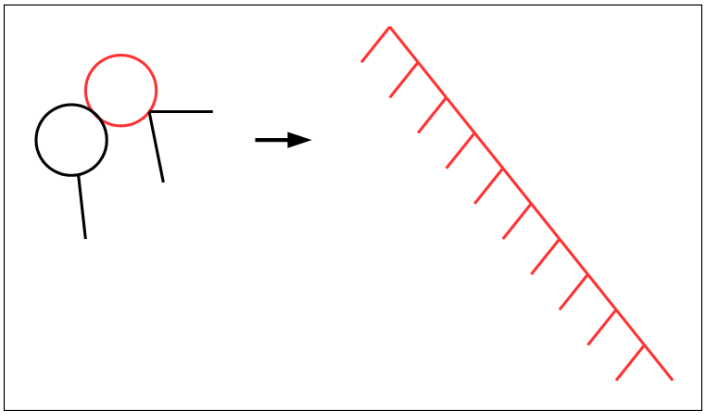

Figure 19: We start with a caterpillar decomposition of one of the cycles.

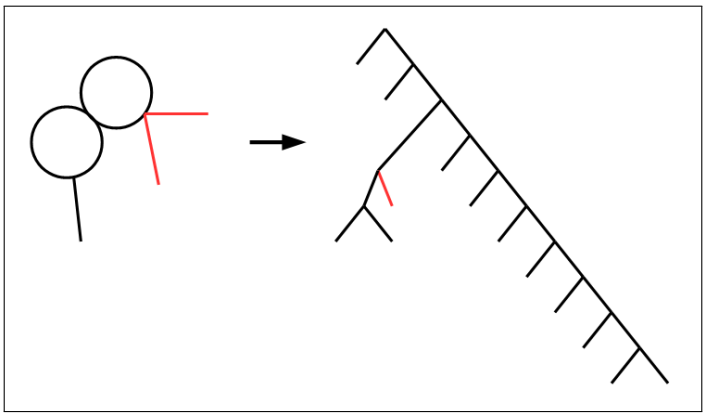

Figure 20: We find two path features attached to the same vertex on this cycle and thus split the edge incident to the leaf node associated to that vertex and make two places to attach decompositions for the two path features.

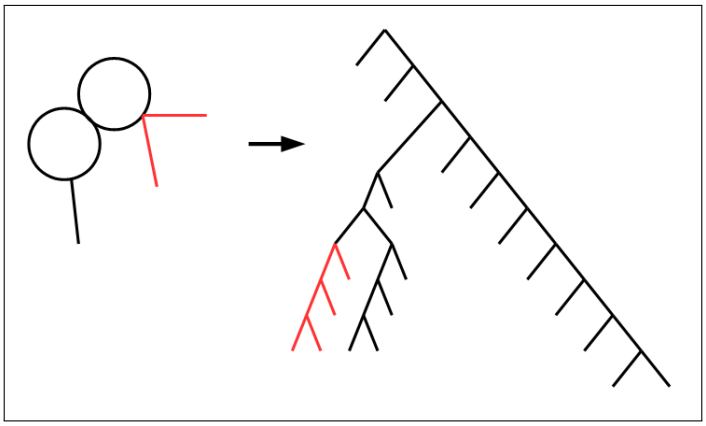

Figure 21: We attach both decompositions of the two path features to the decomposition tree.

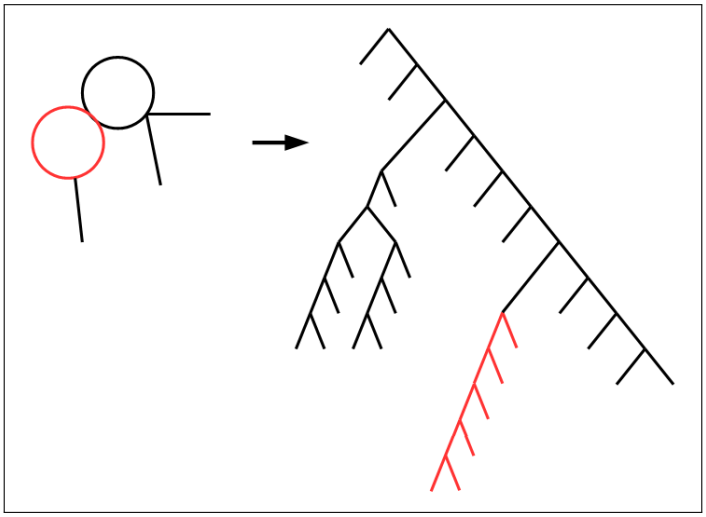

Figure 22: We also find a cycle feature and attach a decomposition for it.

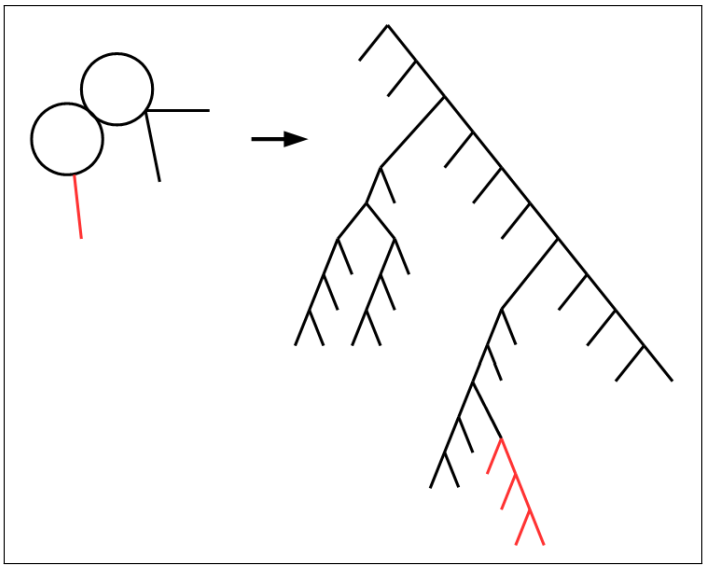

Figure 23: Finally we also add a decomposition for the final feature.

7.7 Clique Graph / Complete Graph

A clique graph is a graph where every vertex is connected with every other vertex. Because every possible edge in the graph is present in the graph, it is also known as a complete graph.

To find the MM-width of a clique graph, let us first look at the maximum matching of a clique consisting of n vertices. Seeing how we can connect any of the unmatched vertices to any other unmatched vertex, we would be able to make $\lfloor n/2 \rfloor$ matchings, before we either end up with a single final unmatched vertex or all vertices matched. This directly implies that for a clique graph G, $MM(G) \leq \lfloor n/2 \rfloor$.

We can actually give a stricter bound though, by using the structure of binary decomposition trees.

Theorem 6. *Let graph G be a clique graph, then $MM(G) = \lceil n/3 \rceil$, where $n = V|(G)|$.*

Proof. First we divide the n vertices over three sets, A, B and C, such that $A \cup B \cup C = V(G)$. We will put any $\lceil n/3 \rceil$ vertices in A, we will put $\lceil (n - \lceil n/3 \rceil)/2 \rceil$ vertices in B and C will contain the rest of the vertices. [Thus, if n is divisible by three: $|A| = |B| = |C|$. Else if $n + 1$ is divisible by three: $|A| + 1 = |B| = |C|$. Otherwise: $|A| = |B| = |C| - 1$.]

Then, to construct (T, δ), we will create a root node and make its left child the root node of any binary tree decomposition over the vertices in set A. We also make an internal node and make it the right child of the main root node. The left child of this internal node will be the root node of any binary tree decomposition over the vertices in set B and the right child the root node of any binary tree decomposition over the vertices in set C.

For all the edges in the binary tree decomposition we built for set A, we know that the result of the function mm will not be larger than $\lceil n/3 \rceil$, because for all the partitions induced by the edges there are at most $\lceil n/3 \rceil$ vertices on one of the sides. Similarly, for all the edges in the binary tree decomposition we built for set B the score will never exceed $\lceil (n - \lceil n/3 \rceil)/2 \rceil$ and for all the edges in the binary tree decomposition we built for set C the score will never be larger than $n - \lceil (n - \lceil n/3 \rceil)/2 \rceil - \lceil n/3 \rceil$.

The edges incident to the root and the internal node that we placed initially will be the highest scoring edges in (T, δ), having the highest scores from the binary tree decompositions we built for the sets: the two edges around the root will have score $\lceil n/3 \rceil$, the edge to the internal node's left child will have score $\lceil (n - \lceil n/3 \rceil)/2 \rceil$ and the edge to the internal node's right child will have score $n - \lceil (n - \lceil n/3 \rceil)/2 \rceil - \lceil n/3 \rceil$.

Thus, each edge in (T, δ) has a mm score $\leq \lceil n/3 \rceil$, thus $MM(G) \leq \lceil n/3 \rceil$.

Now, say that we could construct a binary tree decomposition $(T, \delta)'$ with $mm((T, \delta)') < \lceil n/3 \rceil$. This directly implies that there must be no edges in $(T, \delta)'$ where the induced partition has $\lceil n/3 \rceil$ or more vertices in the smallest of the sets.

Say $(u, v) \in E(T)$ is the edge with the highest induced mm score of $W < \lceil n/3 \rceil$ and say vertex u is on the side of (u, v) with W leaves. Then there must be $n - W$ leaves on v's side of (u, v). Now, denoting the edges incident to v as (u, v), (v, w) and (v, x), the $n - W$ must be divided over the the parts of the tree beyond w and beyond x. As (u, v) has the highest mm score of the tree, (v, w) has at most W leaves beyond it, leaving at least $n - 2W$ leaves to be beyond (v, x).

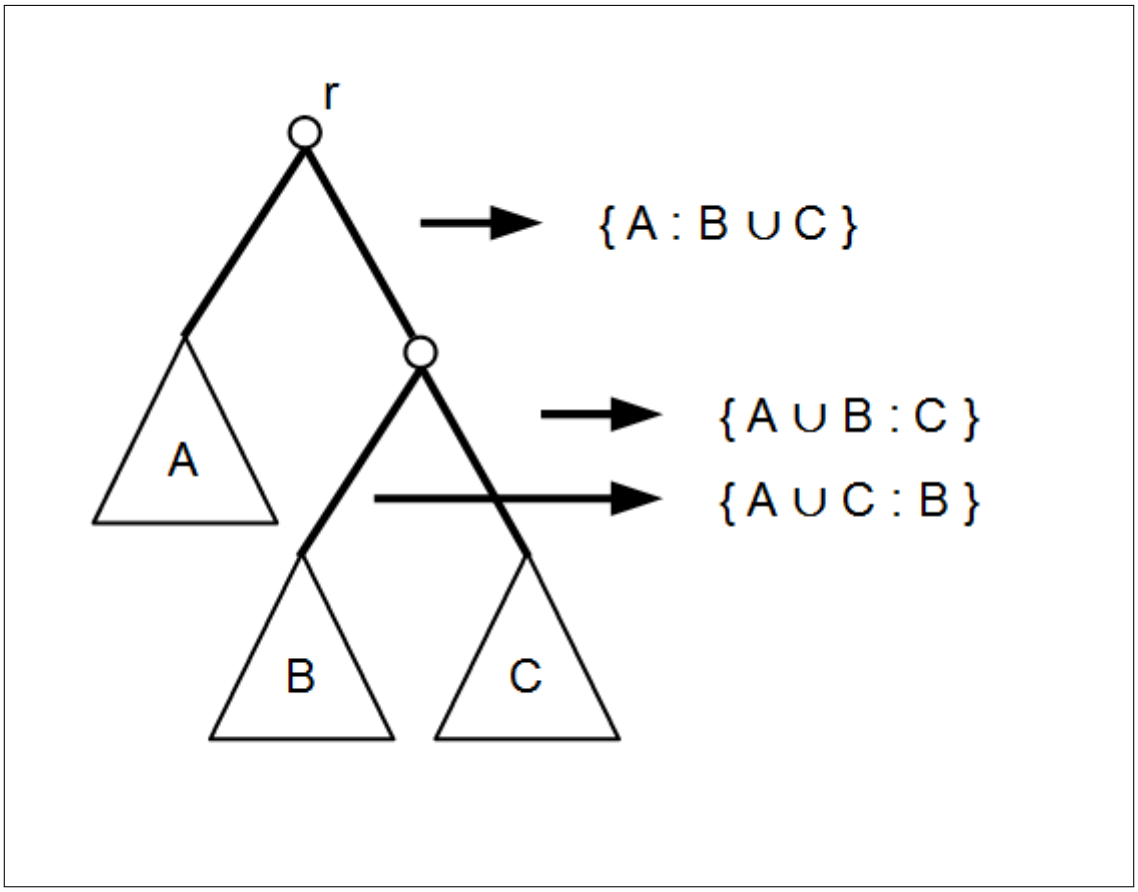

Figure 24: Example of the binary decomposition tree together with the partitions induced by the central three edges. Each triangle indicates a binary decomposition of a set.

If we assume we have W leaves beyond (v, w), then beyond (v, x) there are exactly $n-2W < n-2(\lceil n/3 \rceil - 1) = n+2-2\lceil n/3 \rceil$ vertices. As $n+2-2\lceil n/3 \rceil > \lceil n/3 \rceil$, (v, x) has a higher score than (u, v), leading to a contradiction. Meaning that we cannot build a tree decomposition with mm-width $< \lceil n/3 \rceil$

Because $MM(G) \leq \lceil n/3 \rceil$ and $MM(G) \geq \lceil n/3 \rceil$, $MM(G) = \lceil n/3 \rceil$. □

7.7.1 Upper Bound to MM-width

Using what we have proven above, we are able to derive an upper bound on the value of the MM-width of a graph.

Theorem 7. *For graphs G, H; if $H \subseteq G$, $MM(H) \leq MM(H)$.*

Proof. Say we have constructed any binary decomposition tree (T, δ) of a graph G and we have a graph $H \subseteq G$. If we also use (T, δ) as a binary decomposition tree for H, each edge $e \in E(T)$ induces the same partition $\mathcal{P}_e$ over the vertices of $V(H)$ as it induces over the vertices of $V(G)$, though H may miss some vertices present in G. Assuming those vertices are present however, as $E(H) \subseteq E(G)$, $mm_H(\mathcal{P}_e) \leq mm_G(\mathcal{P}_e)$, where mm_G is the function mm as applied over graph G and mm_H is the function mm as applied over graph H. The removal of the missing vertices can also only decrease the number of matchings that can be made in the

partitions, meaning that for all binary decomposition trees the mm-width must have either lowered or stayed equal, meaning that the minimum binary decomposition tree also followed that trend. □

Theorem 8. *The MM-width of any graph is less than or equal to* $\lceil n/3 \rceil$.

Proof. Assume we have any graph I with $|V(I)| = n$. As we can find a clique graph $J \supseteq I$ by adding edges between all vertices in I and we know that J has a MM-width of $\lceil n/3 \rceil$, $MM(I) \leq \lceil n/3 \rceil$. □

7.8 K-Tree

One of the possible ways of generalizing trees is done by defining them through cliques to create the definition of the k-tree. Following the definition from [12], a graph G is a k-tree if it is either a clique of k vertices, or a graph that contains a vertex v of which the neighborhood in G induces a subgraph that is a clique of k vertices and the removal of v from G results in a k-tree.

Following this definition, the tree we defined earlier is a 1-tree: each of the leaves of the tree has a neighborhood of size 1 (trivially a clique) and we can keep removing vertices of degree 1 from the 1-tree until we end up with with a single vertex.

To construct a k-tree G, we follow the definition backwards: we start with a clique graph consisting of k vertices. Then, we repeat the following an arbitrary number of times: We add a new vertex v to G and select a vertex $w \in V(G)$. Then we select $k-1$ vertices from $N(w)$ (these automatically all neighbor each other as well). Finally we connect v with each of these vertices and w with an edge.

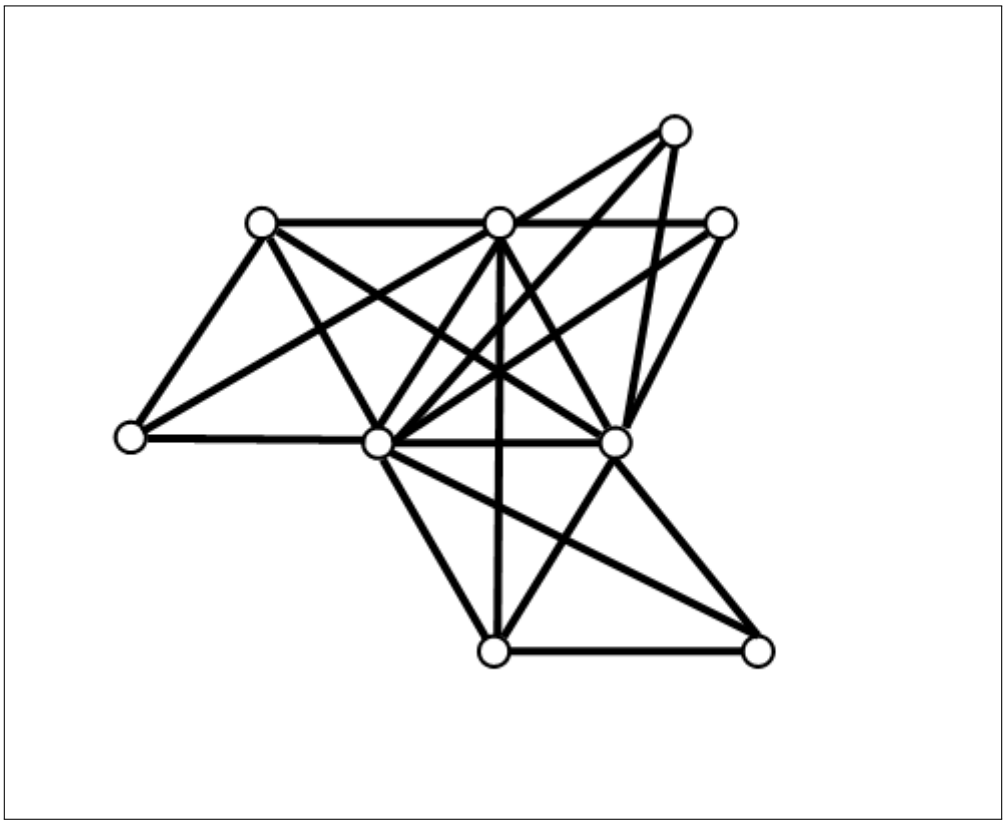

Figure 25: An example of a 3-tree.

In the constructed k-tree we can see the clique which we started with as the *root*. While the root is a single node in a rooted 1-tree, in a rooted k-tree the root is a set consisting out of k vertices. Similarly, we can also find *leaves* in k-trees. These are single vertices with degree k as such vertices are removed successively in the k-tree definition until a clique has been found. This means that the vertices of a k-tree that are not in the root clique, nor have degree k can be seen as *internal vertices/nodes*.

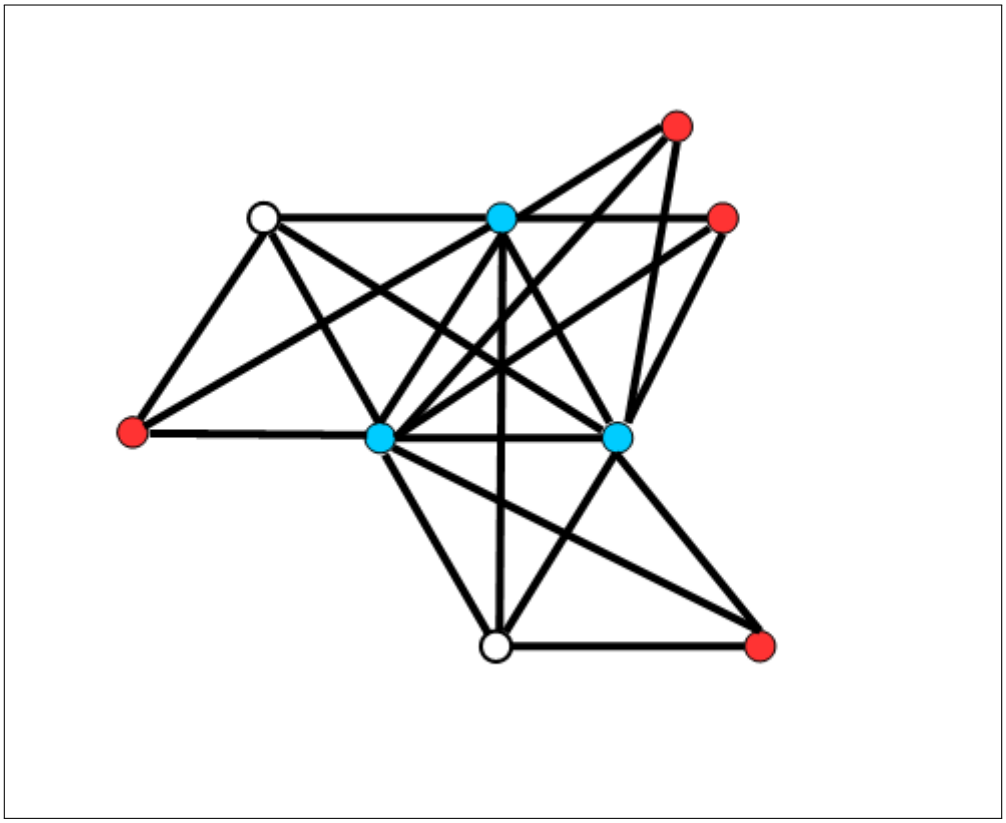

Figure 26: For the 3-tree of figure 25 we have indicated the leaf vertices in red and the three vertices of one of the many possible roots in blue.

7.8.1 MM-width of a K-Tree

To find the MM-width of a k-tree, we will again be building a binary tree decomposition. However, to construct this decomposition, we will be using a method using an advancing front. This front will start around one of the leaves of the k-tree and is moved over the graph. The vertices that are behind the front will have been added to the tree decomposition, while the vertices that are in front of the front do not have associated nodes in the binary tree decomposition yet.

This front that we will be using will be a *clique separator*, which is a special variation of a *separator*. When it stops being a separator during the construction, we have reached a terminating step.

Definition 21. *(Vertex separator)*
A set of vertices $S \subset V(G)$ for non-adjacent vertices u and v is a (u, v)-separator if and only if in $G[V(G) \setminus S]$ u and v are in different connected components.

Definition 22. *(Separator)*
A set of vertices $S \subset V(G)$, where G is a graph consisting of a single connected

component, is a separator if and only if for $H = G[V(G) \setminus S] : \exists u, v \in V(H)$ *such that* u *and* v *are in different connected components of* H*.*

Definition 23. *(Clique separator)*
A set of vertices $S \subset V(G)$*, where* G *is a graph consisting of a single connected component, is a clique separator if and only if for* $H = G[V(G) \setminus S] : \exists u, v \in V(H)$ *such that* u *and* v *are in different connected components and* $G[S]$ *is a clique graph.*

We will be proving that k-trees have MM-width of $\leq k$. This means that for k-trees with $3k$ or less vertices we do not have to prove anything more than we already have, as we have proven earlier that all graphs have MM-width $\leq \lceil n/3 \rceil$. To prove that the MM-width of a k-tree of at least $3k$ vertices is $\leq k$, we will follow an algorithm that we will first roughly sketch here:

The algorithm starts from one of the leaves of the k-tree. This vertex v will immediately get an associated leaf in the binary tree decomposition (T, δ) we are building and a parent node. Then we create the first front F for the algorithm: F will contain exactly each vertex neighboring v at first. Due to the graph being a k-tree, the neighbors of a leaf must be a clique, making F a clique separator.

The front F will be advancing over the graph during the algorithm, which means that we will add vertices to the front while removing others. The basic advance of the front goes as follows: when the front separates the graph in exactly two connected components, then we will choose a vertex y that neighbors all of the vertices in the front and is part of the connected component that does not contain the starting vertex. We will assume for now that $F \cup \{y\}$ also separates the graph into exactly two connected components. Next, we add y to F and choose a vertex x from F, such that $x \neq y$ and $F \setminus \{x\}$ still separates the graph into two connected components, then we remove x from F. We also add an extra leaf node to (T, δ) to associate to x. This new leaf node becomes the other child of the highest internal node of the decomposition tree and we add a new internal node as a parent to that internal node.

Not all advances will be like the basic advance though: the terminating advancing step for example. For the terminating advance, after adding the vertex to the front but before removing the other vertex from it, the graph will not be separated into multiple connected components. In this situation, we will just pick a random vertex from the front to remove (as long as it is not the vertex we just added) and add both an associated leaf and an extra internal node to the decomposition tree. This extra internal node will become the root of (T, δ) and from the other child of the root we will hang any binary tree of k leaves, where the k leaves are associated to the vertices in the final front in any way.

The front can also split the graph in more than two connected components. When this happens, we have found a branching point. To be precise, we have found a clique of size k (the front), from where three or more branches sprout. We cannot continue here as we would during a basic advance, as we would not associate the vertices to leaves in at least one of these branches if we did so. To resolve this, we select one of the branches to continue along and we will decomposition each of the other branches in the same way that we decomposition k-trees. The reason why we can decomposition the branches this way, is because each of the connected components we made by separating the graph with our front is by itself a k-tree when the front is attached to it:

Theorem 9. *$\forall C \in CC(G \setminus F)$, where F is a clique separator and G is a k-tree, $G[V(C) \cup F]$ is a k-tree.*

Proof. After removing all vertices that are in the other branches, we end up with a graph in which each vertex must have been added to k of its neighboring vertices that were a clique. As any vertex in this branch was not connected to anything outside this branch or the separator, these vertices do not lie outside the current graph. Thus the branch can be constructed in the same way as a k-tree, which means that $G[V(C) \cup F]$ is a k-tree. □

However, in this "sub-routine" of the algorithm we do need to enforce that we choose to add and remove vertices from the front of the sub-routine, such that the front of the sub-routine becomes the current front at some point. We do this to prevent a terminating step during the sub-routine.

After decompositioning the extra branches, we need to add these decomposition trees to (T, δ): For each of the branches we decompositioned, we first merge the highest internal nodes of (T, δ) and that branch's decomposition tree and then we add a new internal node as a parent to the merge node.

Finally, it is also possible that by a basic advance, $F \cup \{y\}$ separates the graph into more than two connected components. When this happens, removing any of the vertices of F from $F \cup \{y\}$ can cause us to skip branches and we will need to resolve the branches we will skip separately as well just before we remove a vertex that connects the skipped branches to what we already decompositioned. We will do this in a similar fashion to how we resolved the branches in the previous special case of the advance. However, due to the structure of this type of branching point, during the sub-routines we will need to work towards making the front of the sub-routine consisting out of vertices from $F \cup \{y\}$.

7.8.2 Example of K-Tree Decompositioning

To show how to construct the binary decomposition of a k-tree more clearly, we will be walking through a small example, which is demonstrated in figures 27 through 38 and a case not shown in this demonstration is shown in figure 39.

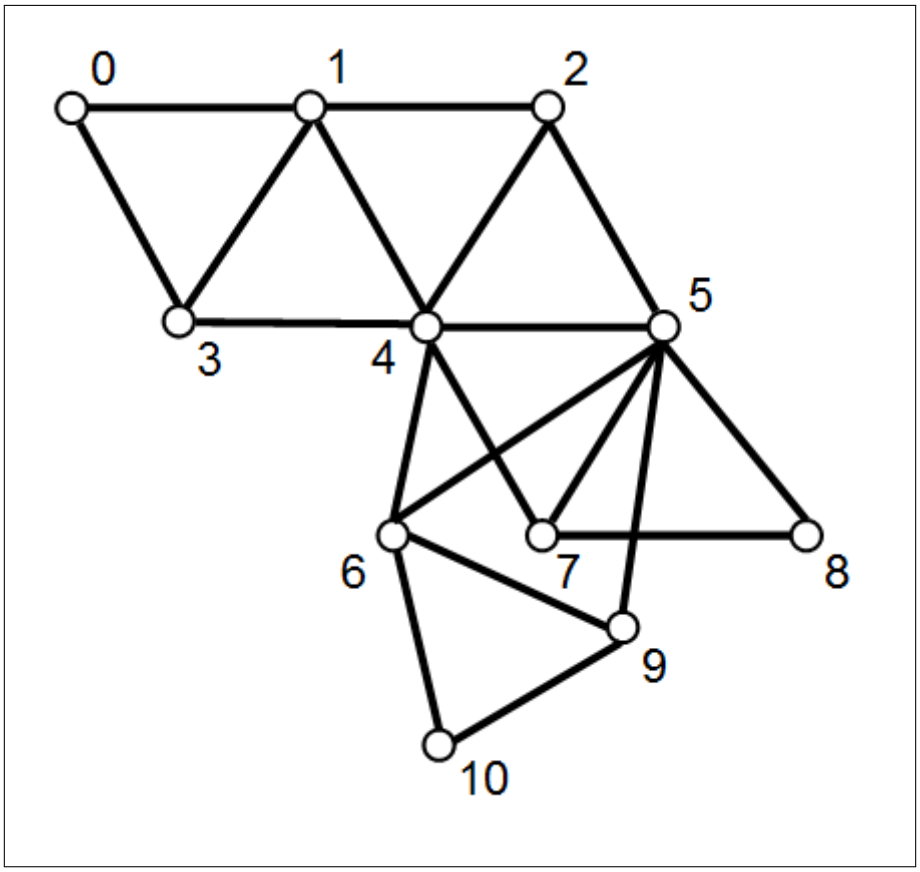

Figure 27: An example of a 2-tree and the k-tree we will be decompositioning for this example.

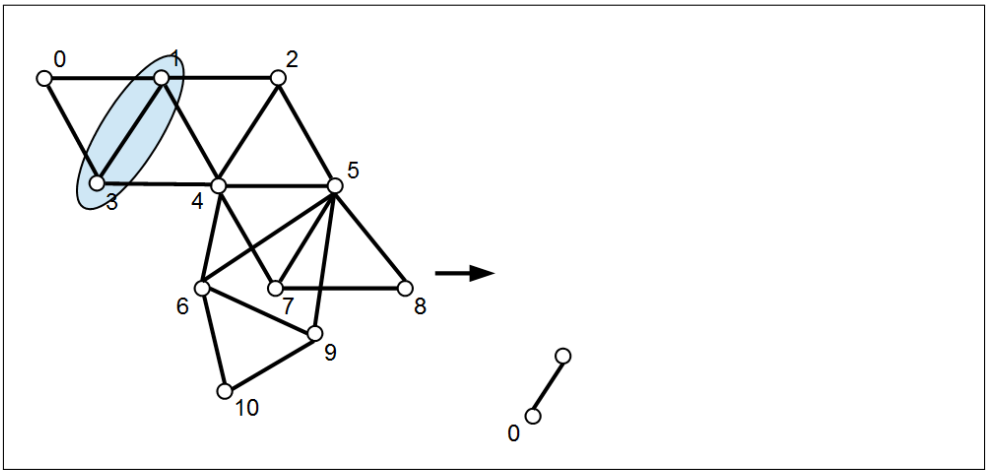

Figure 28: We will start from vertex 0. The two vertices in the ellipse comprise the front. We construct the tree on the right

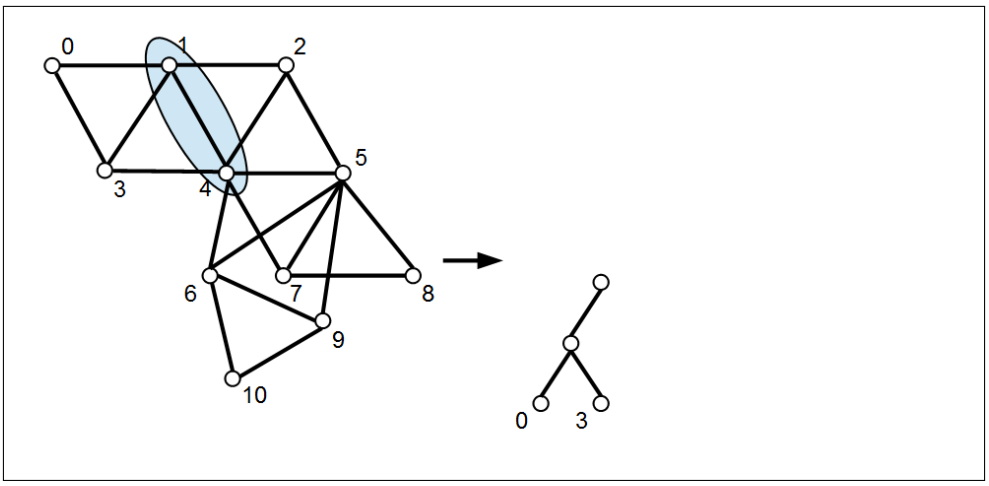

Figure 29: The first advance is a basic one. We add vertex 4 to the front and remove vertex 3. An associated leaf for vertex 3 is also added to the decomposition tree.

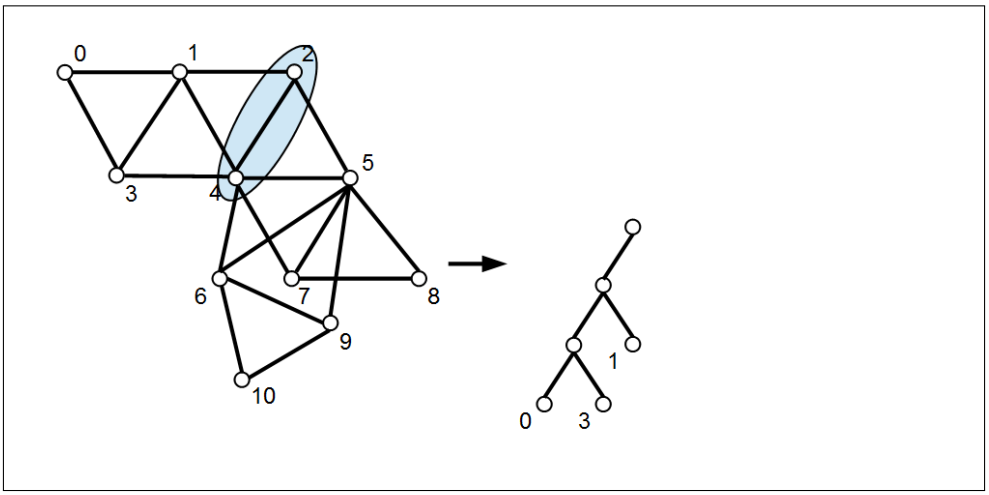

Figure 30: The second advance is also basic, this time vertex 2 is added and vertex 1 removed. A leaf for 1 is added to the tree.

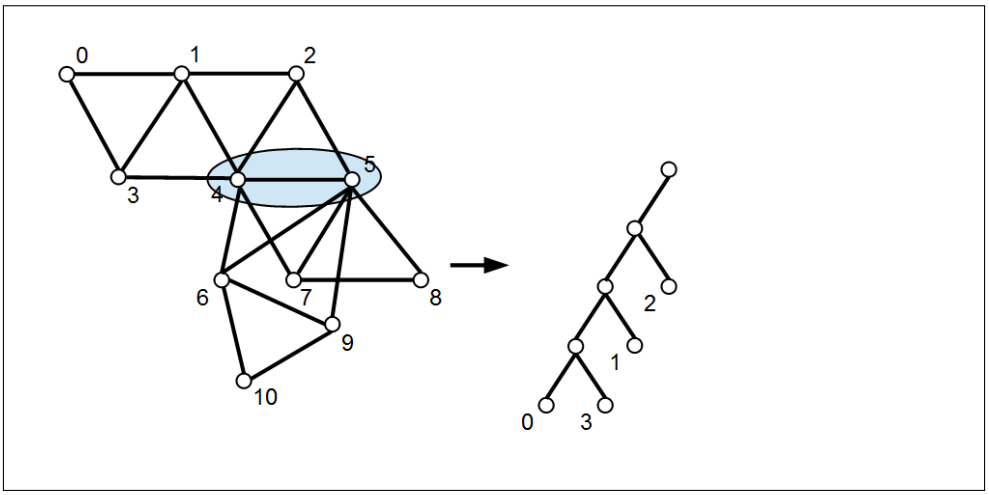

Figure 31: The third advance, another basic one, pushes a leaf for 2 to the tree. The front now separates the graph into three connected components.

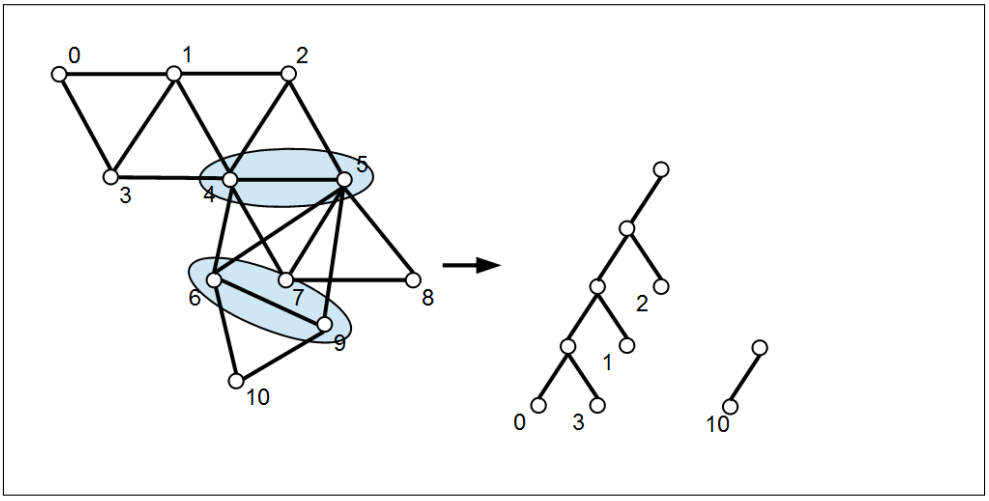

Figure 32: We decided to decompose the branch containing vertex 10 first. An additional front is used to decompose it.

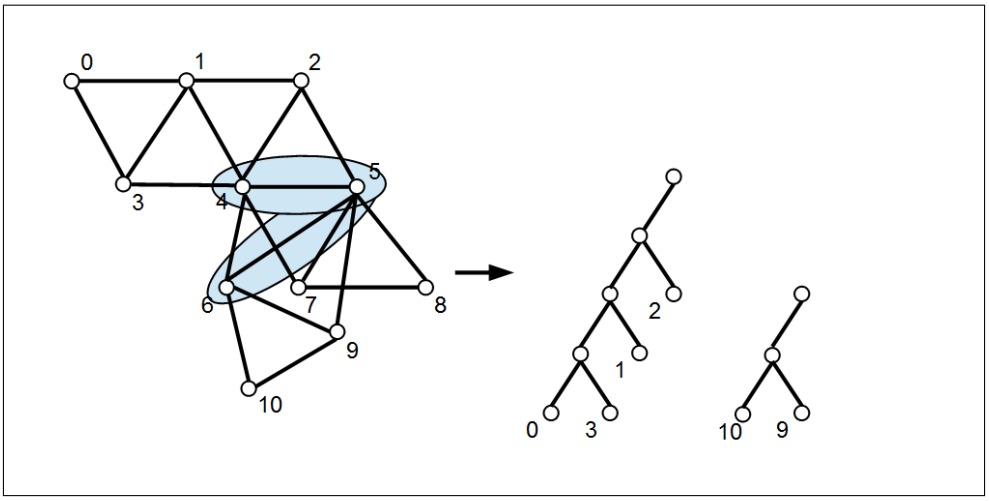

Figure 33: Another basic advance, but the secondary front overlaps with the primary front, so vertex 5 cannot be removed from it anymore.

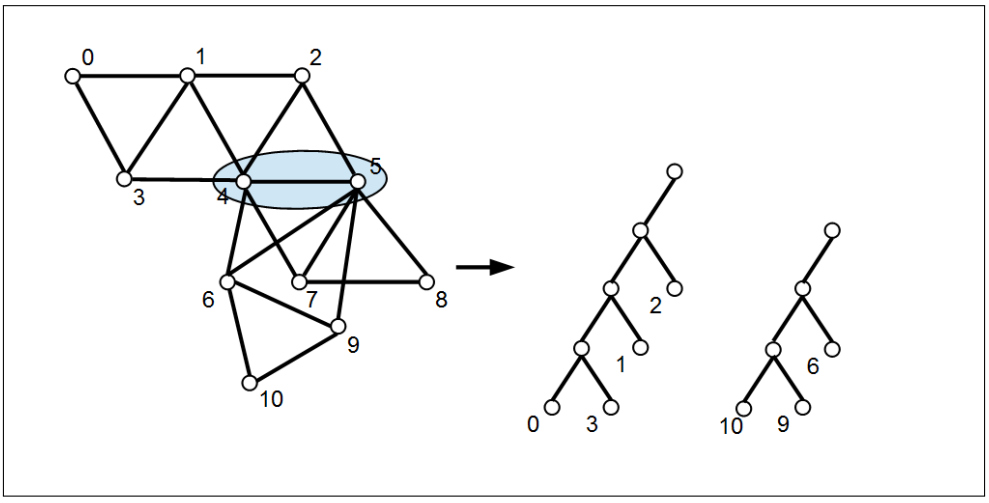

Figure 34: After adding a leaf for vertex 6 to the tree, our primary and secondary front are now equal...

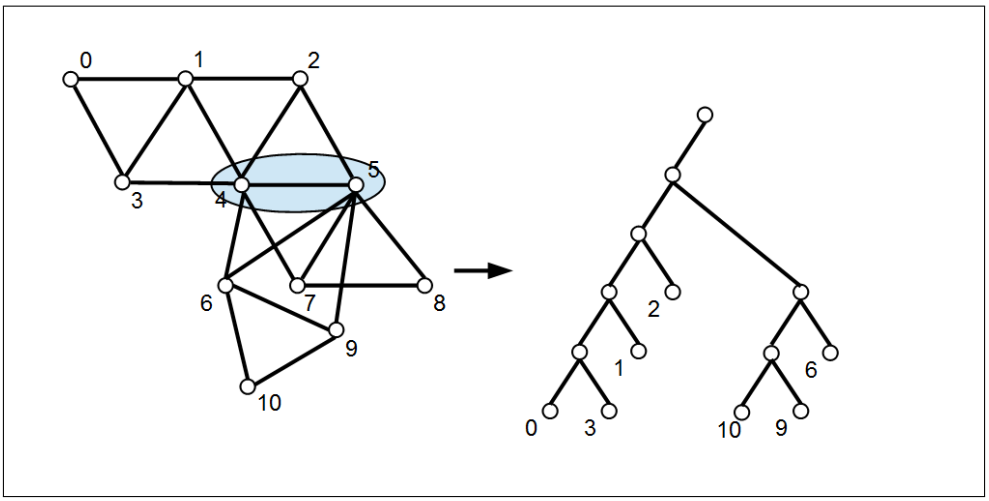

Figure 35: ...thus we connect the two separate trees and continue.

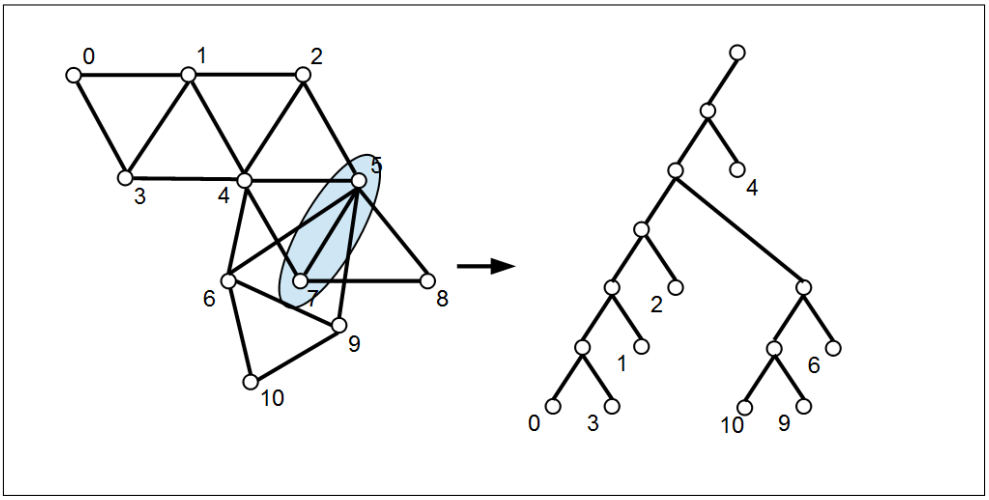

Figure 36: The leaf for vertex 4 is attached above this connection in the tree.

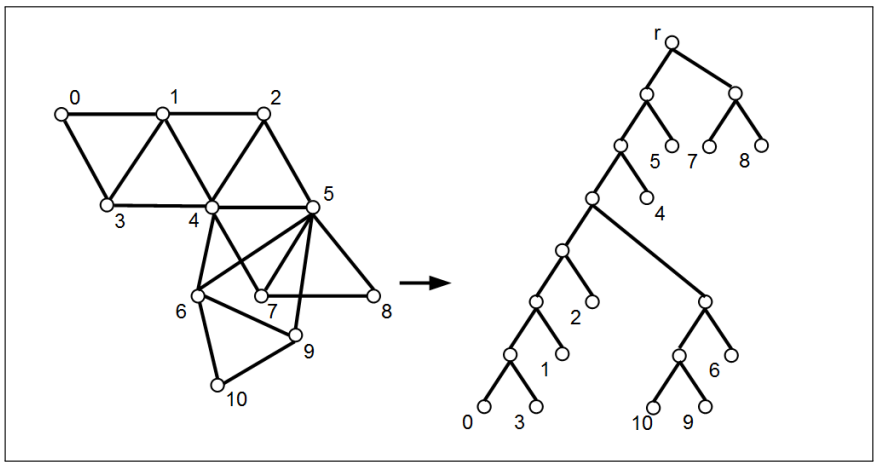

Figure 37: The final advance adds a leaf for vertex 5. Note that the front is now not a separator anymore. For the two vertices of this front, 7 and 8, we construct any binary decomposition tree and add that as the other child of the root.

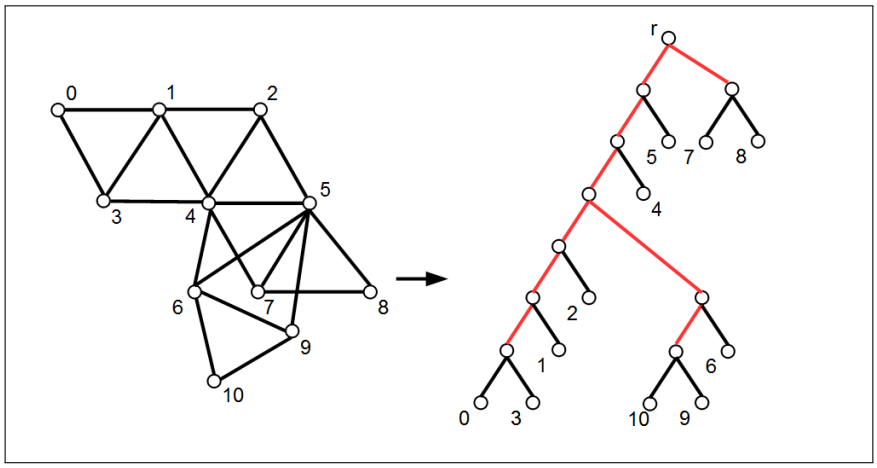

Figure 38: In this figure, the red edges of the decomposition tree indicate edges of mm score 2, the other edges have score 1.

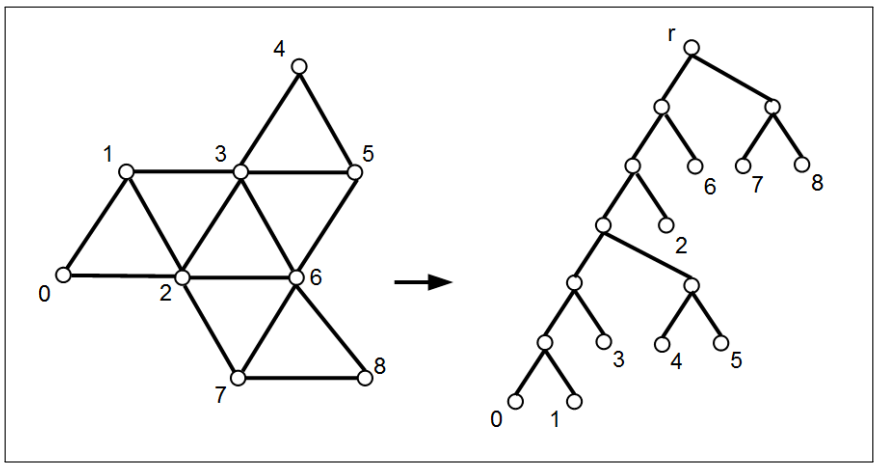

Figure 39: For the other branching possibility (when an advance would skip a branch) you also need to decompose the branches you skipped: Here we started from vertex 0 and found a branching point when we added vertex 6 to the front. Before we added a decomposition tree for the branch containing vertex 4 branching from that branching point, we added a leaf for vertex 3 to the decomposition of the main branch.

7.8.3 MM-Width of the K-Tree

In a moment we will give a proof for the MM-width of the binary decomposition trees we construct with this method, but the intuition for it is relatively simple: for the basic advances, and the locations where the front separates the graph into more than two connected components, we know that the vertices behind the front can only match with vertices in the front. This means that at most k matches can be made at those positions in the decomposition tree.

For the second non-basic advance, the one where $F \cup \{y\}$ separates the graph into more than two connected components, the separator consists of the front and an additional vertex, however each of the branches around this point except for one is still separated from the rest by a separator of size k. The exception is the branch we advanced from as the branch of the decomposition that contains the associated leaves of this branch gains the vertex that was removed from the front during the advance of the front that would have skipped the other branches. This additional vertex removes one of the k matching locations from the branch it was added to while adding a new matching location, making the size of the matchings also at most k.

With this intuition, we can see that the MM-width of a k-tree would be at most k, however the formal proof is found below.

7.8.4 Formal Proof

Theorem 10. *Let graph G be a k-tree, then $MM(G) \leq k$.*

Proof. Let graph G be a k-tree and let vertex v be any vertex from G with degree k. We start constructing the binary decomposition tree (T, δ) by adding a leaf node for vertex v with a parent node r. Let F be a set of vertices that we will call the current front, which will contain exactly all the vertices in $N(v)$ at the start of the algorithm. The current root node of (T, δ) will be denoted with r. Every step of the algorithm, either the front separates the graph into more than two connected components, exactly two connected components or it is not a separator; thus: either (1) $|\mathrm{CC}(G \setminus F)| > 2$, (2) $|\mathrm{CC}(G \setminus F)| = 2$ or (3) $|\mathrm{CC}(G \setminus F)| = 1$).

In case (1): Let $C_0 \in \mathrm{CC}(G \setminus F)$ be the connected component containing v and let C_1 be the connected component we will be advancing to. We will first need to resolve the other connected components in the same way as we handle the rest of the k-tree, but instead when the current front of this recursive call contains a vertex from F, this vertex may not be removed from that current front anymore when advancing. Then we make any binary tree with $|\mathrm{CC}(G \setminus F)| - 2$ leaves and hang the trees created by the recursive calls from the leaves of this tree and make the root of this tree a child of r. Above r we create a new parent node for r, which becomes the new root node of (T, δ) and thus also the new r. After doing this, we need to advance the front as in case 2.

In case (2), we can advance the front F. Let $C_1 \in \mathrm{CC}(G \setminus F)$ be the connected component of which no vertices have been added to (T, δ) yet. Let $N_F = \bigcap_{u \in F} (N(u))$. Remove a vertex x from F and pick a vertex y from $N_F \cap C_1$ to add to F. Then add a leaf for x as child of r to (T, δ), make r the child of a new node which will become the new root of the tree and be denoted as r. If the connected components in $\mathrm{CC}(G \setminus F)$ are the same as they were first, but the connected component containing

v has increased in size by exactly 1, then you are done for this step and we can look at the case distinction again. Else, we have skipped over at least one branch by advancing the front and we will have to resolve the branches in the same way as we handle the rest of the k-tree, but instead when the current front of this recursive call contains a vertex from $F \cup x$, this vertex may not be removed from that current front anymore when advancing. Merge the root of the recursive call with r and give r a new parent, which becomes the new r. Now we can look at the case distinction again. We will refer to a non-simple advance when we have to resolve other branches to advance and a simple advance otherwise.

In case (3), we cannot advance the front any further. Make any binary decomposition tree of F and hang this from r to finish construction of (T, δ).

Now we have constructed (T, δ), we will deduce the mm-width of this binary decomposition tree: First the edges in (T, δ) added to connect multiple branches in case (1). These edges all have the separator that the branches branch out of above it, meaning that each of the edges has mm-width at most k. As the branches that branch from the same separator set hang from the same tree, these branches compete with each other to match.

For every node added by simple advancing in case (2), the edge above the parent of the node has at most mm-width k, as the front at that point, a size k clique separator of the graph, is found completely beyond this edge, while all vertices of the connected component containing the associated vertex of this node is found on this side of the edge.

For a node n added by a non-simple advance in case (2), one that skipped over at least one branch, we can see that the edge above the parent of n must also have at most mm-width k, as originally in the branch that n was added to, $\leq k$ matching could have been made, one of which would have to have been with $\delta(n)$. Adding n to the tree makes it not possible to match with any more from the branch, instead it will now be matching itself with another vertex, meaning that the edge above the parent of n has mm-width at most k.

For all edges created by case (3), it is trivial to see that there are k vertices in the front at all times and therefore these edges have mm-width at most k.

Thus, all vertices in k-tree G have a corresponding leaf in binary decomposition tree (T, δ) and all edges in the binary decompostion tree (T, δ) have mm-width at most k. Therefore G has mm-width at most k.

□

8 Bounded DP MM-Width

We can improve on the performance of the dynamically programmed MM-width algorithm by using upper and lower bounds. The dynamic programming already let us skip mm-width calculations that we already have done, while the lower and upper bounds will allow us to skip calculation of entire possible subtrees with a simple check. We use a slightly adapted dynamically programmed MM-width algorithm and add the checks on the upper and lower bounds to it. We also add an additional dictionary to the algorithm that tracks the mm scores of specific partitions.

8.1 Lower Bound

As we move down the tree making decisions about the partitioning of the vertices, we will now calculate the mm scores of the edges from the nodes directly. We will keep track of the mm score of these edges and pass the largest mm score we have gotten due to partitioning decisions downward into recursive calls with this node as ancestor. Now, when we are calculating the optimal partitioning for a node, before calculating any further mm score for this node we can quickly check whether the mm-width of the subtrees can possibly be larger than what we know the mm-width of the current binary decomposition tree is.

Assuming we calculate the left edge and left subtree of a node first, if we did not skip the calculation of those, we can use also use the mm-width of the left subtree of the node to act as a lower bound on the calculation of the right edge and subtree.

The quick assessment we will be using whether we have to continue down to partition further for calculation will be based on the size of the set that we decided is beyond the edge we will move down past. If it is smaller or equal to the largest mm score we know we have in the tree, we also know that we can never have an edge in that subtree with a mm score higher than that. In addition, we can just give any binary decomposition tree over this subtree, saving on a lot of computations.

8.2 Upper Bound

For each of the edges we calculate the mm score of, we can also check whether or not the score we find is higher than an upper bound. If so, we know we made a bad decision for a partition that will increase our mm-width to above the upper bound. Therefore, we do not have to recurse beyond this edge and we can try another partitioning for this node.

There will be two upper bounds we will be considering in this thesis. The first of these we have already given earlier and is based on the proof that any graph has a MM-width of $\lceil n/3 \rceil$ or less. The second of these is linked to the MM-width of a k-tree.

8.2.1 MM-Width versus Treewidth

Treewidth is a well known graph property and used in parameterized complexity analyses of graph algorithms. In [1], the same work in which MM-width was introduced, the value of MM-width was linked to treewidth and branch-width as follows: $\frac{1}{3}(tw(G)+1) \leq MM(G) \leq \max(brw(G),1) \leq tw(G)+1$, where $tw(G)$ is the

treewidth of a graph G, $brw(G)$ is the branch-width of a graph G and $MM(G)$ is the MM-width of a graph G.

This means first off that if we know the Treewidth of a graph, we can both use the lower bound of $\frac{1}{3}(tw(G)+1)$ in addition to what mm scores we find during the algorithm and $tw(G)+1$ as an upper bound, if it is stronger than the clique bound of $\lceil n/3 \rceil$. However, using our analysis of the MM-width of a k-tree, we can improve on the link between the MM-width and treewidth parameter.

Definition 24. *(Tree Decomposition [Treewidth Decomposition])*
For a graph G a tree decomposition*, which we will name a* treewidth decomposition *to prevent confusion with binary decomposition trees, is a pair $(\{X_i \mid i \in I\}, T = (I, F))$, with $\{X_i \mid i \in I\}$ a collection of subsets of $V(G)$ and T a tree such that:*

- $\bigcup_{i \in I} X_i = V(G)$
- $\forall (u,v) \in E(G) \exists i \in I, v \in X_i$ AND $u \in X_i$
- $\forall i, j, k \in I$, *if i lies on the path between j and k in T, then $X_j \cap X_k \subseteq X_i$*

The width of a treewidth decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ is $\max_{\forall i \in I} |X_i| - 1$.

Definition 25. *(Treewidth)*
The treewidth *of a graph G is the minimum width over all treewidth decompositions of G and is denoted as $tw(G)$.*

Alternatively, one can define treewidth by means of *partial k-trees*:

Definition 26. *(Partial K-Tree)*
A graph G is a partial k-tree *if and only if G is the subgraph of a k-tree.*

In addition, one can prove that a graph G is a partial k-tree if and only if the treewidth of G is at most k. (See [2], [13] and [14])

This leads us to the following:

Theorem 11. *For any graph G, $MM(G) \leq tw(G)$.*

Proof. Let us take any graph G and let H be a k-tree such that $G \subseteq H$ and k is as small as possible. As H is a k-tree, we know that $MM(H) = k$ by theorem and thus that $MM(G) \leq k$ by theorem . As G is a partial k-tree, G has treewidth at most k. As we chose H to be a k-tree with smallest possible k, $tw(G) = k$.

In conclusion, we know that $MM(G) \leq k = tw(G)$, thus for any graph G, $MM(G) \leq tw(G)$. □

This improves on the bound given by Vatshelle in [1], though both our new bound and the old bound can be used to let treewidth be an upper bound in our algorithm or maximum matching width a lower bound on treewidth.

8.3 The Algorithm

Bounded Dynamically Programmed MM-Width is very similar to Dynamically Programmed MM-Width, but it has slightly reordered and extended logic to facilitate the addition of the upper and lower bounds. In addition to the global dictionary MMdict which stores optimal binary subtrees with their associated scores for partitions, we are now also using a global UBSkipDict which stores the mm scores of partitions. We also use Math.Max as a function that returns the maximum value of the two inputs.

(Please note that due to the length of the pseudo code of the algorithm, we had to split it to not fall off the page, meaning that the algorithm is now presented over two pages. To do this split in a natural location and useful location, we had to split inside of the ForEach loop.)

Algorithm 4: Bounded DP MM-width

Data: a graph g
treewidth of graph g *treewidth*
Result: MM-width of the graph and a binary decomposition tree of that score
$cliqueMax \longleftarrow Math.Ceiling(g.vertices.Length/3)$;
$result \longleftarrow BMMDPRecurse(g, \emptyset, -1, Math.Min(treewidth, cliqueMax))$;
return *result*

Algorithm 5: BMMDPRecurse

```
Data: a graph g
the set of vertices not in the subtree parentSet
largest score we forced upon ourselves largestScore
upper bound UB
SetToSplit ⟵ parentSet.Complement();
if SetToSplit.Count = 1 then
    return new Tuple(1, SetToSplit[0].Associated)
if SetToSplit.Count = 2 then
    return new Tuple(1,
      "(" + SetToSplit[0].Associated + ", " + SetToSplit[1].Associated + ")")
BestScore ⟵ int.MaxValue;
BestTree ⟵ " ";
foreach subset S of SetToSplit do
    if S.Count = 0 or (S ∪ parentSet).Count = g.vertices.Length then
        continue;
    scoreSubL ⟵ −1;
    scoreL ⟵ 0;
    skippedLCalc ⟵ false;
    if S.Count > 2 and S.Count ≤ largestScore then
        skippedLCalc ⟵ true;
        tree ⟵ AnyTree(S);
        scoreL ⟵ S.Count;
    else
        if UBSkipDict.ContainsKey(S) then
            leftEdge ⟵ UBSkipDict[S];
        else
            leftEdge ⟵ HopcroftKarp(G[S, S̄]);
            UBSkipDict[S] ⟵ leftEdge;
        if leftEdge > UB then
            continue;
        if MMdict.ContainsKey(S) then
            scoreSubL ⟵ MMdict[S].Item1;
            tree ⟵ "("+MMdict[S].Item2 + ",";
        else
            largerScore ⟵ Math.Max(largestScore, leftEdge);
            tupleL ⟵ BMMDPRecurse(g, S.Complement(), largerScore, UB);
            scoreSubL ⟵ tupleL.Item1;
            tree ⟵ "("+tupleL.Item2 + ",";
        scoreL ⟵ Math.Max(leftEdge, scoreSubL);

    [algorithm continues in this ForEach on the following page]
```

Algorithm 5: BMMDPRecurse (continued)

```
    scoreSubR ⟵ −1;
    scoreR ⟵ 0;
    rightSubSet ⟵ (S ∪ parentSet).Complement();
    if rightSubSet.Count > 2and(rightSubSet.Count ≤
     largestScoreornot(skippedLCalc)andrightSubSet.Count ≤ scoreL)
     then
        tree ⟵ tree + AnyTree(rightSubSet);
        scoreR ⟵ rightSubSet.Count;
    else
        if UBSkipDict.ContainsKey(rightSubSet) then
            rightEdge ⟵ UBSkipDict[rightSubSet];
        else
            rightEdge ⟵ HopcroftKarp(G[rightSubSet, rightS̄ubSet]);
            UBSkipDict[rightSubSet] ⟵ rightEdge;
        if rightEdge > UB then
            continue;
        if MMdict.ContainsKey(rightSubSet) then
            scoreSubR ⟵ MMdict[rightSubSet].Item1;
            tree ⟵ MMdict[rightSubSet].Item2 + ")";
        else
            largerScore ⟵ Math.Max(largestScore, rightEdge);
            tupleR ⟵ BMMDPRecurse(g, rightSubSet.Complement(), largerScore, UB);
            scoreSubR ⟵ tupleR.Item1;
            tree ⟵ tupleR.Item2 + ")";
        scoreR ⟵ Math.Max(rightEdge, scoreSubR);
    score ⟵ Math.Max(scoreL, scoreR);
    if score < BestScore then
        BestScore ⟵ score;
        BestTree ⟵ tree;
result ⟵ newTuple(BestScore, BestTree);
MMdict[parentSet.Complement()] ⟵ result;
return result
```

9 Kernelization of MM-Width

As a final improvement on our exact algorithms, we will be using kernelization rules. Such rules are used to find something called a kernel, which can be seen as the tough part (or parts) of the graph that actually involve the issues that need to be considered specifically to calculate things about the graph. This kernel is generally a smaller graph than the original graph. Because we end up with only the difficult parts of the graph, kernelization is part of the preprocessing part of algorithms.

To facilitate reading the kernelization rules, we have added figures showing graphs and decomposition trees marked to indicate the important vertices and nodes.

9.1 Rule 0: Separate Components

Up until now, we have assumed all graphs we were decompositioning consisted of a single connected component. To be more complete, and because we are going to need to be able to handle multiple connected components for some later rules, we need to figure out how to handle multiple connected components in a graph.

Theorem 12. *For G a graph with two connected components C_0 and C_1, where $G[C_0]$ is the subgraph of G containing only the vertices in C_0 and $G[C_1]$ only the vertices in C_1, $MM(G) = max(MM(G[C_0]), MMG[C_1])$.*

Proof. Let $(T, \delta)_{C_0}$ be an optimal binary tree decomposition of $G[C_0]$ and $(T, \delta)_{C_1}$ be an optimal binary tree decomposition of $G[C_1]$. We will now construct $(T, \delta)'$, a binary tree decomposition of G, by adding a new node r and making its left child the root of $(T, \delta)_{C_0}$ and its right child the root of $(T, \delta)_{C_1}$.

Each cut that can be made below the right edge of r will create a partition in which all vertices in C_0 fall in the same set: these vertices can thus not be matched to any vertex, meaning that the mm scores of these cuts are equal to the scores of the corresponding cuts in $(T, \delta)_{C_1}$. Similarly, each cut below the left edge of r gives an equal score to the corresponding cut in $(T, \delta)_{C_0}$. Both cuts directly below r will give a mm score of 0 as all vertices of C_0 will be in one set and all vertices of C_1 in the other and no edges exist between the two connected components. As the score of a binary decomposition tree is the maximum score found over all cuts, $mm((T, \delta)') \leq max(mm((T, \delta)_{C_0}), mm((T, \delta)_{C_1}))$.

Say that it is possible to find a better binary decomposition tree for G than $(T, \delta)'$, namely $(T, \delta)^*$, with $mm((T, \delta)^*) < mm((T, \delta)')$. We can now construct $(T, \delta)^*_{C_0}$ by removing all leaves associated to vertices of the other connected component and then contracting the nodes with less than two children. As the vertices no longer represented in this tree did not have edges connected to anything in C_0 they can not have increased the score for C_0. Idem for C_1. Thus, either $mm((T, \delta)^*_{C_0}) < mm((T, \delta)_{C_0})$, $mm((T, \delta)^*_{C_1}) < mm((T, \delta)_{C_1})$, or both are true.

However, as $(T, \delta)_{C_0}$ and $(T, \delta)_{C_1}$ were the optimal binary tree decompositions of $G[C_0]$ and $G[C_1]$ respectively, we have found a contradiction, thus we cannot find a better binary decomposition tree for G than $(T, \delta)'$.

Thus $mm((T, \delta)') = max(mm((T, \delta)_{C_0}), mm((T, \delta)_{C_1}))$ and
$MM(G) = max(mm(G[C_0]), mm(G[C_1]))$. □

This proof can be extended to any number of connected components and shows that the MM-width of a graph can be found by calculating the MM-width of each

connected component of a graph separately first. As this is a preprocessing step but does not really change the graph, we will consider this the zeroth step.

Because we can find all connected components in a graph in $O(|V(G)|)$ time by using either depth first search or breadth first search, a singular application of the zeroth rule takes linear time.

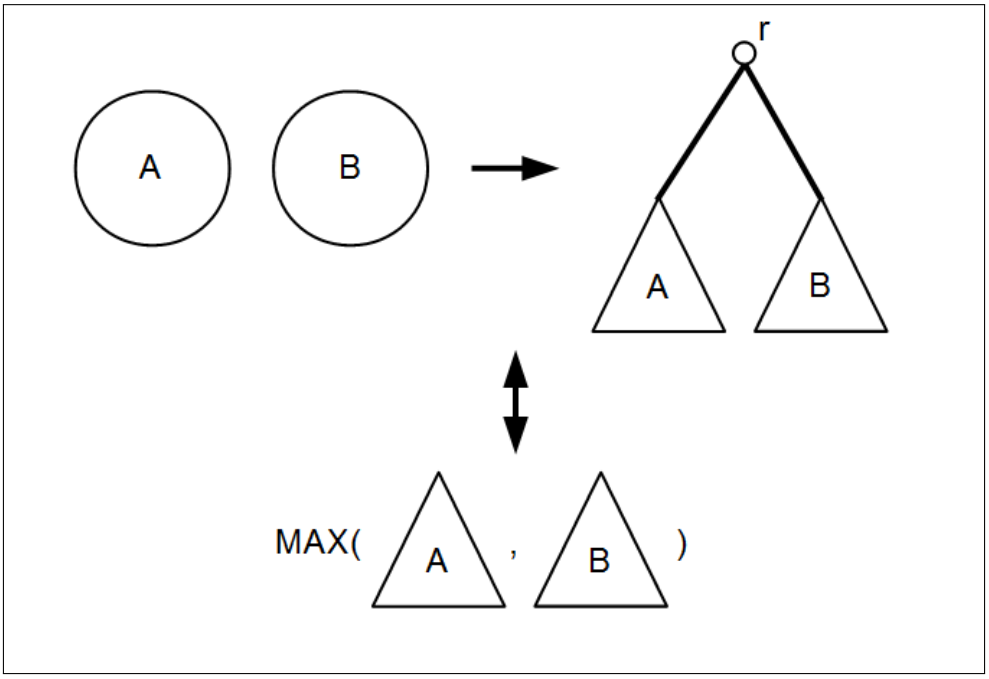

Figure 40: A graph consisting of two connected components, A and B, is decompositioned into a binary tree consisting of the decomposition trees of the separate connected components.

9.2 Rule 1: Dangling Vertices

The first rule we would want to have is one dealing with vertices of degree 1. We figured out in section that 1-trees all have a MM-width of 1. Removing a vertex of degree 1 from a 1-tree creates either a single vertex or another 1-tree. Now, we already know that the MM-width of a graph consisting of a single vertex is 0 as seen in section 7. This means we cannot reduce a graph to the single vertex graph without the MM-width dropping to 0. This implies we are not allowed to remove vertices of degree 1 if their single neighbor has degree 1 as well. Any other vertex of degree 1 can be removed safely.

Theorem 13. *For G a graph with $v \in V(G)$ a vertex of degree 1 with $\forall w \in N(v) degree(w) > 1$, $MM(G) = MM(G[V(G) \setminus \{v\}])$.*

Proof. Say you have an optimal binary decomposition tree $(T, \delta)'$ of graph $G \setminus v$. Now construct (T, δ) from $(T, \delta)'$ by putting an additional node n between $\delta(u)$ and the parent of $\delta(u)$. Also, add a second child to n and associate it with the vertex v. Now (T, δ) is a binary decomposition tree of G. Each partition induced by edges of $(T, \delta)'$ has a corresponding partition induced by (T, δ), but (T, δ) has two additional cut locations, namely between $\delta(u)$ and n, and between $\delta(v)$ and n. For all cuts

in $(T, \delta)'$, in the corresponding cut in (T, δ) u and v will be in the same set of the partition, meaning that no match can be made between u and v and that the score of each cut in $(T, \delta)'$ and the corresponding cut in (T, δ) is the same. For the two new cut locations, either u is all by himself in a set or v is, meaning that the scores of these cuts are at most 1. As $G \setminus v$ contains at least one edge, $MM(G \setminus v) \geq 1$, thus $MM(G) \leq MM(G \setminus v)$.

Because $G \setminus v$ is a subgraph of G, the width of $G \setminus v$ can not be greater than G. Thus $MM(G) = MM(G \setminus v)$. □

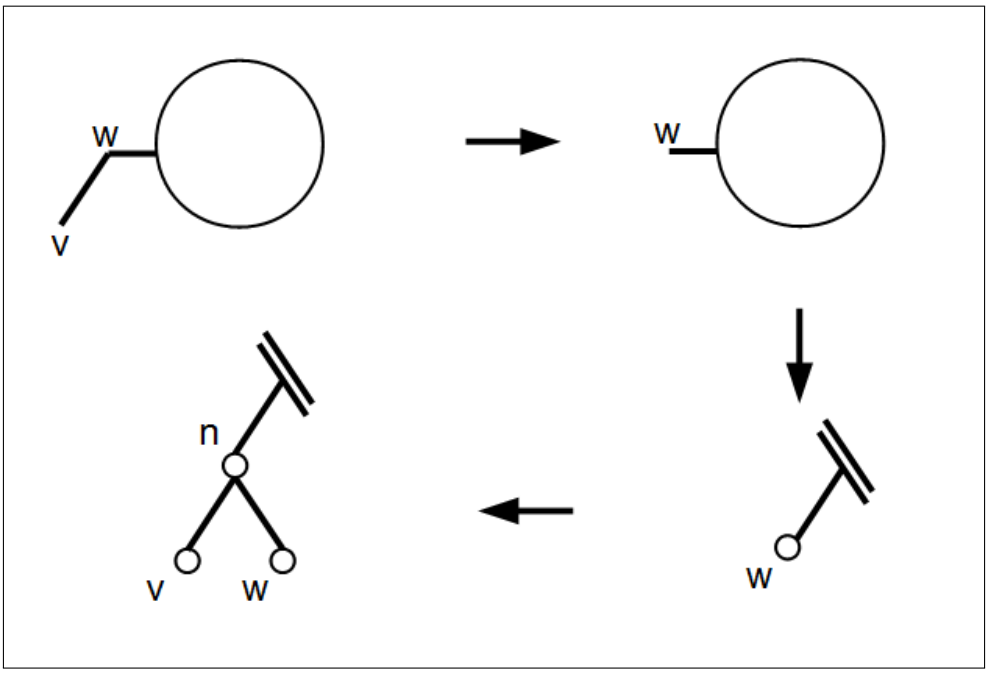

Figure 41: A graph with a degree 1 vertex v connected to a degree 2 vertex w. v is removed from the graph to decompose the graph more easily. It is then reinserted and $\delta(v)$ is inserted into the binary decomposition tree to create a decomposition of the original graph.

Using this proof, it is possible to give alternative proofs for both the MM-width of a path graph and 1-trees by removing vertices of degree 1 until we are left with a single edge graph.

We can find whether there are vertices of degree 1 eligible for deletion in the graph and remove them in $O(|V(G)|)$ time. However, removing a vertex of degree 1 means its singular neighbor may also become a vertex of degree 1. Thus, we can test each vertex of the graph at first and when we remove a vertex from the graph, we also check whether the neighbor of the vertex has become a vertex of degree 1 neighboring a vertex of degree > 1. If so, we also remove that vertex and check again, otherwise we go back to testing the next vertex. This way, we test each vertex at most once in the main loop and for each vertex we remove we test their neighbor. Thus it also takes $O(|V(G)|)$ time to remove all dangling vertices from a graph until there are none left.

Finally, we call these degree 1 vertices "dangling" vertices because they are attached to a larger graph and connect to that graph through a single vertex.

9.3 Rule 2: Dangling Triangles

Logically, the next rule we would want is about vertices of degree 2. The problem is that such vertices can be in more different situations than vertices of degree 1, so we will first look at dangling vertices again. Now, we can't have a vertex of degree 2 attach to a larger graph through a single vertex by itself, so we will have to add at least one additional vertex. If this additional vertex has degree 1, we can simply handle the vertex by applying rule 1 twice. Let us therefore say this additional vertex has degree 2 and is connected to both the vertex we want to remove and the vertex from which it dangles, forming a triangle.

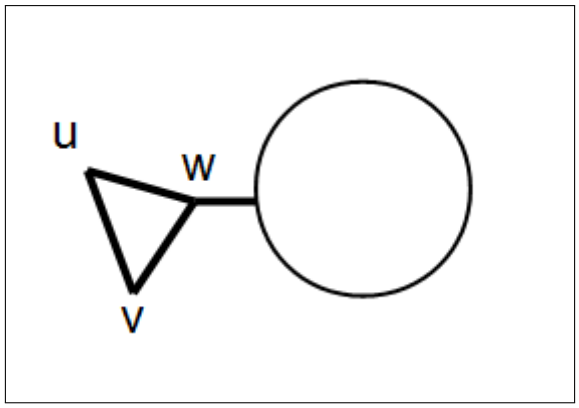

Figure 42: A graph with a dangling triangle.

Theorem 14. *For G a graph with $v \in V(G)$ a vertex of degree 2 with $\exists u, w \in N(v) : degree(u) = 2$ and $degree(w) > 2$, $MM(G) = MM(G[V(G) \setminus \{v\}])$.*

Proof. Say you have an optimal binary decomposition tree $(T, \delta)'$ of graph $G \setminus v$. Now construct (T, δ) from $(T, \delta)'$ by putting an additional node n between $\delta(u)$ and the parent of $\delta(u)$. Also, add a second child to n and associate it with the vertex v. Now (T, δ) is a binary decomposition tree of G. Each partition induced by edges of $(T, \delta)'$ has a corresponding partition induced by (T, δ), but (T, δ) has two additional cut locations, namely between $\delta(u)$ and n, and between $\delta(v)$ and n. For all cuts in $(T, \delta)'$, in the corresponding cut in (T, δ) u and v will be in the same set of the partition, meaning that no match can be made between u and v. In addition, in the partitions that w is in the other set than u and v are, at most one of both will be able to match with w. Thus the score of each cut in $(T, \delta)'$ and the corresponding cut in (T, δ) is the same. For the two new cut locations, either u is all by himself in a set or v is, meaning that the scores of these cuts are at most 1. Thus $MM(G) \leq MM(G \setminus v)$.

As $G \setminus v$ is a subgraph of G, the width of $G \setminus v$ will not be greater than G. Thus $MM(G) = MM(G \setminus v)$. □

After the removal of this vertex, we now have a dangling vertex that we can apply the first rule on. This means that we could also remove both vertices of degree 2 of the dangling triangles from the graph for the second rule, though we have decided on this variation of the rule as it is the simplest variation of the rule.

Finding all vertices of degree 2 in a graph takes $O(n)$ time. Checking for each of the found vertices whether one of their neighbors has degree 2 and the other degree

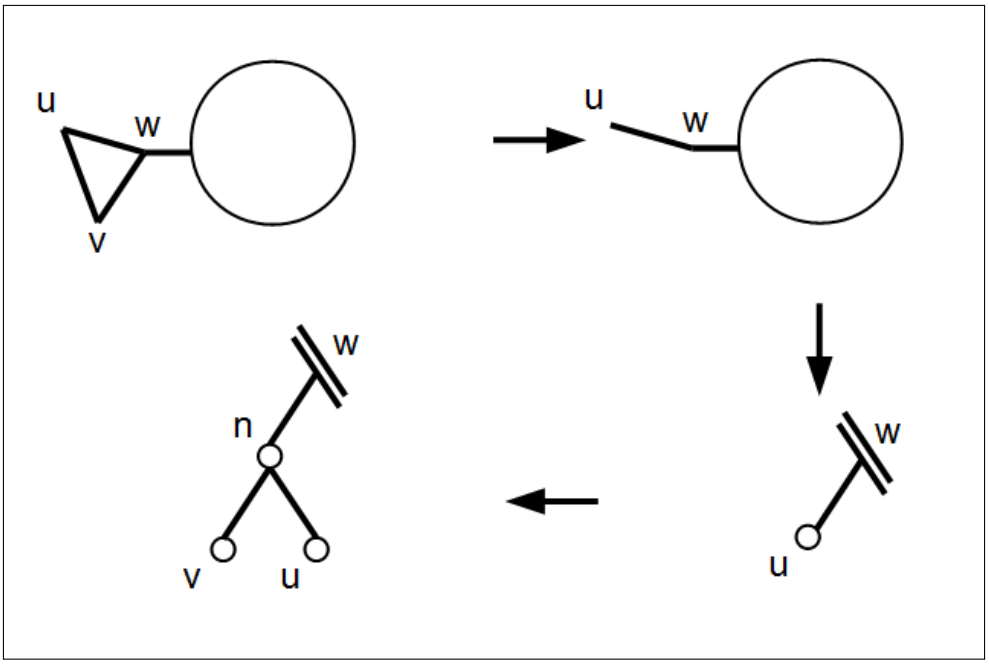

Figure 43: A graph with a dangling triangle. v is removed from the graph to decompose the graph more easily. It is then reinserted and $\delta(v)$ is inserted into the binary decomposition tree to create a decomposition of the original graph.

> 2 does not increase this complexity. In the variation of the rule that we remove only a single vertex, we can not create new situations in which we can delete more vertices by this rule. In the variation that we delete both vertices however, the vertex the removed triangle dangled from can become a vertex that is part of a new dangling triangle. Similarly to the first rule, we can again remove all the dangling triangles as we find and create them without gaining additional complexity.

9.4 Rule 3: Size 1 Separators

We can handle dangling triangles by utilizing rule 2 followed by rule 1, however we can also handle them by splitting the graph on the vertex the triangle dangles from. This results in two connected components: A cycle of three vertices (a triangle) and the graph which we would have gotten if we had removed the vertices by rule 2 and 1. As the graph are now two separate connected components, this implies that the MM-width of the graph is now $max(mm((T,\delta)_{C_3}), mm((T,\delta)_H))$ where $(T,\delta)_{C_3}$ is an optimal binary decomposition of the triangle and $(T,\delta)_H$ is an optimal binary decomposition of the the graph we get after removing the two vertices. As the MM-width of a cycle of three vertices is 1 and the MM-width of the rest of the graph must be at least 1, $max(mm((T,\delta)_{C_3}), mm((T,\delta)_H)) = max(1, MM(H)) = MM(H)$, which is correct as we have seen before.

In fact, we can prove a more general rule for any separator of size 1. If we have a graph G with a separator S of size 1 in it, the MM-width of G is the maximum over the two connected components of $G \setminus S$ with the separator added back to it. But to prove this, we need the following proof first about "rebalancing" binary decomposition trees:

Theorem 15. *For G a graph, (T,δ) an optimal binary decomposition tree of G and*

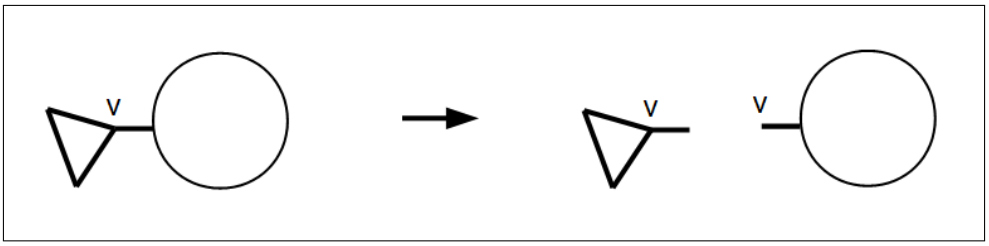

Figure 44: A graph with a dangling triangle is split into two components, this results in the same score as proven before.

$v \in V(G)$, $(T,\delta)^v$ is an (optimal) binary decomposition tree of G with $mm((T,\delta)) = mm((T,\delta)^v)$ and one of the children of the root of $(T,\delta)^v$ is $\delta(v)$. We call $(T,\delta)^v$ v-rebalanced.

Proof. We can construct $(T,\delta)^v$ from (T,δ) as follows: Remove the root node and its two incident edges from (T,δ), then connect the nodes that were originally the root node's children with an edge. This new edge induces the same partition as the two edges we removed, meaning that (T,δ) still has the same MM-width. Then find $\delta(v)$ in (T,δ) and split the edge incident to $\delta(v)$. The new vertex will now be the new root node of (T,δ), while the two new edges have the mm score as the edge they were created from, meaning that (T,δ) still has the same score.

As (T,δ) is still an optimal binary decomposition tree and $\delta(v)$ is now a child of the root node, we have now constructed $(T,\delta)^v$. □

Theorem 16. *For G a graph and S a separator of size 1, $MM(G) = \max_{H \in CC(G \setminus S)}(mm((T,\delta)_{G[H \cup S]}))$, where $(T,\delta)_A$ is an optimal binary tree decomposition of graph A.*

Proof. For each connected component $H \in CC(G \setminus S)$ we can construct an optimal binary tree decomposition $(T,\delta)_{G[H \cup S]}$. For the one vertex $v \in S$, we will v-rebalance $(T,\delta)_{G[H \cup S]}$ and we remove the root node and $\delta(v)$. Then, we create any binary tree with $|CC(G \setminus S)|$ leaves and connect one of the binary tree decompositions for each of the connected components to each of the leaves. Finally, we create a root node to make the parent of the full tree and make $\delta(v)$ one of the children of the root. This new tree is now a binary tree decomposition (T,δ) of G.

As each subgraph $G[H \cup S]$ only has interaction with the others through S, all edges in the connecting binary tree in (T,δ) and the two edges incident to its root have a mm score of 1. In addition, all edges in the subtrees of the subgraphs still have the same score as they had before, meaning that $mm((T,\delta)) = \max_{H \in CC(G \setminus S)}(1, mm((T,\delta)_{G[H \cup S]}))$. We know that any graph with at least one edge has a MM-width of 1, so we can simplify this to $mm((T,\delta)) = \max_{H \in CC(G \setminus S)}(mm((T,\delta)_{G[H \cup S]}))$

Now say that it is possible to find a better binary decomposition tree for G than (T,δ), namely $(T,\delta)^*$ with $mm((T,\delta)^*) < mm((T,\delta))$. We can now construct $(T,\delta)^*_{G[H \cup S]}$ by removing all leaves associated to vertices not in $V(H \cup S)$ and contracting all nodes with less than two children. As the vertices no longer represented in this tree did not have edges connected to anything in H, they could not have increased the score for H. Idem for each $H \in CC(G \setminus S)$. Thus, for at least one of the $H \in CC(G \setminus S)$, $mm((T,\delta)^*_{G[H \cup S]}) < mm(((T,\delta)_{G[H \cup S]})$, which is a contradiction, thus we cannot find a better binary decomposition tree for G than (T,δ).

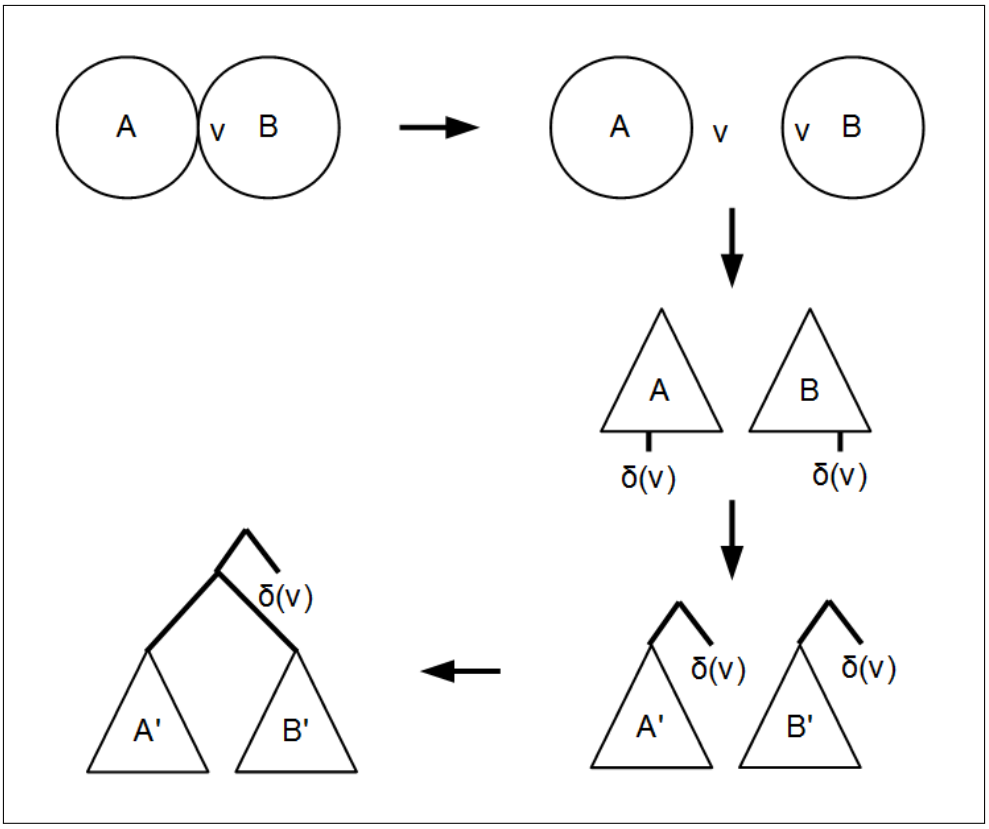

Figure 45: The graph is split on the separator of size 1 and the two components are decompositioned and rebalanced on the single vertex in the separator. Connecting the two trees creates the optimal decomposition tree for the original graph.

Therefore, $MM(G) = mm((T, \delta)) = \max_{H \in CC(G \setminus S)} (mm((T, \delta)_{G[H \cup S]}))$. □

Testing whether or not a vertex is a separator takes $O(|V(G)|)$ time, so testing this for each vertex in the graph takes $O(n^2)$ time, where n is the number of vertices in the graph. So, even though this is a more general rule than the second rule, it is also more expensive. Meaning that both rules are applicable in their own way.

9.5 Rule 4: Cycles of 4 or Greater

Another possible situations for vertices of degree 2 to be in, is part of a cycle. We have already shown how cycle graphs of four or more vertices have MM-width 2, meaning that we can only reduce cycles of five or more vertices without possibly lowering the MM-width of our graph.

For this rule, we will be looking at five consecutive vertices $t, u, v, w, x \in V(G)$, where u, v and w each have degree 2 and both t and x have degree at least 1. As these vertices are consecutive, $u \in N(t)$, $t, v \in N(u)$, $u, w \in N(v)$, $v, x \in N(w)$ and $w \in N(x)$. We will remove vertex v from G to create $G[V(G) \setminus \{v\}]$.

Theorem 17. *For graph G with vertices $t, u, v, w, x \in V(G)$ consecutively on a path, u, v and w each of degree 2 and there is a path from t to x that does not go through u, then $MM(G) = MM(G[V(G) \setminus \{v\}])$.*

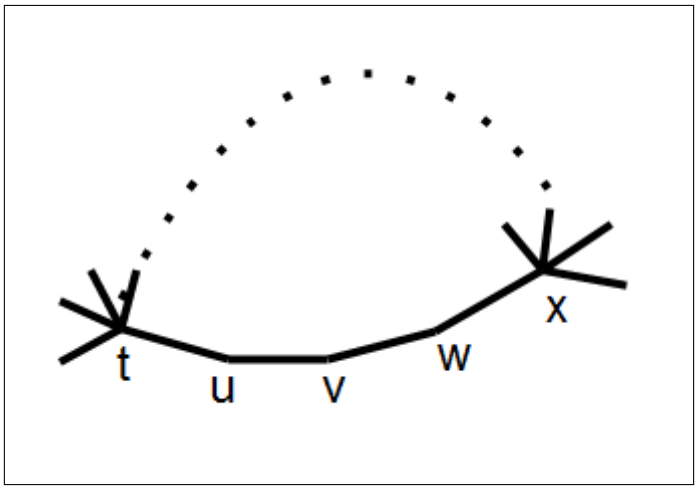

Figure 46: The five consecutive vertices t, u, v, w and x. The dotted line indicates a path between t and x not through u, as the five vertices must be on a cycle.

Proof. To construct an optimal binary tree decomposition $(T, \delta)'$ for G from an optimal binary tree decomposition (T, δ) for $G[V(G) \setminus \{v\}]$ we do the following: rebalance (T, δ) such that $\delta(w)$ is not a child of the root, then identify the path through (T, δ) between $\delta(w)$ and $\delta(x)$. We will insert $\delta(v)$ "next" to $\delta(w)$ on the other side as where the path between $\delta(w)$ and $\delta(x)$ goes. We will denote the edge in (T, δ) we split to add $\delta(v)$ as b and the edge incident to $\delta(w)$'s parent (but not incident to $\delta(w)$ itself) that lies on the path between $\delta(w)$ and $\delta(x)$ as a. Then a' is the same edge as a but in $(T, \delta)'$ and b' and c' are the two edges created by splitting b, where c' is the edge between the parents of $\delta(w)$ and $\delta(v)$.

To figure out the mm-width of $(T, \delta)'$, we need to analyze the edges of the decomposition trees and the partitions they induce. Now, there are many different possible ways for the associated leaves to be placed around the trees and we will be analyzing the different ways they may be and the effects. We will case separate on the cases where w and its two neighbors are located:

(a) First off, say that that $\mathcal{P}_a = (A, B)$, where $x \in A$ and $w, u \in B$ (We will for now shorten this to $\mathcal{P}_a = [u, w; x]$). Then we know that $\mathcal{P}_b = [u; w, x]$ and this means that $\mathcal{P}_{a'} = [x; v, w, u]$, $\mathcal{P}_{c'} = [x, v; w, u]$ and $\mathcal{P}_{b'} = [x, v, w; u]$.

In this situation, $mm(\mathcal{P}_a) = mm(\mathcal{P}_b)$ and $mm(\mathcal{P}_{a'}) = mm(\mathcal{P}_{c'}) = mm(\mathcal{P}_{b'})$, because the amount of matchings remain the same in the graph that can be made by changing between these partitions. However, the addition of v and $\delta(v)$ in these locations mean that the amount of matchings also remain the same, thus $mm(\mathcal{P}_a) = mm(\mathcal{P}_{a'})$. So in this situation the MM-width remains the same.

(b) Otherwise, $\mathcal{P}_a = [u, x; w]$, $\mathcal{P}_b = [u, w, x;]$, $\mathcal{P}_{a'} = [u, x; v, w]$, $\mathcal{P}_{c'} = [u, w, x; v]$ and $\mathcal{P}_{b'} = [u, v, w, x;]$. In this situation, $mm(\mathcal{P}_a) = mm(\mathcal{P}_b) + 1$ as in $\mathcal{P}_a$ w and u can match, this matching is lost in $\mathcal{P}_b$ and no other matching is formed, unless t is in the other set than u, in which case $mm(\mathcal{P}_a) = mm(\mathcal{P}_b)$. Thus we will need to distinguish cases on where t is in the partition:

(b1) Let us first assume $\mathcal{P}_a = [u, x; t, w]$, $\mathcal{P}_b = [u, w, x; t]$, $\mathcal{P}_{a'} = [u, x; t, v, w]$, $\mathcal{P}_{c'} = [u, w, x; t, v]$ and $\mathcal{P}_{b'} = [u, v, w, x; t]$. Now we can see that on all edges between $\delta(u)$'s parent and $\delta(t)$'s parent in (T, δ) (including a and b), there is a matching between t and u because of how the vertices on the path are now in the two sets alternately increasing the score on these edges. However, this local increase is unnec-

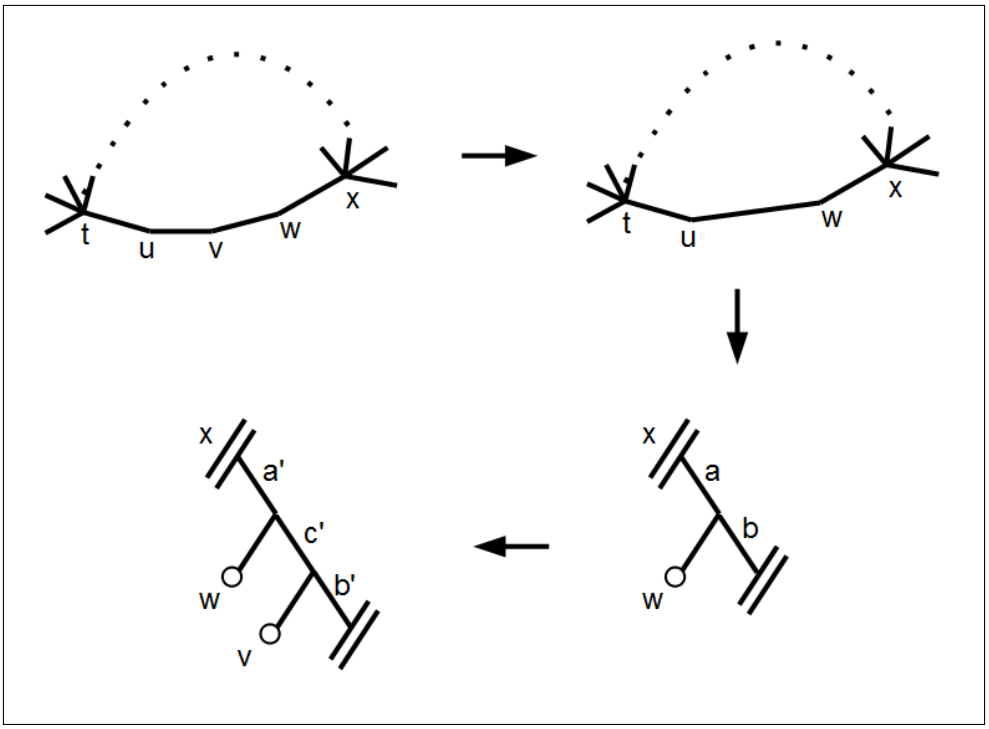

Figure 47: The decomposition notation in the figure can be read as follows: the leaf nodes are marked with the associated vertex. The parts of the tree of which the exact structure is not important or not known are beyond the double lines, next to which is indicated which leaves can be found in the part. In the figure we have also indicated the edges we have denoted specifically above.

essary and there must be another optimal binary decomposition tree of $G[V(G) \setminus v]$ where t is in the same set as u in these partitions, so take that decomposition tree instead.

(b2) Thus we now know that $\mathcal{P}_a = [t, u, x; w]$, $\mathcal{P}_b = [t, u, w, x;]$, $\mathcal{P}_{a'} = [t, u, x; v, w]$, $\mathcal{P}_{c'} = [t, u, w, x; v]$ and $\mathcal{P}_{b'} = [t, u, v, w, x;]$. Now either w is the last element that is being added from this cycle to the partitioning, or x has at least one other neighbor (on the cycle) that is not in the same set as x.

(b2.1) If x has at least one neighbor y other than w that is not in the same set as x, then we have the same issue as in (b1): we have vertices in alternating sets. To be precise, u and t are in the same set as x, but w (and v) in another. There must thus be another optimal binary decomposition tree of $G[V(G) \setminus \{v\}]$ where x has no neighbors other than w not in the same set in these partitions, so take that decomposition instead.

(b2.2) If w is the last element to be added from this cycle to the partition, then $\mathcal{P}_a$ can have one more matching than $\mathcal{P}_b$, however at some point the cycle must have added at least 2 to the score. In addition, $\mathcal{P}_{a'}$ actually gives two matching opportunities, one with w and one with v, so this partition does not exceed what the cycle adds elsewhere. Similar for $\mathcal{P}_{c'}$ and $\mathcal{P}_{b'}$.

All together, we can see that the MM-width of the graph did not increase by changing $G[V(G) \setminus \{v\}]$ to G. Logically, the MM-width does not decrease by adding the vertex, so $MM(G) = MM(G[V(G) \setminus \{v\}])$. □

Unfortunately, to use this rule to simplify the graph to solve a simpler form and

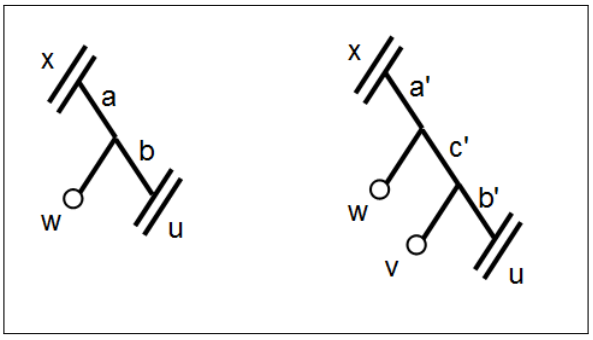

Figure 48: Situation (a).

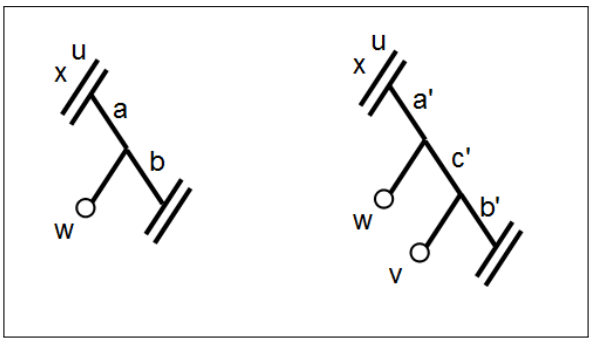

Figure 49: Situation (b).

then finding the decomposition graph of the larger tree can not be easily done with the rule as we present it above, as a few of the possible situations the graph could be in requires us to take another optimal binary decomposition tree of our graph and we can not guarantee that optimal binary decomposition trees we construct are locally optimal as we require to be able not to do that. However, the rule can still be used to find the MM-width of the graph after kernelization.

Because it is non-trivial to construct the optimal binary decomposition tree of the the larger graph, we did not implement it ourselves for any algorithms. However, applying this rule takes linear time to reduce the size of the graph. We can find a vertex of degree 2 in linear time and check in constant time whether it is applicable for removal. We can then check whether a neighbor is also applicable for removal in constant time, meaning that we can remove all occurrences of places to apply this rule in $O(n)$ time.

9.6 Further Rules: Larger Separators

We can actually generalize rule 3 to work for larger separators. If we use a separator S that is larger than one vertex, we can still construct a binary decomposition tree (T, δ) of graph G from each of the binary tree decompositions $(T, \delta)_{G[H \cup S]}$ for $H \in CC(G \setminus S)$, similar to how we did so in the rule with a single separator. However, now we must balance all the binary tree decompositions for a single vertex $v \in S$.

As an example, we will work with two connected components A, B and a separator $S = v, w$ of size 2. If we use a separator of size 2, the MM-width of the resulting decomposition must be at least 2. We will v-balance both $(T, \delta)_A$ and $(T, \delta)_B$, leaving the leaf $\delta(w)$ in a random position in both decomposition trees.

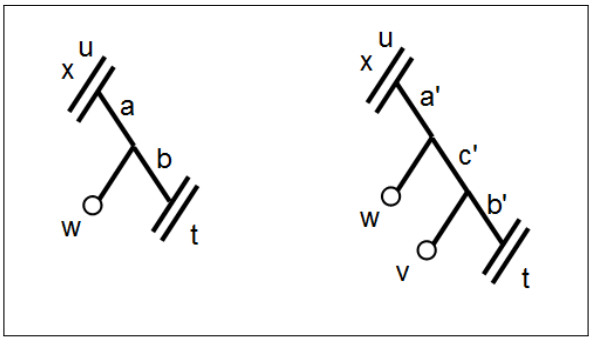

Figure 50: Situation (b1).

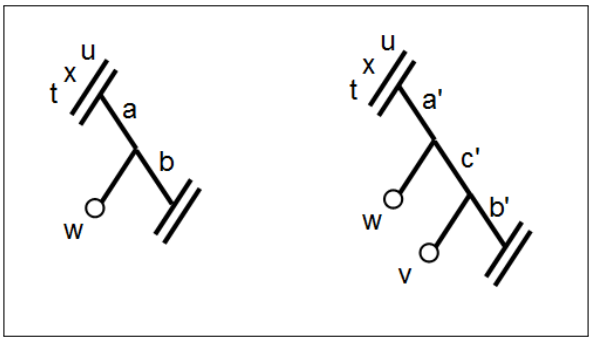

Figure 51: Situation (b2).

When we connect the two trees up however, we also need to remove $\delta(w)$ from one of the two decomposition trees, as the resulting tree can only have one associated leaf for each vertex in the graph. As $\delta(w)$ is now in a possibly sub-optimal position for the decomposition tree we removed it from, the MM-width of the tree we deleted $\delta(w)$ from can has increased by 1.

This means that the MM-width of (T, δ) is either $max(mm((T, \delta)_A), mm((T, \delta)_B)+1)$ or $max(mm((T, \delta)_A) + 1, mm((T, \delta)_B))$ and at least 2. In fact, if $mm((T, \delta)_A) > mm((T, \delta)_B)$ we can simply choose to add the increase in score to $(T, \delta)_B$ and if if $mm((T, \delta)_A) < mm((T, \delta)_B)$ we can simply choose to add the increase in score to $(T, \delta)_A$. Thus, we know that $MM(G) \leq max(min(max(mm((T, \delta)_A), mm((T, \delta)_B)+1), max(mm((T, \delta)_A) + 1, mm((T, \delta)_B))), 2)$.

To find a separator of size s we must select s vertices of G. This means that the complexity of finding all possible separators of that size is $O(n! - (n - s)!)$. Testing whether one of those separator is an actual separator then takes linear time for each of those, making the complete complexity of finding a place to apply this rule $O(n \cdot (n! - (n - s)!))$. Next, figuring out which of the vertices of the separator to rebalance the decomposition trees with and where the associated leaves of the rest will be in the trees then takes $O(2^n)$. This makes this rule quite expensive if you were to use it. Using it for limited size separators is thus a good idea, though we did not implement this beyond separators of size 1.

Note that for these rules we can technically use any size separator, however the mm-width of a solution using such a separator tends to the size of the separator. For implementing such rules, reading relevant material may be helpful. [3] uses separators for preprocessing for treewidth, while [16] uses clique separators for decompositioning of graphs for divide-conquer algorithms.

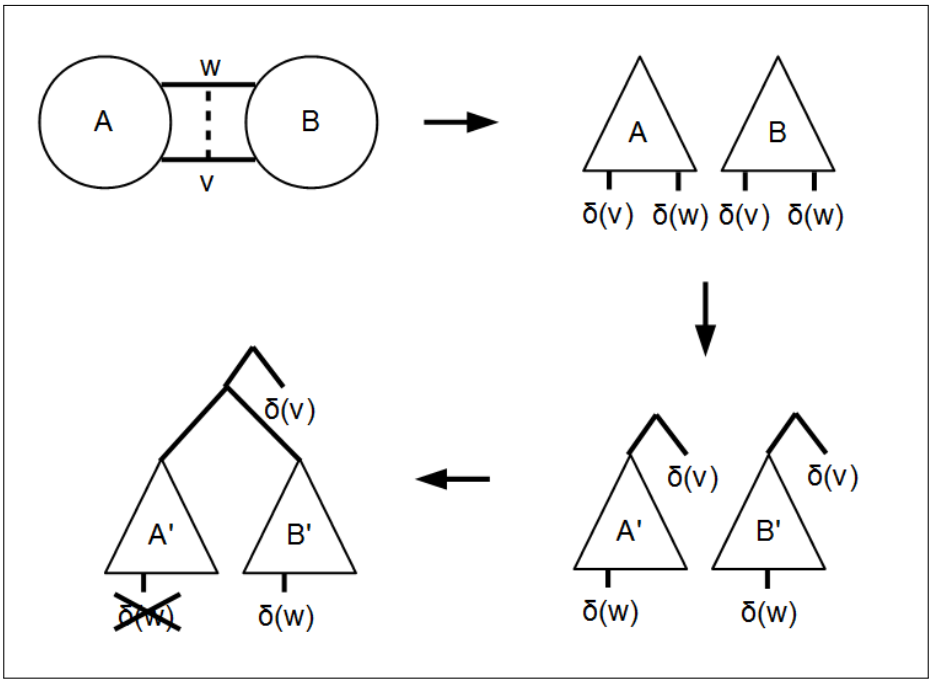

Figure 52: To decomposition the graph with separator of size 2 we decompose the sides of the separator. Then we rebalance and connect the trees and remove duplicate leaves.

9.7 Polynomial Kernels

Something we aimed to do at the start of this project, but were sadly not able to achieve, was to produce a polynomial kernel. A polynomial kernel is a kernel which has a provable size. Because of this provable size, you can give a strict(er) bound on the calculation of graph problems. It is also often a step used for fixed-parameter tractability to be able to give a problem a parameterized complexity, which is a complexity bound on the calculation measured in a function of multiple parameters of the input or output. You can read more about parameterized complexity and kernelization for different widths in [4], [5] and [6].

9.8 Application of Kernelization Rules

When applying the kernelization rules, start with applying rule 0 to separate the graph into separate graphs for each of its connected components. For each of these connected components, apply the first and second rule (and possibly the fourth rule) to reduce their size until that is no longer possible. Then apply rule three to find a separator of size 1. If one is found, replace the bigger graph with the smaller graphs and go back to apply the earlier rules again. Possibly apply rules for larger separators if no smaller separators found and apply earlier rules. Continue until no rules can be applied any more. This can substantially reduce the size of the graph because of the application of rules one, two and four. However, the third rule and any further separator based rules technically increase the amount of vertices of the graph. The running time of a subsequent run of an algorithm will however still speed up, as these rules split the graph in two, reducing the number of vertices in the separate calls.

10 MM-Width by Optimization Algorithms

We have constructed a few variants of an exact algorithm in the previous sections. These different algorithms perform are effective, but are sadly also quite slow and take far too much time even on what we would consider small graphs. What if, instead of trying to guarantee finding the optimal decomposition, we try to find as good a decomposition tree as possible, without looking over all the possible solutions. This will of course not tell us what the MM-width of a graph is as we will not have looked at every possible solution, but it can often be used to give a strict upper bound for our bounded MM-width algorithm, to reduce the running time. To do this, we will start with using local search at first. We will also look at the possibility of using evolutionary computing to find solutions.

You can read more about local search algorithms and genetic algorithms in [15].

10.1 Local Search Basics

To use local search optimization techniques, we are going to need to be able to move to a neighboring solution from a current solution. The solution we are working with is a binary tree decomposition of the graph in our case. We are also going to need local search operators. These operators change a solution into another valid solution, possibly with a different score. We have seen earlier that many of the binary decomposition trees are equivalent, so these operators should not just change a solution into an equivalent solution. Preferably, it should not have any equivalent solutions in the neighbor space.

10.1.1 Starting Solution

For the starting solution, we can either create a random binary decomposition tree, or create a random binary decomposition tree balanced around one of the inner nodes to create a tree that will automatically be an optimal decomposition of a clique graph.

To create a random tree for a graph, we can do the following: create a tree consisting of only a root node. Then choose a random node in the tree that does not have any children and give that node two new child nodes, which increases the amount of leaves of the tree by one. Repeat increasing the leaves of the tree until we are left with a leaf node for each vertex of the graph. Finally, randomly associate the vertices of the graph with the leaf nodes of the tree.

We can use the above method to also create a random binary decomposition tree balanced around one of the inner nodes, by subdividing the vertices of the graph into three balanced sets, then creating a tree for each of those sets and finally attaching these three trees to each other by the root.

10.1.2 Swap Operator

One of the two operators we will use on solutions, is the swap operation. To execute the swap operator, we need to select two subtrees of the current binary decomposition tree that do not have any nodes in common. We can easily select these subtrees by selecting a root node from the solution for the first subtree. All nodes in the full binary decomposition tree with that node as an ancestor then forms the subtree.

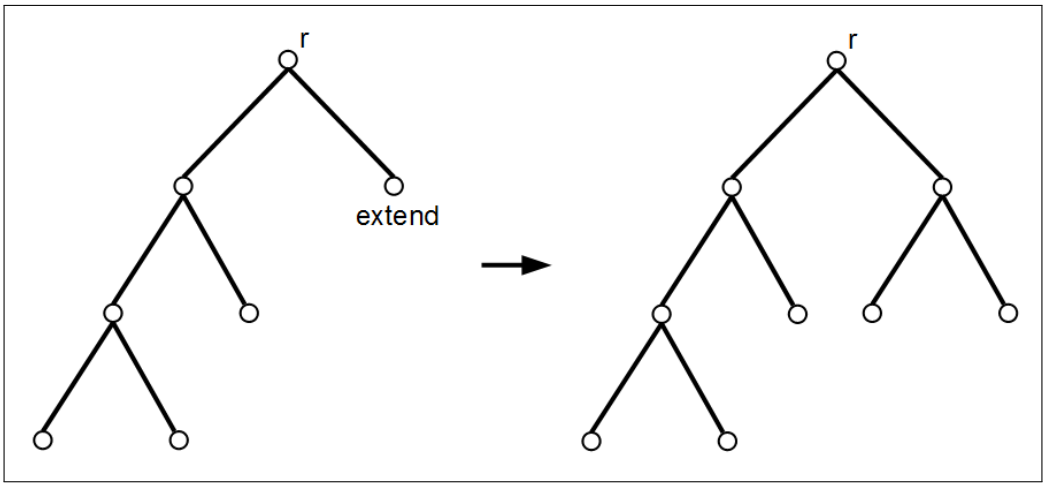

Figure 53: Demonstration of a single step to extend the random tree.

The second root node now has to be selected from all nodes that are not part of the first subtree to create the second subtree in this same way. It should be clear that we cannot select the root node of the full binary decomposition tree for either of these subtrees, as the other subtree then will always have nodes in common with it.

With these two root nodes of the subtrees, we can execute the swap operation by doing the following: Name the two subtrees A and B and denote the parent node of A's root as a and the parent node of B's root as b. Now make b the parent of A's root node and a the parent of B's. The resulting tree is our new solution.

We can revert this operator by simply executing it again on the same nodes.

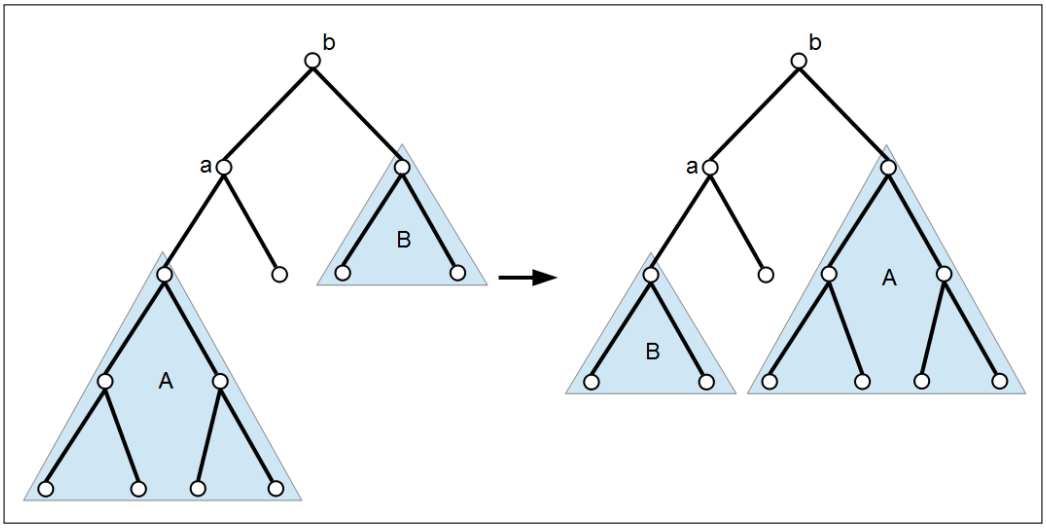

Figure 54: Demonstration of the swap operator.

10.1.3 Move Operator

The other operation we will be using is the move operation. For this one, we also need to select two nodes from the current binary decomposition tree. For the move operator, while the first of these nodes is still the root node of a subtree of the full binary decomposition tree, the second node indicates the edge we will be splitting

to insert the subtree. To be precise: the second node may again not overlap with the nodes in the subtree we picked to move, nor may the second node be a parent or sibling of the root of the subtree, and the edge between the node and its parent is the actual edge we have selected.

Execution of the move operation goes as follows: Name the subtree A and the second node v. Denote the parent node of A's root as a and v's parent node as u. Now, if a is not the root of the binary decomposition tree, then denote the sibling of A's root b and a's parent as c. Remove node a from the tree and make b's parent c. Then add a new node n to the tree and make n's parent u and v's parent n. Finally, make n the parent node of A's root.

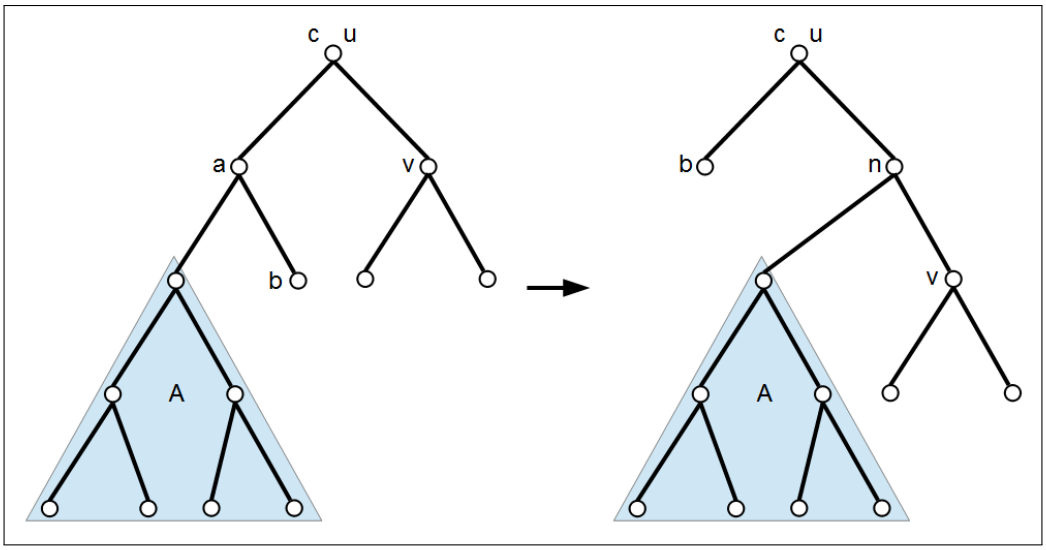

Figure 55: Demonstration of the move operator.

In the situation that a is the root of the binary decomposition tree, we also need to make sure we update what node is considered the root node of the full binary decomposition tree, as the root node of the the tree is removed. As we remove a node and create a new one, we can also technically recycle the node we remove as the new node by simply changing the links around it.

Finally, to revert the move operator, we need to remember the original position of the subtree so we can move the recycled node back to where it came from and reattach the edges as they were.

10.1.4 Score of Solution

We need to score our solutions to find the solution with the lowest mm-width. However, simply using the mm-width of the decomposition tree will result in low granularity, as often neighboring solutions have similar mm-width, which means there is almost no steering when searching through the neighborhood.

To improve the searchability of the solution space, we will need to increase the granularity. To do this, we will instead use the sum of all mm scores of the edges in the decomposition tree in the solution. This means that minimizing the score still tries to minimize the scores on the edges. We want to make sure the highest edge in the decomposition tree is as small as possible though, which is not guaranteed to happen. Because we do want to guarantee that, we will add a component to the

evaluation function for the mm-width of the solution. This component should always outweigh the sum of all the edges. Finally, because the edges incident to the root of the solution always induce the same partition and this would mean optimization would prefer solutions where on of the children of the root is a leaf, we will subtract the score of the left edge incident to the root from the evaluation.

The maximum score of any edge in the binary decomposition tree is $\lfloor n/2 \rfloor$, as this is the maximum amount of matching that can be made in any graph of n vertices. In addition, there are $2n-1$ edges in a binary tree of n leaves, meaning that the sum over all the edges is always lower than n^2. In total, the we will consider the score of one of the solution binary tree decompositions (T, δ) of graph G to be $eval((T,\delta)) = mm((T,\delta)) \cdot |V(G)|^2 + \sum_{e \in E(T)}(mm(\mathcal{P}_e)) - mm(\mathcal{P}_{(T.root, T.root.left)})$.

Using this function, we know that solutions with a lower mm-width will always have a better score, while we also try to minimize the total sum of the edge scores.

10.1.5 Faster Score Calculation

We do not have to calculate our score from scratch when finding a new solution. Instead, we simply need to figure out what edges of the solution have been changed or could have a changed mm score and calculate the mm score of those edges. We simply copy the scores of the other edges of the previous solution to find the score of the rest of the edges. To figure out which edges changed, we need to find out what edges induce a different partition.

It is simple to see that the edges that induce a different partition in the new solution after making a swap operation are the edges on the path between the parents of the roots of the subtrees. For the move operation, we change the partitions induced by all edges between the original parent of the subtree's root and the new parent of the subtree's root. In addition, we also need to score a few edges that have been newly created, although these can be quickly inferred from what we knew before about the mm scores of the edges.

10.2 Hill Climbing (Iterative)

The first local search algorithm we will be looking at will be Hill Climbing. When Hill Climbing, we optimize our solution by finding improvements until we can no longer find a better neighboring solution. The iterative variant of this algorithm always goes through the possible improvements in the same order. To be precise, we first try all move operations before trying all swap operations. We then list all nodes of our current solution by an in-order traversal and pick any node from it that is not the root. For each node we pick for the selected node, we list all nodes in our current solution eligible to be the second selected node under the current chosen operator. Finally, we calculate the score we get from applying the operator on the solution on the two chosen locations.

A well known caveat of Hill Climbing algorithms is that it will not necessarily find the optimal solution to a problem and instead the algorithm will find a local optimum. A local optimum is a situation where a single operation can not change the solution to anything better and instead multiple operations have to be executed to move the algorithm from the local optimum, through worse solutions, to a place where optimization can continue. The iterative variant of Hill Climbing can be considered even worse though, as it will always find the exact same local optimum

for a problem when starting from the same position. This means that starting from a random solution is useful to increase the search space covered by this algorithm.

10.3 Hill Climbing (Randomized)

To fix always finding the same local optimum, we will try random changes to the solution instead of trying the operations in the same order at all times. We can either scramble the orders of the nodes we are looping over and what operation we do, or we can choose a random operation and locations to apply it on. We have chosen the latter as it is the simpler variation. Note that we do now have to put a limit on how many changes we try without finding an improvement to make sure our algorithm does terminate.

10.4 Multi-Start Hill Climbing

The issue that we may get stuck in a local optimum is still present however. We can solve this by starting Randomized Hill Climbing multiple times, possibly from different starting solutions. This means that we can end up in multiple local optima, one of which may be the global optimum. Note however, multi-start algorithms are completely equivalent to simply running your solver multiple times and taking the best result, which we can do for any of our algorithms, even a multi-start algorithm.

10.5 Iterated Local Search

A different way to still use our Hill Climbing algorithms, but prevent getting stuck in the first local optimum we find, is to add an ability to move the current solution away from the local optimum to enable the algorithm to reoptimize and possibly find a better solution. If we do this, we have constructed a variant of Iterated Local Search. With Iterated Local Search we only accept improvements until we can no longer find an improvement, then we execute a number of random operations on our current solution irregardless of the resulting score. These random operations push the solution away from the local optimum and is called a perturbation. We put a limit on the number of perturbations we do to make sure our algorithm terminates and keep track of the best solution we have seen at any point of the algorithm as we are now able to lower the score of our current solution.

It is possible that a problem we are executing Iterated Local Search on has very pronounced local optimum peaks, making it difficult for a series of random changes to push the current solution out of the space that will return to the local optimum. We will therefore increase the number of changes we make during a perturbation if we were unable to push ourselves away from the local optimum (found the same local optimum after hill climbing). We can also choose to increase our perturbation size if we were not able to improve our best solution, though you should note that you then want to make the perturbation from the previous local optimum.

We have to watch out with setting the perturbation size too high though, because if the perturbation size is too large, we may lose too much of the structure that made the previous local optimum so good. This would then destroy all the hard work the algorithm did to find an optimum, making it essentially a Multi-Start Hill Climbing algorithm.

10.6 Simulated Annealing

Instead of only accepting improving changes to the current solution as in the Hill Climbing algorithms, we could also accept some changes that do not improve the solution. We have to watch out however that we do not simply accept any and all changes, because we do want to find a better solution.

To prevent our algorithm from being a Random Walk through the solution space, we will accept all improvements we find but only a fraction of the non improving changes. We want this fraction of worsenings we accept to become smaller as we continue, so our algorithm starts optimizing more as we run and finding better solutions over time. The less worsenings we start accepting however, the less changes we have of moving out of subspaces that converge to certain local optima and into other such subspaces. To change the fraction of worsenings we accept, we will use a temperature variable that we will lower as we continue executing the algorithm. This temperature is often made to decay exponentially, which resembles how metals cool and is therefore the reason why this algorithm is called Simulated Annealing.

Instead of simply changing the fraction of worsenings that we accept, we also want to have a lower chance of accepting worsenings that lower the score more as we do want to optimize in general. This leads us to the following formula to calculate the acceptance chance of a solution:

$$P_{accept} = \begin{cases} if\ eval(S_{next}) \leq eval(S_{current}) & 1 \\ otherwise & e^{-(\frac{eval(S_{next}) - eval(S_{current})}{T})} \end{cases}$$

In this formula, T indicates the current temperature, $eval_{next}$ is the score of the solution we are possibly moving to and $eval_{current}$ is the score of the solution we are currently at. As we want to minimize our score and accept any improvement, we return a chance of 1 of accepting the new solution if it is lower. Otherwise, we calculate a chance that becomes smaller the worse the difference between our current and possible new score is and also becomes smaller the lower our temperature gets.

It should be clear that the value range of the temperature is very important for the functioning of the algorithm. If the temperature starts out too low, our Simulated Annealing approach is not much different from a Hill Climbing algorithm as worsenings we then accept don't make much difference and we don't accept very many worsenings either. If our temperature starts off too high however, we essentially start off with a Random Walk through the search space before our temperature drops to a better level, which wastes time.

In addition to the temperature and the evaluation function we use, there are a few other parameters we need to find values for: We need to determine how strong we cool for our cooling scheme and we need to determine at which points we let the temperature drop. In addition, we are going to need to decide what our terminating condition is.

If our terminating condition is a minimum temperature and we are using exponential decay for our cooling scheme, these parameters relate as such: $T \cdot \alpha^k = t$ and $k = i/Q$ where T is the starting temperature, k is the amount of times we have cooled, t the current temperature, i the iteration we are on and Q the number of iterations between each drop. This means that if we know the starting temperature, what value we use in our cooling scheme, the number of iterations we want to do at most and at what temperature we want to end on, we can calculate the number of

steps we should do between each time we cool as follows: $Q = i_{max}/\log_{\alpha}(\frac{t_{end}}{T})$

10.6.1 Simulated Annealing with Differing Evaluations

During our experiments with Simulated Annealing as we described above we had difficulty finding good values that would make simulated annealing optimize graphs better than the other local search solutions. One of the issues we discovered we had with simulated annealing, was that values that allowed the algorithm to find relatively good solutions in the solution space had a lot of difficulty getting away from such solutions. This led us to making adjustments to the algorithm in addition to adjustments to the parameters. We reasoned that the algorithm does not like moving to a solution with higher MM-width because in the cost function we wrote made the MM-width of the solution guaranteed the most important factor. However, the amount of times we change the MM-width of a solution is far less than the amount of times we change the the sum part of the score. This means that our temperature is thus balanced around changing the score of the sum part, making it so that making the MM-width of our solution worse is near impossible.

To fix this, we will use a second evaluation function:
$eval_2((T, \delta)) = \sum_{e \in E(T)}(mm(\mathcal{P}_e)) - mm(\mathcal{P}_{(T.root, T.root.left)})$. We use this additional evaluation function to determine whether we worsen our score, were we originally used the evaluation function we gave earlier, giving us the following acceptance chance formula:

$$P_{accept} = \begin{cases} if\ eval(S_{next}) \leq eval(S_{current}) & 1 \\ otherwise & e^{-(\frac{eval_2(S_{next}) - eval_2(S_{current})}{T})} \end{cases}$$

Using this acceptance formula in our Simulated Annealing algorithm, we were able to find parameters that allowed Simulated Annealing to find better solutions, though we were generally not able to find solutions as good as our other algorithms give us, nor were we able to find these solutions any faster. Though some of the best solutions found for graphs were found by Simulated Annealing. Finding solutions as good as those was not easily repeatable though.

10.7 Evolutionary Algorithms

There are many more different means of optimization, one of which is the use of Evolutionary Algorithms. Evolutionary Algorithms are algorithms inspired by nature that use a population of solutions, selects solutions from this set to be used to create new solutions, evaluate these new solutions and then fill/replace the population for the next iteration.

To evaluate the solutions in our population, we will be using the same evaluation function we used for the evaluation of solutions during our local search algorithms.

10.7.1 Selection

There are many different ways to select solutions that we use to create new solutions. The simplest of these is to simply sort the solutions by evaluation score and take a number of solutions with the lowest score. This is called truncation selection.

The problem with truncation selection is that solutions with a worse evaluation score have only a small chance of remaining part of the population, disallowing the algorithm to climb out of local optima. We can resolve that by also keeping some higher score solutions in the population, though we should still optimize by mainly selecting better solutions. We can do this by selecting solutions to keep in the population by means of chance, where a better evaluation score gives a higher chance of being selected. This roulette wheel selection works as follows: give every solution in the population a portion of the "roulette wheel" corresponding to the evaluation score of the solution. Then spin the wheel as often as you want to select a solution, selecting the solutions the wheel lands on. In this roulette wheel you can choose to allow the same solution to be selected multiple times or remove an entry from the wheel as soon as it is picked.

Note that in our optimization problem we want to minimize, meaning that we can't base the chance of selection directly on the evaluation score. Instead we will figure out which solution in our population has the highest score $eval(S_{worst})$. The size of a portion of the roulette wheel for a solution S is then $2 \cdot v_{worst} - eval(S)$.

10.7.2 Crossover Operator

To create new solutions during our evolutionary algorithm, we will be using a crossover operator that creates a new solution from two parent solutions. Our crossover solution selects a random subtree from the first of these parents. Then we copy the second parent and remove all leaves from this tree of which the associated vertex has an associated leaf in the subtree we selected, after which we contract nodes in the tree to construct a binary tree. Finally, we connect the subtree and the created tree with a new root node.

This new solution has a lot in common with mainly its first parent. For this first parent, all edges found in the subtree have the same mm score in both the parent and the child solution. For the second parent however, only the relative order of the leaves we did not remove remains the same.

Before we use the crossover operator, we randomly order the selected population and then split the set into pairs of parents. Each pair of parents then creates two children, using both parents as the first parent once.

10.7.3 Representation of Solution

An important issue for Evolutionary Algorithms, and Genetic Algorithms in particular, is the way the solutions are represented. We decided to use the actual binary decomposition trees as representation for our solutions as this is the natural representation for trees and both our optimization algorithms as exact algorithms also represent the decomposition trees. There are also other representations possible however that allow for different crossover operators that preserve different properties of the parents. One such option is presented in [7], we did not investigate other representations however.

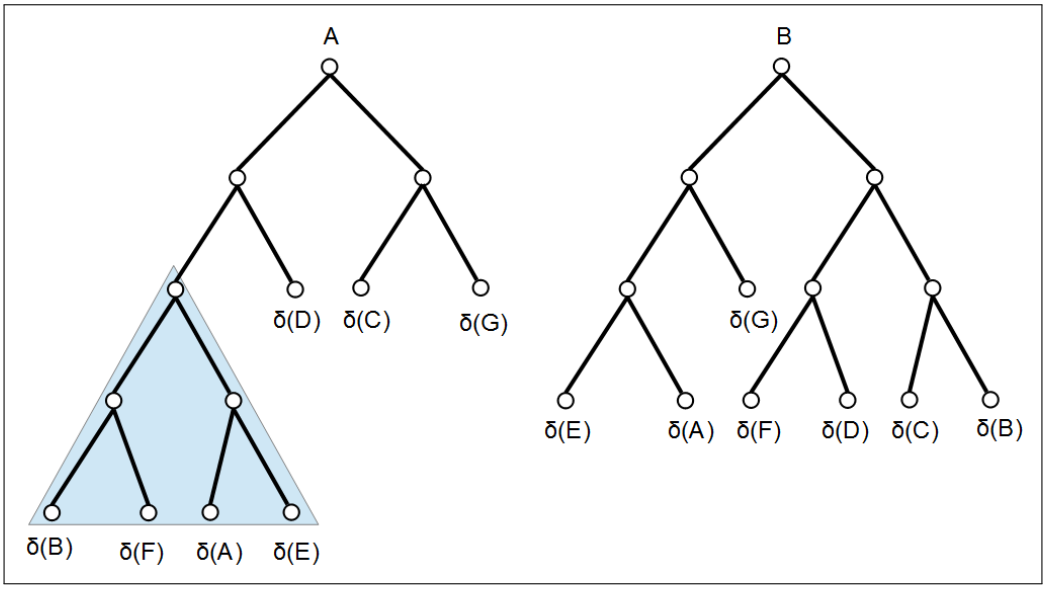

Figure 56: Demonstration of the crossover operator. The chosen subtree of parent A is highlighted in blue.

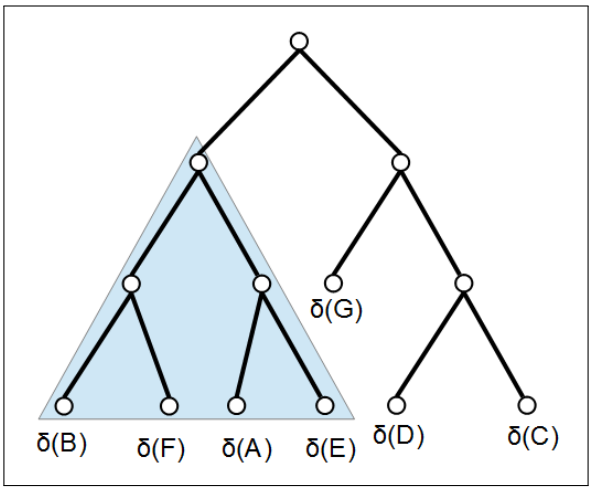

Figure 57: Result of the crossover operator in figure 56.

11 Results

Without testing it is not easy to know how the exact algorithms and optimization algorithms perform, therefore we implemented the algorithms we have discussed in this thesis in C# and used a computer with an Intel i7 processor and 8 GB of RAM to run experiments on these algorithms. A set of representative results of these experiments can be found in this section, where we will first compare the exact algorithms to each other. Then we compare the exact algorithms to the optimization algorithms, after which we look at the performance of the optimization algorithms.

11.1 Exact algorithms

First off, we will be comparing the performance of our exact algorithms to each other. The algorithms we want to compare mostly are Naive MM-Width (MM-Naive), Dynamically Programmed MM-Width (MMDP) and Bounded Dynamically Programmed MM-Width (BMMDP), though we have also chosen to look at the impact of the upper and lower bounds on the performance of Dynamically Programmed MM-Width by running BMMDP without using the upper bound in it (MMDP lb) and running BMMDP without using the lower bound in it (MMDP ub).

We decided to compare the algorithms on clique graphs of increasing size and looking at the amount of time that was needed for the algorithm to finish. Note that we decided to not test an algorithm on a higher size clique graph if a size of clique graph took more than 15 minutes on average to complete, as the order of time needed increases exponentially to complete each larger size of graph, leading to hours of calculations per graph alone. The reason why we decided to show the results of the algorithms on clique graphs, is because these graphs scale in size most naturally and allow us to compare the amount of needed computations. However it may not be as representative for the upper and lower bound comparison. We will compare performance of these further later on.

For Naive MM-Width, the algorithm crashes due to a lack of process memory at K_{10} and beyond which we could have fixed by changing how the algorithm figures out what it should calculate next, though this hardly matters as K_9 takes over 12 hours to complete its calculation as it is.

To generate the entries in the table, we ran each algorithm 10 times on each of the graphs to generate an average running time for each entry in the table. For the upper bound in MMDP ub and BMMDP we use $\lceil n/3 \rceil$. The treewidth of these graphs is $n-1$, meaning that we can't use that for a better upper bound here.

We can see a better representation of the data in figure 58. In this figure we can indeed see that Dynamically Programmed MM-Width improves on the performance of Naive MM-Width, while Bounded Dynamically Programmed MM-Width improves on the performance of both algorithms. To be precise, we can see that MMNaive follows roughly a more than exponential grow before it starts crashing. The other algorithms seem to follow a straight line in the figure though, indicating an exponential growth in running time for these.

Interestingly, we can see a wave-like pattern in both the running time of MMDP ub and BMMDP. These waves are at the highest at the points where the MM-Width of the clique graphs increases by 1 and thus correlate directly to when our upper bound of $\lceil n/3 \rceil$ goes up by 1. It is therefore not surprising that these waves exist,

	MMNaive	MMDP	MMDP lb	MMDP ub	BMMDP
K_2	00:00,0076	00:00,0016	00:00,0022	00:00,0017	00:00,0020
K_3	00:00,0095	00:00,0035	00:00,0056	00:00,0036	00:00,0056
K_4	00:00,0484	00:00,0040	00:00,0061	00:00,0039	00:00,0058
K_5	00:00,2031	00:00,0062	00:00,0061	00:00,0059	00:00,0058
K_6	00:01,2632	00:00,0151	00:00,0074	00:00,0103	00:00,0071
K_7	00:40,5523	00:00,0505	00:00,0108	00:00,0466	00:00,0114
K_8	25:53,7098	00:00,1780	00:00,0204	00:00,1189	00:00,0173
K_9	*	00:00,6286	00:00,0498	00:00,3205	00:00,0298
K_{10}	**	00:02,2391	00:00,1390	00:01,6465	00:00,1219
K_{11}	**	00:08,1472	00:00,4215	00:04,7263	00:00,2650
K_{12}	**	00:28,1353	00:01,2887	00:12,7372	00:00,5961
K_{13}	**	01:39,0747	00:04,0661	01:03,8125	00:02,8893
K_{14}	**	05:43,6491	00:12,7677	02:59,5473	00:06,8782
K_{15}	**	18:39,5527	00:40,6122	08:20,3293	00:16,1941
K_{16}	**	*	02:10,4348	38:11,7188	01:20,1502
K_{17}	**	*	06:54,8677	*	03:16,1813
K_{18}	**	*	24:05,1403	*	07:47,9445
K_{19}	**	*	*	*	39:04,3952

Table 1: Comparing the performance of our exact algorithms on clique graphs.
* *We did not run this experiment.*
** *Algorithm crashes due to lack of memory.*

as the amount of subtrees the two algorithms have to work through increases by a lot when the upper bound is heightened.

We can also see here that (for clique graphs), the addition of upper bounds has a lesser effect on run times than the addition of lower bounds. This also holds for random graphs of random sizes as we can see in table 2 and figure 59.

graph size	MMDP	MMDP lb	MMDP ub	BMMDP
8	00:00,129	00:00,014	00:00,117	00:00,013
9	00:00,447	00:00,042	00:00,289	00:00,027
10	00:01,551	00:00,129	00:01,394	00:00,115
11	00:05,285	00:00,410	00:03,964	00:00,295
12	00:18,838	00:01,252	00:10,851	00:00,657
13	01:06,996	00:04,038	00:53,714	00:03,150

Table 2: Comparing the performance of using upper and lower bounds.

To create table 2 and figure 59, we created a random graph with 13 vertices, connecting each new vertex being added with randomly one to three random other vertices already in the graph. This gave us graph 1, which we present in DIMACS representation on page 78. Then we ran each of the four algorithms 10 times on this graph, giving us the four averages found on the row in table 2 preceded by "13". Removing the highest numbered vertex and all edges connected to it from the graph and running the four algorithms 10 times on the resulting graph then gives us the

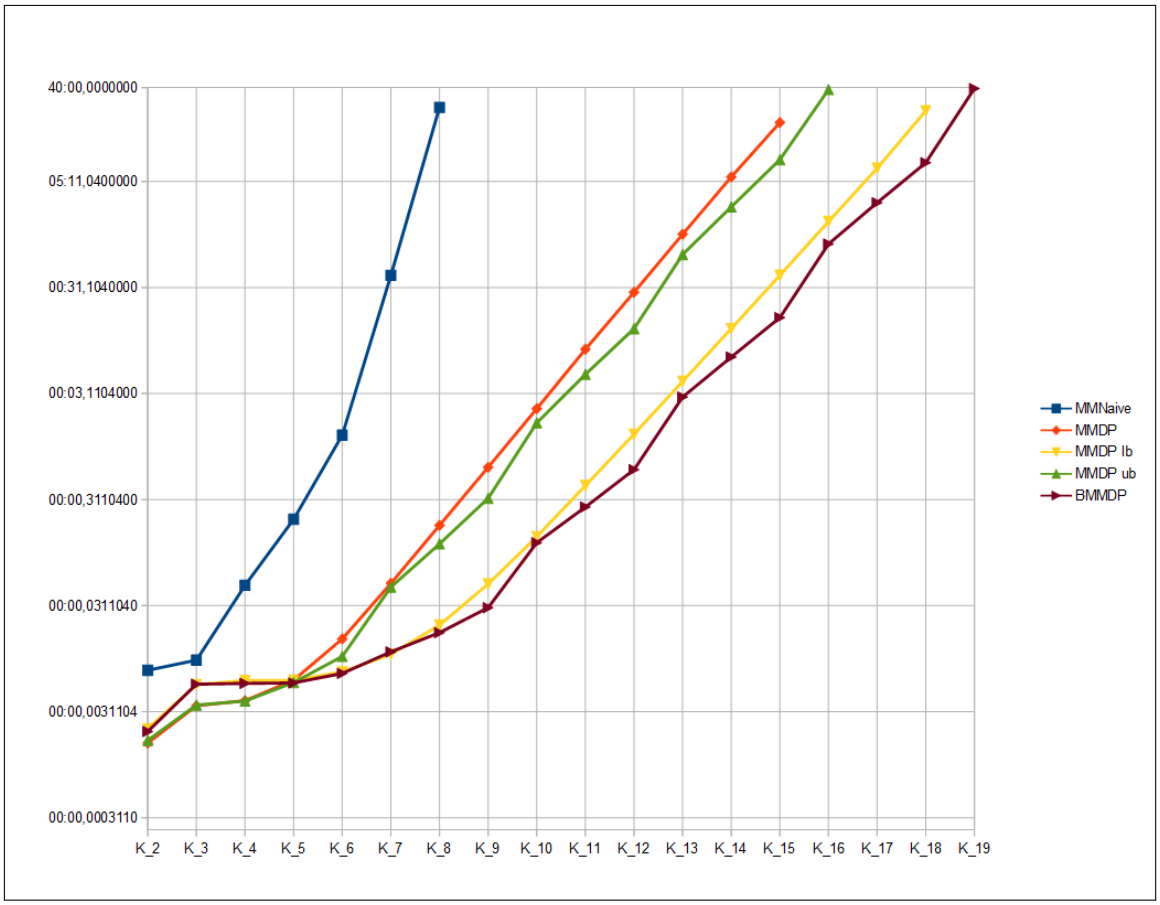

Figure 58: The datapoints in table 1 set against each other on a logarithmic scale.

the averages found on the other rows, each preceded by the number of vertices in the graph at that time.

We can see again that the addition of upper bounds gives a less pronounced difference in performance than the addition of lower bounds. We gain very similar results with other random graphs of the same size, irregardless of changing the number of connections we make for each new vertex. The results in these graphs and figures tell us that the lower bound, which updates while running the algorithm as we make choices to partition our vertices, has a far bigger effect than the strict, non-changing upper bound that we figure out at the start of the algorithm. We should be able to improve further on the upper bound performance if we could discover a way to update it as the algorithm runs. Finding improvements for estimating the lower bound however would also make the performance of BDPMM better.

11.2 Optimization algorithms

Before we compare the optimization algorithms to each other, lets first compare them to our exact algorithms. To do this, we first created another random graph of size 18, creating graph 2 found on page 78. Then we ran both Bounded Dynamically Programmed MM-Width on it 10 times and a few of the more basic optimization algorithms: Iterative and Random Hill Climbing and Iterated Local Search. For each of these we used a random decomposition tree as start solution. For Random Hill Climbing we used 10000 for the maximum amount of tries without a change.

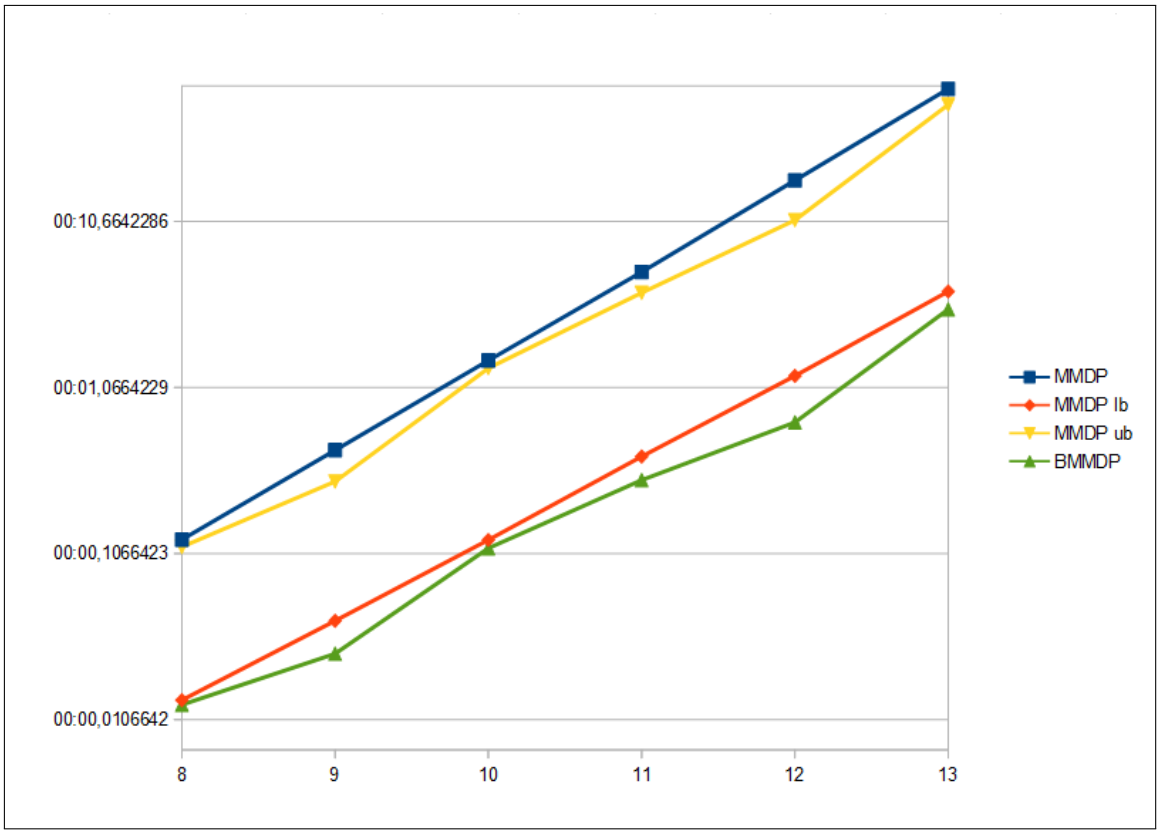

Figure 59: The datapoints in Table 2 set against each other on a logarithmic scale.

We also used this value for the maximum amount of tries without a change for the Iterated Local Search, while for the number of perturbations we used 4 and the base perturbation size and increase in perturbation size we used 4. For Simulated Annealing we have tried many different parameters, but settled on using a starting temperature of 1.6, an ending temperature of 1, a cooling factor of 0.98 and a maximum of one million steps. This because we didn't want Simulated Annealing to simply be Random Hill Climbing and wanted to give the algorithm enough time to find good solutions. For the evolutionary algorithm, we both ran it with truncation selection on a population of size 100 and roulette wheel selection of a population of size 1000. We ran the optimization algorithms 20 times on the graph each.

Algorithm	**Best MM-width**	**Average MM-width**	**Average Time**
BMMDP	4	4	13:40,81
Iterative HC	4	4,55	00:02,69
Random HC	4	4,6	00:04,27
Iterated LS 4, 4	4	4,2	00:15,72
SimAnn	4	4	01:12,39
Truncation100	4	4,7	00:02,14
Roulette1000	4	4,8	00:26,04

Table 3: Comparison of the optimization algorithms on a graph of size 18.

Looking at these results, we can see that with each of the optimization algorithms we were able to find the same MM-width as BMMDP and thus an optimal binary

decomposition tree, though they were not guaranteed to find one. In the situations that they did not find one, they did find a solution with MM-width 5 though, which could be used as an upper bound when running BMMDP as this improves on the clique bound of $\lceil n/3 \rceil = 6$, which would then decrease the running time of the exact algorithm.

Comparing the running times of the algorithms, we can see the optimization algorithms perform a lot faster than our exact algorithms. To be precise, we could run our Iterated Local Search algorithm about 52 times during the time that it took to run BMMDP once, which will only increase the more vertices the graph has that we want to know the MM-width of. Running the Iterated Local Search as a Multi-Start algorithm would have given us an extremely high chance of finding at least one optimal binary decomposition tree here.

With the evolutionary algorithms we can see that we can find the optimal decomposition relatively quickly as well, however not every time. Increasing the population size to 1000 does help the truncation selection variant to find the optimal decomposition almost every time here, however this population size does not help the roulette wheel selection here.

Simulated Annealing always found an optimal solution for us here, though it took quite a bit longer to run than the other algorithms.

However, we would like to use the optimization algorithms to find good binary decomposition trees for graphs that are too large for our exact algorithms. To test this, we will be looking at some bigger graphs and looking at the performance and results of our optimization algorithms. First off, we wanted to test our algorithms on a graph of size 32. Instead of generating a random one, we retrieved a graph of this size from http://treedecompositions.com/ and chose one with a treewidth base bound gap and no exact solution yet. Graph 3 can be found in DIMACS representation in the graph appendix on page 78.

To generate the following graph we used the same parameters as earlier for Random Hill Climbing, Iterated Local Search and Simulated Annealing. We ran each algorithm ten times on the graph.

Algorithm	**Best MM-width**	**Best score**	**Average MM-width**	**Average score**	**Average time**
Iterative HC	5	5231	5,2	5437,2	00:47,0
Random HC	5	5229	5	5230,4	00:18,2
Iterated LS 4, 4	5	5229	5	5229,2	00:59,9
SimAnn	6	6271	6	6272,6	04:01,8
Truncation1000	5	5228	5	5231,6	01:02,2
Roulette1000	6	6278	6,8	7098,2	01:06,8

Table 4: Comparison of optimization algorithms on a graph of size 32.

As you can see in the table, we consistently found the MM-width of the graph to be 5 with the three "basic" optimization algorithms, although Iterative Hill Climbing got stuck on a MM-width of 6 in one instance. Each of the basic algorithms also found these score in less than a minute on average, with Random Hill Climbing never going above 30 seconds.

Simulated Annealing however did not perform well here at all. Each time it

quickly moved towards MM-width scores of 8, after which it would slowly work its way to 6. After arriving at that score, it never found any solution with a MM-width higher than 6. In addition, the scores of each of the found solutions was very similar, even though we gave it quite a lot of time to work on finding better solutions. Making further changes to the temperatures or cooling factor either had the effect of Simulated Annealing not accepting any worsenings, making it perform very similar to Random Hill Climbing, or the algorithm finding higher MM-widths to stabilize at.

For the evolutionary algorithms, we made the population size for both selection methods to 1000. With the truncation selection, this made us find the same score as we found consistently with the "basic" optimization algorithms, but with the roulette selection we were unable to find a score of 5, even when we increased our population size or amount of generations.

We can see similar performance on other graphs of similar sizes. However, as the size of the graph increases, we can also see the performance of the optimization algorithms dwindle in both the run time and the accuracy. The reason why the run time performance dwindles should be clear, as even the Hopcroft Carp algorithm we use to calculate the mm scores of the edges starts taking more time and it is the most called part of the entire algorithm, making it important that we save time by only recalculating the edges that could have actually changed value.

The accuracy on the other hand dwindles because of both the sheer amount of different changes that can be made to the solutions at each point and the local optima that become deeper as the amount of vertices in the graph increase. It appears as if deep local optima are lot more difficult to get out of in our optimization problems as compared to what we see in other optimization problems. This could be because in our main evaluation function, the mm-width of the solutions is extremely important and if one or multiple drops in mm-width are found in the local optimum well, it becomes extremely difficult to climb out. We resolved this partially for our Simulated Annealing algorithm by using a second evaluation function, though if you keep finding those better mm-width solutions while climbing out of the well, you keep falling back into the well.

As a demonstration of some of the problems we get in with a too large graph, we will use a graph of size 64, again from http://treedecompositions.com/ and found as graph 4 in DIMACS representation in the graph appendix on page 78. We decided to use the same settings for the algorithms as in the experiments on the smaller graphs as the settings did not seem to have a lot of influence on the performance here.

Something that is directly clear from this table, is that working on this graph takes a lot of time for both Iterative Hill Climbing and Simulated Annealing. For iterative Hill Climbing this is because just enumerating all possible options and calculating the new scores takes a lot of time and every time it found an improvement it had to start all over again. Each time it found a solution with the same MM-width though. These solutions of this score seem to have a very attractive subspace around it as Random Hill Climbing and Iterated Local Search both ended up in these solutions over and over, even when we changed the perturbation size and amount of Iterated Local Search to larger values.

The evolutionary algorithm did not find the same attractor however, probably because we use a different operator to search through the space, though it still keeps

Algorithm	Best MM-width	Best score	Average MM-width	Average score	Average time
Iterative HC	22	90464	22	90464	30:17
Random HC	22	90464	22	90464	03:24
Iterated LS 4, 4	22	90464	22	90464	12:35
SimAnn	21	86417	21,9	90059,3	56:41
Truncation1000	22	90472	22	90472	16:36
Roulette1000	22	90567	22	90567	16:30

Table 5: Comparison of optimization algorithms on a graph of size 64.

arriving at a mm-width of 22.

Interestingly, Simulated Annealing did find one better solution for this graph during our experiments, one with a MM-width of 21, but it took around an hour to complete that run, with the entire set of the Simulated Annealing runs varying between half an hour and two hours. The other runs of simulate annealing all found solutions with the same score that we have seen over and over. This means that this graph has at most a MM-width of 21, in addition to the reported upper bound to the treewidth of this graph of 25. We were also able to find a solution of mm-width 20, which can be seen in the list of known MM-widths of graphs on page 75

12 Conclusion

In this thesis we have improved the link between the value of MM-width and treewidth and given both exact algorithms and approximation algorithms for calculating the MM-width of a graph.

We have shown that the dynamic programming approach does really improve on the naive approach to find the MM-width of a graph, in addition to the improvement in performance from adding upper and lower bounds to the algorithm. We have also proven the MM-width to some standard variations of graphs, including cactus graphs and k-trees.

There are also still questions related to MM-width open. A selection of those questions brought up from writing this thesis are:
– Does there exist a polynomial kernel for MM-width?
– Can we improve on the upper and lower bounds used in Bounded Dynamically Programmed MM-Width to speed up calculation further?
– Can find an algorithm to approximate MM-width from below?
– Are there other, stricter links between MM-width and the other width parameters?

13 List of Known MM-Widths of Graphs

Finally, mainly for illustration we will list a collection of MM-widths of graphs. We decided to keep this list to all MM-width results of all connected graphs of size 5 and below as the amount of different connected graphs with a certain number of vertices grows exponentially. In addition, we have listed all best results we have found for the graphs found in the graphs appendix on page 78 together with a string representation of the corresponding binary tree decomposition of that MM-width.

13.1 All Connected Graphs of Size ≤ 5

Graph	**MM-Width**
Trivial graph	0
K_2	1
P_3	1
K_3	1
P_4	1
$K_{1,3}$	1
3-*pan*	1
C_4	2
Diamond graph	2
K_4	2
P_5	1
Fork/chair graph	1
$K_{1,4}$	1
C_5	2
4-*pan*	2
Cricket graph	1
Bull graph	1
$(3,2)$-tadpole graph	1
House graph	2
Kite graph	2
Dart graph	2
Gem graph	2
Kite-dart graph	2
W_5	2
$K_5 - e$	2
K_5	2

13.2 Graphs from Graph Addendum

Graph	MM-Width	Decomposition
Graph 1	3	(0, (3, (7, (8, (10, (((1, 2), (4, (6, 9))), (5, (11, 12))))))))
Graph 2	≤ 4	((11, ((13, 12), (((((2, 8), 6), (15, (7, 9))), ((5, 14), (17, (16, (10, 3))))), (4, 0)))), 1)
Graph 3	≤ 5	((((((5, 13), ((16, 2), (11, (26, (0, 12))))), (((25, (20, 9)), (10, 27)), ((((22, 19), 8), ((31, 18), 7)), (24, 29)))), (((30, 15), 4), (3, 28))), ((6, 23), (14, 17))), (21, 1))
Graph 4	≤ 20	((((((43, (15, 20)), (30, (29, 27))), ((45, 42), (24, 21))), (((0, 3), (6, 12)), ((1, 4), (18, 35)))), ((((40, 28), (19, 31)), ((7, 16), (22, 34))), (((5, (2, 11)), ((26, 14), (23, 17))), ((25, 32), (39, 36))))), (((((57, 9), (50, 58)), ((51, 54), (48, 55))), (((13, 10), (46, 53)), ((33, 8), (37, 56)))), ((((44, 49), 52), (38, 41)), (((47, 59), 60), (61, (62, 63))))))

14 Algorithms Appendix

Algorithm 6: AllBinaryTrees

Data: number of leaves required n
Result: a set containting all binary trees with that number of leaves
$result \longleftarrow emptyset$;
if $n = 1$ **then**
    $result.Add(newNode())$;
    return *result*
for $i \longleftarrow 1 \textbf{to} n-1$ **do**
    $left \longleftarrow AllBinaryTrees(i)$;
    $right \longleftarrow AllBinaryTrees(n-i)$;
    foreach *lefttree in left* **do**
        foreach *righttree in right* **do**
            $node \longleftarrow newNode()$;
            $node.leftchild \longleftarrow lefttree$;
            $lefttree.parent \longleftarrow node$;
            $node.rightchild \longleftarrow righttree$;
            $righttree.parent \longleftarrow node$;
            $result.Add(node)$;
return *result*

Algorithm 7: AllOrdersOfLength

Data: length of orders required n
Result: set containting all possible orderings of the numbers 0 through n - 1
return *AllOrdersRecurse({ 0, 1, .. n - 1})*

Algorithm 8: LabelTree

Data: tree to label *tolabel*
list of labels *labels*
Result: set containting all possible orderings of the numbers 0 through n - 1
$LabelTreeRecurse(tolabel, labels, 0)$;
return *tolabel*

Algorithm 9: AllOrdersRecurse

Data: set of elements to give all orders of *input*
result $\longleftarrow$ *emptyset*;
if *input.Count = 2* **then**
    result.Add(*input.Copy*());
    result.Add({*input*[0], *input*[1]});
foreach *element in input* **do**
    rest $\longleftarrow$ *input.Copy*();
    rest.Remove(*element*);
    foreach *tail in AllOrdersRecurse(rest)* **do**
        tail.Insert(0, *element*);
        result.Add(*tail*);
return *result*

Algorithm 10: LabelTreeRecurse

Data: tree to label *tolabel*
list of labels *labels*
index of next label to use *next*
Result: index of next label to use
if *tolabel.leaf* **then**
    tolabel.label $\longleftarrow$ *labels*[*next*];
    return *next + 1*
tolabel.label $\longleftarrow -1$;;
$next \longleftarrow LabelTreeRecurse(tolabel.left, labels, next)$;
$next \longleftarrow LabelTreeRecurse(tolabel.right, labels, next)$;
return *next*

Algorithm 11: TreeScorer

Data: a graph g
a labeled tree *labeledtree*
Result: mm-width of the tree
if *labeledtree.leaf* **then**
    return *-1*
$left \longleftarrow MMTreeTester(g, labeledtree.left)$;
$right \longleftarrow MMTreeTester(g, labeledtree.right)$;
$scoreleft \longleftarrow HopcroftKarp(G[V_{labeledtree.left}, V_{\bar{labeledtree.left}}])$;
$scoreright \longleftarrow HopcroftKarp(G[V_{labeledtree.right}, V_{\bar{labeledtree.right}}])$;
$result \longleftarrow left$;
if $right > result$ **then** $result \longleftarrow right$;
if $scoreleft > result$ **then** $result \longleftarrow scoreleft$;
if $scoreright > result$ **then** $result \longleftarrow scoreright$;
return *result*

15 Graphs Appendix

Graph 1:

p edge 13 22

e 1 4	e 1 7
e 1 9	e 1 12
e 2 3	e 2 5
e 2 7	e 3 4
e 3 10	e 3 12
e 4 6	e 4 8
e 4 12	e 5 6
e 5 10	e 5 13
e 6 7	e 6 11
e 7 13	e 8 9
e 9 11	e 11 13

Graph 2:

p edge 18 36

e 1 2	e 1 3
e 1 4	e 1 5
e 1 7	e 1 11
e 2 3	e 2 6
e 2 9	e 2 13
e 3 4	e 3 8
e 3 9	e 3 11
e 4 5	e 4 6
e 4 17	e 4 18
e 5 9	e 5 14
e 6 7	e 7 8
e 7 10	e 7 11
e 7 12	e 8 10
e 8 16	e 9 12
e 10 12	e 10 16
e 11 15	e 11 17
e 12 13	e 12 14
e 12 15	e 13 14

Graph 3:

p edge 32 43

e 1 13	e 1 27
e 2 14	e 2 24
e 3 17	e 3 27
e 4 6	e 4 17
e 4 18	e 4 31
e 5 16	e 5 31
e 6 23	e 7 18
e 7 24	e 8 20
e 8 30	e 9 20
e 9 23	e 10 21
e 10 26	e 11 21
e 11 30	e 12 13
e 12 14	e 12 19
e 13 32	e 14 15
e 15 16	e 15 18
e 16 17	e 19 20
e 19 21	e 22 23
e 22 24	e 22 25
e 25 26	e 25 28
e 26 27	e 28 29
e 28 32	e 29 30
e 29 31	

Graph 4:

p edge 64 536

e 1 2	e 1 3
e 1 5	e 1 16
e 1 17	e 1 19
e 1 20	e 1 23
e 1 24	e 1 37
e 1 41	e 2 3
e 2 4	e 2 5
e 2 8	e 2 12
e 2 13	e 2 14
e 2 15	e 2 16
e 2 17	e 2 18
e 2 19	e 2 20
e 2 21	e 2 23
e 2 37	e 2 38
e 2 41	e 2 42
e 3 4	e 3 5
e 3 6	e 3 8
e 3 13	e 3 16
e 3 17	e 3 20
e 3 37	e 3 40
e 3 41	e 3 42
e 3 43	e 3 44
e 3 46	e 4 5
e 4 6	e 4 7
e 4 8	e 4 9
e 4 10	e 4 11
e 4 12	e 4 13
e 4 14	e 4 16
e 4 17	e 4 18
e 4 21	e 4 34
e 4 37	e 4 38
e 4 39	e 4 41
e 4 42	e 4 43
e 4 45	e 4 49
e 4 58	e 4 64
e 5 6	e 5 7
e 5 8	e 5 9
e 5 12	e 5 13
e 5 16	e 5 43
e 6 7	e 6 8
e 6 43	e 6 44
e 6 45	e 6 47
e 7 8	e 7 9

Graph 4 (cont.):

e 7 12 e 7 43
e 7 45 e 7 64
e 8 9 e 8 10
e 8 11 e 8 12
e 8 13 e 8 15
e 8 16 e 8 17
e 8 42 e 8 43
e 9 10 e 9 11
e 9 12 e 9 13
e 9 14 e 9 15
e 9 16 e 9 43
e 9 63 e 9 64
e 10 11 e 10 12
e 10 13 e 10 14
e 10 15 e 10 17
e 10 18 e 10 34
e 10 38 e 10 39
e 10 41 e 10 42
e 10 43 e 10 45
e 10 46 e 10 49
e 10 52 e 10 56
e 10 58 e 10 59
e 10 60 e 10 61
e 10 62 e 10 63
e 10 64 e 11 12
e 11 13 e 11 14
e 11 15 e 11 18
e 11 38 e 11 58
e 11 59 e 11 61
e 11 63 e 11 64
e 12 13 e 12 14
e 12 15 e 12 16
e 12 17 e 12 19
e 13 14 e 13 15
e 13 16 e 13 17
e 13 18 e 13 21
e 13 34 e 13 37
e 13 38 e 13 39
e 13 40 e 13 41
e 13 42 e 13 43
e 13 49 e 13 58
e 14 15 e 14 16
e 14 17 e 14 18
e 14 19 e 14 21
e 14 22 e 14 26
e 14 31 e 14 34
e 14 37 e 14 38
e 14 39 e 14 42

Graph 4 (cont.):

e 14 49 e 14 52
e 14 56 e 14 57
e 14 58 e 14 59
e 14 61 e 14 62
e 14 63 e 15 16
e 15 17 e 15 18
e 15 19 e 15 20
e 15 22 e 15 58
e 16 17 e 16 18
e 16 19 e 16 20
e 16 23 e 16 37
e 16 38 e 16 42
e 17 18 e 17 19
e 17 20 e 17 21
e 17 22 e 17 24
e 17 26 e 17 30
e 17 31 e 17 34
e 17 37 e 17 38
e 17 39 e 17 40
e 17 41 e 17 42
e 17 43 e 17 49
e 17 58 e 18 19
e 18 20 e 18 21
e 18 22 e 18 26
e 18 28 e 18 30
e 18 31 e 18 34
e 18 37 e 18 38
e 18 58 e 19 20
e 19 21 e 19 22
e 19 23 e 20 21
e 20 22 e 20 23
e 20 24 e 20 26
e 20 30 e 20 37
e 20 38 e 20 41
e 21 22 e 21 23
e 21 24 e 21 25
e 21 26 e 21 27
e 21 28 e 21 30
e 21 31 e 21 32
e 21 33 e 21 34
e 21 37 e 21 38
e 21 41 e 21 58
e 22 23 e 22 24
e 22 25 e 22 26
e 22 31 e 22 34
e 22 58 e 23 24
e 23 25 e 23 26
e 24 25 e 24 26

Graph 4 (cont.):

e 24 27 e 24 30
e 24 31 e 24 33
e 24 37 e 25 26
e 25 27 e 25 30
e 25 31 e 26 27
e 26 28 e 26 29
e 26 30 e 26 31
e 26 33 e 26 34
e 26 37 e 26 38
e 26 58 e 27 28
e 27 29 e 27 30
e 27 31 e 27 32
e 27 33 e 28 29
e 28 30 e 28 31
e 28 32 e 28 34
e 28 57 e 28 58
e 29 30 e 29 31
e 29 32 e 29 33
e 29 35 e 29 36
e 29 55 e 30 31
e 30 32 e 30 33
e 30 34 e 30 36
e 30 37 e 30 38
e 31 32 e 31 33
e 31 34 e 31 35
e 31 37 e 31 38
e 31 57 e 31 58
e 32 33 e 32 34
e 32 35 e 32 36
e 32 54 e 32 55
e 32 56 e 32 57
e 32 58 e 33 34
e 33 35 e 33 36
e 33 37 e 33 38
e 33 40 e 33 54
e 34 35 e 34 36
e 34 37 e 34 38
e 34 39 e 34 42
e 34 49 e 34 52
e 34 56 e 34 57
e 34 58 e 34 59
e 34 60 e 35 36
e 35 37 e 35 38
e 35 39 e 35 40
e 35 49 e 35 50
e 35 52 e 35 53
e 35 54 e 35 55
e 35 56 e 35 57

Graph 4 (cont.):

e 35 58
e 36 37
e 36 38
e 36 39
e 36 40
e 36 41
e 36 49
e 36 50
e 36 54
e 37 38
e 37 39
e 37 40
e 37 41
e 37 42
e 37 49
e 37 58
e 38 39
e 38 40
e 38 41
e 38 42
e 38 49
e 38 52
e 38 56
e 38 57
e 38 58
e 38 62
e 39 40
e 39 41
e 39 42
e 39 46
e 39 47
e 39 48
e 39 49
e 39 50
e 39 52
e 39 53
e 39 54
e 39 56
e 39 58
e 39 60
e 39 62
e 40 41
e 40 42
e 40 46
e 40 49
e 40 50
e 41 42
e 41 43
e 41 46
e 41 49
e 42 43
e 42 44
e 42 46
e 42 49
e 42 50
e 42 52
e 42 56
e 42 58
e 43 44
e 43 45
e 43 46
e 43 47
e 43 48
e 43 49
e 43 64
e 44 45
e 44 46
e 44 47
e 44 48
e 44 50
e 45 46
e 45 47
e 45 48
e 45 51
e 45 62
e 45 63
e 45 64
e 46 47
e 46 48
e 46 49
e 46 50
e 47 48
e 47 49
e 47 50
e 47 51
e 47 62
e 48 49
e 48 50
e 48 51
e 48 52
e 48 56
e 48 60
e 48 61
e 48 62
e 48 63
e 48 64

Graph 4 (cont.):

e 49 50
e 49 51
e 49 52
e 49 53
e 49 54
e 49 56
e 49 57
e 49 58
e 49 60
e 49 61
e 49 62
e 50 51
e 50 52
e 50 53
e 50 54
e 50 56
e 50 62
e 51 52
e 51 53
e 51 56
e 51 62
e 52 53
e 52 54
e 52 55
e 52 56
e 52 57
e 52 58
e 52 59
e 52 60
e 52 61
e 52 62
e 53 54
e 53 55
e 53 56
e 53 60
e 53 62
e 54 55
e 54 56
e 55 56
e 55 57
e 55 60
e 56 57
e 56 58
e 56 59
e 56 60
e 56 61
e 56 62
e 57 58
e 57 59
e 57 60
e 57 61
e 57 62
e 58 59
e 58 60
e 58 61
e 58 63
e 59 60
e 59 61
e 59 62
e 59 63
e 60 61
e 60 62
e 61 62
e 61 63
e 62 63
e 63 64

References

[1] Martin Vatshelle, *New Width Parameters of Graphs* (2012).

[2] Hans L Bodlaender, *A partial k-arboretum of graphs with bounded treewidth*, Theoretical computer science **209** (1998), no. 1, 1–45.

[3] Hans L Bodlaender and Arie MCA Koster, *Safe Separators for Treewidth*, Discrete Mathematics **306** (2006), no. 3, 337–350.

[4] Hans L Bodlaender, Bart MP Jansen, and Stefan Kratsch, *Preprocessing for Treewidth: A Combinatorial Analysis through Kernelization*, SIAM Journal on Discrete Mathematics **27** (2013), no. 4, 2108–2142.

[5] Hans L Bodlaender, Bart MP Jansen, and Stefan Kratsch, *Kernel Bounds for Structural Parameterizations of Pathwidth*, Algorithm Theory–Swat 2012 (2012), 352–363.

[6] Bart MP Jansen, *On Sparsification for Computing Treewidth*, Algorithmica **71** (2015), no. 3, 605–635.

[7] Charles C Palmer and Aaron Kershenbaum, *Representing Trees in Genetic Algorithms*, Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on (1994), 379–384.

[8] John E Hopcroft and Richard M Karp, *A n5/2 Algorithm for Maximum Matchings in Bipartite Graphs*, Switching and Automata Theory, 1971., 12th Annual Symposium on (1971), 122–125.

[9] John H Conway and Richard Guy, *The Book of Numbers*, Springer Science & Business Media, 2012.

[10] Gordon E Moore, *Cramming More Components onto Integrated Circuits*, Proceedings of the IEEE **86** (1998), no. 1, 82–85.

[11] Thomas H Cormen, *Introduction to Algorithms*, MIT press, 2009.

[12] Lilian Markenzon, Claudia M Justel, and Newton Paciornik, *Subclasses of K-Trees: Characterization and Recognition*, Discrete Applied Mathematics **154** (2006), no. 5, 818–825.

[13] Petra Scheffler, *Die Baumweite von Graphen als ein Maß für die Kompliziertheit Algorithmischer Probleme* **4** (1989).

[14] J van Leeuwen, *Graph Algorithms*, North-Holland Handbook of theoretical Computer Science; Algorithms and Complexity Theory, Amsterdam (1990), 527–631.

[15] Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 1995.

[16] Robert E Tarjan, *Decomposition by Clique Separators*, Discrete mathematics **55** (1985), no. 2, 221–232.

[17] Jisu Jeong, Seongmin Ok, and Geewon Suh, *Characterizing Graphs of Maximum Matching Width at most 2*, arXiv preprint arXiv:1606.07157 (2016).

[18] Jisu Jeong, Sigve H Sæther, and Jan A Telle, *Maximum Matching Width: New Characterizations and a Fast Algorithm for Dominating Set*, arXiv preprint arXiv:1507.02384 (2015).