DIMITRI RUSIN

TREE DECOMPOSITIONS: MINIMIZING THE SIZE OF JOIN NODES

TREE DECOMPOSITIONS: MINIMIZING THE SIZE OF JOIN NODES

DIMITRI RUSIN

Master of Science (M.Sc.)
Theoretical Computer Science Department
Computer Science
RWTH Aachen University

April 30, 2020

Dimitri Rusin: *Tree Decompositions: Minimizing the Size of Join Nodes*
Master of Science (M.Sc.), © April 30, 2020

SUPERVISORS:
Prof. Peter Rossmanith
Prof. Martin Grohe
Jan Dreier, M.Sc.

LOCATION:
Aachen

ABSTRACT

This thesis deals with tree decompositions and, more specifically, the question how to improve a graph algorithm, that exploits a tree decomposition to compute its result. The usual measurement for the value of a tree decomposition is its width. But in this thesis we propose another measurement for the value of a tree decomposition, its **join width**: This is the maximum bag size of nodes of degree at least three. We evaluate the runtime of the graph algorithm `Sequoia` using minimal-width tree decompositions and using tree decompositions of lower joinwidth, whose width is not necessarily minimal. These runtimes and their conclusions are presented in the thesis.

ZUSAMMENFASSUNG

In dieser Masterarbeit geht es um Baumzerlegungen und die Frage, wie man einen Algorithmus, der eine Baumzerlegung nutzt, effizienter machen kann. Das übliche Maß für den Wert einer Baumzerlegung ist ihre Weite. Aber in dieser Masterarbeit schlagen wir ein anderes Maß für den Wert einer Baumzerlegung vor, ihre **Join-Weite**: Das ist die maximale Taschengröße aller Join-Knoten der Baumzerlegung. Wir messen die Laufzeit des Algorithmus `Sequoia`, wobei wir auf der einen Seite Baumzerlegungen mit optimaler Weite und auf der anderen Seite Baumzerlegungen mit nicht-optimaler Weite, aber mit besserer Join-Weite, verwenden. Diese Laufzeiten und unsere Schlussfolgerungen werden in dieser Arbeit vorgestellt.

ACKNOWLEDGMENTS

I'd like to express my thanks to Jan Dreier for reviewing and evaluating my progressing work on this Master's thesis :)

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

WELCOME!

Welcome to my Master's thesis. Be ready to be taken on a stroll through the world of tree decompositions and computer program runtimes!

1 INTRODUCTION

This thesis is about graphs, more precisely about tree decompositions of graphs. We define a new property of tree decompositions and compute tree decompositions with this property to evaluate the runtime of `Sequoia` on these tree decompositions. The program `Sequoia` takes as input a graph and a tree decomposition to compute some NP hard parameter of the graph using that tree decomposition. Will `Sequoia` run faster using tree decompositions with this new property?

Now, what is this new property? But first: What is the old and known property of tree decompositions?

- *(Old)* Width: How many vertices can a bag fit at maximum (minus one)?
- *(New)* Join width: How many vertices can a bag fit at maximum (minus one), if this bag lies at an intersection with at least three other bags?

The ultimate question is: If we want a better runtime of `Sequoia`, that is, a program which uses this tree decomposition, can we sacrifice width for join width?

The conclusion is: There are indeed graph and formula combinations, that yield better runtimes, if the join width is smaller than the width by at least 2, or if the tree decomposition is a plain path. But on many, many cases it is better to just use an optimum-width tree decomposition.

The details of the thesis are: The author has implemented an algorithm, that, given a graph, computes a tree decomposition, whose join width is at most its tree width minus one. This algorithm was tested on 70 graphs. On 19 of them, it ran for longer than one minute, so it was stopped prematurely. On the other 51 graphs, it halted with a tree decomposition. These 51 graphs yielded a sequence of runtime diagrams, that are presented in this thesis.

Each graph and tree decomposition were used by another program to compute a graph parameter. This other program is Sequoia together with another special input. This special input defines the specific graph parameter, that Sequoia is computing. It is an MSO formula. There are 15 MSO formulae. Each MSO formula, together with Sequoia, yields a program, that uses a tree decomposition, to compute an NP hard parameter of a given graph.

In this thesis, a decomposer is an algorithm that, given a graph, computes a tree decomposition of that graph. There are many decomposers on the Internet. The author has collected three of them from the

PACE website. PACE is an organization, that organized two Software competitions to compute tree decompositions. The two competitions took place in 2016 and 2017. The submitted decomposers were made publicly available.

These decomposers compute tree decompositions of minimum width. But the algorithm, that has been written for this thesis, works in a different way. Let's say, our given graph is G and has tree width k_G. Then the algorithm computes some tree decomposition with

- width k' at most $k_G + 1$,
- and join width j' at most $k_G - 1$.

This means, that we have these two properties satisfy $j' \leq k' - 2$. The least we can say about this approach is that this is a new kind of requirement on a tree decomposition, that might make it better, in the sense that Sequoia might make more efficient use of this tree decomposition, when computing an NP hard graph parameter.

Note, that we cannot easily switch from minimizing the width of a tree decomposition to minimizing the join width instead. Because the minimum join width of any graph is -1: Make a single-node tree decomposition and put all vertices into this single node. Since there are no join nodes, the join width is -1. That's why the join width of a tree decomposition can only be meaningfully minimized, after we enforce a specific width.

The author chose to use the tree width as that specific width. But the author did not minimize the join width with respect to this specific width, but only searched for a tree decomposition, whose join width is at most this specific width minus two, as shown in the above equation. This way, the author tried to improve the join width at least a little bit, instead of optimizing the join width with respect to one or several specific widths.

Probably, the best diagram in this thesis is 5.1. This shows, how well minimizing the size of join nodes in a tree decomposition can work. Other diagrams, where it does not work so well, are also presented in chapter 5.

2 DEFINITIONS

2.1 BASICS

The set of natural numbers $0, 1, 2, \ldots$ is denoted by $\mathbb{N}$.

Let V be some set. Let $v_1, v_2 \in V$. Then we denote $v_1v_2 := (v_1, v_2)$.

Let $E \subseteq V^2$. Let $m \in \mathbb{N}$ with $m \geq 1$. Let $s := v_1, \ldots, v_m$ be a sequence of elements of V with the following two properties:

1. For all $i \in [1, m-1]$, we have $v_1v_{i+1} \in E$.
2. $|\{v_1, \ldots, v_{m-1}\}| = |\{v_2, \ldots, v_m\}| = m - 1$.

We might have $v_1 = v_m$. Then s is called an E-path from v_1 to v_m.

Let $v, w \in V$. If $vw \in E$, then v is an E-predecessor of w, and w an E-successor of v.

Define

$$vE := \{x \in V | vx \in E\}$$

and

$$Ew := \{y \in V | yw \in E\}.$$

2.2 GRAPHS

Let V be some set and $E \subseteq V^2$. If for every $vw \in E$ we have $wv \in E$, then $G := (V(G), E(G)) := (V, E)$ is called a graph. Elements of $V(G)$ are *vertices* and elements of $E(G)$ are *edges*. For $vw \in E(G)$, we might write $\{v, w\} := vw$.

Let G, H be graphs. H is a subgraph of G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$.

Let $W \subseteq V(G)$. Define

$$\begin{aligned} G[W] &:= (W, E(G) \cap W^2), \\ G \backslash W &:= G[V(G) \backslash W], \\ G \cup H &:= (V(G) \cup V(H), E(G) \cup E(H)). \end{aligned}$$

We call $G[W]$ the W-induced subgraph of G. If $V(G) = \emptyset$, then G is *empty*.

A connected component C of G is a subgraph of G, such that

1. for all $v, w \in V(C)$, there is an $E(C)$-path from v to w,
2. and for all $v \in V(G) \setminus V(C)$ and $w \in V(C)$, we have $vw \notin E(G)$.

The graph G has a certain set of connected components $C_1, \ldots, C_m$ (with an arbitrary ordering and $m \in \mathbb{N}$), so that we have

$$G = \dot{\bigcup_{i \in [1,m]}} C_i.$$

If $m = 1$, G is *connected*.

Let $F \subseteq V^2$ and $r \in V$. A *rooted tree* is a structure $((V, F), r)$ such that for all $v \in V$, there is exactly one F-path from r to v. The *graph underlying the rooted tree* is the graph $(V, F \cup F^{-1})$, where

$$F^{-1} := \{wv | vw \in F\}.$$

Let T be a tree. Vertices of T are called *nodes* and vertices, that have at most one neighbour are called *leaves*.

A path P is a graph if its vertex set can be written $V(P) = \{v_1, \ldots, v_n\}$ so that its edge set is equal to

$$E(P) = \{v_i v_{i+1} | i \in [1, n-1]\}.$$

A 2-*subdivision* H of G is another graph, which is created from G by replacing every edge of G with a path of length 2. A 2-subdivision might have more vertices than the original graph.

Part II

HOW TO MINIMIZE THE SIZE OF JOIN NODES

In this part, we will design a very simple algorithm based on the cops-and-robber game.

3

HABIMM DECOMPOSER

3.1 DEFINITION OF A TREE DECOMPOSITION

Let G be a graph. A tree decomposition for G is a triple (T, r, β), where (T, r) is a rooted tree and β is a mapping $V(T) \longrightarrow 2^{V(G)}$ satisfying the following two properties:

1. For all $vw \in E(G)$, there is a node $t \in V(T)$ such that $v \in \beta(t)$ and $w \in \beta(t)$.
2. For all $v \in V(G)$, the induced subgraph $T[\{t \in V(T) | v \in \beta(t)\}]$ is a non-empty tree.

In section 3.8, we will define the cops-and-robber game formally. But before we dive into formalities, come with me on a journey with pictures and intuitive explanations.

3.2 EXAMPLE OF A TREE DECOMPOSITION

Let's turn our attention to the forest in Figure 3.1.

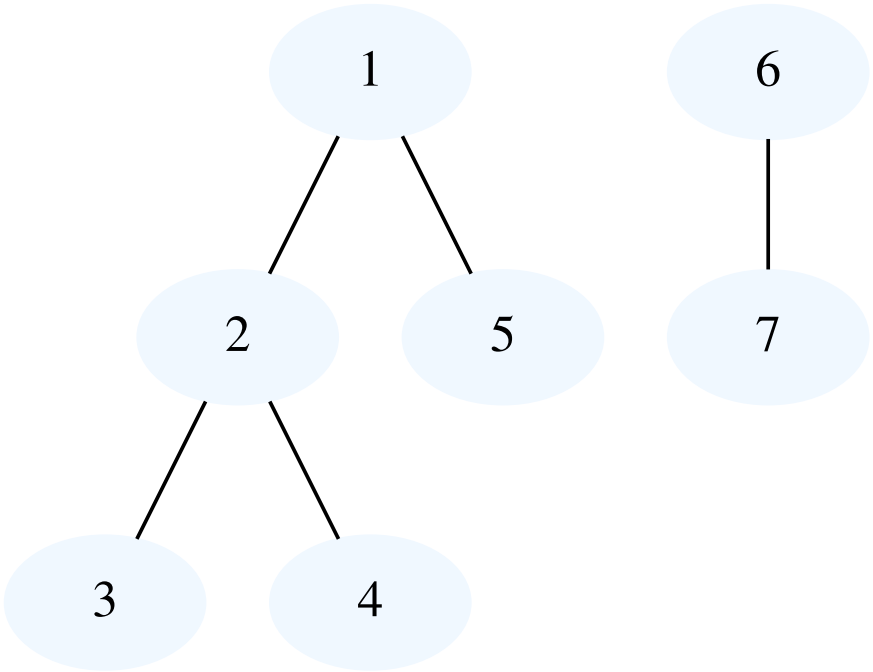

Figure 3.1: A forest; call it G_0

Now, Figure 3.2 shows a tree decomposition of this forest.

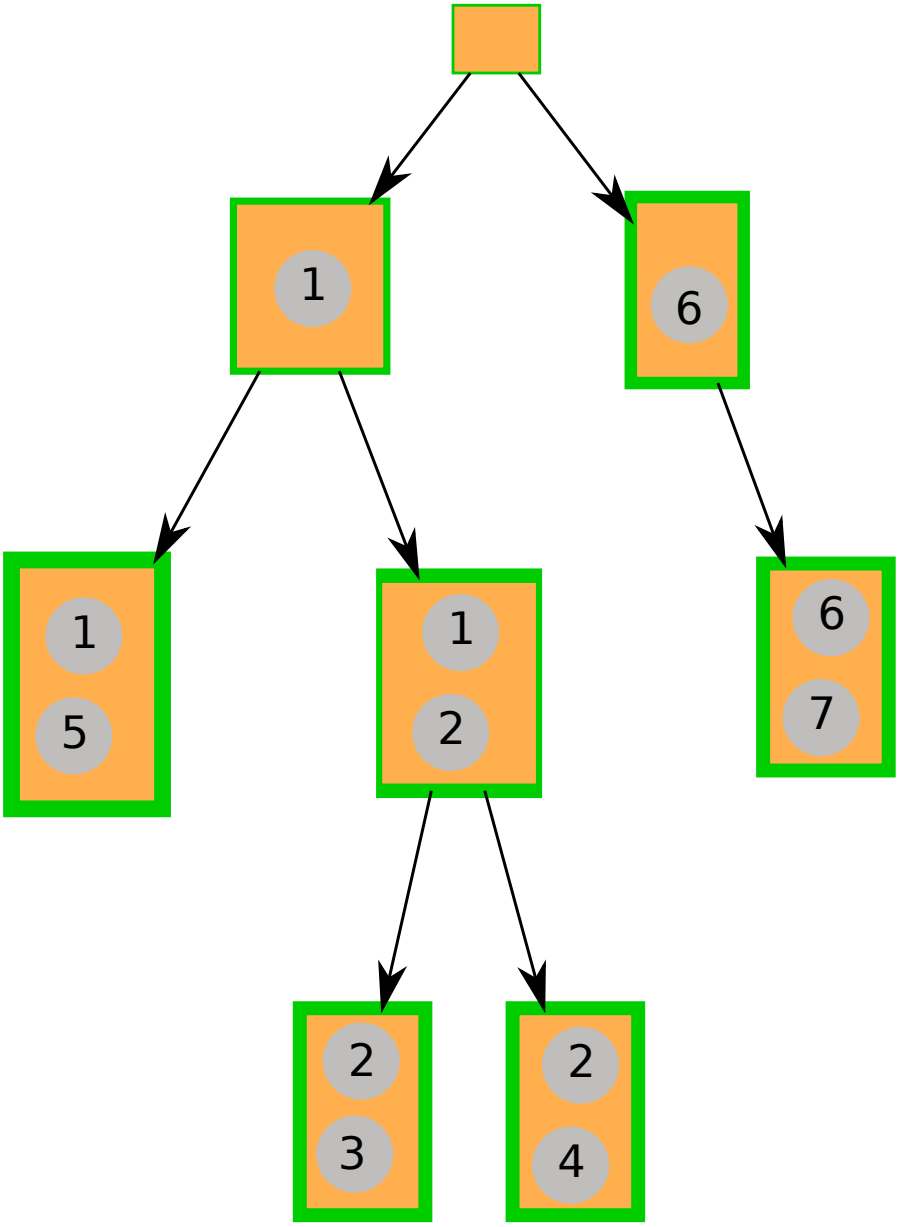

Figure 3.2: Tree decomposition (T_0, r_0, β_0) of forest G_0

This tree decomposition covers all five edges with five 2-vertex bags. For example, the edge $\{1, 5\}$ is covered by the left-most node. Thus, the first condition on β_0 is fulfilled. Furthermore, every vertex spans a non-empty subtree of T_0. For example, vertex 1 spans the 3-node subtree in the middle. Thus, the second condition on β_0 is fulfilled.

In the next sections, we will enrich this tree decomposition with more structure by labeling its nodes and edges and adding more subtrees. The result will be a search tree, that, when fully expanded, includes all tree decompositions of this forest. Our Habimm decomposer will do nothing but construct this search tree, until the first tree decomposition is found, that satisfies our bound requirements.

3.3 LABELING THE NODES OF A TREE DECOMPOSITION

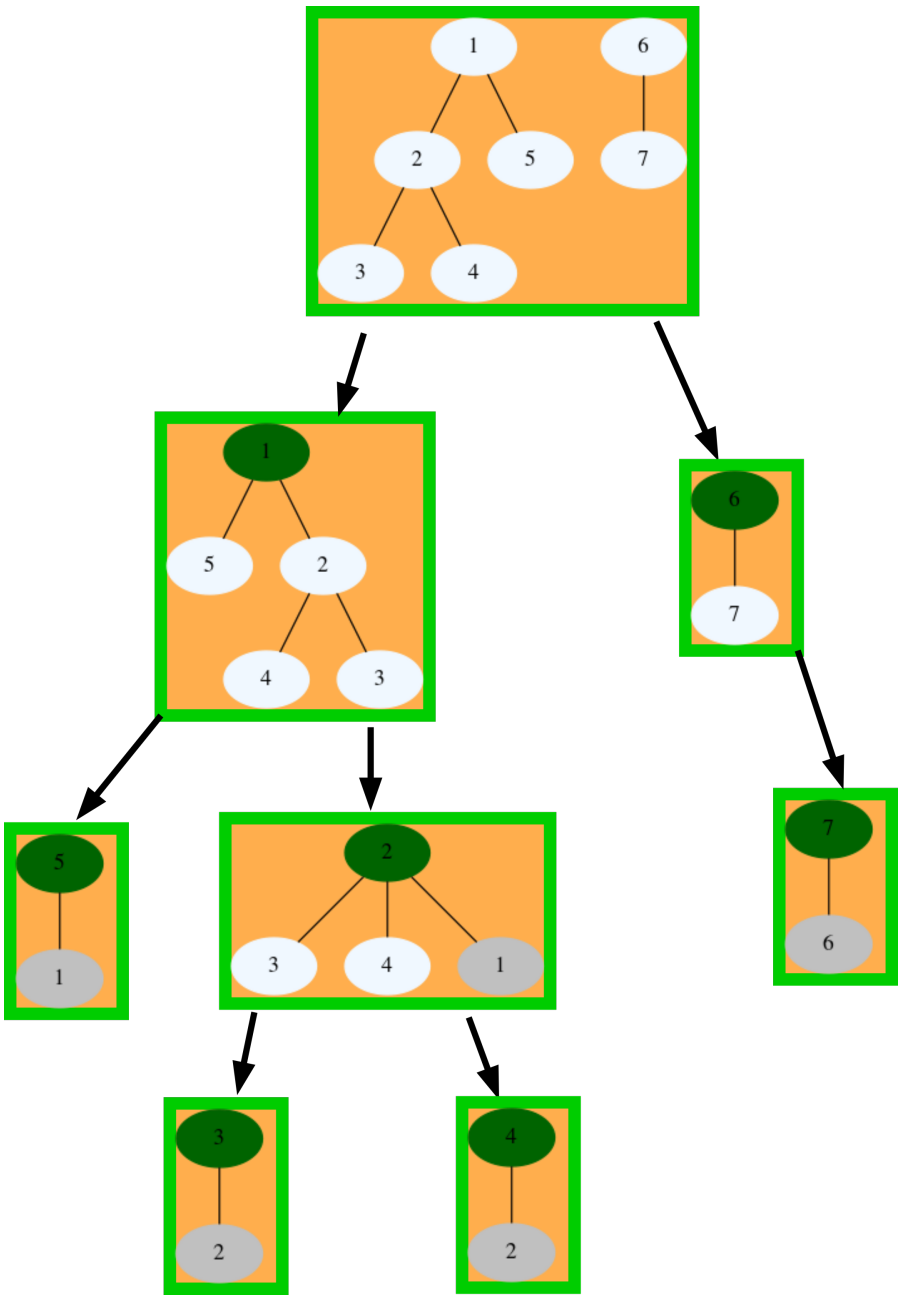

Figure 3.3: The attack of the cops on the forest G_0

In Figure 3.3, in each node $t \in V(T)$, we replace the bag $\beta(t)$ with some subgraph of G_0 containing this bag. In the tree decomposition 3.2, only one new vertex is added to the bag per level of the tree. This new vertex is colored green and the other vertices in the bag are colored gray. The set of all the green or gray vertices is the bag.

Let's make this structure consisting of a graph and a bag more precise. A *cop graph* H is a triple $(V, E, \textbf{cops})$, where

- (V, E) is a graph,
- **cops** $\subseteq V$ marks some vertices of this graph, the *cop vertices*.

We need the following short definition in order to concisely describe the robber components of a cop graph. Let W_1, W_2 be two sets and $E \subseteq W_1 \times W_2$. Then

$$W_1 \cap_E W_2 := \{w_1 \in W_1 | w_1 E \neq \emptyset\}.$$

Note, that we do not generally have $W_1 \cap_E W_2 = W_2 \cap_E W_1$, and that $W_1 \cap_E W_2 \subseteq W_1$.

Let H be a cop graph. Let $H' := H \setminus \mathbf{cops}(H)$. Let $C'_1, \ldots, C'_m$ be the connected components of the graph H'. Then the *robber components* of the cop graph H are $C_1, \ldots, C_m$ with

$$C_i := H[V(C'_i) \cup (\mathbf{cops}(H) \cap_{E(H)} V(C'_i))]$$

for all $i \in [1, m]$. Examples of robber components are shown in blue nodes in Figure 3.4.

3.4 LABELING THE EDGES OF A TREE DECOMPOSITION

In Figure 3.4, we label the edges of the tree decomposition as well. For that we form a 2-subdivision of the former structure. Then we get additional nodes for the edges. Then we label these additional nodes, which we will call *edge nodes* for now.

An edge node is labeled as follows: A original node of the tree decomposition contains a cop graph. This cop graph can be decomposed into its robber components. Now every robber component of this cop graph becomes the label of some outgoing edge node of this original node.

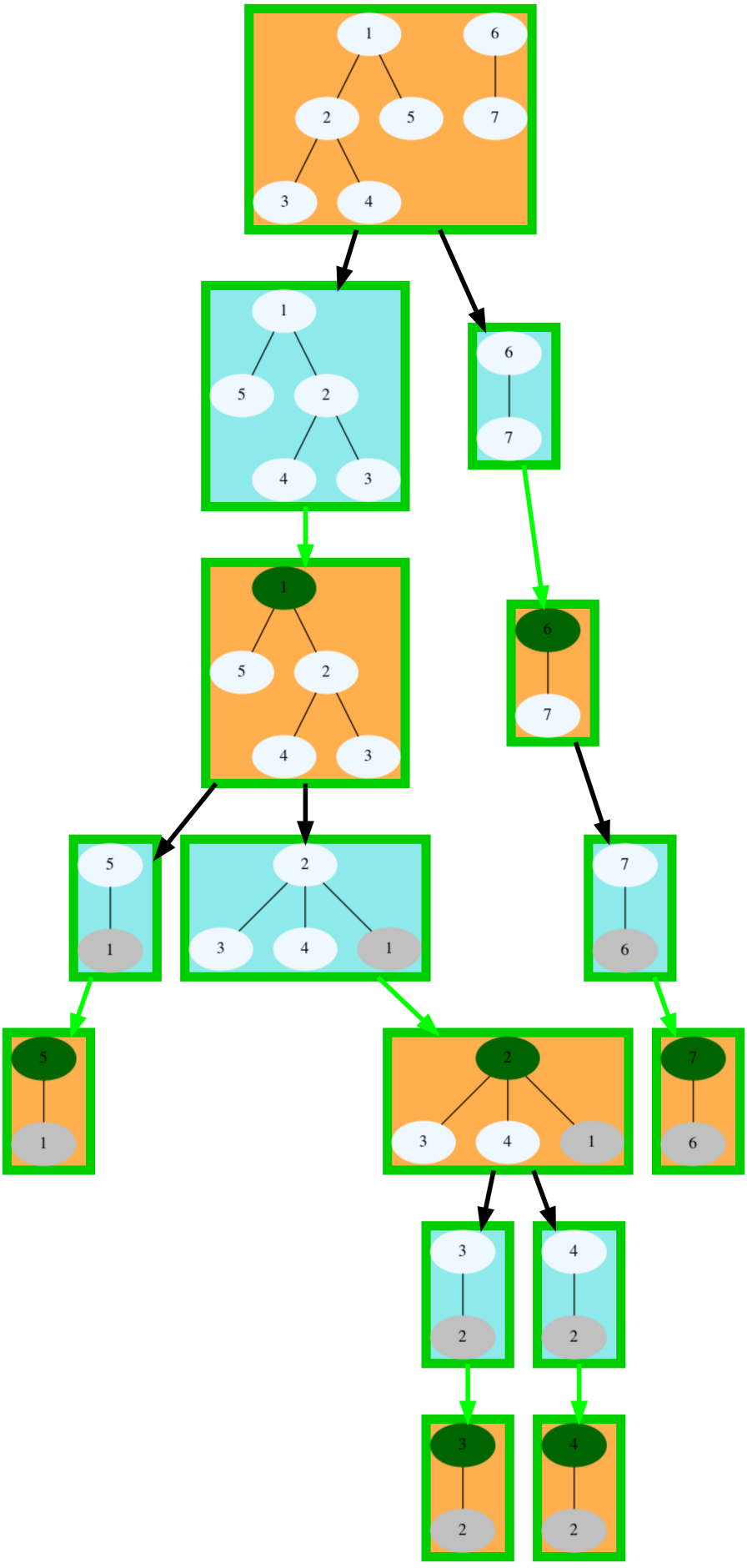

Figure 3.4: Labeling the edges of the tree decomposition with robber components

3.5 DECOMPOSITION INTO CONNECTED COMPONENTS

Let H be a cop graph. A vertex f of H with $f \notin \mathbf{cops}(H)$ is called a *robber vertex of* H.

Algorithm 1 Decomposition of a cop graph into robber components

```
1:  function DecomposeCopGraph(CopGraph)
2:      decomposition ← empty list []
3:      for robber vertex f in CopGraph
4:          fresh ← True
5:          for component from decomposition
6:              if vertex f in the component
7:                  fresh ← False
8:          if fresh is True                          ▷ induce a robber
9:              vertex set ← []                       ▷ component from
10:             worklist ← [f]                        ▷ fresh vertex f
11:             while worklist is not empty
12:                 pop a vertex v off the worklist
13:                 add vertex v to the vertex set
14:                 if v is a robber vertex in CopGraph
15:                     for vertex v's neighbour w in CopGraph
16:                         if neigbour w not in the vertex set
17:                             add neighbour w to the worklist
18:             component ← empty cop graph
19:             for vertex v in the vertex set
20:                 add vertex v to the component
21:                 if v is a robber vertex in CopGraph
22:                     make v a robber vertex in the component
23:                 else
24:                     make v a cop vertex in the component
25:                 for vertex v's neighbour w in CopGraph
26:                     add edge vw to the component
27:             add component to the decomposition
        return decomposition
```

Algorithm 1 uses the definition of robber components to yield, given a cop graph, a list of its robber components.

3.6 EXAMPLE OF A SEARCH TREE

In Figure 3.5, we have added a failed path to the search tree. This is a path, that leads to an edge, that is labeled with a connected component, that cannot be decomposed within the given bounds on the width or join width. In this case, the connected component cannot

be decomposed, because the children of this cop node would have at least 3 bag vertices, and that would break the bound 1 on the width.

A search tree is successful, if every blue node is on some path that ends with an orange node. Blue are the nodes, from which the cops take their steps. Orange are the nodes, from which the robber takes his turn. We are the cops and we want to catch the robber; that's why we want to end up in a node, where the robber can no longer escape from. This is cops-and-robber game terminology, which is explained in Section 3.8.

The bulk of the work consists in finding those paths for all the blue nodes. This work consists in selecting the next robber node from several consecutive cop nodes, that is, selecting a robber vertex to place our cop on. In this thesis, the author did not create any elaborate heuristic for that. We just choose some robber vertex.

On first sight, there is no need to put any effort into the selection of the next cop node from a robber node, because every connected component has to be decomposed eventually. But indeed, this task is also very important because we want to fail as fast as possible to save computation time. So if from placing a cop, we get three connected components, two of which are decomposable within the given boundaries, but one is not, ideally we would like to select the non-decomposable one immediately, so that we can backtrack up again as soon as possible to try to place this cop elsewhere. So, as the cops, we want to correctly decompose the given graph; and as the robber, we want to save time, that is spent for the decomposition of this graph.

Note that, a successful search tree will have a guide for each blue node to find its correct path. Just follow the green arrows of each cop node to find these correct paths. The green arrows are captured in the structure **choice**, that is defined in Section 3.9. This way, a tree decomposition satisfying all its bounds can be recovered from a successful search tree. Now, we can focus just on constructing a successful search tree, which could contain any correct tree decomposition.

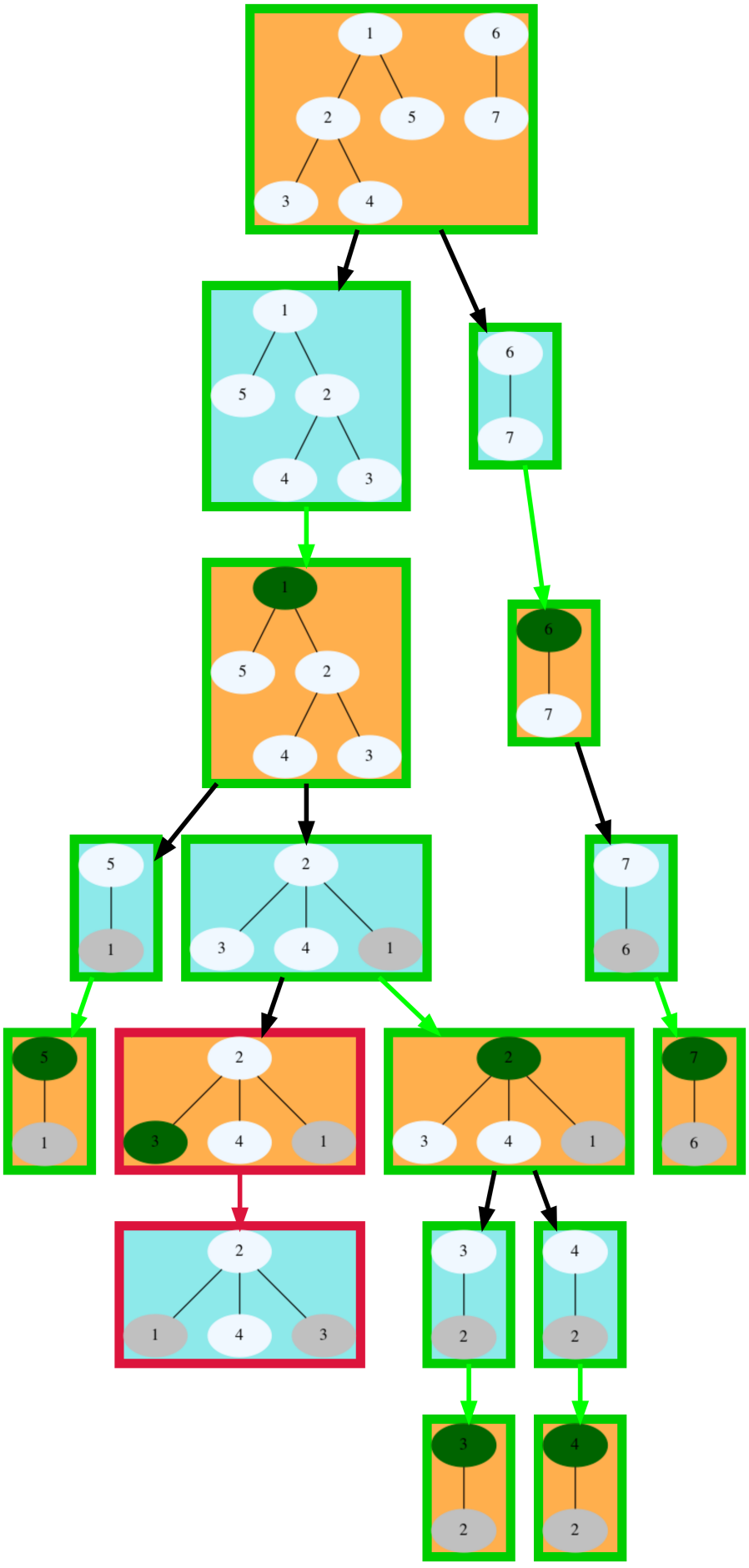

Figure 3.5: Example of a search tree for forest G_0

3.7 RESTRICTING THE GROWTH OF A SEARCH TREE

A cop graph H is *connected*, if $H \setminus \mathbf{cops}(H)$ is connected.

Let H be a connected cop graph. Let $l := |\mathbf{cops}(H)|$, which is the number of cops in this cop graph.

Let $c \in V(H)$. Then define $H[c \leftarrow \mathbf{cop}]$ to be the cop graph with

1. $V(H[c \leftarrow \mathbf{cop}]) = V(H)$,
2. $E(H[c \leftarrow \mathbf{cop}]) = E(H)$,
3. $\mathbf{cops}(H[c \leftarrow \mathbf{cop}]) = \mathbf{cops}(H) \cup \{c\}$.

So, $H[c \leftarrow \mathbf{cop}]$ is the same cop graph as H, if c is a cop vertex in H. Otherwise $H[c \leftarrow \mathbf{cop}]$ has one more cop vertex.

Let $j \leq k \in \mathbb{N}$. (k represents our bound on the width and j our bound on the join width.) We define the subset $D_{k,j}(H)$ of the vertices of H, the set of *choosable robber vertices*.

$$D_{k,j}(H) := \begin{cases} \emptyset & l \geq k \\ V(H) \setminus \mathbf{cops}(H) & l < j \\ \{c \in V(H) \setminus \mathbf{cops}(H) | H[c \leftarrow \mathbf{cop}] \text{ is connected}\} & j \leq l < k \end{cases}$$

The choosable robber vertices are those robber vertices of H, on which a cop can be placed without violating the bound k on the width and the bound j on the join width of the tree decomposition. We will use this concept in 3.9.2 to create the children of a cop node.

3.8 THE COPS-AND-ROBBER GAME

Everything, that we described earlier, is just a game. It's a game between the cops and the robber on our given graph. Let's define, what this means.

Let G be a graph. Let $j \leq k \in \mathbb{N}$. The (k, j)-cops-and-robber game on G is a tuple $\mathcal{C}_k = (P, E, q, \mathbf{copGraph})$, where

1. $((P, E), q)$ is a rooted tree,
2. **copGraph** is a function from P to the set of all cop graphs, whose vertex set is a subset of $V(G)$.

There are two players: The cops and the robber. Every node from P belongs to exactly one of the players, i.e. a certain player takes a turn from a certain node. The root belongs to the robber and then the players take turns alternatingly, that is, every node with an even distance to the root belongs to the robber and the others to the cops. Let P_C denote the nodes, at which the cops take their turn, and P_R the nodes, at which the robber takes his turn. These two sets satisfy $P_C \dot{\cup} P_R = P$.

We will define **copGraph** and E in the following. The root q has $\textbf{copGraph}(q) = (V(G), E(G), \textbf{cops})$ with $\textbf{cops} = \emptyset$. The children of a robber node r are defined through the robber components of $\textbf{copGraph}(r)$ as explained in 3.3 and computed using Algorithm 2. The children of a cop node c are defined through $D_{k,j}(\textbf{copGraph}(c))$, as explained in 3.7 and computed using Algorithm 3.

The leaves of the tree (P, E) are terminal positions. A terminal position, at which the cops take a turn, is called a losing position for the cops, and similarly for the robber.

A function $s_C : P_C \longrightarrow P_R$, such that for all $p \in P_C$, we have $s_C(p) \in pE$, is called a strategy for the cops. A function $s_R : P_R \longrightarrow P_C$ with the analogous property is called a strategy for the robber.

Given a strategy s_C for the cops and a strategy s_R for the robber, we can take the appropriate turns and build a sequence of positions from P, that starts at the root and ends at some terminal position. This sequence is called the game episode for s_C and s_R. The current player at the terminal position of this game episode is called the loser of this game episode. The other player is the winner of this game episode.

A winning strategy for the cops is a strategy s_C^*, such that for all strategies s_R for the robber, the winner of the game episode for s_C^* and s_R are the cops.

A winning strategy for the cops in the (k, j)-cops-and-robber game on G corresponds to one of its tree decompositions of width at most k and join width at most j. For a partial proof, refer to Theorem (4.18) of [1].

In the next section, we present the whole algorithm, on how to construct the search tree for a given graph G and bounds k and j. This algorithm uses all previously presented algorithms.

3.9 BUILDING A SEARCH TREE

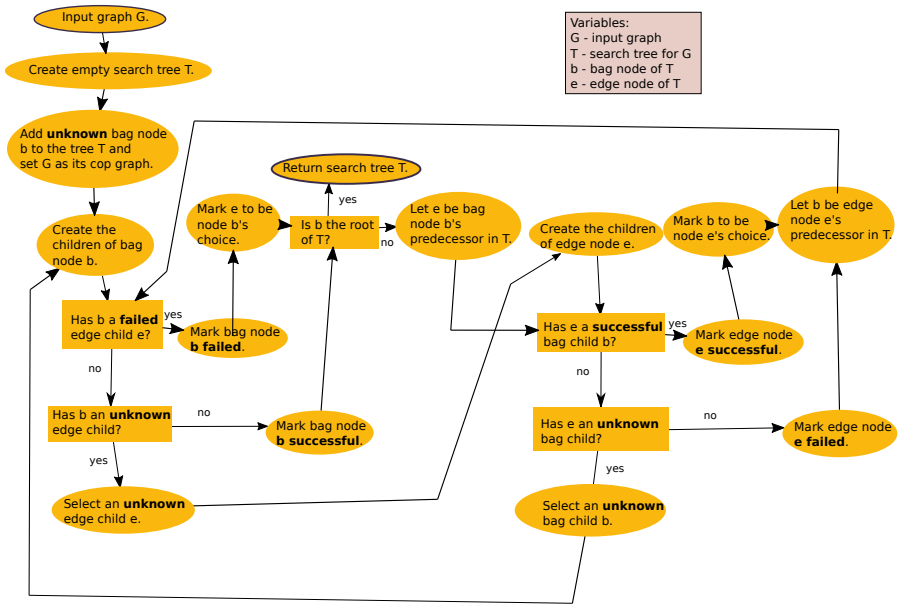

Figure 3.6: Flow diagram of the Habimm decomposer

Flow diagram 3.6 describes the Habimm decomposer. A circle in this flow diagram represents an action, that the Habimm decomposer performs on its state. A rectangle represents a control block, which steers the Habimm decomposer according to its state.

There are 16 actions and 5 control blocks in the Habimm decomposer. The state of the Habimm decomposer consists of the input graph, the being-constructed search tree, a current node of the search tree, which can be a cop or robber node, and the action or control block, that the Habimm decomposer is currently at.

Let G be a graph, for which we seek a tree decomposition. The search tree is a tuple $\mathcal{T} = (P, E, r, \textbf{copGraph}, \textbf{status}, \textbf{choice})$ where

1. $((P, E), r)$ is a rooted tree,
2. **copGraph** is a function from P to the set of all cop graphs, whose vertex set is a subset of $V(G)$,
3. **status** is a function from P to the set $\{\textbf{unknown}, \textbf{failed}, \textbf{successful}\}$,
4. **choice** maps a node from P to one of its E-successors.

Let $q \in P(\mathcal{T})$. Then $\textbf{copGraph}(q)$ is the cop graph at q, $\textbf{status}(q)$ is the status at q, $\textbf{choice}(q)$ is a step in the winning strategy of the player at q.

3.9.1 *Expanding a robber node*

Let $\mathcal{T}$ be a search tree. Let r be a robber node of $\mathcal{T}$. Let $H := \textbf{copGraph}(r)$. Let $C_1, \ldots, C_m$ be the robber components of H. Each robber component yields one child of the robber node. This child corresponds to an edge in the tree decomposition as seen in Figure 3.4. The child p, which corresponds to the robber component C_i, has

- $\textbf{copGraph}(p) = C_i$,
- $\textbf{status}(p) = \textbf{unknown}$,
- $\textbf{choice}(p) = \textbf{unknown}$.

Algorithm 2 Creation of children of a robber node

```
1: function ExpandRobberNode(r)
2:     children ← []
3:     decomposition ← DecomposeCopGraph(copGraph(r))
4:     for component from decomposition
5:         create unknown edge child e
6:         copGraph(e) ← component
7:         add e to children
       return children
```

Algorithm 2 shows, how to create the children of a robber node.

3.9.2 *Expanding a cop node*

Let $\mathcal{T}$ be a search tree. Let p be a cop node of the search tree $\mathcal{T}$. Let $H := \textbf{copGraph}(p)$. We know that H is a connected cop graph. Let $l := |\textbf{cops}(H)|$, which is the number of cops in the cop graph belonging to p.

Recall our definition 3.7 of choosable robber vertices. Each choosable robber vertex defines a child of node p. Let $c \in D_{k,j}(H)$. This defines a robber child r of p with

- $\textbf{copGraph}(r) = H[c \leftarrow \textbf{cop}]$,
- $\textbf{status}(r) = \textbf{unknown}$,
- $\textbf{choice}(r) = \textbf{unknown}$.

Node p has $|D_{k,j}(H)|$ children. Algorithm 3 receives a robber node p and computes its children using this definition.

Algorithm 3 Creation of children of a cop node

```
1:  function ExpandCopNode(p, k, j)
2:      l := |cops(copGraph(p))|
3:      if l ≥ k  return []
4:      children ← []
5:      for robber vertex v of copGraph(p)
6:          make a cop graph copy H of copGraph(p)
7:          mark v a cop vertex in H
8:          decomposition ← DecomposeCopGraph(H)
9:          if l < j or |decomposition| ≤ 1
10:             create unknown robber child b
11:             copGraph(b) ← H
12:             add b to children
          return children
```

Part III

IS MINIMIZING THE SIZE OF JOIN NODES MEANINGFUL?

In this part, we want to evaluate the runtime of the Habimm decomposer on the one hand, and the runtime of Sequoia on the tree decompositions computed by the Habimm decomposer on the other hand. We expect to understand, whether or not minimizing join nodes in a tree decomposition shortens the runtime.

4

EMPIRICAL EVALUATION OF THE HABIMM DECOMPOSER

We have constructed an algorithm to compute width and join width bounded tree decompositions. Now, we want to evaluate its runtime.

Before the author wrote the Habimm decomposer, he already had access to 51 graphs from the PACE 2016 tree width challenge. Then, the author again used the same 51 graphs as his test bed to evaluate the runtime of the Habimm decomposer.

But our Habimm decomposer needs more input than just a graph. The Habimm decomposer takes a triple (G, k, j), where G is the graph to be decomposed, k is the upper bound on the width and j is the upper bound on the join width of the desired tree decomposition. Where do we get those (k, j) from?

First, we download other decomposers from the Internet, that compute a tree decomposition of minimal width of an input graph. This minimal width is called the tree width of the graph. Then we compute the tree widths of all our graphs. In our case, the tree widths for our graphs could be computed in only a few seconds.

Then, for each graph G, we also have its tree width k_G. So the Habimm decomposer gets as input $(G, k_G + 1, k_G - 1)$, that is, we allow the width of the tree decomposition to become non-optimal, while we want the join width to be better than the tree width. The author could have used the actual join width of the optimal tree decomposition, that has been computed by a decomposer; but he decided to keep it simple.

So, to recap, for each of our 51 graphs, every downloaded decomposer will get G as input and output some tree decomposition. Our Habimm decomposer will get $(G, k_G + 1, k_G - 1)$ as input and output some some tree decomposition. Each decomposer will take some time to compute its result and this is the time, that we measure.

For this reason, the Habimm decomposer gets an unfair advantage in the diagram 4.3.

First, we present the downloaded decomposers. The author has collected three decomposers from the PACE website. PACE is an organization, that moderated two Software competitions to compute tree decompositions. The two competitions took place in 2016 and 2017. The submitted decomposers were made publicly available.

Another decomposer was already built into Sequoia. So, the author has taken three decomposers from those competitions, one decomposer from Sequoia and has implemented one himself:

1. The Meiji 2017 decomposer[6]
    - computes a tree decomposition of minimal width,
    - won the second prize at the PACE 2017 challenge,

- is written by Hiromu Ohtsuka and Hisao Tamaki from the Meiji University.

2. The Luebeck decomposer[4]
   - computes a tree decomposition of minimal width,
   - won the third prize at the PACE 2017 challenge,
   - is written in Java by Max Bannach, Sebastian Berndt and Thorsten Ehlers from University of Lübeck and University of Kiel,
   - relies on a SAT solver to find the optimal elimination order.
3. The Meiji 2016 decomposer[5]
   - computes a tree decomposition of minimal width,
   - won the first prize at the PACE 2016 challenge,
   - is written in C by Hisao Tamaki from the Meiji University in Japan,
   - is based on a modified version of the brute force approach of Arnborg et al. in the paper **Complexity of finding embeddings in a k-tree** from 1987.
4. The Sequoia decomposer[8]
   - is a heuristic to quickly compute a tree decomposition of reasonable width,
   - is built into Sequoia.
5. The Habimm decomposer[2]
   - has been implemented in Python for this thesis,
   - computes a tree decomposition bounded by given width and join width,
   - receives as input (G, k, j), where
     - G is the graph to be decomposed,
     - k is the upper bound on the width of the desired tree decomposition,
     - j is the upper bound on the join width of the desired tree decomposition.

Now, let's look at the two important features of the tree decompositions, that these decomposers have computed: the join widths and the widths. We don't consider the Sequoia decomposer, because Sequoia only uses its heuristic tree decomposition internally and does not output it for further analysis.

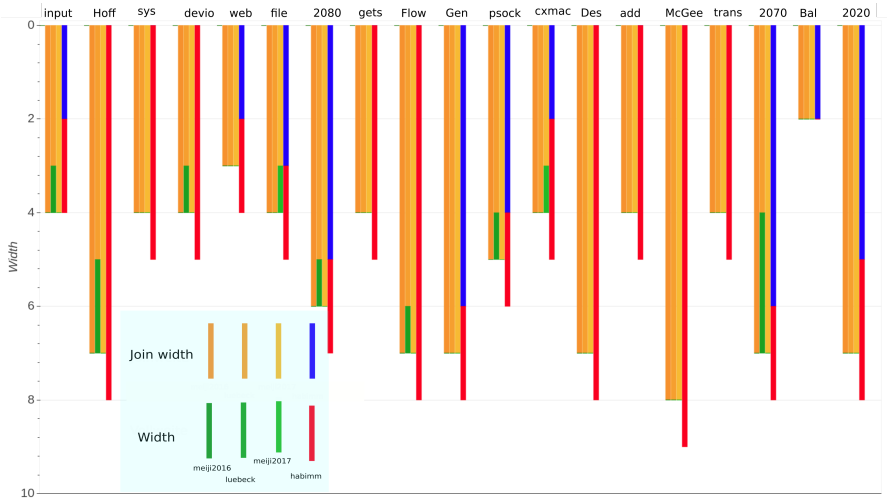

Figure 4.1: Widths and join widths of the four decomposers: **meiji2016**, **luebeck**, **meiji2017**, **habimm**

Figure 4.1 shows the widths and join widths of the tree decompositions, that were computed by the different decomposers for a selected subset of the 51 graphs. One column represents a tree decomposition. Each column is divided into a bottom part and a top part, each of which are colored differently. The top part's height represents the join width of this tree decomposition. The bottom part's height represents the difference between the width and join width of this tree decomposition, which means that the height of the whole column represents the width of this tree decomposition.

Every set of four columns is for one graph. This diagram shows 19 graphs. So there are 19 sets of four columns in this image. Each set of four columns represents the tree decompositions of the four decomposers **meiji2016**, **luebeck**, **meiji2017** and **habimm** for this particular graph. The four columns and their respective top and bottom parts are distinguished by color, as shown in the legend.

Note that the widths of the tree decompositions belonging to the decomposers other than Habimm are equal. This is because the three decomposers **meiji2016**, **luebeck**, **meiji2017** always compute minimum-width tree decompositions.

Furthermore, we can see that on most graphs, the width of the Habimm tree decomposition is greater by 1 than the fixed width of the other tree decompositions. This is because we allow the width to deteriorate. But this is not always the case, as we can see with the graph **input**, where the Habimm decomposer managed to compute a tree decomposition with lower width than all the other decomposers and still maintain the same width.

And we can see that on most graphs, the join widths of the Habimm tree decompositions are significantly smaller than the join widths of the other tree decompositions. This is because the Habimm tree decomposition has been specifically constructed to shrink the join

width below this graph's tree width. This is also not always the case, as we can see with the graph **2070**. There, the green top is higher than the blue bottom, which means that the Luebeck tree decomposition has an even lower join width than the Habimm tree decomposition on this particular graph **2070**. This is because we did not force the Habimm decomposer to have a join width, that is smaller than the join widths of the other tree decompositions.

In the following, we present even more diagrams. Every diagram will use the legend in Figure 4.2.

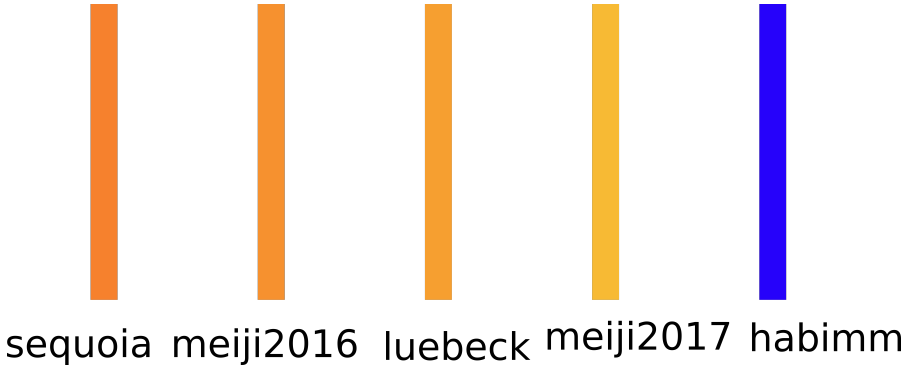

Figure 4.2: Legend for all the following diagrams

The names **sequoia**, **meiji2016**, **luebeck**, **meiji2017**, **habimm** denote the Sequoia decomposer, the Meiji 2016 decomposer, the Luebeck decomposer, the Meiji 2017 decomposer and the Habimm decomposer respectively. The first four decomposers will be represented by similar colors to visually merge them into one counter player for the Habimm decomposer.

All runtime measurements were performed on the author's laptop, which has an Intel(R) Core(TM) m3-6Y30 CPU with two cores at 0.90GHz and 8GiB of system memory.

Note: The author measured every single column several times and then took the average to print on the Y-axis as the runtime in milliseconds. **And** the author verified that every single of those executions resulted in the same result, meaning in the same tree decomposition encoded in the same sequence of UTF-8 characters.

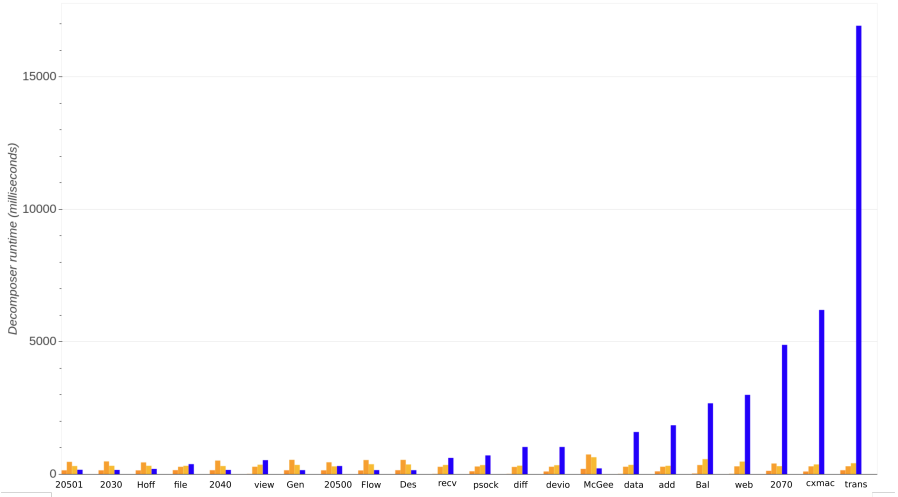

Figure 4.3: Runtimes of the four decomposers: **meiji2016**, **luebeck**, **meiji2017**, **habimm**

Figure 4.3 shows the runtime of four decomposers on a selected subset of our 51 graphs. We can see that the runtime of the Habimm decomposer is similar to the runtime of the other decomposers on most graphs. But on the graphs **trans**, **cxmac** and **2070** the runtime is a lot larger, although the input for the Habimm decomposer is richer than the inputs for the other decomposers.

5 EMPIRICAL EVALUATION OF SEQUOIA

5.1 DEFINITION OF SEQUOIA

`Sequoia`[8] is a program written by Alexander Langer. Its underlying theory is explained in [3]. `Sequoia` can take an MSO formula with a binary `adj` predicate and a graph to assign the free first-order variables of the formula to vertices and the free second-order variables to sets of vertices of the graph, or say that not every free variable can be assigned. `Sequoia` can maximize the cardinality of the assignment of free second-order variables. This way, an NP hard optimization problem like finding a maximum clique in a graph can be expressed by just defining what a clique is:

```
clique(U) := All x All y (x = y or x notin U or y notin U or
    adj(x,y))
```

5.2 SEQUOIA INPUTS

The author has chosen the 70 graphs from the PACE 2016 tree width challenge [7]. The Habimm decomposer ran for longer than one minute on 19 of those graphs, so the author stopped those runs, so their runtime has not been evaluated. We present runtimes on some of the remaining 51 graphs.

Sequoia accepts four or five arguments:

1. The graph.
2. The formula.
3. One of `Bool`, `MinCard`, `MaxCard`, `MinCardSet`, `MaxCardSet`, `Witness` to tell Sequoia, whether to just compute an assignment or to compute an assignment maximizing some function.
4. A flag whether to use the given graph or a 2-subdivision of the given graph.
5. Optionally, a tree decomposition of the given graph.

5.2.1 *Sequoia input: The graph*

Table 5.1: Graphs

Graph	# Vertices	# Edges	Tree width	Graph	# Vertices	# Edges	Tree width
Bal	64	62	1	**Flow**	34	34	6
next	69	78	1	**newf**	74	73	2
menu	27	27	2	**clock**	20	29	2
cxmac	27	29	3	**trans**	85	88	3
dhcpc	29	29	2	**bfind**	119	129	3
req	25	25	2	**devio**	364	363	3
cfs	90	97	3	**diff**	27	28	1
input	31	31	3	**sys**	46	48	3
pol	108	109	2	**file**	27	26	3
stats	31	32	2	**gets**	61	68	3
start	71	67	2	**get**	24	23	2
psock	20	21	4	**rand**	28	30	2
put	31	35	2	**stat**	20	24	1
nack	32	35	1	**chown**	20	44	2
send	20	30	1	**Gen**	24	36	6
runi	52	51	1	**2010**	20	30	3
view	48	44	2	**2020**	20	30	6
mesh	20	46	2	**2030**	47	47	8
recv	20	73	2	**2040**	30	29	10
echo	46	47	2	**Hoff**	25	27	6
reg	16	32	2	**McGee**	26	27	7
add	20	73	3	**20500**	26	25	9
data	20	21	2	**20501**	20	78	8
init	52	57	2	**2070**	61	62	6
web	25	25	2	**2080**	45	48	5
Des	20	56	6				

5.2.2 *Sequoia input: The formula*

In our setting, 15 MSO formulae were used:

Table 5.2: MSO formulae

Formula	Nickname	Is the graph 2-subdivided?	Number of free variables	Quantifier rank	Length
3col-free2	**cf2**	no	2	2	151
3col-free	**cf**	no	0	5	233
3col2	**c2**	no	0	4	159
3col	**c**	no	3	2	218
bipartite	**bi**	no	0	3	75
clique	**cli**	no	1	2	54
connected-domset	**d1c**	no	1	3	183
connected	**con**	no	0	3	84
d2-dominating-set	**d2**	no	1	3	70
d3-dominating-set	**d3**	no	1	4	115
dominating-set	**d1**	no	1	2	35
hamiltonian-cycle	**ham**	yes	1	4	361
independent-set	**ind**	no	1	2	47
longest-cycle	**lon**	yes	1	4	314
vertex-cover	**ver**	no	1	2	41

A formula's **Length** is the number of actual, non-whitespace UTF-8 characters, that are passed to Sequoia. This feature is supposed to measure the complexity of a formula.

5.2.3 *Sequoia input: Evaluation of a formula*

For every formula, **Bool** was passed to Sequoia, so Sequoia tried to find *some* assignment for the free variables of the formula, without maximizing any function. This means that Sequoia didn't actually compute any NP hard parameter at all. But on several chosen instances, the author measured the time, that Sequoia needed, when passed **Bool** and the time, that Sequoia needed, when passed **MaxCard**, so that in the latter case, Sequoia actually output some NP hard parameter of the graph. But the two times were almost the same. So the author passed **Bool** for every formula to keep it simple.

5.2.4 *Sequoia input: 2-subdivision of the graph*

Some formulae require that the input graph be 2-subdivided. For example, it is impossible to capture the definition of a cycle in an MSO formula, which only uses the **adj** binary predicate over the vertices. Since we still want to use an MSO formula to capture a cycle, we must slightly alter the graph, which the MSO formula is applied upon: We 2-subdivide the graph to obtain a slightly bigger graph. Then we mark those *edge nodes*, that have additionally been added to the graph, with a new **E** unary predicate to distinguish them from the original nodes, that have already been present in the original graph.

Now, if a formula has **no** in the corresponding column, the MSO signature is $\{adj\}$. And if a formula has **yes** in the corresponding column, the MSO signature is $\{adj, E\}$.

5.2.5 *Sequoia input: The tree decomposition*

To get a tree decomposition for a graph, we can do it in at least five different ways. We can use the Meiji 2016, the Luebeck or the Meiji 2017 decomposer to compute a minimal-width tree decomposition. Or we can input nothing and Sequoia will heuristically compute some tree decomposition. Or we can use the Meiji 2016 decomposer to compute a tree decomposition, look up its width to get the tree width of the graph, and launch the Habimm decomposer on this graph and the tree width value to compute a low-join-width tree decomposition.

5.3 ONE GRAPH, MANY FORMULAE

In the following, we will present some runtime diagraphs. Each diagram is for one fixed input graph. In the diagrams, there are columns. Each column represents an MSO formula together with some tree decomposition. The column colors distinguish between the five different tree decompositions; whereas the column positions distinguish between the fifteen different MSO formulae.

Thus, a column represents a triple consisting of a graph, an MSO formula and tree decomposition, which is the input to Sequoia. A column's height in this diagram represents the runtime of Sequoia in milliseconds on the corresponding input.

Note: The author measured every single column several times and then took the average to print on the Y-axis as the runtime in milliseconds. **But** the author did not verify that every single of those executions resulted in the same result.

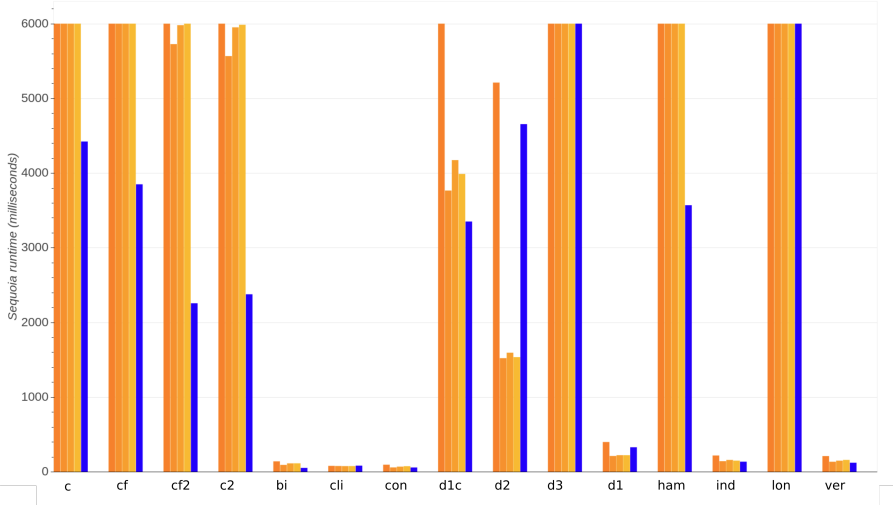

Figure 5.1: Runtime on selected MSO formulae and the graph *Hoff*

In Figure 5.1, we see the Sequoia runtimes on the Hoffman graph. Here, we see how well minimizing the size of join nodes in a tree decomposition can work. In fact, here, the tree decomposition computed

by the Habimm decomposer was a path. In the first four columns, which correspond to the four formulae for 3-colorability, the Habimm tree decomposition yields significantly lower runtimes than other tree decompositions.

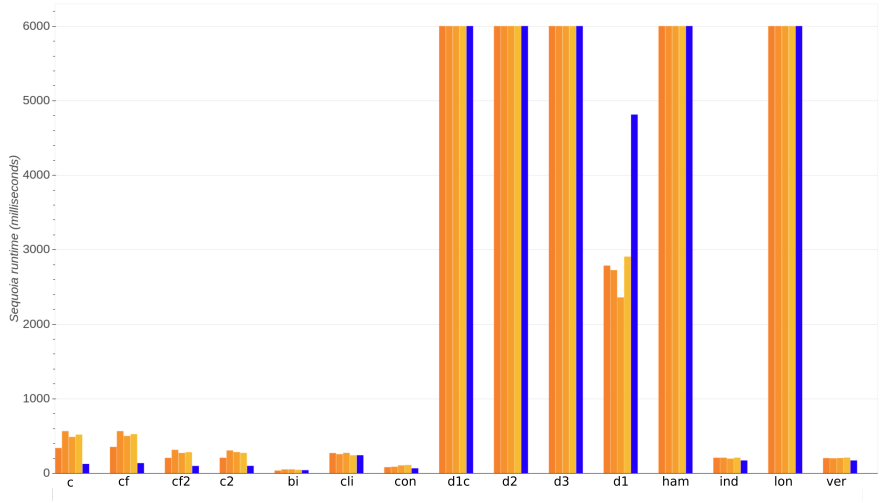

Figure 5.2: Runtime on selected MSO formulae and the graph *2040*

In Figure 5.2, there is a randomly generated graph. The exact strategy to generate this graph from a sequence of random numbers is not known to the author. But on the four formulae for 3-colorability, the runtimes of Sequoia on the Habimm tree decompositions are lower than the runtimes of Sequoia on any other tree decomposition respectively. But on formula **d1**, the runtime of Sequoia on the Habimm tree decomposition is higher than the runtimes of Sequoia on any other tree decomposition.

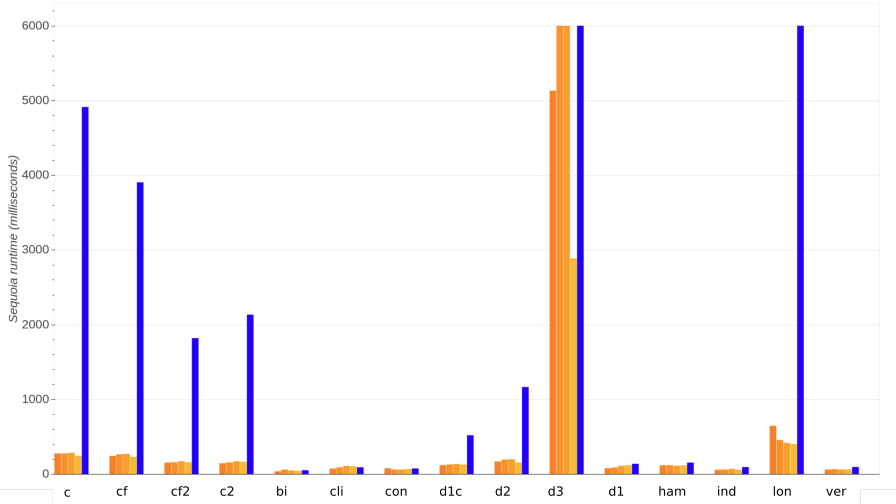

Figure 5.3: Runtime on selected MSO formulae and the graph *cxmac*

But in Figure 5.3, we see that the Habimm decomposer has created a very bad tree decomposition, that does not work well with the four formulae for 3-colorability on the *cxmac* graph. The runtime on the Habimm tree decomposition is many times higher than the highest runtime of the other tree decompositions.

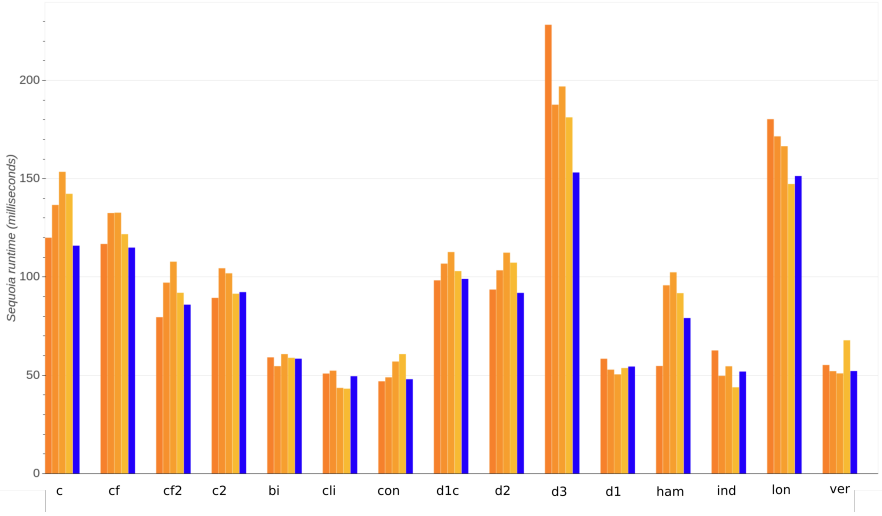

Figure 5.4: Runtime on selected MSO formulae and the graph *chown*

In Figure 5.4, we see the *chown* graph. On this graph, Sequoia actually runs fastest on the Habimm decomposition on the formula **d3**, but on the formula **ham**, Sequoia unexpectedly runs fastest on the heuristic tree decomposition, that has been computed by the Sequoia-internal decomposer.

5.4 ONE FORMULA, MANY GRAPHS

In the following, similar diagrams are presented. But instead of being for one fixed graph, each diagram is for one fixed MSO formula.

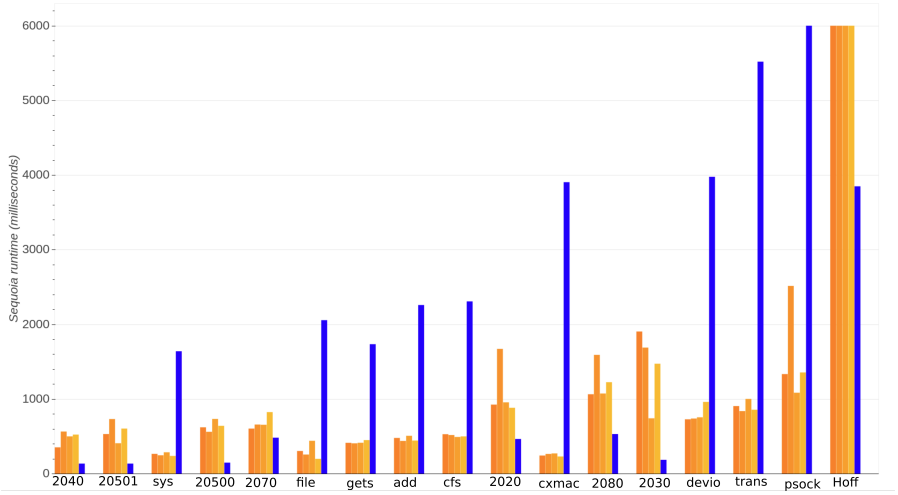

Figure 5.5: Runtime on selected graphs and the MSO formula *3col-free*

In Figure 5.5, we see the runtimes of Sequoia on the *3col-free* formula. On the formulae **2020**, **2080**, **2030**, **Hoff**, Sequoia runs faster on the Habimm tree decomposition than on any other tree decomposition.

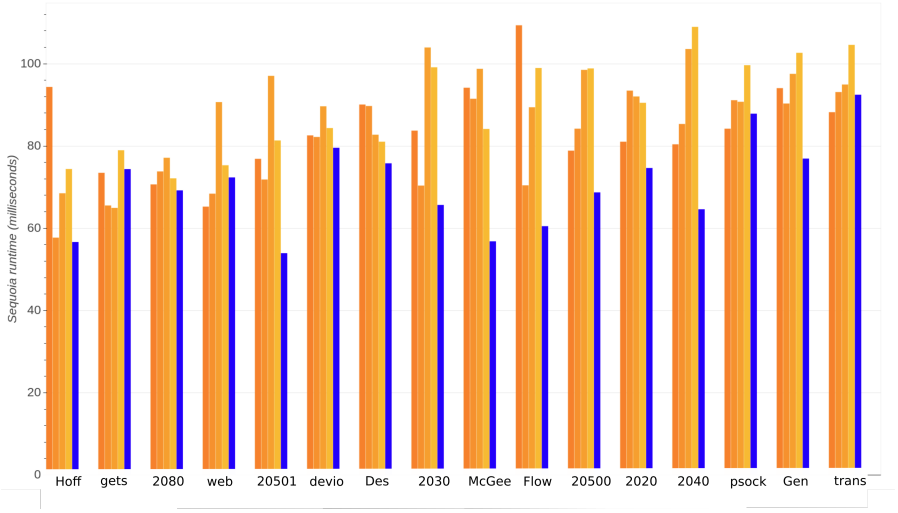

Figure 5.6: Runtime on selected graphs and the MSO formula *connected*

In Figure 5.6, there is the diagram for the formula *connected*. Here, the differences between all the runtimes are only in the order of tens of milliseconds. Hence, we can say that the runtimes on the Habimm tree decompositions are at least not much worse than the runtimes on the other tree decompositions. But, if we say that ten milliseconds matter, Sequoia really runs faster on the Habimm tree decompositions than on any other tree decomposition on the formulae **McGee**, **20500**, **Gen** and **20501**.

In Figure 5.7, we see a diagram, that is not so good. Here, Sequoia runs significantly slower on the Habimm tree decompositions than on any other tree decomposition on all the shown graphs.

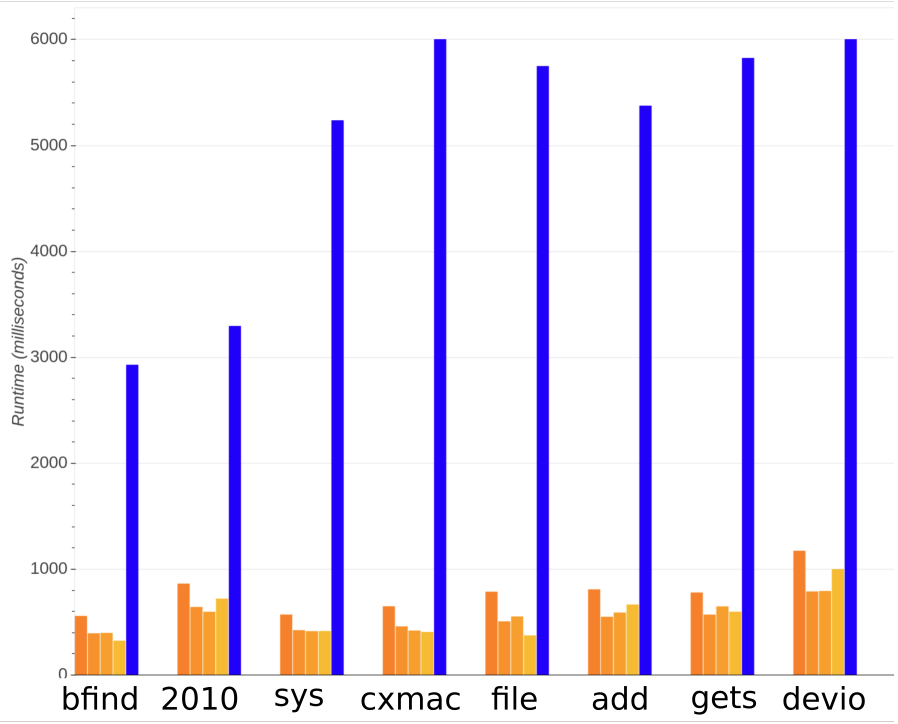

Figure 5.7: Runtime on selected graphs and the MSO formula *longest-cycle*

Part IV

WHAT DO WE LEARN FROM THIS?

To round it off, we discuss the results of this Master's thesis.

6 CONCLUSION

We conclude that

1. there are graph and formula combinations, on which Sequoia performs better, if the supplied tree decomposition has a smaller join width and a greater width, and
2. there are graph and formula combinations, on which Sequoia performs worse, if the supplied tree decomposition has a smaller join width and a greater width.

The natural question is: Why is that so? The author has the following observations:

- The Hoffman graph, on which several formulae yielded better Sequoia runtimes, has many automorphisms, which means that this graph is highly symmetric. Maybe, it is better if a symmetric graph is decomposed into a path, rather than an asymmetric one.
- The four formulae for 3-colorability had several graphs, on which Sequoia yielded better runtimes. The author does not know, how these formulae differ from the other formulae.

To evaluate the utility of low join width even better, one could compute, for a given graph, all the pairs (k, j) such that there is a tree decomposition of width k and join width j, but no tree decomposition of width k and join width $j-1$ or of width $k-1$ and join width j. One could call these pairs the *pareto-front* of this graph with respect to width and join width. And then one could try to insert the corresponding tree decompositions into Sequoia and compare the runtimes. Is it useful for any graph, if the join width is lowered? Or are there graphs, for which Sequoia works better if the join width is higher?

One could also try other graph parameter programs other than Sequoia. This irons out the possibility that Sequoia is implemented to work with specific tree decompositions.

It's me, Mario!

BIBLIOGRAPHY

[1] Martin Grohe. *Graph Decompositions and Algorithmic Applications*. Lecture notes for a course held at the RWTH Aachen. WS 2016/17.

[2] *Habimm decomposer*. https://github.com/Jachtabahn/treedec. Accessed: 2020-04-24.

[3] Alexander Langer, Felix Reidl, Peter Rossmanith, and Somnath Sikdar. "Practical algorithms for MSO model-checking on tree-decomposable graphs." In: *Computer Science Review* 13 (2014), pp. 39–74.

[4] *Luebeck decomposer*. https://github.com/maxbannach/Jdrasil. Accessed: 2020-04-24.

[5] *Meiji 2016 decomposer*. https://github.com/TCS-Meiji/treewidth-exact. Accessed: 2020-04-24.

[6] *Meiji 2017 decomposer*. https://github.com/TCS-Meiji/PACE2017-TrackA. Accessed: 2020-04-24.

[7] *PACE 2016: Tree width instances*. http://bit.ly/pace16-tw-instances-20160307. Accessed: 2020-02-27.

[8] *Sequoia*. https://github.com/sequoia-mso/sequoia-core. Accessed: 2020-04-24.