Vrije Universiteit Amsterdam

Master Thesis

Solving the Maximum Weight Connected Subgraph Problem Using Color Coding

Author:
Eline van Mantgem

Supervisors:
Prof. Dr. G.W.Klau
Prof. Dr. W.J.Fokkink

A thesis submitted in fulfilment of the requirements
for the degree of Master of Computer Science

in the

Faculty of Sciences
Department of Computer Science

May 2019

Declaration of Authorship

I, Eline van Mantgem, declare that this thesis titled, 'Solving the Maximum Weight Connected Subgraph Problem Using Color Coding' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a Master's degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

VRIJE UNIVERSITEIT AMSTERDAM

Abstract

Faculty of Sciences
Department of Computer Science

Master of Computer Science

Solving the Maximum Weight Connected Subgraph Problem Using Color Coding

by Eline van Mantgem

Analysis of large networks has become a major research topic in systems biology. The Maximum Weight Connected Subgraph (MWCS) problem can be applied to identify functional modules in protein-protein interaction networks. Given a vertex weighted graph, MWCS identifies a connected subgraph that maximizes the sum of the vertex weights. Different methods have been introduced to solve the MWCS problem, such as integer linear programming and color coding. We propose a method that applies a preprocessing scheme to reduce the input graph, decomposes the graph into its biconnected components, and finds an optimal solution using an algorithm based on color coding. We improve both the preprocessing scheme as well as the color coding algorithm as compared to previously published methods. We test our algorithm on the ACTMOD benchmark instances of the 11th DIMACS implementation challenge and compare the results and performance to an already existing integer linear programming method for solving the MWCS problem. We show that even though our algorithm cannot compete with the performance of the integer linear programming method, it is still feasible as a heuristic for finding the location of an optimal solution.

Acknowledgements

I would like to express my appreciation to my thesis advisor Gunnar Klau for his support and assistance during the development of this research. I would also like to acknowlegde Wan Fokkink as the second reader, I am grateful for his very valuable comments on this thesis.

I would also like to thank Femke van Raamsdonk for all her help and support throughout my years of study.

Finally, a very special gratitude goes out to Natalia Silvis for her emotional support and for believing in me.

Eline

Contents

List of Figures

List of Tables

To my Kleinie

Chapter 1

Introduction

In recent years, construction and analysis of large networks have become major research topics in different fields such as systems biology, network design, social networks and facility location planning. The Conservation Planning problem, also known in literature as site selection or reserve network design, is one example of this kind of problem that arises in the field of biology. The problem consists of selecting a set of land parcels for conservation to ensure the ability for species to survive. Given a set of land parcels, a set of biologically important land parcels, the cost of each parcel, and the habitat suitability, the goal is to select a set of land parcels that connects all biologically important parcels while keeping the cost within a limited budget [3]. The Conservation Planning problem, like many decision and optimization problems, can be transformed to finding a connected subgraph of a larger input graph satisfying constraints on cost and/or benefit. These costs and benefits are associated with the vertices, edges or both, depending on which specific problem out of this family of problems is used. Examples of this family of problems are the Prize Collecting Steiner Tree Problem, the Minimum Steiner Tree problem, the Point-to-Point Connection problem and the Maximum Weight Connected Subgraph problem. In this thesis, we are concerned with the Maximum Weight Connected Subgraph problem, where given an input graph with positive and negative weights on the vertices, the goal is to find a connected subgraph that maximizes the sum of the vertex weights.

In systems biology, research has shifted more and more to the integrated analysis of large datasets. Due to the exponential growth of gene expressions and protein-protein interaction (PPI) data, the identification of functional modules in protein-protein interaction (PPI) networks has become of more interest in recent years. The Maximum Weight Connected Subgraph problem can be applied to identify functional modules in such networks. The identification of these functional models can be used to develop

biomarkers for cancer diagnosis and treatement, and to identify genes that can be used as predictors of drug response in cancer. Furthermore, it can be used in the analysis of survival data of cancer to identify mutations that are clinically relevant. A wide range of different methods have been studied to solve this problem such as integer linear programming [4], heuristics [5] and greedy search strategies [6]. Here, we will solve the Maximum Weight Connected Subgraph problem using the color coding technique [7], a fixed parameter tractable method for finding patterns in graphs.

In Chapter 1 we discuss related work, our contribution and the goal of this thesis. In Chapter 2 we discuss a collection of topics that are important to understand for the remainder of the thesis. These topics include parameterized complexity, the Maximum Weight Connected Subgraph problem and graph connectivity: biconnected components. We present our method in Chapter 3 and Chapter 4 where we discuss both the theory (Chapter 3) as well as some implementation details (Chapter 4). In Chapter 5 present our test results. Lastly, in Chapter 6 we discuss our conclusions.

1.1 Related Work

Color coding has been applied to the Maximum Weight Connected Subgraph (MWCS) problem in previous work to find high scoring subgraphs for bioinformatics applications. Dao et al. [8] describe a network-based classification algorithm to identify discriminative subnetwork markers in protein-protein interaction (PPI) networks. They apply their algorithm to drug response studies. They show that their algorithm provides a better and more stable performance and produces predictive markers that are more reproducible across independent cohorts of breast cancer patients treated with chemotherapy compared to other subnetwork methods. The authors introduce the Optimal Discriminating k-Subnetwork problem (ODkS) which asks to compute the connected subnetwork from a graph G such that it distinguishes samples from different classes optimally. The Optimal Discriminating k-Subnetwork problem (ODkS) can be formalized as follows:

Definition 1.1 (ODkS). Given a PPI network $G = (V, E)$, a weight function w on subnetwork S, and a restriction $|S| \leq k$, find the connected subnetwork S_{opt}, with $|S_{opt}| \leq k$, such that S_{opt} maximizes the score $w(S_{opt})$ for all $w(S)$ in G. S_{opt} is called the optimally discriminative subnetwork.

The weight function w on subnetwork S is defined as the difference between the total distance between samples from different classes and the total distance between samples from the same class. The authors show that weight function w can be decomposed such

that it can be written as a sum of gene scores ($w(g_i)$). Thus the discriminative score of a connected subnetwork S can be rewritten as follows:

$$w(S) = \sum_{\forall i: g_i \in S} w(g_i)$$

This is equivalent to finding the connected subnetwork for which the total weight of the vertices is maximum, i.e., MWCS.

In [9] Hansen and Vandin describe NoMAS (Network of Mutations Associated with Survival), an algorithm based on color coding for finding the highest scoring subnetwork in a graph in order to identify subnetworks of a large gene-gene interaction network that have mutations associated with survival in cancer. Given an undirected graph $G = (V, E)$, an $n \times m$ mutation matrix M, and the survival information for the patients in M, NoMAS first identifies a high scoring subnetwork by solving MWCS using a color coding based algorithm. Then, NoMAS uses a permutation test to assess the significance of the subnetwork. The authors propose a log-rank statistic as a measure of the association between mutations in a group of genes and survival. As opposed to the approach in [8], the log-rank statistic score of the subnetwork is not set additive and they prove that there is a family of instances for which the algorithm cannot identify the optimal solution. The differences between the color coding algorithms described in both [8] and [9] compared to our algorithm are discussed in more detail in Chapter 3.

In previous work, MWCS has also been solved using other approaches. In [4], Dittrich et al. proposed a method to identify functional modules in PPI networks which delivers provably optimal and suboptimal solutions to the MWCS problem by integer-linear programming (ILP) in reasonable running time. In [2], El Kebir and Klau extended this method by adding a preprocessing scheme and by decomposing the input instance into its biconnected and triconnected components.

1.2 Our Contribution

The method described in [2] makes use of an ILP to solve the MWCS instance. Closed source commercial software and the commercial license that comes along with it are needed to execute their approach. We describe another approach to solve the MWCS problem which does not make use of an ILP, the software is completely open source and available without a commercial license. To solve the MWCS problem, we apply the color coding technique. To this end, we integrated our color coding implementation into Heinz[1], a software tool written by El-Kebir, which implements the ILP as described in [2].

[1] https://github.com/ls-cwi/heinz

Applying color coding to solve MWVS has been done before. We slightly improved the color coding algorithm as described by Hansen et al. [9] and Dao et al. [8]. Furthermore, we also implemented a rooted version of the color coding algorithm. We decomposed the input graph into its biconnected components before running a block-cut color coding algorithm using both the rooted as well as the unrooted version on the data. We ran experiments using both the unrooted color coding algorithm as well as the block-cut variant and compared the results. Additionally, we made a minor improvement to the preprocessing scheme as proposed by El-Kebir et al. [2] and combined this scheme with our color coding implementation. We ran experiments using the preprocessing scheme and all versions of the algorithm and analyzed the results. To test the correctness of our solution, we implemented a minor adjustment to the ILP as implemented in Heinz such that we were able to find a correct solution for any given size using the ILP. We compared these results to the results found by our algorithm.

1.3 Goals

The main goal of this thesis is to investigate whether a color coding based algorithm can compete with the aforementioned ILP approach. Additionally, we want to improve on previously developed algorithms for solving MWCS using color coding. First, we need to get an understanding of the color coding technique in order to develop and implement our own algorithm for applying color coding to solve an MWCS instance. Second, we want to study the preprocessing scheme. We want to improve the scheme and study the impact of applying the preprocessing scheme on the input instance before running color coding by comparing it with results found while running the algorithm on the original input instance. Third, we want to devise a two-layer scheme for solving MWCS based on graph decomposition and the three-layer scheme as described by El-Kebir [2].

The application for which our algorithm can be used in the field of bioinformatics, i.e., the identification of optimal PPI subgraphs, as well as the statistics behind the calculation of the vertex weights are out of the scope of this thesis. A more detailed discussion of both can be found in [4].

Chapter 2

Preliminaries

In order to fully understand the methods and algorithms described in Chapter 3 some theory knowledge is necessary. In this chapter we will discuss parameterized complexity and parameterized algorithms (Section 2.1), the Maximum Weight Connected Subgraph problem (Section 2.2) and graph connectivity, specifically biconnected components (Section 2.3).

2.1 Parameterized Complexity

NP-hard problems are a set of problems to each of which any other problem in NP can be reduced to polynomial-time. Multiple strategies have been devised for solving NP-hard problems. One approach is to sacrifice solution quality for efficiency, for example, by employing heuristic algorithms, or approximation algorithms. Another possibility is to insist on exact solutions and to accept inefficiency for some inputs. In addition, a commonly used strategy is to reduce the problem to 'general-purpose problems' such as integer linear programming or satisfiability solving. Another alternative can be to design a fixed-parameter tractable algorithm.

In a classical setting, the complexity of computational problems is only measured as a function in the input size. Parameterized complexity focuses on the complexity of computational problems with respect to multiple parameters of the input or output. By measuring complexity as a function in these parameters, classification of NP-hard problems can be done on a finer scale. Some problems can be solved by algorithms that are polynomial in the input size and exponential (or at least super polynomial) in the size of a fixed parameter only. These algorithms are called fixed-parameter algorithms. Problems which can be solved efficiently when the size of a fixed parameter is small are

called fixed-parameter tractable (FPT). In [1] Hüffner et al. give the following definition of FPT:

Definition 2.1 (**FPT**)**.** A parameterized problem instance consists of a problem instance I of size $|I|$, and a parameter k. A parameterized problem is fixed-parameter tractable if it can be solved in $f(k) \cdot |I|^{\mathcal{O}(1)}$ time for all instances I, where f is a (computable) function solely depending on the parameter k.

Typically the function $f(k)$ is thought of as single exponential like 2^k. However, the definition allows functions to grow even faster. The crucial part of the definition is to exclude functions of the form $f(n, k)$, such as n^k, where $n = |I|$.

2.1.1 Kernelization

Kernelization is the reduction of data in polynomial-time with guaranteed effectiveness. The idea behind kernelization is to presolve those parts of the problem instance that are easy to cope with, leaving only the parts that form the hard core, the kernel, of the problem. The computationally expensive algorithms then only need to be applied to this kernel. Guaranteed effectiveness means that it is often possible to prove that a kernel with guaranteed bounds on the size of a kernel as a function of a fixed parameter k can be found in polynomial-time. More formally, kernelization is defined as follows:

Definition 2.2 (**Kernelization**)**.** Let I be an instance of a parameterized problem with given parameter k. A reduction to a problem kernel (kernelization) is a polynomial-time algorithm that replaces I by a new instance I' and k by a new parameter k' such that

- the size of I' and the value of k' are guaranteed to only depend on some function of k, and
- the new instance I' has a solution with respect to the new parameter k' if and only if I has a solution with respect to the original parameter k.

Example: Vertex Cover. For an example of kernelization consider the Vertex Cover problem. Finding the minimal vertex cover is a classical NP-complete problem since it was one of Karp's 21 NP-complete problems [10]. Furthermore, many discoveries that influenced the field of fixed-parameter research originated from this problem. A vertex cover of a graph is a set of vertices of a graph so that every edge is incident with at least one vertex of the set, more formally:

Definition 2.3 (**Vertex Cover**)**.** A vertex cover V' of an undirected graph $G = (V, E)$ is a subset of V such that $uv \in E \rightarrow u \in V' \vee v \in V'$.

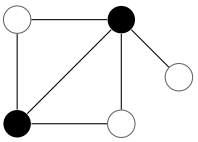

FIGURE 2.1: A graph with a vertex cover (black vertices) of size 2

An example of a vertex cover for a graph is shown in Fig. 2.1.

To cover an edge in the graph one of the two endpoints must be in the vertex cover. In case one of the two endpoints has a degree of 1, the other endpoint potentially covers more edges than this degree-1 vertex. This realization leads to a first data reduction rule:

Reduction Rule 1. If there is a degree-1 vertex, then put its neighboring vertex into the cover.

After applying Reduction Rule 1 we can add the following rule in the fixed-parameter setting where we need to find a vertex cover of size at most k:

Reduction Rule 2. If there is a vertex v of degree at least $k+1$, then put v into the cover.

If there is a vertex v in G with a degree $\delta \geq k+1$ which is not put in the cover, then the $k+1$ neighbors of v have to be in the cover in order to cover all edges incident with v. Clearly, the resulting cover is not a solution since the maximum allowed size of the cover is k.

After applying both reduction rules exhaustively, all vertices in the remaining graph have at most a degree of k (Reduction Rule 2). Thus, by adding another vertex to the cover, at most k edges can be covered. Since a vertex cover can not have more than k vertices, the graph only has a solution of size k if the remaining graph has at most k^2 edges. Because all vertices have a minimum degree of 2 (Reduction Rule 1) and there can be at most k^2 edges, the remaining graph can contain at most k^2 vertices if it has a solution set of maximum k vertices. Thus, after applying two polynomial-time reduction rules what is left is a reduced instance whose size can be expressed solely in terms of the parameter k. According to Definition 2.2 this resulting instance is a kernel of the original instance.

2.1.2 Fixed-parameter Dynamic Programming

Dynamic programming is a method used for solving complex problems. The main idea behind it is to recursively break down the problem into possibly overlapping subproblems. Each of these subproblems is solved only once and, to avoid recalculation, the

solutions are stored in a table. This technique is called memoization. Using memoization saves computation time at the cost of a hopefully modest expenditure in storage space. Since the running time of fixed-parameter dynamic programming algorithms mainly depends on the table size, the main trick to obtaining useful fixed-parameter dynamic programming algorithms is to bound the size of the table by a function of the parameter times a polynomial in the input size. In many cases, the table size is obviously bounded in the parameter. If this is not the case, two methods that could be used for this are tree decomposition and color coding. Of interest for this thesis is color coding described in the following section.

2.1.3 Color Coding

Color coding is a technique that can be used to find small patterns in graphs, e.g. to detect simple paths or cycles of length k. A naive approach to finding a simple path or cycle of length k with certain properties in a graph is to combinatorially try all $\binom{n}{k}$ possibilities of selecting k out of n vertices. However, this approach leads to a combinatorial explosion, so a more sophisticated method is needed. The color coding technique has two phases:

1. Randomly color all vertices of the graph in one of k colors.
2. Detect whether there exists a "colorful" path or cycle of length k.

A "colorful" path or cycle is one where all vertices have a distinct color. Since the coloring is random, many coloring trials have to be performed to make sure the correct path is found with a high probability. The number of trials needed to do so also depends on k as will be shown later in this section. For now, suffice it to say that even very low error probability rates, e.g. 0.1%, do not incur excessive running time costs.

Given a fixed coloring of vertices, the problem of finding a simple path or cycle can be solved using dynamic programming. In the naive algorithm, all vertices visited need to be stored, using $\mathcal{O}(n^k)$ time and space. Using color coding, only the possible sets of vertices of distinct colors need to be stored, using $\mathcal{O}(2^k) \cdot n^{\mathcal{O}(1)}$ time and space. Choosing the number of colors poses an important trade-off: increasing k leads to a decreased number of trials since chances of the target path or cycle becoming colorful increase. However, this also leads to an increased running time and increased memory requirements during dynamic programming.

Example: Minimum Weight Path. Consider the minimum weight path problem for an example of color coding.

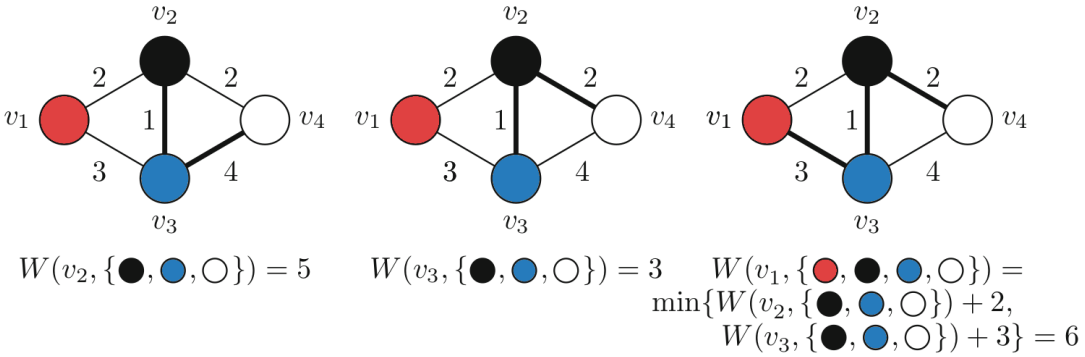

FIGURE 2.2: An example from [1] for solving Minimum Weight Path using color coding. The new table entry (*right*) is calculated using two already know entries (*left* and *middle*)

Definition 2.4 (**Minimum Weight Path**)**.** A minimum weight path of an undirected graph G with edge weights $w : E \to \mathbb{Q}^+$ and a non-negative integer k is a simple length-k path in G that minimizes the sum over its edge weights.

The minimum weight path problem is NP-hard due to the requirement of finding a *simple* path [11]. It can be solved using the color coding technique and dynamic programming. Assume we have a fixed coloring of vertices using k colors. Let S be a subset of the k colors and let $\mathrm{c}(v)$ denote the color of v. Furthermore, let $W(v, S)$ denote the minimum weight of a path that uses each color in S exactly once and ends in v. Using dynamic programming we can compute a value $W(v, S)$ for every vertex $v \in V$ and every cardinality-i subset S of colors. $W(v, S)$ is computed as follows:

$$W(v, S) = \min_{e=\{u,v\} \in E} (W(u, S \setminus \{c(v)\}) + w(e))$$

The resulting path is a simple path since no color is used more than once. See Fig. 2.2 for an example. In case we have a successful coloring (i.e. a coloring that colors a simple length-k path P of minimum weight in different colors) the algorithm will find this path in $\mathcal{O}(2^k|E|)$ time. Furthermore, there are k^k ways to arbitrarily color k vertices with k colors and $k!$ ways to color them. Thus the probability of any length-k path being colorful in a single trial is $\frac{k!}{k^k} \geq e^{-k}$. In Chapter 3 we will discuss how to apply color coding to solve the Maximum Weight Connected Subgraph problem.

2.2 Maximum Weight Connected Subgraph Problem

The MWCS-problem can be described as follows: given an undirected, vertex weighted graph, find a subset of vertices that induces a connected subgraph having a maximal sum

of vertex weights. More formal definitions for both the unrooted and rooted versions of MWCS are given by:

Definition 2.5 (**MWCS**)**.** Given a connected undirected vertex-weighted graph $G = (V, E, w)$ with weights $w : V \to \mathbb{R}$, find a connected subgraph $T = (V_t, E_t)$ of G, with $V_t \subseteq V$, $E_t \subseteq E$, such that the score $w(T) := \sum_{v \in V_t} w(v)$ is maximal.

Definition 2.6 (**R-MWCS**)**.** Given a connected undirected vertex-weighted graph $G = (V, E, w)$ with weights $w : V \to \mathbb{R}$ and a node set $R \subseteq V$, find a connected subgraph $T = (V_t, E_t)$ of G, with $V_t \subseteq V$, $E_t \subseteq E$, such that $R \subseteq V_t$ and the score $w(T) := \sum_{v \in V_t} w(v)$ is maximal.

In case all vertex weights are positive an optimal solution can simply be found by determining any spanning tree. When positive and negative weights are involved, however, the problem is not so easy to solve. In fact, in case of positive and negative vertex weights, the MWCS problem becomes an NP-hard problem [5].

The MWCS problem is closely related to the well-known Prize-Collecting Steiner Tree problem (PCST) which is defined as follows:

Definition 2.7 (**PCST**)**.** Given an undirected graph $G = (V, E)$ with vertex profits $p : V \Rightarrow \mathbb{R}_{\geq 0}$ and edge costs $c : E \Rightarrow \mathbb{R}_{\geq 0}$, find a connected subgraph $T = (V^*, E^*)$ of G such that $p(T) := \sum_{v \in V^*} p(v) - \sum_{e \in E^*} c(e)$ is maximal.

The PCST problem is frequently encountered in Operations Research where profit generating customers and the cost of creating a connecting network have to be chosen in the most profitable way, e.g. applications such as planning district heating or telecommunications networks.

In [4] Dittrich et al. describe a reduction from MWCS to PCST. Let $G = (V, E)$ and let (G, w) be an instance of MWCS with both positive and negative vertex weights and let $w' = \min_{v \in V} w(v)$ be its smallest vertex weight. Construct an instance (G, p, c) of PCST by setting the vertex profits to $p(v) = w(v) - w'$ for all $v \in V$ and all edge costs $c(e) = -w'$ for all $e \in E$. All vertex weights and edge costs are positive and as such this is a valid PCST instance.

Theorem 2.8. A Prize-Collecting Steiner Tree T in the transformed instance $(G = (V, E), p, c)$ is a connected subgraph in $(G = (V, E), w)$ with $w(T) = p(T) - w'$.

Proof. Prize-Collecting Steiner Tree T is a connected subgraph. Since T is a tree, the profit $p(T)$ of T is given by:

$$p(T) = \sum_{v \in V^*} p(v) - \sum_{e \in E^*} -w' = \sum_{v \in V^*} p(v) + |V^* - 1|w'$$

The score of T is given by:

$$w(T) = \sum_{v \in V^*} w(v) = \sum_{v \in V^*} (p(v) + w') = \sum_{v \in V^*} p(v) + |V^*|w' = p(T) - w'$$

□

In [2] a reduction from PCST to MWCS is given. Let $G = (V, E)$ and let G, p, c be an instance of PCST. Construct an instance G, w of MWCS by splitting each edge $(v, w) \in E$ into two edges (v, u) and (u, w) by introducing a split-vertex u, and set the weight of u to $w(u) = -c(e)$.

Theorem 2.9. A maximum weight connected subgraph T' in the transformed instance corresponds to an optimal prize-collecting Steiner tree T in the original instance, and $w(T') = p(T)$

Proof. Since for split vertex u we have $w(u) = -c(e)$, it holds that if $u \in T'$, its neighbors v and w must also be in T', otherwise $T' \setminus u$ would be a better solution. Each split vertex can then be mapped back to its original edge. The solution has profit $p(T) = w(T')$ and is optimal since a more profitable subgraph with respect to p would correspond to a higher scoring subgraph with respect to w, contradicting the optimality of T. □

2.3 Graph Connectivity: Biconnected Components

In graph theory, graph connectivity is an important concept which often arises in problems surrounding network reliability. As we will see in the next chapter, the notion of graph connectivity can also be used to decompose a large input graph into smaller subgraphs to be processed individually. The simplest form of a graph connectivity problem is the question whether or not a graph is connected, i.e. the number of connected components in a graph.

Definition 2.10 (**Connected component**)**.** Given an undirected graph $G = (V, E)$ a connected component of G is a subgraph $G' = (V', E')$ which contains a path between all vertex pairs $\{u, v\} \in V'$.

For each connected component, one could then examine if the there exists one weak link in the component, i.e., see whether the deletion of a single vertex is sufficient to disconnect the component.

Definition 2.11 (**Cut vertex**)**.** Given a graph $G = (V, E)$ a cut vertex of G is a vertex $v \in V$ that, when removed, increases the number of connected components of G.

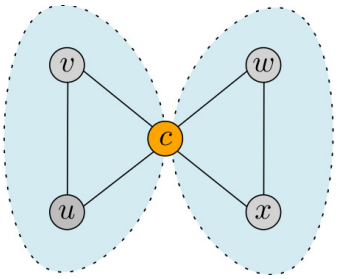

FIGURE 2.3: A graph G with cut vertex c and two biconnected components $\{u, v, c\}$ and $\{w, x, c\}$

The graph in Fig. 2.3 has one connected component. However, when removing vertex c (and its incident edges) the resulting graph consists of two connected components. If such a cut-vertex c does not exist, the graph is biconnected.

Definition 2.12 (**Biconnected component**)**.** A graph $G = (V, E)$ is biconnected if G does not contain any cut vertices. A biconnected component of a graph is a maximally biconnected subgraph.

Any connected graph can be decomposed into a tree of biconnected components called a block-cut tree. A block-cut tree of graph $G = (V, E)$ is a tree such that each vertex in the tree represents either a biconnected component or a cut vertex of G. A vertex representing a cut vertex is connected to all vertices representing biconnected components containing that cut vertex.

Definition 2.13 (**Block-cut tree**)**.** Given a graph $G = (V, E)$, let $C \subseteq V$ be the set of all cut vertices of G and let B be the set of all biconnected components of G, where each biconnected component b is the set of all vertices $v \in b$. The block-cut tree T of G is then given by the (bipartite) graph $T = (C \cup B, \{(c, b) \mid b \in B, c \in B, c \in C\})$.

A block-cut tree describes the structure of the biconnected components and the cut-vertices of a connected graph (or a connected component). See Fig. 2.4 for an example of a graph and its block-cut tree.

For each biconnected component as well as for each cut-vertex, a vertex is added to the block-cut tree. An edge is added between each biconnected component and each cut-vertex that belongs to it. In [12] Hopcroft and Tarjan describe a linear-time algorithm for finding the biconnected components of a graph. The algorithm is an adapted version of a depth-first search (DFS) and breaks the graph into its biconnected components. Upon reaching a new vertex during the DFS, it is placed on a stack and its DFS-number and low point (LP) are stored. The DFS-number is used to keep track of the DFS tree, the lowpoint of a vertex is the lowest point on the stack to which it is connected. When there is no new vertex reachable from the vertex on top of the stack (TOS), the lowpoint

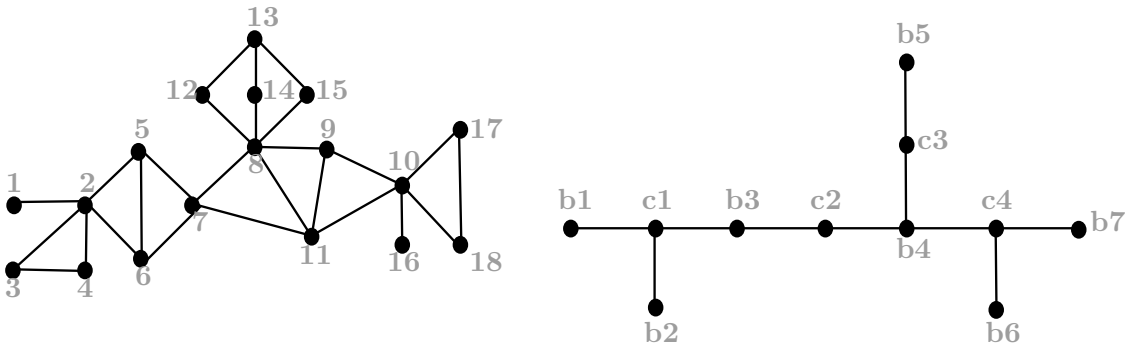

FIGURE 2.4: A graph (left) and its block-cut tree (right).
The bi-connected components are $b_1 = \{1, 2\}, b_2 = \{2, 3, 4\}, b_3 = \{2, 5, 6, 7\}, b_4 = \{7, 8, 9, 10, 11\}, b_5 = \{8, 12, 13, 14, 15\}, b_6 = \{10, 16\}, b_7 = \{10, 17, 18\}$. The cut-vertices are $c_1 = 2, c_2 = 7, c_3 = 8, c_4 = 10$.

of the current top of stack is checked, if the vertex does not connect to a vertex lower than the second vertex on the stack, then this second vertex is a cut vertex of the graph. The top of stack is then popped from the stack and the DFS is continued from the new top of stack. During the DFS, all edges visited are placed on a separate stack in order to retrieve the edges of the corresponding biconnected component in case a cut vertex is found. When the stack with vertices has only one vertex left, a complete DFS of a connected component has been performed. In case there are no other connected components the algorithm will terminate, otherwise, an unvisited vertex is chosen as a new starting point. Pseudocode for the algorithm can be found in Algorithm 1.

Algorithm 1 CONSTRUCTBLOCKCUTTREE($G = (V, E)$)

```
1:  vertexStack, edgeStack ← the empty stack
2:  DFS-count ← 0
3:  cutVertices ← ∅
4:  mark all v ∈ V unvisited
5:  while ∃v ∈ V s.t. v is unvisited do
6:      vertexStack, edgeStack ← the empty stack
7:      DFS(v) ← DFS-count++
8:      vertexStack.PUSH(v)
9:      if TOS has outgoing edge e = (TOS, x) then
10:         PROCESSEDGE((TOS, x))
11:     else
12:         PROCESSVERTEX(v)
13:     end if
14: end while
```

15: **function** PROCESSEDGE($e = (TOS, x)$)
16: $G \leftarrow G = (V, E \setminus e)$
17: *edgeStack*.push(e)
18: **if** x is unvisited **then**
19: DFS(x) $\leftarrow$ *DFS-count++*
20: LP(x) $\leftarrow$ DFS(TOS)
21: *vertexStack*.PUSH(x)
22: **else**
23: **if** DFS(v) $<$ LP(TOS) **then**
24: LP(TOS) $\leftarrow$ *DFS(v)*
25: **end if**
26: **end if**
27: **end function**

28: **function** PROCESSVERTEX(v)
29: **if** *vertexStack*.SIZE > 1 **then**
30: **if** LP(TOS) == DFS(TOS -1) **then**
31: $(a, b) \leftarrow$ *edgeStack*.PEEK()
32: *biconnectedComponents*.ADD(FINDBICONNECTEDCOMPONENT((a, b)))
33: **else**
34: **if** LP(TOS $<$ LP(*TOS*)-1) **then**
35: LP(TOS-1) $\leftarrow$ LP(TOS)
36: **end if**
37: **if** $\exists u$ s.t. u is a child of v AND LP(u)$\geq$ DFS(v) **then**
38: *cutVertices*.ADD(v)
39: **end if**
40: *vertexStack*.POP(v)
41: **end if**
42: **end if**
43: **end function**

44: **function** FINDBICONNECTEDCOMPONENT($e = (a, b)$)
45: *biconnectedComponent* $\leftarrow \emptyset$
46: **while** $a! = TOS - 1$ **do**
47: *biconnectedComponent*.ADD((a, b))
48: *edgeStack*.POP()
49: $(a, b) \leftarrow$ *edgeStack*.PEEK()
50: **end while**
51: **return** *biconnectedComponent*
52: **end function**

Chapter 3

Methods and Algorithms

In this chapter we present our method for solving MWCS using color coding. We first discuss a preprocessing scheme to reduce the input instance (Section 3.1). In Section 3.2 we discuss the main algorithm of our method, color coding. We describe unrooted color coding and introduce block-cut color coding, which uses both unrooted as well as rooted color coding.

3.1 Preprocessing

In [2] El-Kebir et al. describe a set of reduction rules to simplify an instance of MWCS. The reduction rules make use of two different operations on vertex sets: Merge and Remove. Given a graph $G = (V, E)$ and a vertex set $V' \subseteq V$, Remove(V') removes all vertices $v \in V'$ from V and their incident edges from E. Merge(V') calls the operation Remove(V') and combines all vertices $v \in V'$ into one supervertex s with weight $w(s) = \sum_{v \in V'} w(s)$. For every edge $(u, v) \in E$ with $u \in V' \wedge v \notin V'$ an edge (u, s) is added to E. The supervertex gets a label containing all names of all vertices in the supervertex, e.g., Merge(v, w, z) creates a supervertex with label $\{v\ w\ z\}$.

The preprocessing scheme consists of three increasingly complex phases. Fig. 3.1, Fig. 3.2, and Fig. 3.3 contains examples for the rules of Phase I, Phase II, and Phase III respectively.

- Phase I consists of three simple rules.

    1. *Remove isolated negative vertices*: Let $v \in V$ be an isolated vertex with $w(v) < 0$. Since v will never be part of an optimal solution v can safely be

removed by calling REMOVE($\{v\}$). Identifying all isolated negative nodes can be done in $\mathcal{O}(|V|)$ time.

2. *Merge adjacent positive vertices*: Let $(u, v) \in E$ be an edge with $w(u) \geq 0$ and $w(v) \geq 0$. Since both vertices have a positive weight, if one vertex is part of the optimal solution, the other vertex will be as well. Thus, MERGE($\{u, v\}$) can be called. Identifying all adjacent positive vertices can be done in $\mathcal{O}(|E|)$ time.
3. *Merge negative chains*: Let $P \subseteq V$ be a chain of degree 2 vertices, all with a negative weight. Either all vertices in P are part of the optimal solution since they connect two positive weighted subgraphs, or none of the vertices in P are part of the optimal solution. As such, it is safe to call MERGE(P). Identifying all negative chains can be done in $\mathcal{O}(|E|)$ time.

- Phase II consists of one rule.

1. *Remove mirrored vertex*: Let $u, v \in V, u \neq v$ be adjacent to the same vertices and let $w(u) < 0$. Without loss of generality, we can assume that $w(u) < w(v)$. Since v will always be preferred over u for the optimal solution and u, v are adjacent to the same vertices, we can safely call REMOVE($\{u\}$). Finding all pairs of mirrored hubs takes $\mathcal{O}(\Delta \cdot |V|^2)$ where Δ is the maximum degree of the graph.

- Phase III consists of one expensive rule.

1. *Least-cost rule*: This rule is an adaptation from the least-cost test for the vertex-weighted Steiner tree problem as described in [13]. Let $(u, v) \in E$ and $(v, w) \in E$, with $w(v) < 0$ and $\deg(v) = 2$. We try to find the least-cost path from u to w as follows: we construct a directed graph $G' = (V, A)$ where A is the arc set obtained by introducing two directed arcs (a, b) and (b, a) for each undirected edge $(a, b) \in E$. Let the weight of arc $w(a, b) = \max\{-w(b), 0\}$. If the least-cost path has a weight smaller than $-w(v)$, v will not be part of the optimal solution. Thus, REMOVE($\{v\}$) can safely be called. Applying the least-cost rule takes $\mathcal{O}(|V'| \cdot (|E| + |V| \log |V|))$ time where V' is the set of all negative vertices with degree 2.

The *Remove mirrored vertex* rule is slightly different from the rule as described by El-Kebir. In [2], El-Kebir states that both $u, v \in V$ need to be negatively weighted in order for this rule to be triggered. It is however sufficient for only one vertex to be negative as is described here. This allows for more relaxed constraints, i.e., only one of the vertices has to be negative, for the application of this rule.

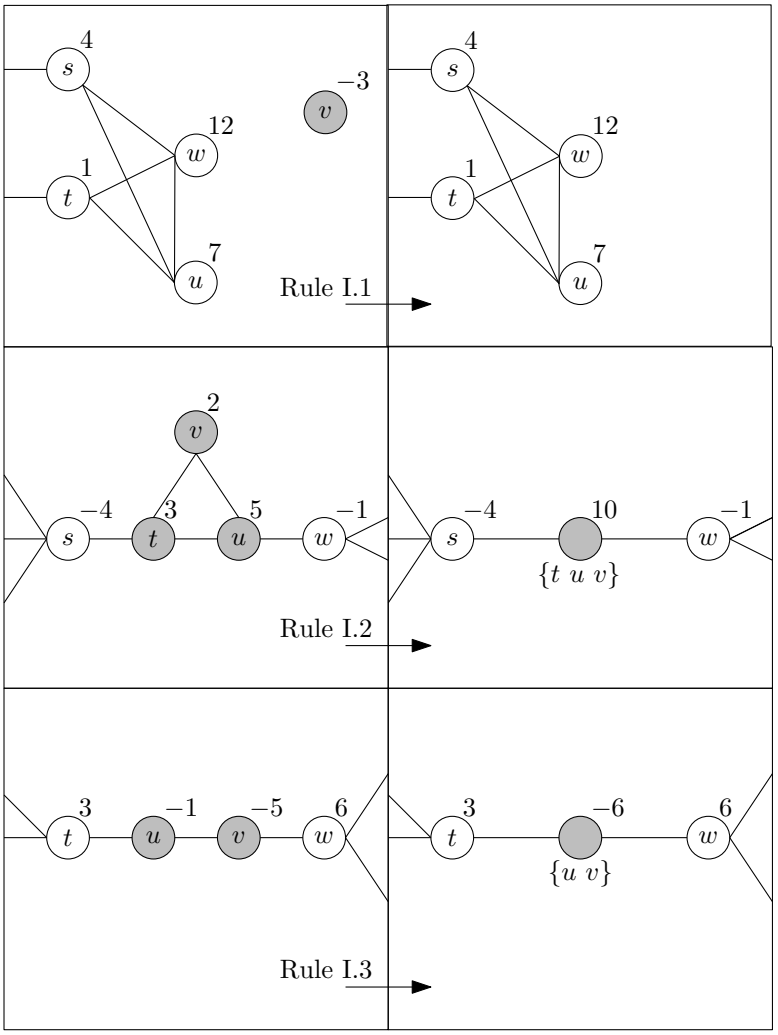

FIGURE 3.1: An example for each of the rules of Phase I. On the left the graph is shown before application, on the right the result is shown after the specific rule is applied. The first example shows the removal of isolated negative vertex v. The second example merges adjacent positive vertices t, u, v into a supervertex labeled $\{t\ u\ v\}$ with weight $w(t) + w(u) + w(v)$. The last example merges negative chain u, v into a supervertex labeled $\{u\ v\}$ with weight $w(u) + w(v)$.

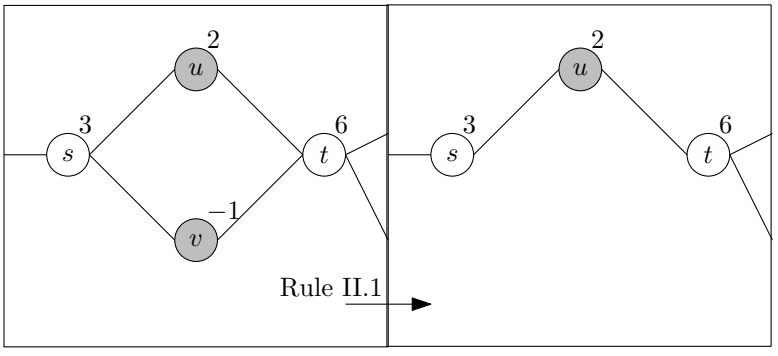

FIGURE 3.2: An example for the rule of Phase II. On the left the graph is shown before application, on the right the result is shown after negative mirrored vertex v is removed.

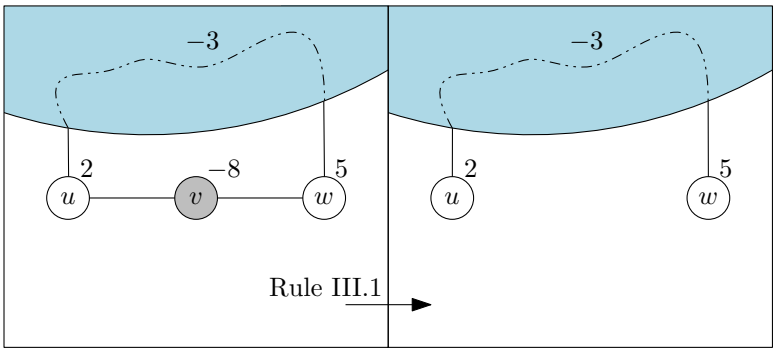

FIGURE 3.3: An example for the rule of Phase III. On the left the graph is shown before application, on the right the result is shown after the least-cost rule is applied, removing vertex v.

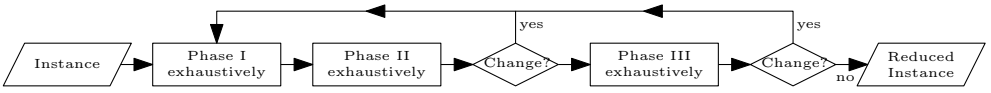

FIGURE 3.4: The preprocessing scheme adapted from [2]: an MWCS instance passes through each phase of rules that are applied exhaustively.

These phases are applied to an MWCS instance exhaustively until no rules of that phase can be applied anymore. A more detailed scheme of how exactly these phases are applied can be seen in Fig. 3.4. Phases I and II are applied exhaustively, repeatedly, in that order. If no changes are made in the last iteration, Phase III is applied. If changes were made, the whole process repeats again. Otherwise the preprocessing is finished. The result after preprocessing is a reduced MWCS instance.

3.2 Color Coding for MWCS

The color coding technique has been described in Chapter 2, where it was applied to find the minimum weight path of a graph. Here, we will describe a dynamic-programming algorithm based on the color coding technique to solve MWCS. We will describe algorithms to solve both the unrooted and rooted versions of MWCS (Sections 3.2.1 and 3.2.2 resp.). In addition, in Section 3.2.3 we will describe an algorithm for solving MWCS which first creates a block-cut tree of the input graph, after which it runs both the unrooted and rooted versions on each block (biconnected component) separately in order to find an optimal solution.

3.2.1 Unrooted Color Coding

In this section, we describe a dynamic-programming algorithm that uses the color coding technique to solve the unrooted MWCS problem (Def. 2.5). Our proposed algorithm

finds an optimal solution up to size k for the unrooted MWCS problem, given a certain error probability. Similar to the color coding algorithm as described in Chapter 2 our algorithm runs for a predefined number of iterations, which depends on the error probability. Since we have no information on the number of vertices in an optimal solution, we need to account for the worst-case scenario. This means that, for each iteration, the probability that an optimal solution of at most k is colorful, i.e., all vertices have a different color, is $\frac{k!}{k^k} > e^{-k}$. Thus, to ensure that the success probability of finding an optimal solution is at least $1 - \delta$, given an error probability δ, we need $\mathcal{O}(\ln(1/\delta)e^k)$ iterations.

During each iteration the algorithm builds a dynamic-programming table (DP-table). Given the input instance $G = (V, E)$, vertex weights w and the number of colors k, let C be a set of k colors and $w(v)$ the weight of $v \in V$. In addition, let $P \subseteq C$ and let S be a connected subnetwork of G with $S = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$. Entries in the DP-table are of the form $T[v, P] = S$ indicating that an optimal colorful connected subnetwork $S = (V', E')$ has been found containing a vertex of each color $c \in P$ exactly once and $v \in V'$. The weight of subnetwork S is given by $w(S) = \sum_{v \in S} w(v)$. Pseudocode for the algorithm is shown in Algorithm 2.

Algorithm 2 solveUnrooted$(G = (V, E), w, \delta, k)$

for each *iteration* $\leq \ln(1/\delta)e^k$ **do**
  $C \leftarrow$ a set of colors with $|C| = k$
  colorVertices(V, C)
  Table $T \leftarrow$ initializeTable(V, w, C)
  for $2 \leq i \leq k$ **do**
    for each vertex $v \in V$ **do**
      for each $P \subseteq C$ with $|P| = i$ **do**
        if $c(v) \in P$ **then**
          $T[v, P] \leftarrow$ dpLookup(G, v, w, P)
        end if
      end for
    end for
  end for
  return $T[v, P] : \max_{v \in V} \{ \max_{P \neq \emptyset, P \subseteq C, |P| \leq k} \{w(T[v, P])\}\}$
end for

function colorVertices(V, C)
  for each $v \in V$ **do**
    $c(v) \leftarrow$ a random color $c \in C$
  end for
end function

Let each vertex $v \in V$ be colored uniformly at random with one of k colors at the start of each iteration. To initialize the DP-table the following entries are inserted into the

```
21: function INITIALIZETABLE(V, w, C))
22:     for each vertex v do
23:         for each color c ∈ C do
24:             if c(v) then
25:                 T[v, {c(v)}] ← {v}
26:             end if
27:         end for
28:     end for
29: end function
```

DP-table for each $v \in V$ and each color set $P \subseteq C$ with $|P| = 1$:

$$T[v, \{c\}] = \begin{cases} \{v\} \text{ if } c(v) \in P \\ \emptyset \text{ otherwise} \end{cases}$$

Algorithm 2 INITIALIZETABLE(V, w, C) shows the pseudocode for initializing the DP-table. In contrast to the pseudocode as described by Hansen [9] our algorithm fills the DP-table sparsely. We only create an entry $T[v, \{c(v)\}]$ if vertex v has color c. As such, our DP-table will take up less memory than if we would fill the DP-table densely. Furthermore, the number of DP-lookups that need to be done later in our algorithm is greatly decreased.

After inserting all entries for $|P| = 1$ into the DP-table, entries for $|P| = 2, \ldots, k$ are inserted. Here, we only need to perform a DP-lookup if $c(v) \in P$, i.e., if the color of v is an element of the current color set. This additional step is also a slight improvement to the algorithm since it decreases the number of DP-lookups that have to be performed.

The intuition behind a DP-lookup for each entry $T[v, P]$ is the following: for each neighbor u of v with $c(u) \in P \setminus c(v)$ and for each non-empty subset $Q \subset P$, the entries $T[v, Q]$ and $T[u, P \setminus Q]$ are looked up in the DP-table. Note that the first requirement, $c(u) \in P \setminus c(v)$, is again a minor improvement of the algorithm as described in both [9] and [8]. By adding this constraint we prevent that a DP-lookup takes place for a neighbor u of vertex v that cannot be part of a solution since $c(u) \notin P \setminus c(v)$. Those entries whose union has a maximum weight are then stored in the table as a new entry $T[v, P] = (T[v, Q] \cup T[u, P \setminus Q])$. More formally, new entries are inserted into the DP-table according to the following recurrence:

$$T[v, P] = \left\{ T[v, Q] \cup T[u, R] : \max_{\forall u : uv \in E} \left\{ \max_{\forall Q, R : Q \cap R = \emptyset, Q \cup R = P} \left\{ w(T[v, Q]) + w(T[u, R]) \right\} \right\} \right\}$$

First, for each vertex $v \in V$ and each set P_i of $i = 2$ colors, $T[v, P_2]$ is computed. After each round, i is increased by one and $T[v, P_i]$ is computed and inserted into the

DP-table for each vertex $v \in V$ and each set $P_i \subset C$ of i colors. In the final round, $T[v, P_k] = T[v, C]$ is computed for each vertex $v \in V$. Let S^* be a colorful maximum weight subgraph of G. When the table is completely filled, S^* can be computed as follows:

$$S^* = T[v, P] : \max_{\forall v : v \in V} \left\{ \max_{\forall P : P \neq \emptyset, |P| \leq k} \{T[v, P]\} \right\}$$

The score of S^* is $w(S^*) = \sum_{v \in S^*} w(v)$.

```
30: function DPLOOKUP(G = (V, E), v, w, P))
31:     currentBest ← ∅
32:     for each neighbor u of v do
33:         if c(u) ∈ P \ c(v) then
34:             for each Q ⊂ P with Q ≠ ∅ do
35:                 R ← P \ Q
36:                 if c(v) ∈ Q and c(u) ∈ R then
37:                     candidate ← { max_{∀Q,R:Q∩R=∅,Q∪R=P} {w(T[v, Q]) + w(T[u, R])} }
38:                     if w(candidate) > w(currentBest) then
39:                         currentBest = candidate
40:                     end if
41:                 end if
42:             end for
43:         end if
44:     end for
45:     return currentBest
46: end function
```

Let $\deg(u)$ be the degree of vertex u. It takes $\mathcal{O}(\deg(u)2^{|T|}) = \mathcal{O}(\deg(u)2^k)$ time to compute $w(u, T)$ for each vertex u and each color set T. Given a coloring, the computation of the DP-table can be performed in $\mathcal{O}(2^k \sum_{u \in V} \deg(u)2^k) = \mathcal{O}(|E|4^k)$. Since $\mathcal{O}(\ln(1/\delta)e^k)$ iterations are sufficient to ensure that the probability of an optimal solution S^* being found is $\geq 1 - \delta$, the overall time complexity is $\mathcal{O}(\ln(1/\delta)|E|(4e)^k)$.

3.2.2 Rooted Color Coding

The rooted color coding algorithm differs from the unrooted version in that it takes one extra parameter as input, a set of root vertices. Instead of finding any optimal solution we are now only interested in an optimal solution containing each node in the root set (Def 2.6). The DP-table is initialized and created the same way as described in the previous section. However, when returning an optimal solution we have an extra constraint – we return an entry in the DP-table which has maximum weight and contains all vertices of root set X. Thus S^* can be computed as follows:

$$S^* = T[v, P] : \max_{\forall v : v \in V} \left\{ \max_{\forall P : P \neq \emptyset, |P| \leq k, \forall u \in X : u \in P} \{w(T[v, P])\} \right\}$$

The score of S^* is $w(S^*) = \sum_{v \in S^*} w(v)$.

Algorithm 3 SOLVEROOTED$(G = (V, E), w, \delta, k, X)$

for $1 \leq iteration \leq \ln(1/\delta)e^K$ **do**
  $C \leftarrow$ a set of colors with $|C| = k$
  COLORVERTICES(V, C)
  Table $T \leftarrow$ INITIALIZETABLE(G, w, C)
  for $2 \leq i \leq k$ **do**
    for each vertex $v \in V$ **do**
      for each $P \subseteq C$ with $|P| = i$ **do**
        if $c(v) \in P$ **then**
          $T[v, P] \leftarrow$ DPLOOKUP(G, v, w, P)
        end if
      end for
    end for
  end for
  return $T[v, P] : \max_{\forall v: v \in V} \left\{ \max_{\forall P: P \neq \emptyset, |P| \leq k, \forall u \in X: u \in P} \{w(T[v, P])\} \right\}$
end for

3.2.3 Blockcut Color Coding

Apart from the preprocessing scheme to reduce an MWCS instance, El Kebir et al. [2] also introduce a two-layer divide-and-conquer scheme for solving MWCS to provable optimality based on decomposing the input graph into its connected and biconnected components (Algorithm 4). In the first layer, all connected components of the input graph are considered individually. For each connected component, a block-cut tree (Def. 2.13) is constructed. This block-cut tree is then processed from the leaves up to the root, thus only those blocks with degree 0 or 1 are allowed to be processed. For each biconnected component (block) it is first checked whether it is block-negative. If for all $v \in B$ we have $w(v) \leq 0$, i.e. the block is negative, processing it can be skipped – since the blocks are processed in a bottom up fashion, an entirely negative block can never be part of any optimal solution. Otherwise, the block does contain positive vertices and it needs to be processed. After processing a block, an optimal solution for the block is stored and the block-cut tree is updated, i.e. the processed block and its incident edges are removed from the tree. Fig. 3.5 shows an example of how color coding is applied to a block-cut tree.

In the second layer the biconnected components are considered (Algorithm 4 function PROCESSBICOMPONENT$(G = (V, E), w)$). Cut-vertices are special vertices that are processed as part of the blocks they belong to. We distinguish between two cases, either the block B has a cut-vertex c, or block B does not have a cut-vertex. If block B does have a cut-vertex c, maximal solutions for both the unrooted color coding algorithm (V_{u}^*)

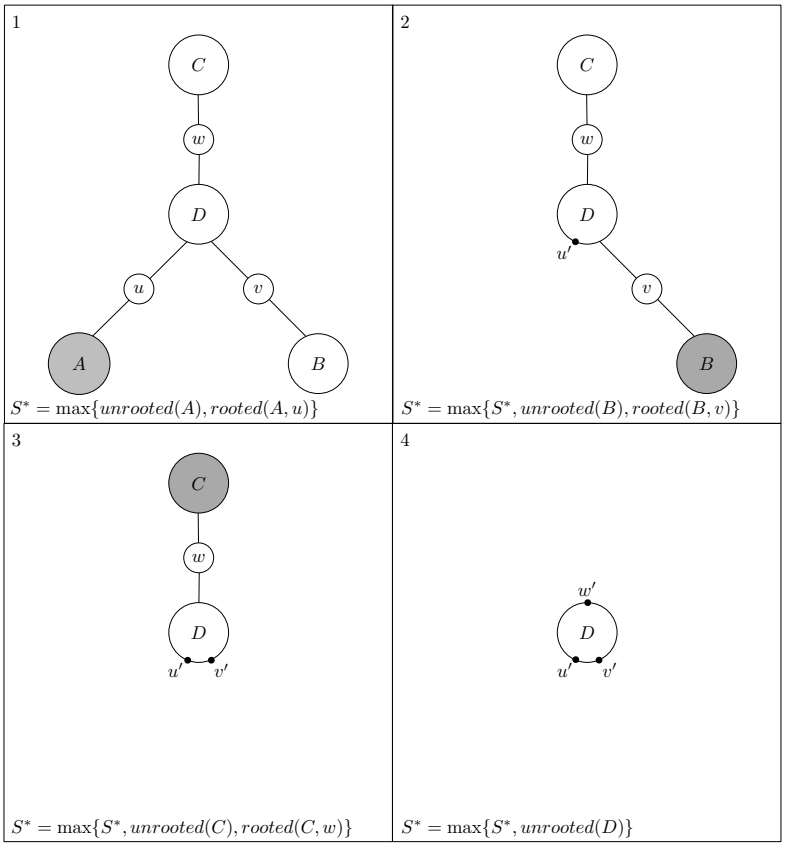

FIGURE 3.5: An example of how our algorithm is applied to a block-cut tree. One of the leafs of the tree, block A, is processed first, both unrooted color coding as well as the rooted version with cut-vertex u as the root are executed. A maximum solution of these two calls is stored in S^*. Furthermore, the weight of cut-vertex u is updated to the weight of the solution found by the rooted color coding (shown here as u'). This process is repeated for all leafs. After each processed block, a maximal solution found up until that point is stored in S^* and the weight of the cut-vertex is updated. Finally, only the unrooted version has to be executed on the last remaining block D and an optimal solution for the entire graph is stored in S^*.

Algorithm 4 SOLVEBLOCKCUT$(G = (V, E), w, \delta, k)$

```
1: V* ← ∅
2: for each connected component C of G do
3:     PREPROCESS(C)
4:     T_B ← block-cut vertex tree of C
5:     while biconnected component B ∈ T_B has degree 0 or 1 do
6:         if B ≠ block-negative then
7:             V* ← PROCESSBICOMPONENT(B, w, δ, k)
8:             update T_B
9:         end if
10:    end while
11: end for
12: return V*
```

as well as for the rooted color coding algorithm with c as the root (V_{r}^*) will be stored. The maximal solution for the entire block is then given by $\mathrm{argmax}_{V^* \in \{V_{\mathrm{u}}, V_{\mathrm{r}}\}} \{\sum_{v \in V^*} w(v)\}$. As a last step, the cut-vertex is updated to V_{r}^* if $\sum_{v \in V_{\mathrm{r}}^*} w(v) > w(c)$ to make sure this possible partial solution will be taken into account whenever a biconnected component containing the same cut-vertex c is being processed.

```
13: function ProcessBicomponent(G = (V, E), w, δ, k)
14:     let c be the corresponding cutnode, if applicable
15:     if c ∈ V then
16:         V_u^* ← solveUnrooted(G, w, δ, k)
17:         V_r^* ← solveRooted(G, w, δ, k, c)
18:         V^* ← argmax_{V^* ∈ {V_u, V_r}} {Σ_{v ∈ V^*} w(v)}
19:         if Σ_{v ∈ V_r^*} w(v) > w(c) then
20:             c ← V_r^*
21:         end if
22:     else
23:         V^* ← solveUnrooted(G, w, δ, k)
24:     end if
25:     return V^*
26: end function
```

Chapter 4

Implementation

In this chapter we will discuss how our method was integrated into Heinz[1] and we will briefly describe the implementation itself. After discussing the Heinz integration in general (Section 4.1) we explain the data structures used (Section 4.2) and we describe the adapter needed for the Heinz integration in more detail (Section 4.3).

4.1 Heinz integration

Our algorithm is implemented as part of Heinz and implemented using C++. All three versions of the algorithm as described in Chapter 3 (unrooted, rooted and blockcut color coding) have been implemented. Furthermore, the number of iterations and the number of colors that should be used while executing the algorithm can be passed to Heinz as arguments. The reason for choosing the number of iterations as a parameter over error probability δ has to do with the fact that it gives the user more control over the running time of the algorithm since this is the most restricting factor of the algorithm. The user can run the algorithm for a specific k and 1 iteration and calculate the number of possible iterations i that can be performed in the desired running time. The error probability δ can then be calculated by $\delta = e^(- i/e^k)$.

Heinz makes use of Lemon (Library for Efficient Modeling and Optimization in Networks) for the implementation of graphs. Lemon is a C++ template library providing efficient implementations of common data structures and algorithms with focus on combinatorial optimization tasks connected mainly with graphs and networks. For the implementation of color coding, however, using a library like Lemon proved too slow and thus we decided to implement our algorithm with our own data structure. This data structure is described in Section 4.2.

[1] `https://github.com/ls-cwi/heinz`

The preprocessing algorithm (Section 3.1) as well as the algorithm to create a blockcut tree of the input graph were already implemented in Heinz and as such, these implementations were re-used. However, because of the different data structures used in the already existing ILP implementation of Heinz and our new color coding implementation as described in the next section, the processing of the blockcut tree had to be implemented differently. In order to be able to work with the two different data structures (one for the preprocessing and creating the blockcut tree, and one for color coding algorithm) and to make the integration as modular as possible, an adapter was written to which control is given in case the color coding algorithm has to be run.

For testing purposes we adjusted the existing ILP to be able to restrict the size, the number of vertices, of the optimal solution found.

4.2 Data structures

Using Lemon for our implementation of color coding proved to slow. In order to increase the performance of our algorithm we adjusted the data structure used for the graph. To store all data needed about the vertices we used the class vector of the standard library of C++. For storing the labels of the vertices and their weights we use two separate vectors, `labels` and `weights` resp., such that the data of each vertex is stored at the same index position of each vector, i.e. $w($`labels[`v`]`$) =$ `weights[`v`]`. A label can be seen as the name of the vertex. It is either the name of the original vertex from the input graph, or the names of the merged vertices inside a supervertex created by the preprocessing phase, e.g., $\{u\ v\ w\}$. Since we need to look up the neighbors of each vector often during the execution of the algorithm we also used a vector `neighbors` such that `neighbor[`v`]` contains a vector with the index positions of all neighbors of v allowing for direct lookup of the data of these neighbors in all vectors.

The DP-table is implemented as a 3-dimensional vector. The indices of the first vector correlate to the indices of the vertices of the graph. Then, each index position contains a vector in which all color sets for which there are entries (the DP-table is filled sparsely) for the current vertex are stored. Each color set also contains a vector storing the weight in the first index position for quick access and the indices of the neighbors that are part of this subgraph in the following indices.

Each color set is stored as an unsigned integer of 32 bits (`uint_32`), where each bit represents a color. If a bit is set to 1 this means this color is present in the set, if it is set to 0 the color is not an element of the set. This allows for quick comparison between sets using bit manipulation and it requires very little memory to store each color set.

As will be shown in Chapter 5 pushing the algorithm to run with 9 colors is already a far stretch and thus being able to store a maximum 32 colors should be more than sufficient.

4.3 Adapter

Whenever Heinz is called with the color coding flag set control is passed to the color coding adapter. The adapter checks the version of color coding it needs to run. In case of block cut color coding, the adapter processes each connected component of the input graph individually. For each component the adapter creates a subgraph using Lemon, preprocesses the subgraph if needed, and calls the function to create the block cut tree from this subgraph. For each block with degree 0 or 1 the block negativity is checked. If the block is not negative, the adapter transforms the data structure from a Lemon graph into the data structure as described in the previous section. The block is then processed by calling the unrooted and, if needed, rooted color coding functions. If only the rooted or unrooted color coding versions need to be executed, the adapter preprocesses the graph if needed, transforms the graph from Lemon into the color coding data structure and calls either the unrooted or the rooted color coding function.

Chapter 5

Results and Discussion

We tested the performance of our algorithm using the ACTMOD dataset of the DIMACS11 challenge. The full names and the abbreviations used in this thesis can be found in Table 5.1.

Name	Abbr.
drosophila001	dro1
drosophila005	dro5
drosophila0075	dro75
HCMV	HCMV
lymphoma	lymph
metabol_expr_mice_2	meta2
metabol_expr_mice_3	meta3

TABLE 5.1: The full name and abbreviation for each dataset.

All experiments were run on a single CPU of a machine with 16 CPUs of the type Intel Xeon ES-2667 v4 (3.20GHz) and 512GB of memory. We adjusted the ILP in Heinz such that it returned an optimal solution up to a specific k in order to compare our results to the optimal solution of size k. We set δ to 0.1, resulting in a success probability of 0.9 for each experiment. We chose a cut-off time of two hours for our tests. In the case that an experiment would run longer than two hours, the maximum possible number of iterations were calculated such that a single run would not exceed the two hour mark. As such, δ was increased for these runs. To determine the number of iterations for each test, we ran 1 iteration. Let t_1 be the time it takes to run exactly 1 iteration of the algorithm. We then calculated $it_{time} = 7200/t_1$, the maximum number of iterations that could be run in 2 hours. We also calculated it_δ, the number of iterations needed to get a maximum error probability δ of 0.1. We then chose the minimum of both, i.e., $\min(it_{time}, it_\delta)$.

We tested our algorithm from $k = 4$ to $k = 9$ for each dataset. Table 5.2 shows the adjusted values of the success probability per k per dataset. For $k = 4$ and $k = 5$ we were able to run all iterations needed to keep the success probability at 0.9. For $k \geq$ however, we had to decrease the success probability to make sure a single run would not take longer than two hours.

	4	5	6	7	8	9
drosophila001	0.9	0.9	0.3	0.04	0.003	0.0002
drosophila005	0.9	0.9	0.31	0.03	0.003	0.0002
drosophila0075	0.9	0.9	0.32	0.04	0.003	0.0002
HCMV	0.9	0.9	0.9	0.32	0.03	0.002
lymphoma	0.9	0.9	0.9	0.37	0.04	0.003
metabol_expr_mice_2	0.9	0.9	0.9	0.9	0.21	0.02
metabol_expr_mice_3	0.9	0.9	0.9	0.9	0.69	0.03

TABLE 5.2: The success probability $(1 - \delta)$ per k per dataset.

All instances were preprocessed before the color coding algorithm was applied. Table 5.3 shows the size of the reduced graph after preprocessing. Table 5.4 and Table 5.5 show

	$\|V\|$	$\|E\|$	$\|C\|$	prep($\|V\|$)	prep($\|E\|$)	prep($\|C\|$)
dro1	5226	93394	1	3796	88810	1
dro5	5226	93394	1	3743	86505	1
dro75	5226	93394	1	3702	84854	1
HCMV	3863	29293	78	1387	8188	4
lymph	2034	7756	1	1308	6721	1
meta2	3514	4332	166	637	1029	1
meta3	2853	3335	166	426	722	1

TABLE 5.3: Number of vertices ($|V|$), edges ($|E|$) and components ($|C|$) for each dataset, before and after the preprocessing phase.

the number of vertices in the (optimal) solutions per k found by color coding and the ILP respectively. It is clear that preprocessing the input instance has significant consequences for the possible size of the found solution. Without preprocessing, any solution found would have a maximum size of k. Using preprocessing, solutions of up to 79 vertices were found.

We ran both the unrooted as well as the block-cut versions of our algorithm on the data and found that there is no significant difference between the two methods, both with respect to the running time as well as with respect to the solution found. The differences that we did find are the result of the error probability $\delta \geq 0.1$ combined with luck depending on the randomized assignment of colors to the vertices. Running the algorithm multiple times on smaller graphs (blocks) does not improve the running time of the algorithm as a whole. A reason for this could be the fact that even though the

	4	5	6	7	8	9
dro1	1	1	1	1	1	9
dro5	29	30	40	41	43	43
dro75	58	59	72	72	78	79
HCMV	4	5	5	7	10	5
lymph	13	16	18	19	20	21
meta2	6	10	10	12	15	15
meta3	17	24	24	26	26	26

TABLE 5.4: Solution size in number of vertices while running color coding after preprocessing.

	4	5	6	7	8	9	∞
dro1	1	1	1	1	9	9	38
dro5	29	30	40	41	43	48	175
dro75	58	59	72	73	79	79	240
HCMV	4	5	5	7	10	11	17
lymph	13	16	18	19	20	21	46
meta2	6	10	10	12	15	15	24
meta3	17	24	24	26	26	28	87

TABLE 5.5: Solution size in number of vertices while running the ILP after preprocessing.

graph is decomposed into smaller subgraphs, one large block with a size almost as big as the entire (preprocessed) graph remains in all datasets. These large blocks account for the biggest part of the running time. Table 5.6 shows the total number of vertices left after preprocessing and the largest block size in each graph.

	prep($\|V\|$)	blocks	size of largest block
dro1	3796	21	3738
dro5	3743	21	3685
dro75	3702	21	3644
HCMV	1387	8	1342
lymph	1308	1	1308
meta2	637	1	637
meta3	426	1	426

TABLE 5.6: The number of vertices after the preprocessing phase, the number of vertices in the largest block per dataset and the number of blocks.

As an example, Table 5.7 and Table 5.8 show the running time, the optimal weight found and the number of iterations for the unrooted color coding version and the block-cut color coding version for the dataset HCMV. It is clear that the differences found in both the running time as well as the optimal weight found are negligible. The other datasets show the same behavior – Appendix A shows the test results on all datasets. Since these

two versions of color coding do not show any significant difference in their behavior or results, we will not make any distinction between the two from here on.

	4	5	6	7	8	9
Weight	5.50	5.70	5.70	6.02	6.51	5.70
Time	26.26	305.04	3471.11	6579.98	7024.49	6968.51
Iterations	125	341	928	423	104	22

TABLE 5.7: Running time and optimal weight found for the HCMV dataset using unrooted color coding

	4	5	6	7	8	9
Weight	5.50	5.70	5.70	6.02	6.52	5.70
Time	26.24	303.80	3437.00	6477.19	6997.82	6909.38
Iterations	125	341	928	423	104	22

TABLE 5.8: Running time and optimal weight found for the HCMV dataset using blockcut color coding

From our experiments it has become clear that the color coding algorithm cannot compete with the ILP based algorithm as implemented in Heinz. In all cases, the optimal solution found by the ILP without a restriction on k was much better than what our algorithm found on any of the runs.

As long as we were able to keep the success probability at 0.9 our method was able to find the optimal solution for a given k, albeit in a much slower running time than the ILP did. As soon as the two hour mark was hit and the success probability was decreased, only one of all solutions found by our algorithm was an optimal solution. The exponential increase in running time can be seen in the table in Appendix A for each dataset, even though it is capped off at two hours (7200sec).

The results of our experiments for each dataset can be found in Table 5.9. The left top corner shows the name of the dataset and each column represents a value for k. The first row (ILP) show the optimal weight found by the ILP and the second row shows the optimal weight found by color coding. To get a better insight into the relation between these weights, the third row shows the relative weight of the color coding solution to the ILP solution, as given by $\frac{w(\text{CC})}{w(\text{ILP})}$. For all tests with $1 - \delta = 0.9$ the optimal solution for that k was found. For all but one dataset (drosophila005), optimal solutions were found by color coding as long as the success probability was above 0.3. In some cases, the optimal solution was found with even lower success probabilities, such as for drosophila005 with $k = 8$ and $1 - \delta = 0.003$ and metabol_expr_mice_2 with $k = 9$ and $1 - \delta = 0.02$. Furthermore, in all cases, the relative weight found was at least 80% of the optimal solution.

dro1	4	5	6	7	8	9
ILP	11.9359	11.9359	11.9359	11.9359	14.6241	14.6241
CC	11.9359	11.9359	11.9359	11.9359	11.9359	13.231
weight score	1	1	1	1	0.81	0.90
J(labels)	1	1	1	1	0	0.78
J(vertices)	1	1	1	1	0	0.8

dro5	4	5	6	7	8	9
ILP	40.131	42.6671	50.2389	55.868	58.4041	67.2555
CC	40.131	42.6671	49.3262	52.7423	58.4041	59.3297
weight score	1	1	0.98	0.94	1	0.88
J(labels)	1	1	0.71	0.56	1	0.5
J(vertices)	1	1	0.95	0.87	1	0.78

dro75	4	5	6	7	8	9
ILP	77.6151	80.5567	92.6747	98.6696	103.359	112.632
CC	77.6151	80.5567	92.6747	95.0692	102.001	103.387
weight score	1	1	1	0.96	0.96	0.91
J(labels)	1	1	1	0.56	0.6	0.5
J(vertices)	1	1	1	0.93	0.94	0.92

HCMV	4	5	6	7	8	9
ILP	5.49602	5.70381	5.70381	6.27261	6.69949	7.00598
CC	5.49602	5.70381	5.70381	6.27261	6.01626	5.70381
weight score	1	1	1	1	0.89	0.81
J(labels)	1	1	1	1	0.78	0.27
J(vertices)	1	1	1	1	0.82	0.23

lymph	4	5	6	7	8	9
ILP	30.0599	31.5504	35.2917	39.8447	43.0887	45.1728
CC	30.0599	31.5504	35.2917	39.8447	40.5045	40.1543
weight score	1	1	1	1	0.96	0.88
J(labels)	1	1	1	1	0.6	0.64
J(vertices)	1	1	1	1	0.82	0.83

meta2	4	5	6	7	8	9
ILP	148.751	162.361	162.361	190.1	214.472	214.472
CC	148.751	162.361	162.361	190.1	214.472	214.472
weight score	1	1	1	1	1	1
J(labels)	1	1	1	1	1	1
J(vertices)	1	1	1	1	1	1

TABLE 5.9: The weights found by both the ILP and color coding, the relative weight and both Jaccard indices for all datasets.

To get a better understanding of the overlap between the subgraphs themselves found by both the ILP and color coding, we calculated the Jaccard indices. The Jaccard index

meta3	4	5	6	7	8	9
ILP	252.293	356.923	356.923	373.095	373.095	376.932
CC	252.293	356.923	356.923	373.095	373.095	373.095
weight score	1	1	1	1	1	0.98
J(labels)	1	1	1	1	1	0.78
J(vertices)	1	1	1	1	1	0.93

TABLE 5.9: The weights found by both the ILP and color coding, the relative weight and both Jaccard indices for all datasets (continued).

is a measuring statistic for the similarity of finite sets, given by:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

The next two rows of Table 5.9 show Jaccard indices for each dataset. The first index (J(labels)) is based on the labels as created during the preprocessing phase. Each label can contain one or more vertices from the original input graph. Even when only taking these merged vertices into account, the subgraph found by color coding shows a large overlap with the labels found by the ILP. The last row (J(vertices)) shows the Jaccard indices for the single vertices contained inside all labels of the solution. Thus, this shows the similarity between the vertices in the solutions and the vertices as they existed in the original input graph. As can be expected, the Jaccard indices show results similar to the weights. With a success probability of 0.9 color coding found the same optimal solutions as the ILP did. Here the positive influence of the preprocessing becomes clear. The Jaccard indices based on the labels are considerable lower when compared to those of the actual vertices found. Apart from the test with $k = 9$ for the HCMV dataset, which has a Jaccard index of 0.23 based on the vertices, the similarity of the ILP solutions and the color coding solutions is at least 0.78. This shows that for higher k and lower success probabilities our algorithm is able to at least identify the location of an optimal solution.

To compare the results of the optimal solution found by the ILP without any restrictions on k and the solution found by our algorithm, we used the Tversky index [14] based on the optimal ILP solution and the solutions found by our algorithm for $k = 9$. We decided to use the results for $k = 9$ since this is the furthest we could push the running time of our algorithm. The Tversky index is an asymmetric similarity measure on sets that compares a variant to a prototype. Here, the prototype is the optimal solution and the variant is the solution of our algorithm. The Tversky index is given by:

$$\mathrm{T}_{(\alpha,\beta)}(X,Y) = \frac{|X \cap Y|}{|X \cap Y| + \alpha|X - Y| + \beta|Y - X|}$$

In our calculations, X represents the optimal solution and Y represents our solution. Setting $\alpha = 1, \beta = 0$ then indicates the similarity between the vertices in the optimal solution and the vertices in the solution found by color coding, i.e., $\mathrm{T}_{(1,0)}(X,Y) = 1$ means all vertices that are part of the optimal solution are found by our algorithm, whereas $\mathrm{T}_{(1,0)}(X,Y) = 0$ means that none of the vertices of the optimal solution are found by our algorithm. As can be seen in Table 5.10 in the first column labeled $\mathrm{T}_{(1,0)}(X,Y)$, our algorithm does not perform very well since only a small part of the optimal solution is found. Only in one case $\mathrm{T}_{(1,0)}(X,Y)$ is larger than 0.5. Setting $\alpha = 0$, $\beta = 1$, the Tversky index $\mathrm{T}_{(0,1)}(X,Y)$ gives us an indication of how many vertices of our solution are indeed part of the optimal solution, i.e., $\mathrm{T}_{(0,1)}(X,Y) = 1$ indicates that the entire subgraph found by our algorithm is part of the optimal solution, whereas $\mathrm{T}_{(0,1)}(X,Y) = 0$ indicates that none of the vertices found by our algorithm are part of the optimal solution. The second column of Table 5.10, labeled $\mathrm{T}_{(0,1)}(X,Y)$, shows that the subgraph found by our algorithm in most cases is almost completely a part of the optimal solution found by the ILP without any restrictions on k.

	$\mathrm{T}_{(1,0)}(X,Y)$	$\mathrm{T}_{(0,1)}(X,Y)$
dro1	0.16	0.67
dro5	0.23	0.93
dro75	0.31	0.95
HCMV	0.29	1
lymph	0.41	0.90
meta2	0.62	1
meta3	0.23	0.77

TABLE 5.10: The Tversky indices for all datasets

Chapter 6

Conclusions

Our algorithm consist of a 2-layer scheme which was introduced in [2]. There, the authors actually went a step further and implemented a 3-layer scheme which also decomposes the input instance into its triconnected components using SPQR-trees. However, since decomposing the input instance into its triconnected components did not work there for similar reasons as mentioned here for the biconnected components, we decided not to implement the 3-layer scheme. A detailed description on decomposing an input graph into its triconnected components can be found in [2].

Our experiments show that our algorithm cannot compete with the ILP method. One of the main reasons for this is the exponential explosion in running time with increasing k and as such the lower number of iterations resulting in a lower success probability. Here, luck with the random coloring of the vertices plays a role. To decrease the running time one could look into parallelizing the implementation since the algorithms is very suitable for parallelisation. Furthermore, another direction would be to investigate possible heuristics to improve the coloring of the vertices, or derandomization of the vertices.

Even though color coding cannot compete with the ILP approach, it does have benefits and might serve well as a heuristic. Similar to the ILP method, color coding is able to give intermediate results and as such can give a quick indication of the location of an optimal solution in the input instance. Furthermore, the Jaccard indices indicate that even for a high k and a low δ our algorithm is able to identify a large part of the subgraph containing an optimal solution of roughly the same size. Even in case the optimal solution is much larger than any solution our algorithm is able to find due to a restriction on k, the Tverski indices show that a significant part of the vertices found are part of the optimal solution. Our algorithm also does not make use of any commercial, closed source licenses such as CPLEX in case of the ILP implementation in Heinz. In

the end, the size of the solutions found with color coding combined with preprocessing make it a suitable heuristic to use for applications where an indication of the location of an optimal solution would be sufficient.

Appendix A

Test results

This appendix contains results of all experiments. Each table contains the data for one dataset and one version of the color coding algorithm, i.e., unrooted or blockcut. In each table, k is the number of colors, i is the number of iterations, w is the weigth found, s is the time in seconds it took to perform the test, and *vertices* is the optimal solution found by the algorithm.

drosophila001 — unrooted

k	i	w	s	vertices
4	125	11.9359	270.825	(1869)
5	341	11.9359	3172.38	(1869)
6	147	11.9359	5798.42	(1869)
7	41	11.9359	6847.38	(1869)
8	9	12.5998	6574.49	(372) (4272) (3114) (4858) (4330) (1575) (1852) (1822)
9	2	6576.16	11.9359	(1869)

drosophila001 — blockcut

k	i	w	s	vertices
4	125	11.9359	276.009	(1869)
5	341	11.9359	3213.68	(1869)
6	146	11.9359	7200	(1869)
7	41	11.9359	6891.82	(1869)
8	10	11.9359	7200	(1869)
9	2	13.231	6874.12	(1575) (4309) (3152 2262) (3675) (4272) (372) (168) (1822)

drosophila005 — unrooted

k	i	w	s	vertices
4	125	40.131	264.527	(4152 4903 1413 2842 802 1899 739 2381 3332 3634) (4748) (4698 4799 2635 616 2136 1214 4272 372 4157 3114 4483 4422) (3411 2937 2922 2921 2593 2556)
5	341	42.6671	3107.12	(444) (4748) (4698 4799 2635 616 2136 1214 4272 372 4157 3114 4483 4422) (4152 4903 1413 2842 802 1899 739 2381 3332 3634) (3411 2937 2922 2921 2593 2556)
6	146	50.1371	5668	(4152 4903 1413 2842 802 1899 739 2381 3332 3634) (4748) (4698 4799 2635 616 2136 1214 4272 372 4157 3114 4483 4422) (3093) (1168 572 4163 3551 883 1421 5088 1787 128 109) (3411 2937 2922 2921 2593 2556)
7	42	55.868	6846.65	(3411 2937 2922 2921 2593 2556) 4748 (4698 4799 2635 616 2136 1214 4272 372 4157 3114 4483 4422) (3022) (1168 572 4163 3551 883 1421 5088 1787 128 109) (3152 2262) (4152 4903 1413 2842 802 1899 739 2381 3332 3634)
8	9	55.2977	6422.75	1117 (4698 4799 2635 616 2136 1214 4272 372 4157 3114 4483 4422) (2979) (2584 2564 933 2450 464 103) (1168 572 4163 3551 883 1421 5088 1787 128 109) (4748) (4152 4903 1413 2842 802 1899 739 2381 3332 3634) (3411 2937 2922 2921 2593 2556)
9	2	56.2043	6705.28	(3411 2937 2922 2921 2593 2556) (4748) (4698 4799 2635 616 2136 1214 4272 372 4157 3114 4483 4422) (3148) (1168 572 4163 3551 883 1421 5088 1787 128 109) (3920) (4152 4903 1413 2842 802 1899 739 2381 3332 3634) (4858) (1575)

drosophila005 — blockcut

k	i	w	s	vertices
4	125	40.131	268.852	(3411 2937 2922 2921 2593 2556) (4748) (4698 4799 2635 616 2136 1214 4272 372 4157 3114 4483 4422) (4152 4903 1413 2842 802 1899 739 2381 3332 3634)
5	341	42.6671	3170.8	(4748) (4698 4799 2635 616 2136 1214 4272 372 4157 3114 4483 4422) (4152 4903 1413 2842 802 1899 739 2381 3332 3634) (3411 2937 2922 2921 2593 2556) (444)
6	150	49.3263	5846.4	(4748) (4152 4903 1413 2842 802 1899 739 2381 3332 3634) (3100) (4698 4799 2635 616 2136 1214 4272 372 4157 3114 4483 4422) (1168 572 4163 3551 883 1421 5088 1787 128 109) (3411 2937 2922 2921 2593 2556)
7	37	52.7423	6569.55	(4152 4903 1413 2842 802 1899 739 2381 3332 3634) (4748) (4948) (4698 4799 2635 616 2136 1214 4272 372 4157 3114 4483 4422) (3148) (1168 572 4163 3551 883 1421 5088 1787 128 109) (3411 2937 2922 2921 2593 2556)
8	9	58.4041	6640.23	(1168 572 4163 3551 883 1421 5088 1787 128 109) (3022) (3152 2262) (4698 4799 2635 616 2136 1214 4272 372 4157 3114 4483 4422) (4748) (4152 4903 1413 2842 802 1899 739 2381 3332 3634) (3411 2937 2922 2921 2593 2556) (444)
9	2	59.3297	6712.63	(4748) (4698 4799 2635 616 2136 1214 4272 372 4157 3114 4483 4422) 1186 (1168 572 4163 3551 883 1421 5088 1787 128 109) (1802) (1575) (4152 4903 1413 2842 802 1899 739 2381 3332 3634) (3411 2937 2922 2921 2593 2556) (444)

drosophila0075 — unrooted

k	i	w	s	vertices
4	125	77.6151	259.74	(3411 2937 2922 2921 2593 2556) (4748) (2584 2564 933 2450 464 103 3669 4992 4326 4698 3945 4272 372 1356 1852 1822 1980 3912 1340 4175 4435 4976 2635 2341 726 260 813 2136 2266 3804 3114 4517 1582 616 1214 4799 4483 4422 4157) (4152 2985 2970 4903 1413 2842 802 1899 739 2381 3332 3634)
5	341	80.5567	3051.17	(444) (4748) (2584 2564 933 2450 464 103 3669 4992 4326 4698 3945 4272 372 1356 1852 1822 1980 3912 1340 4175 4435 4976 2635 2341 726 260 813 2136 2266 3804 3114 4517 1582 616 1214 4799 4483 4422 4157) (4152 2985 2970 4903 1413 2842 802 1899 739 2381 3332 3634) (3411 2937 2922 2921 2593 2556)
6	156	92.6747	5922.71	(2584 2564 933 2450 464 103 3669 4992 4326 4698 3945 4272 372 1356 1852 1822 1980 3912 1340 4175 4435 4976 2635 2341 726 260 813 2136 2266 3804 3114 4517 1582 616 1214 4799 4483 4422 4157) (268 2578) (1168 572 4163 3568 3551 883 1160 1421 5088 1787 128 109) (4748) (4152 2985 2970 4903 1413 2842 802 1899 739 2381 3332 3634) (3411 2937 2922 2921 2593 2556)
7	43	94.875	6870.8	(444) (4748) (2584 2564 933 2450 464 103 3669 4992 4326 4698 3945 4272 372 1356 1852 1822 1980 3912 1340 4175 4435 4976 2635 2341 726 260 813 2136 2266 3804 3114 4517 1582 616 1214 4799 4483 4422 4157) (1117) (1168 572 4163 3568 3551 883 1160 1421 5088 1787 128 109) (4152 2985 2970 4903 1413 2842 802 1899 739 2381 3332 3634) (3411 2937 2922 2921 2593 2556)
8	10	96.4936	6995.06	(3411 2937 2922 2921 2593 2556) (4748) (2584 2564 933 2450 464 103 3669 4992 4326 4698 3945 4272 372 1356 1852 1822 1980 3912 1340 4175 4435 4976 2635 2341 726 260 813 2136 2266 3804 3114 4517 1582 616 1214 4799 4483 4422 4157) (4647) (3323) (1168 572 4163 3568 3551 883 1160 1421 5088 1787 128 109) (4152 2985 2970 4903 1413 2842 802 1899 739 2381 3332 3634)
9	2	94.6256	6578.64	(3829) (3410) (2584 2564 933 2450 464 103 3669 4992 4326 4698 3945 4272 372 1356 1852 1822 1980 3912 1340 4175 4435 4976 2635 2341 726 260 813 2136 2266 3804 3114 4517 1582 616 1214 4799 4483 4422 4157) (1168 572 4163 3568 3551 883 1160 1421 5088 1787 128 109) (2180) (1609) (3411 2937 2922 2921 2593 2556) (1575) (4152 2985 2970 4903 1413 2842 802 1899 739 2381 3332 3634)

drosophila0075 — blockcut

k	i	w	s	vertices
4	125	77.6171	264.463	(4748) (2584 2564 933 2450 464 103 3669 4992 4326 4698 3945 4272 372 1356 1852 1822 1980 3912 1340 4175 4435 4976 2635 2341 726 260 813 2136 2266 3804 3114 4517 1582 616 1214 4799 4483 4422 4157) (4152 2985 2970 4903 1413 2842 802 1899 739 2381 3332 3634) (3411 2937 2922 2921 2593 2556)
5	341	80.5567	3070.41	(2584 2564 933 2450 464 103 3669 4992 4326 4698 3945 4272 372 1356 1852 1822 1980 3912 1340 4175 4435 4976 2635 2341 726 260 813 2136 2266 3804 3114 4517 1582 616 1214 4799 4483 4422 4157) (4748) (4152 2985 2970 4903 1413 2842 802 1899 739 2381 3332 3634) (3411 2937 2922 2921 2593 2556) (444)
6	150	92.6747	5738.87	(3411 2937 2922 2921 2593 2556) (4748) (2584 2564 933 2450 464 103 3669 4992 4326 4698 3945 4272 372 1356 1852 1822 1980 3912 1340 4175 4435 4976 2635 2341 726 260 813 2136 2266 3804 3114 4517 1582 616 1214 4799 4483 4422 4157) (268 2578) (1168 572 4163 3568 3551 883 1160 1421 5088 1787 128 109) (4152 2985 2970 4903 1413 2842 802 1899 739 2381 3332 3634)
7	42	95.0692	7200	(1168 572 4163 3568 3551 883 1160 1421 5088 1787 128 109) (3093) (2584 2564 933 2450 464 103 3669 4992 4326 4698 3945 4272 372 1356 1852 1822 1980 3912 1340 4175 4435 4976 2635 2341 726 260 813 2136 2266 3804 3114 4517 1582 616 1214 4799 4483 4422 4157) (4748) (4152 2985 2970 4903 1413 2842 802 1899 739 2381 3332 3634) (3411 2937 2922 2921 2593 2556) (444)
8	10	102.001	7200	(5018) (4873 3067 1209 4500 440 557) (4748) (3094) (2584 2564 933 2450 464 103 3669 4992 4326 4698 3945 4272 372 1356 1852 1822 1980 3912 1340 4175 4435 4976 2635 2341 726 260 813 2136 2266 3804 3114 4517 1582 616 1214 4799 4483 4422 4157) (1168 572 4163 3568 3551 883 1160 1421 5088 1787 128 109) (3411 2937 2922 2921 2593 2556) (4152 2985 2970 4903 1413 2842 802 1899 739 2381 3332 3634)
9	2	103.387	7200	(4748) (3411 2937 2922 2921 2593 2556) (2584 2564 933 2450 464 103 3669 4992 4326 4698 3945 4272 372 1356 1852 1822 1980 3912 1340 4175 4435 4976 2635 2341 726 260 813 2136 2266 3804 3114 4517 1582 616 1214 4799 4483 4422 4157) (1101) (130) (4873 3067 1209 4500 440 557) (4594) (1168 572 4163 3568 3551 883 1160 1421 5088 1787 128 109) (4152 2985 2970 4903 1413 2842 802 1899 739 2381 3332 3634)

HCMV — unrooted

k	i	w	s	vertices
4	125	5.49602	26.2604	(1632) (1405) (2733) (972)
5	341	5.70381	305.039	(972) (1405) (2733) (1632) (3176)
6	928	5.70381	3471.11	(972) (1405) (2733) (1632) (3176)
7	423	6.0226	6579.98	(972) (1232) (2733) (1632) (695) (1954) (1270)
8	104	6.51791	7024.49	(920) (1232) (2733) (2243) (1270) (1954) (1632) (972)
9	22	5.70381	6968.51	(972) (1405) (2733) (1632) (3176)

HCMV — blockcut

k	i	w	s	vertices
4	125	5.49602	26.2442	(972) (1405) (2733) (1632)
5	341	5.70381	303.797	(2733) (1405) (1632) (3176) (972)
6	928	5.70381	3437	(3176) (1405) (2733) (1632) (972)
7	423	6.27261	6477.19	(2733) (1232) (2243) (1270) (1954) (1632) (972)
8	104	6.01626	6997.82	(1954) (695) (2162 3125 2746) (1232) (2733) (1632) (920) (972)
9	22	5.70381	6909.38	(3176) (1405) (2733) (1632) (972)

lymphoma — unrooted

k	i	w	s	vertices
4	125	30.0599	23.0169	(380 59 58 57) (776) (814 63 808 4 696 528) (615 62)
5	341	31.5504	251.438	(380 59 58 57) (782) (650) (814 63 808 4 696 528) (1155 429 28 927)
6	928	35.2917	2864.39	(380 59 58 57) (1360) (1155 429 28 927) (776) (814 63 808 4 696 528) (615 62)
7	553	39.8447	7100.88	(380 59 58 57) (512) (1155 429 28 927) (615 62) (681) (931) (814 63 808 4 696 528)
8	126	39.9158	7024.3	(380 59 58 57) (512) (896 556) (1155 429 28 927) (615 62) (681) (931) (814 63 808 4 696 528)
9	27	40.1543	7048.33	(380 59 58 57) (512) (551) (542) (473) (1155 429 28 927) (615 62) (793) (814 63 808 4 696 528)

lymphoma — blockcut

k	i	w	s	vertices
4	125	30.0599	21.1206	(776) (814 63 808 4 696 528) (380 59 58 57) (615 62)
5	341	31.5504	250.087	(814 63 808 4 696 528) (782) (650) (1155 429 28 927) (380 59 58 57)
6	928	35.2917	2833.1	(1360) (380 59 58 57) (776) (814 63 808 4 696 528) (615 62) (1155 429 28 927)
7	514	39.8447	6644	(1155 429 28 927) (512) (380 59 58 57) (681) (931) (814 63 808 4 696 528) (615 62)
8	126	40.5045	7037	(931) (151) (1155 429 28 927) (512) (1769) (380 59 58 57) (615 62) (814 63 808 4 696 528)
9	27	40.1543	7052.31	(380 59 58 57) (512) (551) (542) (473) (1155 429 28 927) (615 62) (793) (814 63 808 4 696 528)

metabol_expr_mice_2 — unrooted

k	i	w	s	vertices
4	125	148.751	3.87601	(214 181 178 177) (690) (691)
5	341	162.361	41.0248	(214 181 178 177) (690) (691) (216) (831 2946 60)
6	928	162.361	460.302	(214 181 178 177) (690) (691) (216) (831 2946 60)
7	2525	190.1	5089.36	(2835 825 2834) (661) (74) (75) (71) (216) (214 181 178 177)
8	720	214.472	6242.05	(71) (831 2946 60) (216) (214 181 178 177) (75) (74) (661) (2835 825 2834)
9	180	214.472	7200	(71) (831 2946 60) (216) (214 181 178 177) (75) (74) (661) (2835 825 2834)

metabol_expr_mice_2 — blockcut

k	i	w	s	vertices
4	125	148.751	3.87652	(214 181 178 177) (690) (691)
5	341	162.361	41.048	(214 181 178 177) (690) (691) (216) (831 2946 60)
6	928	162.361	456.891	(214 181 178 177) (690) (691) (216) (831 2946 60)
7	2525	190.1	5089.36	(2835 825 2834) (661) (74) (75) (71) (216) (214 181 178 177)
8	730	214.472	6423.5	(71) (831 2946 60) (216) (214 181 178 177) (75) (74) (661) (2835 825 2834)
9	185	214.472	7200	(71) (831 2946 60) (216) (214 181 178 177) (75) (74) (661) (2835 825 2834)

metabol_expr_mice_3 — unrooted

k	i	w	s	vertices
4	125	252.293	2.6583	(195) (194 582 164 165 161 160) (67 66 1 106 2285 58 68 570 107 105)
5	341	356.923	28.6636	(404 2174 686 2173 71 102) (72) (67 66 1 106 2285 58 68 570 107 105) (195) (194 582 164 165 161 160)
6	928	356.932	321.122	(67 66 1 106 2285 58 68 570 107 105) (72) (404 2174 686 2173 71 102) (195) (194 582 164 165 161 160)
7	2525	373.095	3561.29	(194 582 164 165 161 160) (195) (67 66 1 106 2285 58 68 570 107 105) (45) (2183) (72) (404 2174 686 2173 71 102)
8	1107	373.095	6709.29	(67 66 1 106 2285 58 68 570 107 105) (45) (2183) (72) (404 2174 686 2173 71 102) (195) (194 582 164 165 161 160)
9	251	373.095	7069.88	(67 66 1 106 2285 58 68 570 107 105) (45) (2183) (72) (404 2174 686 2173 71 102) (195) (194 582 164 165 161 160)

metabol_expr_mice_3 — blockcut

k	i	w	s	vertices
4	125	252.293	2.67325	(195) (194 582 164 165 161 160) (67 66 1 106 2285 58 68 570 107 105)
5	341	356.923	28.7284	(195) (194 582 164 165 161 160) (67 66 1 106 2285 58 68 570 107 105)
6	928	356.923	326.719	(404 2174 686 2173 71 102) (72) (67 66 1 106 2285 58 68 570 107 105) (195) (194 582 164 165 161 160)
7	2525	373.095	6712.32	(195) (194 582 164 165 161 160) (67 66 1 106 2285 58 68 570 107 105) (45) (2183) (72) (404 2174 686 2173 71 102)
8	1055	373.095	7051.72	(195) (194 582 164 165 161 160) (67 66 1 106 2285 58 68 570 107 105) (45) (2183) (72) (404 2174 686 2173 71 102)
9	218	373.095	6855.87	(67 66 1 106 2285 58 68 570 107 105) (45) (2183) (72) (404 2174 686 2173 71 102) (195) (194 582 164 165 161 160)

Bibliography

[1] Falk Hüffner, Christian Komusiewicz, Rolf Niedermeier, and Sebastian Wernicke. *Parameterized Algorithmics for Finding Exact Solutions of NP-Hard Biological Problems*, pages 363–402. Springer New York, New York, NY, 2017. ISBN 978-1-4939-6613-4. doi: 10.1007/978-1-4939-6613-4_20. URL `https://doi.org/10.1007/978-1-4939-6613-4_20`.

[2] Mohammed El-Kebir and Gunnar W Klau. Solving the maximum-weight connected subgraph problem to optimality. *CoRR*, abs/1409.5308, 2014. URL `http://arxiv.org/abs/1409.5308`.

[3] Bistra Dilkina and Carla P Gomes. Solving connected subgraph problems in wildlife conservation. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 102–116. Springer, 2010.

[4] Marcus T Dittrich, Gunnar W Klau, Andreas Rosenwald, Thomas Dandekar, and Tobias Müller. Identifying functional modules in protein–protein interaction networks: an integrated exact approach. *Bioinformatics*, 24(13):i223–i231, 2008.

[5] Trey Ideker, Owen Ozier, Benno Schwikowski, and Andrew F Siegel. Discovering regulatory and signalling circuits in molecular interaction networks. *Bioinformatics*, 18(suppl_1):S233–S240, 2002.

[6] Şerban Nacu, Rebecca Critchley-Thorne, Peter Lee, and Susan Holmes. Gene expression network analysis and applications to immunology. *Bioinformatics*, 23(7): 850–858, 2007.

[7] Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995.

[8] Phuong Dao, Kendric Wang, Colin Collins, Martin Ester, Anna Lapuk, and S Cenk Sahinalp. Optimally discriminative subnetwork markers predict response to chemotherapy. *Bioinformatics*, 27(13):i205–i213, 2011.

[9] Federico Altieri, Tommy V Hansen, and Fabio Vandin. Nomas: A computational approach to find mutated subnetworks associated with survival in genome-wide cancer studies. *Frontiers in genetics*, 10, 2019.

[10] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[11] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447.

[12] John Hopcroft and Robert Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, June 1973. ISSN 0001-0782. doi: 10.1145/362248.362272. URL `http://doi.acm.org/10.1145/362248.362272`.

[13] Cees W Duin and Ton Volgenant. Some generalizations of the Steiner problem in graphs. *Networks*, 17(3):353–364, 1987.

[14] Amos Tversky. Features of similarity. *Psychological review*, 84(4):327, 1977.