Efficient Implementation of Modular Graph Decomposition

Bachelor Thesis of

Luis Göppel

At the Department of Informatics and Mathematics
Chair of Theoretical Computer Science

Reviewer:	Prof. Dr. Ignaz Rutter
Advisor:	Miriam Münch

Time Period: 1st December 2023 – 29th February 2024

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

[illegible]

Passau, February 29, 2024

Abstract

An important operation to perform on a graph, is to locate sets of vertices with similar properties and to condense them into a single vertex. The process of modular decomposition is responsible for such operations, as it takes a set of vertices that share the same connections outside the set and contracts them into a module, that can be represented by a single vertex. Modular decomposition plays a crucial role in various significant problems within algorithmic graph theory. These include tasks such as transitive orientation, identifying specific graph classes, and solving certain combinatorial optimization problems. As a consequence, a lot of research has gone into the development of algorithms that solve this problem. However, despite considerable effort, a simple and efficient solution has remained elusive for a long time. In this work, we will analyse an algorithm that promises to be the first simple, linear-time approach to modular decompose a graph. We will provide an implementation of this algorithm in $C++$ that turns out to be among the first to compute the modular decomposition of any simple, undirected graph in an efficient time.

Deutsche Zusammenfassung

Ein wichtiger Prozess bei der Bearbeitung von Graphen besteht darin, Gruppen von Knoten mit ähnlichen Eigenschaften zu identifizieren, und diese zu einem einzigen Knoten zusammenzufassen. Der Prozess der modularen Zerlegung ist für solche Operationen verantwortlich. Er kombiniert Knoten mit gleichen Kanten zu Modulen, welche wiederum durch einzelne Knoten repräsentiert werden können. Die modulare Zerlegung spielt eine entscheidende Rolle in verschiedenen wichtigen Problemen der algorithmischen Graphentheorie. Zu diesen gehören Aufgaben wie die transitive Orientierung, die Identifizierung bestimmter Klassen von Graphen und das Lösen bestimmter kombinatorischer Optimierungsprobleme. Trotz erheblicher Bemühungen, Algorithmen für dieses Problem zu entwickeln, blieb eine einfache und effiziente Lösung lange Zeit schwer zu fassen. In dieser Arbeit werden wir einen Algorithmus analysieren, der verspricht, der erste einfache, linearzeitliche Ansatz zur modularen Zerlegung eines Graphen zu sein. Wir werden eine Implementierung dieses Algorithmus in $C++$ bereitstellen, die sich als eine der ersten herausstellt, um jeden einfachen, ungerichteten Graphen in effizienter Zeit modular zu zerlegen.

Contents

1. Introduction

Graphs, as mathematical representations of relationships, undergo various transformations to unveil their underlying structures. One such fundamental operation for a graph G involves selecting a vertex v and replacing it with another graph G', where the neighbors of v become universal to the vertices of G'. In contrast to that, modular decomposition takes a captivating inverse approach: rather than expanding the graph, it seeks to identify sets of vertices that share common neighbors outside the set, which are called *modules*. These modules act as cohesive units, allowing for the contraction of a complex network into a more manageable representation.

Essentially, a graph's modules create a *partitive family* [CHM81], a breakdown that helps analyzing the graph's complex relationships. This breakdown can be imagined as a tree, with *maximal strong modules* — distinct groups of vertices that cannot be extended and that don't overlap each other — standing out. The presence of such decomposition trees arises from the theorems regarding unique decomposition, as outlined in Cunningham and Edmonds [CE80]. We will operate under the assumption that the constructed decomposition trees are uniquely defined, allowing for isomorphism. Breaking down this modular decomposition is akin to solving a puzzle, where comprehending one part provides insight into the other. This process, efficiently capturing the graph's structure, becomes a crucial first step in various algorithmic applications.

Since Gallai [Gal67] noticed that modular decomposition is vital in the domain of comparability graphs, the stage for its role in algorithmic graph theory has been set. The close link between modular decomposition and efficient transitive orientation algorithms [MS00] shows its essentiality in deciphering complex relationships in graphs. Its versatility extends to identifying different graph families, like interval graphs [Möh85], permutation graphs [PLE71], and cographs [CPS85], showcasing its adaptability across various domains. Beyond its foundational role, modular decomposition proves powerful in solving many combinatorial optimization problems. Möhring and Radermacher [MR84] presented an excellent survey of its numerous applications which are both, theoretical and practical. Recent uses in graph drawing [PV07] and bioinformatics [SPH14] highlight its evolving importance, emphasizing modular decomposition's resilience and relevance in contemporary research.

The task of computing modular decomposition, akin to addressing planarity testing and interval graph recognition, has witnessed significant strides. The initial algorithm in the early 1970s achieved polynomial time at $O(n^4)$ [JSC72], gradually evolving through incremental improvements in subsequent work [HM79] [MS89]. A milestone was reached in 1994 with the introduction of the first linear-time algorithms by McConnell and Spinrad [MS94] and Cournier and Habib [CH94]. Despite their theoretical significance, these solutions were complex, prompting a search for simpler alternatives. More recent attempts (e.g. [MS00] or [DGM01]), build on the approach pioneered by Ehrenfeucht et al. [EGMS94] and refined by Dahlhaus [Dah95]. These contributions highlight the ongoing effort to balance efficiency and simplicity in tackling the modular decomposition problem. In 2008, Tedder, Corneil, Habib and Paul [TCHP08] proposed an algorithm, which is said to combine both simplicity and a linear running time. If this is indeed the case, remains to be seen in this work.

This thesis will introduce the reader to the principles of modular graph decomposition in Chapter 3, along with presenting several use-cases of it in larger detail. Chapter 4 is the main chapter of the thesis. Here, we will start by exploring different approaches to modular decomposition over time and then move on to an in-depth exploration of the algorithm by Tedder, Corneil, Habib and Paul. Currently, this algorithm has been developed on paper, yet we are not aware of any practical implementation that is publicly available. Therefore, the main contribution of this work is an implementation of the named algorithm in $C++$. This implementation will be described and evaluated in Chapter 5. We will ensure the algorithm's correctness on multiple input graphs of different structure and size and analyse the implementation's running time, which turns out to be within quadratic bounds. Finally, we will describe approaches to implement the algorithm within a linear time constraint in Chapter 6.

2. Preliminaries

This work centers around graphs, which will be referred to as $G = (V, E)$, with V denoting the set of G's vertices and E describing the set of G's edges. An edge $e \in E$ is composed of a pair of vertices $e = (u, v)$, where $u, v \in V$. The letter n will be used to denote the number of vertices in G, i.e. $n = |V|$, and the letter m will be used to denote the number of edges in G, i.e. $m = |E|$. A subgraph $G' = (V', E')$ of G is a graph with $V' \subseteq V$ and $E' = \{(u, v)|(u, v) \in E \text{ and } u \in V', v \in V'\}$.
If not mentioned otherwise, the graphs considered in this work will be simple and undirected. A *simple graph* is a graph that contains no loops and no doubled edges. This means that E does not contain an edge $e = \{u, u\}$ with $u \in V$, and that for every edge $e = \{u, v\}, u, v \in V, e \in E$, there cannot be another edge $e' = \{u', v'\}, u', v' \in V$ with $u = u'$ and $v = v'$ in E. The complement graph of G is the graph $G' = (V, E')$, where $(u, v) \in E'$, if $(u, v) \notin E$. Furthermore, this work will involve *connected components* and *complement-connected components.* A connected component is a maximal set V' of vertices, with $V' \subseteq V$, where any two vertices $v_1, v_2 \in V'$ in the subgraph induced by V' are connected by a path. In a complement-connected component, any two vertices $v_1, v_2 \in V'$ are connected by a path in the complement of the graph induced by V'. For simplicity, connected components will be referred to as *components* and complement-connected components will be referred to as *co-components.*
In a graph G, every vertex x has a neighborhood $N(x) = \{v \in V|(x, v) \text{ is and edge in } G\}$. This neighborhood will be denoted as $N(x)$ and refers to the set of vertices that contains all vertices that are connected to x.
A *module* is a set $V' \subseteq V$ of vertices, that are indistinguishable by any vertices outside the set. This means, that for any two vertices $v_1, v_2 \in V'$ and any vertex $u \notin V'$ either $(v_1, u) \in E$ and $(v_2, u) \in E$ or $(v_1, u) \notin E$ and $(v_2, u) \notin E$. Let M be a module of G. If $M \cap M' = \emptyset$ for every other module M' with $M \nsubseteq M'$ and $M' \nsubseteq M$, then M is called a *strong module.* If there is no vertex $v' \in V \setminus V'$ for which $M \cap \{v'\}$ is a module as well, M is called a *maximal module.*

3. The Modular Decomposition

Modular decomposition is an important concept in graph theory that involves decomposing a graph and thereby showing its underlying structure. It is mostly performed on simple, undirected graphs. This chapter will provide an introduction to the topic of modular decomposition by starting with an explanation of its principles and progressing towards modern day applications of it.

3.1 What is Modular Decomposition?

To be able to work with modular decomposition, or to implement any modular decomposition algorithm, one needs to understand the principle behind it first. That is what this chapter deals with.
As the name already tells, modular decomposition aims to decompose a graph into modules. Given a graph $G = (V, E)$, a module M is a set containing vertices of G, $M \subseteq V$, which are indistinguishable by all vertices outside of M: Let w be a vertex with $w \in V$, but $w \notin M$. Then w has to be either connected to all vertices in M or to none of them. Following this definition, a set v with $v \in V$ is a module. Such modules, that only consist of one vertex of G, are called *trivial*. Hence, it is legitimate to say that every graph can be decomposed into modules. However, the process of modular decomposition focuses specifically on a graph's *maximal strong modules*. *Strong modules* are the modules, that don't overlap each other. Let's consider an example to better understand these concepts. Imagine a complete Graph $G = (V, E)$ (i.e. a graph in which every vertex $v \in V$ is connected to all vertices $u \in V$, with $u \neq v$) with n vertices. In this graph, any two vertices can be seen as a module, as there is no outside vertex that can distinguish them. In fact, any m vertices, with $1 \leq m < n$ form a module as well. However, these modules overlap each other and are therefore not strong. Furthermore, one can see that all these modules could also easily be extended by adding at least one vertex while keeping their module properties. Thus, these modules are not *maximal* as well. In this graph, the only *maximal strong module* is the one containing all n vertices.

Keeping these definitions in mind, we can consider our first example for a modular decomposition. We do this by decomposing the graph, that is portrait in Figure 3.1 into its modules.

To start with, we need to identify the graph's maximal strong modules of minimal size. On first thought, the vertices a, b, c, d, e form a complete (sub)graph and should therefore

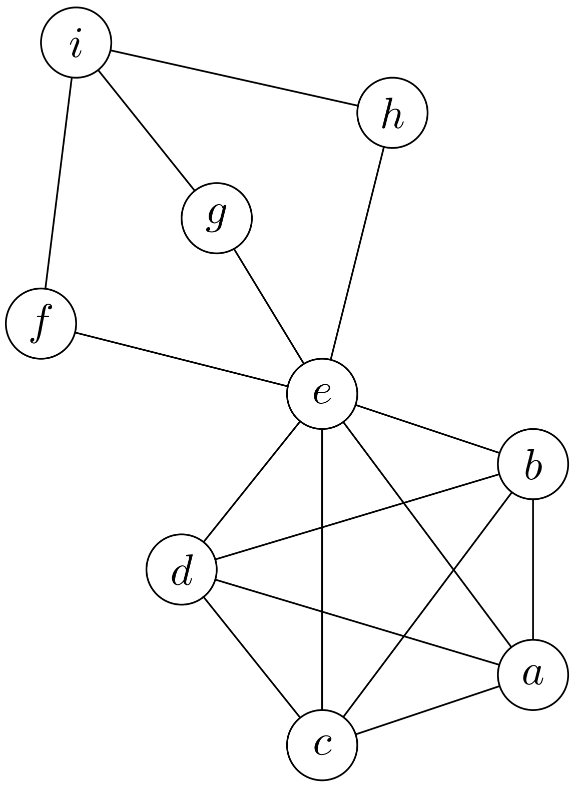

Figure 3.1: A simple, undirected graph

be able to form a strong module. However, the vertex e is connected to the three vertices f, g, h, while a, b, c, d are not. This means, that adding e to the current strong module implies adding f, g, h as well. In this case, vertex i could distinguish the module's vertices (as i is connected to f, g, h, but not to a, b, c, d, e), so i had to be included too. The whole graph is a strong module, but is not minimal. Therefore, e can not be in the same minimal strong module as a, b, c, d. Nonetheless, these vertices (a, b, c, d) are all connected to e and are all disconnected to every other vertex in the graph. They cannot be distinguished by any outside vertex. Removing a vertex from this module leads to it not longer being strong and as already discussed, this module cannot be expanded. Thus, the first strong module of the graph is found. We can replace all vertices in the module (a, b, c, d) with a vertex labeled $M1$, representing the module as a whole. The result can be seen in Figure 3.2.

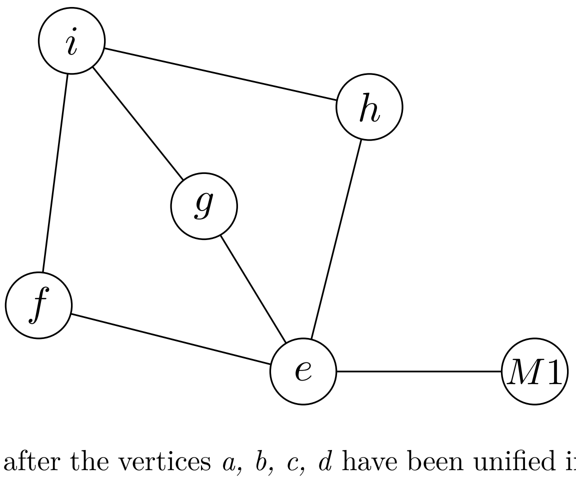

Figure 3.2: The graph after the vertices *a, b, c, d* have been unified into a new module, *M1*

In the same way as before, we can identify the graph's next maximal strong module. The vertices f, g, h are all connected to e and i and disconnected to any other of the graph's vertices. As a consequence, a new module, $M2$, containing these vertices, can be formed. The resulting graph is shown in Figure 3.3.

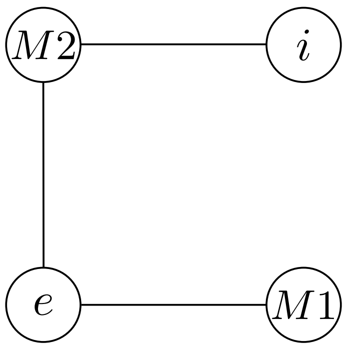

Figure 3.3: The graph with the vertices *a, b, c, d* as *M1* and the vertices *f, g, h* as *M2*

Now, it is easy to see that the graph cannot be decomposed any further, although it is possible to interpret the four remaining vertices as a module. Therefore, the process of modular decomposition is finished. Figure 3.3 can be seen as the "final (decomposed) form" of the graph. However, this representation has lost a lot of information compared to the original graph in Figure 3.1. Additionally, this was a very simple example. In a larger graph with more vertices, there might be several modules that are formed by using other modules, which could also contain modules themselves (and so on). Hence, it would be wise for a modular decomposition algorithm not to return the graph in its final decomposed form, but rather a representation of the graph that shows all its maximal strong modules while maintaining information about its original form. Such a representation is the *modular decomposition tree*. In a modular decomposition tree, all vertices of the original graph are leaf-nodes, while all inner nodes of the tree represent the graph's strong modules. These modules can be distinguished into different types:

1. **Parallel**: In a PARALLEL module, all vertices are disconnected from each other. The graph induced by those vertices is not connected, whereas its complement is. In simpler terms, the complement graph contains the same vertices as its origin graph, but has its edges swapped. Such a graph is said to be *complement-connected*. Regarding the previous example, one can see that the vertices f, g, h form a PARALLEL module.

2. **Series**: In a SERIES module, all vertices are connected to each other. In this case, the graph induced by the module's vertices is connected, but not complement-connected. Regarding the previous example, one can see that the vertices a, b, c, d form a SERIES module.

3. **Prime**: In a PRIME module, some of the module's vertices are connected, while others are not. If a module is neither SERIES nor PARALLEL, it is called PRIME. The graph induced by the modules vertices is connected and complement-connected. Regarding the previous example, the module containing all the vertices of the graph's final (decomposed) form (Figure 3.3) is PRIME. Therefore, the root of the graph's modular decomposition tree will be marked as 'PRIME' as well.

Keeping these definitions in mind, we can take a look at the graph's modular decomposition tree, displayed by Figure 3.4.

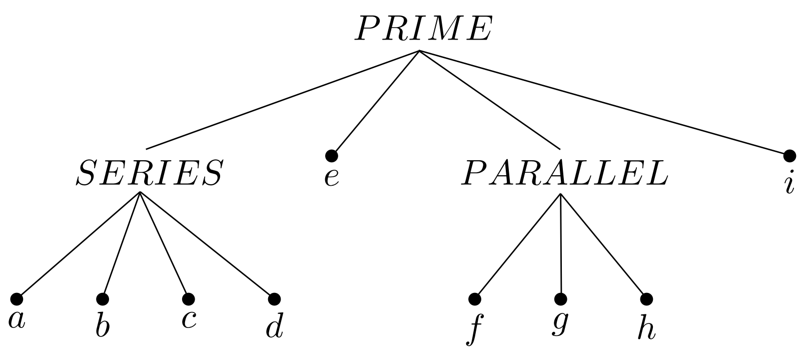

Figure 3.4: The modular decomposition tree of the graph shown in Figure 3.1

This representation of the graph contains the information about the graph's modules (the inner nodes and the root, marked with either 'PARALLEL', 'SERIES', or 'PRIME'), while keeping information about the graph's vertices (the leaf nodes). To be able to reconstruct the original graph from its modular decomposition tree, the sub-graph induced by its PRIME nodes has to be given as well. The sub-graph induced by SERIES or PARALLEL nodes is trivial, as its vertices are either all connected or all disconnected. Due to its benefits, the modular decomposition tree is used as representation for a graph's modular decomposition in almost every work centered around this topic. Thus, when literature refers to the modular decomposition of a graph, the modular decomposition tree is meant.

In literature, the process of modular decomposition has several names. Other names for modular decomposition include *substitution decomposition*, *disjunctive decomposition*, *X-join* or *ordinal sum*. Modular decomposition is not only good to visualize a graph's structure, it has multiple applications in and even out of graph theory. This will be the content of the next chapter.

3.2 Applications of Modular Decomposition

The process of modular decomposition manages to reduce the size of a graph significantly, while keeping all information about its vertices' connections. It shows, how multiple parts (vertices) of the graph share closer connections to each other, while maintaining a larger distance to other parts. As modular graph decomposition allows modules to contain modules themselves, it basically gives the opportunity to zoom into and out of a graph's structure, enhancing its simplicity on one side, and showing greater detail up to the graph's original vertices on the other.
Nowadays, graphs are used in many areas of science, technology, and business, providing valuable insights and facilitating data-driven decision-making. And wherever graphs are present, modular decomposition can help by providing a simplified view on their structure.

These use-cases include network analysis, where graphs are used to represent complex networks such as social networks, communication networks or biological networks. Here, modular decomposition allows researchers to identify cohesive groups within these networks, making community detection and network organization easier.

Modular decomposition techniques can also be applied in electronic circuit design. As such circuits can be represented as graphs, modular decomposition can be used to analyse the connectivity between the different components of the circuit. When optimizing a circuit's structure, this understanding is crucial and has the benefits of reducing an electronic circuit's area, minimizing the wire length and overall improving its performance.

In the same way, modular decomposition can also be used in the field of data mining and clustering, by identifying cohesive groups of clusters within large data-sets. Data points and their relationships can be represented as a graph, which allows modular decomposition methods to reveal underlying patterns and structures in the data. This knowledge can be used to optimize performance and lower costs on these tasks.

Furthermore, modular decomposition has several use-cases within algorithm design and graph theory itself. Many graph-algorithms rely on modular decomposition to exploit the modular structure of a graph and solve problems efficiently. We will take a look at some of these graph algorithms and their reliance on modular decomposition on the next few pages.

3.2.1 The maximum clique problem

In a graph $G = (V, E)$, a clique C is a subset of vertices that are all connected to each other. Hence, C is a complete sub-graph of G. Every graph has a maximum clique, which is determined by the amount of vertices it contains. In weighted graphs, a maximum clique can also be determined by the weights of the connections of its vertices. The maximum clique problem is NP-hard. This means, that with increasing size of input, the computation time increases exponentially for each exact solver, making it basically impossible to solve on large inputs within a reasonable time.

In her paper [Utk17], I. Utkina describes an algorithm, that works recursively as a depth-first search on the modular decomposition tree of a given graph. It considers each node and bases its calculation on its type. This approach helps solving the problem, as the calculation for PARALLEL or SERIES nodes is relatively simple. Furthermore, modular decomposition is used to break the original graph into smaller parts, making it easier to solve, especially on graphs without PRIME nodes.

The maximum clique problem still remains NP-hard, but this approach can be used to get good solutions in a reasonable amount of time.

3.2.2 Transitive Orientation and Permutation Graphs

A directed graph is transitive if, whenever there is a directed edge from vertex A to vertex B and a directed edge from vertex B to vertex C, there is also a directed edge from A to C. This condition has to hold for every triple of vertices $\{A, B, C\}$ in the graph. Given an undirected graph, finding a transitive orientation is similar to assigning directions to its edges, in a way that the resulting directed graph is transitive. A graph that has a transitive orientation is called *comparability graph*. Transitive orientations have several applications and are used in various areas of computer science and mathematics, including comparing rankings, database management or network routing.

Finding a transitive orientation for a graph and finding its modular decomposition are in some sense dual problems. Most of the approaches for transitive orientation compute the modular decomposition tree for the given graph first, and try to find a transitive orientation based on this tree. On the other hand, the modular decomposition of a graph can be constructed if the transitive orientations of the graphs' edges are known. Some algorithms even try to solve both problems, the modular decomposition and transitive orientation of a given graph simultaneously.

As this chapter is about the use of modular decomposition, only a brief explanation on how to get a graph's transitive orientation based on its modular decomposition will be provided. Given the modular decomposition tree of a graph, a possible approach uses recursion to start at the tree's leafs and to progress towards the root. At each step, the orientations that were obtained from the child nodes have to be combined, according to the type of the module. When dealing with a PARALLEL module, the orientations of its children are independent from another, as there are no connections between them. Here, each child's orientation can be considered separately. In a SERIES module, all nodes are connected, meaning that the elements in one module are comparable to those in the next. After directions in each node have been assigned, the last vertex of one module can be connected to the first vertex of the next module in series. Given a PRIME module, a standard algorithm for finding a transitive orientation within it can be applied. When the root is reached, the process is finished and the orientation that is obtained represents the transitive orientation of the entire graph [MS00].

One other problem in graph theory, that is closely related to finding a transitive orientation for a given graph, is determining whether a given graph is a permutation graph. To understand, what a permutation graph is, let's consider a set of n numbers labeled 1 to n and a permutation P of these n numbers. Imagine two parallel lines, where on one the numbers $1, ..., n$ are spread with equal distances in their correct order, and on the other the numbers are spread in the order of the given permutation P. Now, for each number a line is drawn, connecting its appearances on both of the parallel lines. This is known as a *permutation diagram.* In a corresponding graph, we have n vertices labeled $1, ..., n$. Two of these vertices are connected, if and only if the lines of their numbers cross in the previously described permutation diagram. This graph is a permutation graph, based on P. Hence, a graph with n vertices is a permutation graph, if and only if there is a permutation of the numbers 1 to n, so that the crossing lines in the resulting permutation diagram correspond exactly with the graph's connections.
If a graph is known to be a permutation graph, many otherwise NP-complete problems can be solved efficiently on it. Furthermore, permutation graphs are helpful in modeling and solving various problems due to their special structural properties. These problems, for instance, involve finding intersection-avoiding layouts for connection boards or figuring out the best schedules for reallocating memory space in a computer [PLE71].

Transitive orientation - and therefore modular decomposition - makes determining whether or not a given graph is a permutation graph easy, as a well-known theorem states that a graph G is a permutation graph, if and only if both G and its complement graph G_c have a transitive orientation. Hence, modular decomposition can help to solve this problem as well, contributing to the numerous benefits associated with it.

3.2.3 Graph Drawing

Many applications need to draw or visualize graphs. Randomly placing all vertices on a 2D-plane and connecting them respectively would (at least in most cases) lead to many edges crossing other edges and vertices, making the graph somewhat unreadable and hard to process. Therefore, algorithms that provide a clear and readable layout of a graph and its complex structures are needed. Although there are multiple approaches to do so, a particularly promising one focuses on the use of modular decomposition to highlight the global structure of a graph and reveal regular structures within it. Starting from bottom to top, the modules provided by the modular decomposition tree are drawn separately, by using different techniques for each module type. Combined with already existing methods like grid placement or circular drawing, this algorithm manages to achieve aesthetically pleasing results which emphasize the structure of the graph. Furthermore, unlike in most other graph drawing algorithms, the graph can be visualized in various levels of abstraction [PV07].

3.2.4 Biology and Bioinformatics

This last example for the use of modular decomposition leaves the field of regular mathematics and computer science, and steps into the domain of biology. Here, modular decomposition can be used for identifying protein complexes and the way they share components, which is an essential step in describing biology on a molecular basis. In a bodies' cells, proteins combine into tiny complexes that execute essential tasks like replication, transcription or protein transport. Such macromolecular complexes are built from a network of protein-protein interactions, which is typically represented by a graph with proteins as vertices and their interactions as edges. Modular decomposition helps to logically organize these interaction networks, creating paths for targeted therapies, biomarker discovery, and a more profound understanding of life at the molecular level. This knowledge not only drives advancements in medical research but also holds the potential to improve disease treatment strategies and personalized and precision medicine, based on individual genetic and molecular profiles [GKBC04].

The previous paragraphs showed multiple examples on how the use of modular decomposition can help to solve problems in and outside of graph theory. And even though a lot of use-cases were listed, there are still several that remain unmentioned. However, the previous sections should have shown, that the advantages of modular decomposition are worth the research. The next big question is, how modular decomposition can be implemented. This will be discussed in the next chapter.

4. Computing the Modular Decomposition

The previous chapter showed the basics of modular decomposition and emphasized its value on several problems and tasks. Consequently, computing the modular decomposition for a given graph as efficiently as possible is an important problem. This chapter will explore different approaches and concepts for modular decomposition, as well as concrete algorithms. Starting with an overview of different approaches that have been developed over time, Section 4.1 will give ideas on how modular decomposition can look like. In the main part of this chapter, Section 4.2 will deep-dive into one specific algorithm and explore its concepts in detail. An example input and its modular decomposition process will be provided as well.

4.1 Modular Decomposition Algorithms

Starting in the early 1970s, many algorithms with different approaches and various time-complexities have been presented. This section aims to show how such algorithms have developed over the decades, contributing to the goal of a simple algorithm that solves the modular decomposition problem in linear time. This will be done by discussing four representative approaches, with different levels of difficulties and time-complexities.

4.1.1 The first Algorithm for Modular Decomposition

The first algorithm for modular decomposition was developed in 1972, by L.O. James, R.G. Stanton and D.D. Cowan [JSC72]. It has a time complexity of $O(n^4)$. The authors state that the algorithm converts a "(Michigan) graph to a graph structure form". Considering the input, the algorithm deals with simple, undirected graphs, as most modular decomposition algorithms do. On the side of the output however, the paper fails to mention a modular decomposition tree or a similar tree-like structure, as this concept was likely not yet developed at the time. The paper and its given pseudo-code leave the concrete form of the output unclear and up to the algorithm's implementations. In general, the presented algorithm consists of four sub-algorithms.

The first sub-algorithm, which the paper calls "Algorithm 2.2" calculates the smallest *condensible* subgraph of a graph G, that contains the vertices of a vector v. A subgraph is called condensible, if and only if no two vertices within it can be distinguished by any vertex outside of it. This definition corresponds exactly to our definition of a module from

Chapter 3. Finding this subgraph is achieved by iteratively expanding v, including vertices whose neighborhood intersection with the (outside) neighbourhood of updating vertices in v is not empty. This process takes a linear amount of steps.

The purpose of the second sub-algorithm (in the paper: "Algorithm 3.5") is to find the set of condensible cliques and independent sets. In cliques, every vertex is connected to every other vertex, hence "condensible clique" refers to a SERIES module. In a similar way, "condensible independent set" refers to a PARALLEL module, as in an independent set no two vertices are connected. Finding these condensible cliques and independent sets however, requires a cubic time, as the sub-algorithm iterates over every pair $< i, j >$ of vertices in the graph and checks if the $i - th$ and $j - th$ lines in the graph's adjacency-matrix are equal. Based on that knowledge, it can add j to the clique (or independent set) of i or create a new clique (or independent set) with those vertices.

Another sub-algorithm ("Algorithm 3.6") determines the set of minimal condensible subgraphs in G. It starts by calculating the minimal condensible subgraph (minimal module) for each edge of the graph, using Algorithm 2.2. Then, it associates a minimal condensible subgraph with every vertex, based on the condensible subgraphs of its incident edges. By checking every vertex v again, it removes vertices from the condensible subgraphs of v's neighbors or deletes the minimal condensible subgraph induced by v from consideration. This sub-algorithm has a quadratic time-complexity.

The fourth and final sub-algorithm ("Algorithm 3.4") is responsible for calculating the final result. While the graph contains condensible subgraphs, it uses Algorithm 3.5 to find the current condensible cliques and independent sets. These can be condensed, forming either a PARALLEL or a SERIES module. Algorithm 3.6 is used to check for and form PRIME modules. As long as a new module M can be formed (or a subgraph can be condensed), the algorithm tries to form modules containing M. If neither Algorithm 3.5 nor Algorithm 3.6 can form a new module, Algorithm 3.4 is finished. Figure 4.1 shows the dependencies of these algorithms and the structure of the process in a visual way. It is possible to create a modular decomposition tree with this algorithm or to just condense the graph as seen in the first example of Chapter 3. The algorithm does not force either.
Figure 4.1 shows the process in a visual way.

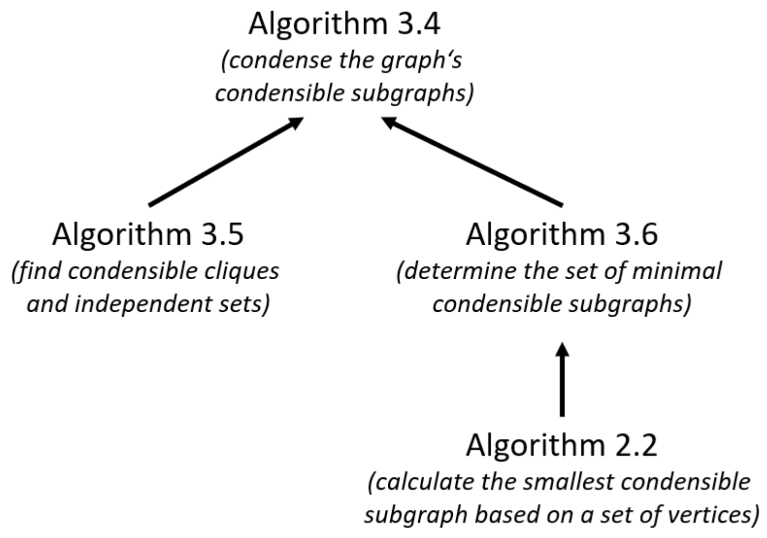
Algorithm 3.4
(condense the graph's condensible subgraphs)
Algorithm 3.5
(find condensible cliques and independent sets)
Algorithm 3.6
(determine the set of minimal condensible subgraphs)
Algorithm 2.2
(calculate the smallest condensible subgraph based on a set of vertices)

Figure 4.1: The structure of the first algorithm for modular decomposition, with a time-complexity of $O(n^4)$

This algorithm marks the start for all following modular decomposition algorithms. Compared to other algorithms, it is relatively simple and straight-forward. However, this simplicity results in an undesirable time-complexity of $O(n^4)$, making the algorithm too slow to get results for larger graphs in a reasonable amount of time.

4.1.2 Incremental Modular Decomposition

Incremental modular decomposition describes a specific way on how to create a modular decomposition tree. Algorithms that are based on incremental modular decomposition take an arbitrary vertex v from the input graph and create a simplistic modular decomposition tree containing only v. Then, all other vertices are added to the tree, one after another. This section will explore incremental modular decomposition by analysing the algorithm of J.H. Muller and J.P. Spinrad from 1989 [MS89].

Over 15 years have passed since James, Stanton and Cowan constructed the first polynomial time algorithm for modular decomposition. In this time, multiple improved algorithms have been developed, with their time-complexity lying in $O(n^3)$. The algorithm of Muller and Spinrad reduces the time required to find a modular decomposition to $O(n^2)$. Using incremental modular decomposition, n vertices have to be inserted into the tree. Thus, each vertex must be inserted within a time boundary of $O(n)$.
The main algorithm takes three parameters as input: An undirected graph, that is given by its adjacency matrix (used for fast look-ups of connections between two vertices), a representation of an already existing modular decomposition of a graph $G = (V, E)$, and a vertex v that is not yet in V, along with its edges E_v to V. The algorithm returns a modular decomposition of a graph $G' = (V \cup v, E \cup E_v)$. Another procedure is responsible for giving this algorithm all vertices of an input graph - one after another - to process.
As the algorithm for adding a vertex to the current modular decomposition is quite complicated, we will not look at it in detail and rather discuss general aspects of it.
One reason for the algorithm's complexity lies in its representation of the modular decomposition tree. Here SERIES and PARALLEL modules are represented using *N-ary trees*, which are similar to the representation that was established in Chapter 3. PRIME modules (which are referred to as *Neighborhood modules* in the paper) however, are harder to represent, as they contain connected and disconnected pairs of vertices. As the knowledge of a PRIME modules structure is essential for the algorithm, the *N-representation* is introduced to depict PRIME modules in the modular decomposition tree. In this representation, each node is labeled with a vertex's value and has up to two children. Vertices, that are connected to the node's value are placed in the nodes left- and disconnected vertices are placed in its right sub-tree. To achieve the desired time-bound, additional edges are added as "shortcuts" to this representation, changing it from a tree to a directed, acyclic graph with a maximum outdegree equal to two. An example for such a representation is shown in Figure 4.2

Whenever a vertex z gets added to the modular decomposition, it "travels" from the current tree's root to its module, taking the left or right subtree based on its connection to the inner node's vertex label. Adding z to its module while keeping the tree structure and N-representation intact is quite challenging and takes different approaches for different module types. Additionally, already existing modules in the structure have to be split, re-grouped or re-formed, as z can be connected to some of their members and disconnected from others, making their components distinguishable from the outside. Here, different approaches for different modules types have to be considered as well.

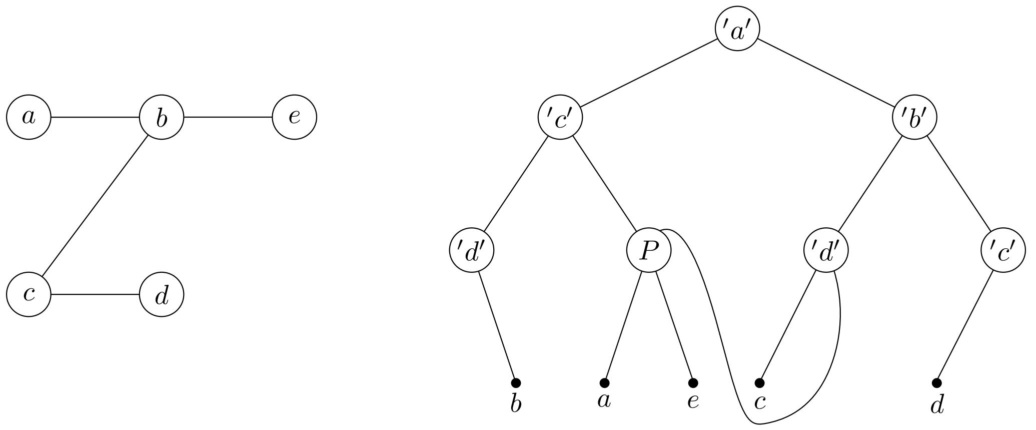

Figure 4.2: A simplistic PRIME module along with one of its N-representations. This representation manages to capture the inner structure of a PRIME module, but can get quite complicated with a growing amount of vertices.

Once, the last vertex has been added, the algorithm manages to calculate the modular decomposition in a quadratic amount of time. Keeping in mind that the first implementation only managed to find the modular decomposition in $O(n^4)$ and following implementations still had a cubic time-complexity, this achievement is quite considerable. However, having larger graph's as input, this algorithm still struggles to find the solution in a reasonable amount of time.

4.1.3 First linear algorithm for Modular Decomposition

In 1994, the first linear-time algorithms for modular decomposition were released. A. Cournier and M. Habib [CH94], as well as R.M. McConnel and J. P. Spinrad [MS94] developed such algorithms independently. This section will focus on the *Decompose2 algorithm* that was presented by Cournier and Habib in their paper "A New Linear Algorithm for Modular Decomposition" [CH94].
The authors see the ingenuity of their algorithm in the combination of two modular decomposition paradigms. At the time of the paper's release, every efficient Modular Decomposition (MD) algorithm could be categorized into one of two classes. The first class comprises incremental algorithms, as discussed in Section 4.1.2, which iteratively build the decomposition tree by adding vertices or edges. The second class consists of algorithms that incrementally expand the size of a prime subgraph found within the original input graph, emphasizing the identification and enlargement of these fundamental structures. The *Decompose2 algorithm* combines elements of both classes by maintaining a collection of prime subgraphs and a collection of so called *local cotrees decomposition.*

A *cograph* is a graph, that has only SERIES and PARALLEL nodes in its modular decomposition tree. A *cotree* is the MD tree of such a graph. A linear-time algorithm that is able to recognize cographs and compute their modular decomposition tree is already well-established. Constructed by Corneil, Perl and Stewart, it is often referred to as the *CPS-algorithm. Decompose2* uses the CPS-algorithm to check if its input graph is a cograph and compute the modular decomposition tree in the positive case. The main challenge of the algorithm lies in the incorporation of PRIME modules.
Therefore, the paper introduces the $P4$-graph, which is the smallest non-trivial prime graph. *Prime graphs* are the connected components of the original graph that cannot be further decomposed. P_4 is a simple path of length four, displayed by Figure 4.3.

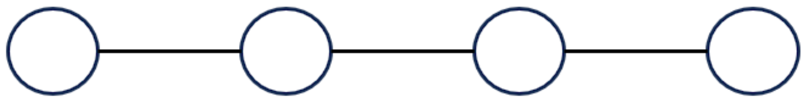

Figure 4.3: The P_4 graph, which is the first non-trivial prime graph

Every undirected prime graph contains four vertices a, b, c, d, that form a P_4. Using this knowledge, the *Decompose2 algorithm* computes PRIME modules by starting with a P_4 and incrementally adding vertices. Each added vertex v requires one recursive computation and complex algorithms for updating the existing modules based on v's connections. One central approach is the usage of cotrees to find P_4's in a graph and to verify module attributes. The algorithm groups vertices into classes based on their properties, and each class has its unique cotree. Managing these cotrees, as its classes are splitted, is a complicated procedure and adds to the overall complexity of the algorithm.

This algorithm, along with the linear-time approach from McConnel and Spinrad is complex to a degree, that it can be primarily seen as a theoretical contribution rather than an actual algorithm. However, it shows that calculating the modular decomposition in linear time is possible. Many subsequent algorithms however, have fallen short, either by not achieving linear-time or by relying on advanced data structure techniques to do so.

4.1.4 Sequential Modular Decomposition

This section describes one subsequent algorithm, as mentioned at the end of Section 4.1.3 A sequential algorithm for modular decomposition processes the elements of an input graph in a specific order, one after another, to organize the modular structure of the graph. The order in which the vertices and edges are considered is an essential aspect of such algorithms and the sequential processing ensures a systematic exploration of the graph's connectivity.

This section's algorithm was developed by E. Dahlhaus, J. Gustedt and R. M. McConnell and released in their paper "Efficient and Practical Algorithms for Sequential Modular Decomposition" in the year 2001 [DGM01]. Based on an already known strategy by Ehrenfeucht et al., it recursively generates the modular decompositions for two induced subgraphs, which are then combined to produce the modular decomposition of the entire graph. The general structure of the algorithm can be seen in Algorithm 4.1. In this algorithm, $N(v_0)$ represents all vertices that are connected to v_0, whereas $\overline{N}(v_0)$ contains all vertices that are disconnected to v_0. $G|N(v_0)$ and $G|\overline{N}(v_0)$ refer to the graphs induced by these vertices. $M(G, v_0)$ denotes $\{v_0\}$ and the set of maximal modules of G that do not contain v_0. $G/M(G, v_0)$ refers to the graph G without those modules.

At its core, the algorithm partitions the vertices of a graph G, based on an arbitrary vertex v_0 and recursively computes the modular decomposition of the graph induced by the vertices that are connected to v_0 and the graphs induced by vertices that are disconnected to v_0. After the resulting modular decomposition trees are restricted, based on the connections between their vertices and $G/M(G, v_0)$ is computed, the resulting MD trees can be assembled to form the final result. The complexity of this algorithm lies in its approaches and data structures that are used to achieve a preferable time-bound.

For example, the algorithm relies on an advanced *union-find* structure to efficiently maintain and track the connected components of the graph during the decomposition process.

Algorithm 4.1: Dahlhaus

```
Data: An undirected graph $G$
Result: The graph's modular decomposition
1 if $G$ has only one vertex $v$ then
2     return $v$ as a tree node
3 else
4     Recurse: Select a vertex $v_0$, and recursively compute the modular
        decompositions of $G|N(v_0)$ and $G|\overline{N}(v_0)$.
5     Restriction Step: Using these trees, find $M(G, v_0)$ and for each
        $X \in M(G, v_0)$ the modular decomposition of $G|X$.
6     $v_0$-Modules Step: Compute the modular decomposition of $G/M(G, v_0)$.
7     Assembly Step: Assemble these modular decompositions to form the
        modular decomposition of $G$ based on the approach of Ehrenfeucht et al.
```

The union-find data structure is a structure that helps keeping track of a partition of a set into disjoint subsets. It supports two main operations: *Union*, where two subsets are combined into one, and *Find*, where the set that contains a particular element has to be determined. In the algorithm, union operations are performed on the endpoints of *active edges*. Active edges are those, that are connecting two vertices of different recursively calculated MD-trees. The find operation is used to determine the representative (or root) of the set to which a particular vertex belongs.

Furthermore, a topological sort is used to find the topological order of the component graph of the *forcing graph*. The forcing graph is a specific graph that is constructed for the modular decomposition to represent relationships between certain components or vertices in the original input graph. The topological order is essential for correctly organizing the components of the graph and updating parent pointers during the traversal of the decomposition tree. Hence, topological sort is used to efficiently determine the order in which the graph's components should be processed.

Each vertex in the graph is assigned a unique preorder number during the traversal of the decomposition tree. Here, radix sort is used to sort the vertices based on these preorder numbers in linear time. This sorting is crucial for subsequent steps, especially when selecting the initial pivots for the recursive calls and ensuring the correctness of the least common ancestor computations.

These where only three of many structures and operations that the algorithm uses to achieve an efficient time-bound. Considering that, the authors describe two versions of the algorithm: A more practical variant has a time-bound of $O(n + m\alpha(m, n))$. Here, the α is an extreme slow-growing but unbounded function, that has a maximal value of four in any practical application. A second, less practical version of the algorithm uses advanced data structures and tricks to avoid touching vertices and tree nodes that have no incident active edges during the inductive step. This allows this version to achieve a linear time-bound.

Overall, this chapter provided an overview on different approaches to modular decomposition. The first algorithm, that was described in Section 4.1.1 was - compared to the other algorithms - relatively simple and straightforward. However, it had an undesirable high time-complexity. Other algorithms that followed managed to reduce the time complexity up to linear time, but they are often complex to a point where an efficient implementation comes with several weeks of work and many other unfortunate trade-offs. The rest of this work will provide deeper insights, as well as an implementation of a newer algorithm, that manages to achieve linear time without relying on multiple highly advanced data structures and complicated operations. Compared to the algorithms shown in Section 4.1.2, Section 4.1.3 and Section 4.1.4, it can be regarded as a simple, linear time algorithm for modular decomposition.

4.2 The Algorithm

Now, it is finally time to look at an algorithm in detail, with the aim of implementing it. The algorithm that was chosen for this work was developed by Marc Tedder, Derek Corneil, Michel Habib and Christophe Paul and described in their paper "Simple, Linear-time Modular Decomposition" [TCHP08]. In the conclusion of this paper, the authors claim to be "confident of having achieved the first simple, linear-time algorithm for transitively orienting a graph." As already described in Chapter 3, this is equivalent to having achieved the first simple, linear time algorithm for modular decomposition.

According to the algorithms description in the paper [TCHP08], a considerable amount of work goes in the development of a *factorizing permutation* of the input graph's modules. As this is a key concept of the paper's algorithm, we should take some time to understand it.
Every graph $G = (V, E)$ can be depicted through an MD-tree T_G. In this tree, the leaves correspond exactly to vertices in G. Every inner node n of T_G corresponds to a set X of vertices that is a subset of V. Here, X contains exactly the vertices that are descendants of n (that can be reached from n by only going "down"). If the edges that have both their ends in X are added, one gets a sub-graph G_n based on the node n. Therefore, every graph contains modules, which correspond to the inner nodes of the modular decomposition tree. A *factorizing permutation* is a permutation of the vertices of G, in which all vertices of every module appear consecutively.
If a permutation of the vertices $x_1, x_2, x_3, x_4, \ldots, x_n$ is given, it is a factorizing permutation if and only if it is possible to place alternating opening- and closing parenthesis, such that all vertices of a module are within two parenthesis.
As example, consider a graph $G = (V, E)$, where $V = \{A, B, C, D, E, F\}$ and modules $\{A, B\}$, $\{C, D\}$, $\{E, F\}$. In this case, $\{D, C, E, F, B, A\}$ is a valid factorizing permutation because it is possible to place parenthesis in the earlier described way: $\{(D, C), (E, F), (B, A)\}$. Taking a look at e.g. the permutation $\{A, C, B, D, E, F\}$, it is not possible to insert parenthesis as described above; thus, it is not a valid factorizing permutation.

In the algorithm's last step, the paper uses a factorizing permutation to build a list of (co-)components, in which $\mu - values$ are used to assemble the final modular decomposition tree. Unfortunately, this way seems not intuitive and, additionally, the papers' sections on this step contain errors. For those reasons, we contacted the authors [Pau23] and learned, that they are currently working on an improved way to assemble the modular decomposition tree, in which the factorizing permutation plays a less crucial role. After communicating with Christophe Paul, who kindly shared his current progress with us, we are able to present a better, and far more intuitive approach for the final assembly of the modular decomposition tree. Thus, the procedure that will be used in Section 4.2.1.4 and Section 4.2.2.4 is not the same as the one the authors of [TCHP08] originally described.

With that being said, it is finally time to study the algorithm in detail.

4.2.1 Algorithm Description

In this section, we will analyse the algorithm from Tedder, Corneil, Habib and Paul [TCHP08]. In general, it consists of four steps:

1. Recursion
2. Refinement

3. Promotion
4. Assembly

At the time of publication, the first step is already well-known. In it, all vertices are grouped based on their distance to an arbitrarily selected pivot-vertex x. A variant of this approach was also used in the algorithm of Section 4.1.4. Step 4 is known as well: It creates an MD-Tree based on a factorizing permutation. The ingenuity of the algorithm lies in uniting these two already known concepts. Steps 2 and 3 take the responsibility for that.

4.2.1.1 Step 1: Recursion

This step aims to reduce the complexity of the problem by recursively removing vertices from the computations. At the start of the process, an arbitrarily selected vertex x is chosen as pivot element. The remaining vertices are grouped into sets, based on their distance to x.

- $N_0 = \{n \mid \text{starting at } x,\ \text{vertex } n \text{ can be reached using only one edge}\}$,
- $N_1 = \{n \mid \text{starting at } x,\ \text{the shortest path to } n \text{ has exactly two edges}\}$,
- $N_2 = \{n \mid \text{starting at } x,\ \text{the shortest path to } n \text{ has exactly three edges}\}$,
- $\ldots$

Finally, the MD-trees $T(N_i)$ of these N_i are computed recursively.

But what is this separation of the graph's vertices good for? The algorithm aims to represent the modules of the graph through inner nodes in one of the recursively computed MD-trees. Remember, that a (strong) module M is characterized by the fact, that every vertex outside of M is either connected to all the vertices in M or to none of them. So, let M be a strong module that doesn't contain x. After recursion, this module is completely in one of these $T(N_i)$. This can be proved through simple logic: Let n_1 and n_2 be two vertices that lie in different $T(N_i)$. This means, that these two vertices can be reached from x through a different amount of edges. Therefore, at least one vertex v exists that is connected to n_1 but not n_2 (or the other way around). Including v in M would lead to the inclusion of multiple other vertices from different $T(N_i)$, and ultimately the whole graph. Even though including the whole graph in a module is possible, it is not the desired outcome of a modular decomposition, thus v has to remain outside of M. This infers that n_1 and n_2 cannot be part of the same module.

Algorithm 4.2 sketches a way to implement this procedure. After the algorithm is terminated, one obtains a list that looks as follows:

$$T(N_0),\, x,\, T(N_1),\, ...,\, T(N_k)$$

Here, $T(N_0)$ is the MD-tree of the neighborhood of x, whereas the vertices in N_1, ..., N_k are all non-neighbors of x. Starting at x, vertices in the set N_a can be reached in exactly $a + 1$ steps. The base case for this recursion is trivial: If the input only consists of one vertex, a modular decomposition tree with that vertex as only node is returned. In regard to the efficiency of the algorithm, one can notice that every vertex becomes the pivot element at most once throughout the whole procedure.

In this chapter, we assume that the input graph is connected, but due to the recursive nature of the algorithm, this might not be the case in every (sub)step. Handling disconnected graphs however, is not complicate. If the algorithm gets a disconnected graph as input, it

Algorithm 4.2: Recursion

```
Data: A simple, connected graph
Result: A forest of MD-trees, a set of active edges
1 Take an arbitrarily chosen vertex $x$ as the pivot element.
2 Compute the sets $\overline{N(x)}$ and $N(x)$, where $\overline{N(x)}$ contains all vertices that are
    connected to $x$. $N(x)$ contains all remaining vertices. The correct order of these
    sets is $\overline{N(x)}$, $x$, $N(x)$.
3 Calculate the MD-tree for the subgraph that is induced by the set $\overline{N(x)}$
    recursively. Additionally, construct two new sets $\overline{N_A(x)}$ and $\overline{N_N(x)}$, where
    $\overline{N_A(x)}$ contains all vertices in $N(x)$ that have a neighbor in $\overline{N(x)}$. $\overline{N_N(x)}$
    contains the remaining nodes of $N(x)$.
4 Continue with this process by looking at the set $\overline{N_A(x)}$. Calculate the MD-tree for
    this set recursively and determine the set of vertices in $\overline{N_N(x)}$, that have a
    neighbor in $\overline{N_A(x)}$. This newly calculated set is the new $\overline{N_A(x)}$, whereas the old
    $\overline{N_A(x)}$ becomes the set $N_1$. Remove all elements of (the new) $\overline{N_A(x)}$ from $\overline{N_N(x)}$.
5 Continue as described in Step 4. Increment the numbers when renaming the sets
    into $N_i$. The process is finished when the newly created set $\overline{N_N(x)}$ is empty.
```

has to determine all its connected components and calculate their modular decomposition trees separately. After that, the computed MD trees can be united under a common node with a PARALLEL label. This node represents the root of the resulting modular decomposition tree.

4.2.1.2 Step 2: Refinement

Step 1 of the algorithm provides a list of modular decomposition trees. This list of MD-trees will also be referred to as *forest* in the following sections. Every module that does not contain the previous pivot element x now lies completely in one of the $T(N_i)$'s. This step aims to refine the trees, so that every module that does not contain x is correctly represented by one node in the forest.

In general, *Refinement* can be seen as the process of partitioning already partitioned sets even further, with the aim of including new information. The algorithm uses this concept by refining the forest based on a set of so called *active edges*. An edge of the graph is called active, if and only if it is adjacent to x (pivot element from step 1) or connects two vertices from different N_i.

To be able to separate all modules not containing x in the forest, every vertex (except for x) has to be looked at again. Let n be a vertex and let $T(N_k)$ be the tree that contains n. In general, n is connected to some nodes in $T(N_{k-1})$ (if that tree exists) and disconnected to others in that tree. The same can be said about nodes in $T(N_{k+1})$. The vertex n cannot be connected to any other node outside of those trees, as - in this case - it would provide a faster way to reach these sets, starting from x. Refinement takes the vertex n and checks, if there are nodes h_1, h_2 in $T(N_{k-1})$ or $T(N_{k+1})$, such that n is connected to h_1 but not h_2. This is done using the incident active edges of n. If such a connection exists, the algorithm knows that h_1 and h_2 cannot be in the same module. Therefore, h_1 and h_2 must either be separated into different trees, or a new node that separates h_1 and h_2 must be inserted into either $T(N_{k-1})$ or $T(N_{k+1})$, depending on the current situation.

After this procedure is repeated for all vertices, the algorithm can be assured that all modules not containing x are correctly represented in the forest, as connections to all vertices (x in Step 1, every other vertex in Step 2) have been checked. Recall that in order to claim that two vertices n_1 and n_2 are not in the same module, it is necessary to have a witness (vertex) k that is connected to n_1 but not to n_2, and that such a vertex can no longer exist.

The algorithm for Step 2 is separated into two sub-algorithms, seen in Algorithm 4.3 and Algorithm 4.4

Algorithm 4.3: Refinement, Part 1

```
Data: A forest resulting from step 1, a set containing all active edges
Result: The refined forest
1  foreach vertex $v$ (except $x$) do
2      Determine $\alpha(v)$, which represents the incident active edges of $v$
3      Refine the forest with $(v)$ and Algorithm 4.4, such that:
4      if $v$ is to the right of $x$ and a tree to the right of $x$ is refined then
5          Refine using right-splits and mark nodes with "right"
6      else
7          Refine using left-splits and mark nodes with "left"
```

Algorithm 4.4: Refinement, Part 2

```
Data: A forest resulting from Step 2, a set $X$ of incident active edges,
      information about using "left" or "right" for splits and markings
Result: The forest after being refined with the given set of incident active edges
1  Determine the set of sub-trees $T_1$, ..., $T_k$, where all the leaf-descendants are
   contained in $X$
2  Determine the set of parent nodes $P_1$, ..., $P_j$ of the $T_i$. It applies that $j \leq k$, as
   some $T_i$ can have the same parent node
3  for Every $P_i$ that is not prime do
4      Determine the sets $A$ and $B$. $A$ contains all children of $P_i$, that can be found
       in the set of maximal sub-trees $T_1$, ..., $T_k$. $B$ contains the remaining children
       of $P_i$. If $A$ is empty or $B$ is empty, the refinement for $P_i$ stops
5      Determine the trees $T_a$ and $T_b$ depending on $A$ and $B$. If there is only one tree
       in $A$ ($B$), then this tree is $T_A$ ($T_B$). If there are multiple trees in $A$ ($B$), these
       trees must be united under a common node. This node becomes the root of
       $T_A$ ($T_B$)
6      Assign the same label (SERIES, PARALLEL or PRIME) as $P_i$ to $T_A$ and $T_B$
7      if $P_i$ is the root of a $T(N_i)$ then
8          if left-split (see Algorithm 4.3) then
9              Replace $P_i$ with $T_a$, $T_b$ in the ordered list of trees
10         if right-split (see Algorithm 4.3) then
11             Replace $P_i$ with $T_b$, $T_a$ in the ordered list of trees
12     else
13         Replace the children of $P_i$ with $T_a$ and $T_b$
14     Mark the root and all its ancestors (up to the root of the current $T(N_i)$) of $T_a$
       and $T_b$ with "left" or "right", depending on the information of Algorithm 4.3
15 for Every $P_i$ that is prime do
16     Mark $P_i$, as well as all its children and ancestors with "left" or " right",
       depending on the information of Algorithm 4.3
```

After refinement, the strong modules not containing x appear consecutively in the forest. This means, that a pre-order of all trees of the forest in order would list all elements of the modules not containing x one after another. A more detailed explanation will be provided at the end of Step 3. Furthermore, the nodes in the forest that do not have marked children correspond exactly to the strong modules not containing x.

4.2.1.3 Step 3: Promotion

The aim of this step is to split the trees of the forest, so that every remaining tree corresponds to a component of the final MD tree. Most of the work was already done in Step 2, but some work still remains. Refinement created new nodes to separate siblings that do not belong to the same strong module and left markings on those nodes. As a following, these markings show where the different strong modules are placed in the forest, and where the trees should be separated from each other. Promotion uses this information in the following way: While there is a pair of a child- and a parent node with the same marking, the connection between these nodes has to be removed, resulting in two new trees. These trees should replace the old one in the forest. The algorithm uses the information on the marking to determine the places of the new trees. If the child- and parent node are marked with "left", the tree with the former child node as root will be placed on the left of the tree with the parent node. Otherwise, if the nodes are marked with "right", the tree with the former child as root will be placed on the right of the parent-node's tree. Algorithm 4.5 sketches a possible implementation for this step.

Algorithm 4.5: Promotion

```
Data: The forest resulting from Step 2
Result: A fully refined forest that provides a *factorizing permutation*
1 while There is a root $r$ with a child $c$ both marked with "left" do
2     Remove the connection between $r$ and $c$. This results in a new sub-tree, that
        has $c$ as its root
3     Place this new sub-tree right before the tree that contains the root $r$
4 while There is a root $r$ with a child $c$ both marked with "right" do
5     Remove the connection between $r$ and $c$. This results in a new sub-tree, that
        has $c$ as its root
6     Place this new sub-tree right after the tree that contains the root $r$
7 Remove all marked roots in the forest, that have exactly one child. Place this
    child at the former position of the root
8 Remove all marked roots in the forest, that have no children (Except the root is a
    former leaf the contains the data of one vertex)
9 Delete all markings
```

After the promotion algorithm has terminated, the result of traversing all leaves of the forest in a pre-order is a *factorizing permutation.*

4.2.1.4 Step 4: Assembly

In the last step of the algorithm, the individual trees of the forest resulting from Step 3 have to be combined into a final modular decomposition tree. As already mentioned, the paper uses the resulting factorizing permutation to build a list of (co-)components, in which $\mu - values$ are used to assemble the trees. Due to our conversation with Christophe Paul, we know of an improved way to execute this step, hence the procedure that will be described in this section is not the same as the one the authors of [TCHP08] originally used.

At this point in time, the algorithm knows the individual pieces of the final tree. What is left to do is combining these, by constructing the spine of the tree. To get a better understanding of where we are at, it is useful to take a look at Figure 4.4

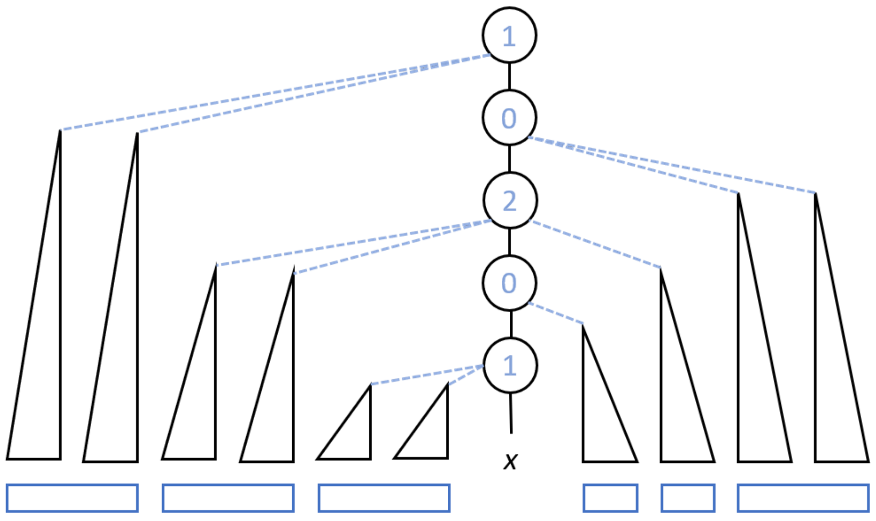

Figure 4.4: The spine of a modular decomposition tree. "0" indicates a PARALLEL-, "1" a SERIES- and "2" a PRIME module

In this Figure, the black triangles are (smaller) modular decomposition trees, representing maximal modules. Each triangle in the graphic corresponds to one tree of our current forest. The correct order of the triangles is also given by the order of the trees. The x is representing the pivot-element; its place between the modules is also known. The trees to its left consist of elements that are connected to x in the original input graph, while the trees to its right contain the elements that are not. Everything that is represented in blue (the dashed lines, the numbers in the circles and the boxes below the triangles) has to be computed in this final step.
Step 4: Assembly is composed of two sub-steps. In the first, left- and right-pointers have to be assigned to all maximal modules in the following way:

1. The left-pointer of a maximal module M points to the module M' with the highest index, so that all modules to the left of M' are adjacent to M. The left-pointer can never be to the right of M.

2. The right-pointer of a maximal module M points to the module M' with the smallest index, so that all modules to the right of M' are not adjacent to M. The right-pointer can never be to the left of M.

The module indices increase from left to right. Two modules M_1 and M_2 are adjacent, if and only if at least one element of M_1 is connected to at least one element of M_2. Algorithm 4.6 shows how the left- and right - pointers can be computed.

Now, these pointers can be used in a second substep that is responsible for the construction of the final tree. The algorithm starts at the position of the pivot-element x. From there, it forms the MD-tree in multiple steps. In each step, a new inner node of the tree is created, which has the roots of some trees from the forest as children. Figure 4.4 shows, that PARALLEL nodes only add trees to right of x, while SERIES nodes only add trees to x's left. PRIME nodes have children on both, the left and the right of x. This is used to determine the label of the new node. In every step, a parallel module has the highest priority. This means, that the algorithm tries to include the element to the right of either

Algorithm 4.6: Assembly, Part 1

```
Data: The forest resulting from Step 3, the original graph
Result: A set of left- and right-pointers for each tree in the forest
1 for every tree $T$ in the forest do
2     Determine all adjacent trees (maximal modules) of the vertices in $T$
3     Determine the left-pointer of $T$ by traversing the list of trees from left to right,
        stopping at the first non-adjacent tree
4     Determine the right-pointer of $T$ by traversing the list of trees from right to
        left, stopping at the first adjacent tree
```

x, or the lastly formed module. If a new tree is included in the current module, every tree up to its left- and up to its right-pointer has to be included as well. Every tree that gets added like this has to check its left- and right-pointers too. After no tree can be added anymore, the module can be formed. This process is successful, if it only added trees to the right of x (or to the right of the last formed module). If that is not the case, the algorithm has to start again, this time by adding the tree to the left. Following the same rules for including new modules based on the left- and right-pointers, the new module can be created. It gets labeled with PARALLEL, if it only contains elements to the right of either x or the last formed module, SERIES, if it only contains elements to the left, and PRIME otherwise. This process is repeated until every tree has been included.
It is possible that the root of one of the included trees has the same label as the created module. In this case, these nodes can be unified without the loss of information. This is done by replacing both nodes with a new node, that has the same label as both old nodes, as well as both their children. After this, the final modular decomposition tree is returned. Algorithm 4.7 displays these steps.

Algorithm 4.7: Assembly, Part 2

```
Data: The forest resulting from Step 3, a set of left- and right pointers for each
      tree in the forest
Result: The final modular decomposition tree
1  Initialize a left- and a right-index to surround the pivot $x$
2  while the left- and right-index do not surround the whole list do
3      if going to the right is possible without adding elements on the left then
4          Add the element to the right of the current right-index to a queue
5      else
6          Add the element to the left of the current left-index to a queue
7      while there are elements left in the queue do
8          Take one element of the queue and move the algorithm's indices according
             to the element's pointers
9          Add all new elements within the indices to the queue
10     Create a new node and label it 'PARALLEL' if only elements to the right were
         included, 'SERIES' if only elements to the left were included, and 'PRIME' if
         elements both, to the left and the right were included
11     Add all not-already-added trees between the left- and right-indices, as well as
         the module node of the last iteration (or the node containing the pivot) as
         children to the newly created node
12     If the new node has the same label as one of its children, unify these two nodes
         into a node with the same label and both their children
```

4.2.2 An Example

Understanding how an algorithm works can often be difficult, without seeing what it does when real input is given. Therefore, this section will provide the calculation of the modular decomposition of an example graph, along with the algorithm. Consider the graph shown in Figure 4.5

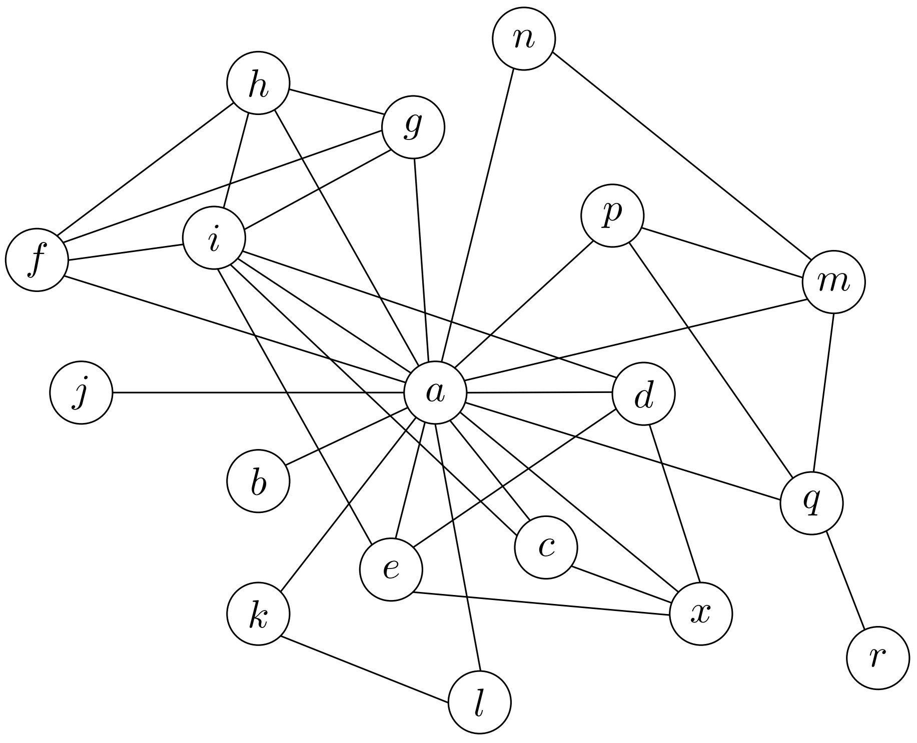

Figure 4.5: A simple, undirected graph

4.2.2.1 Step 1: Recursion

The algorithm starts by selecting an arbitrary vertex as pivot-element. In this example, the pivot element will be the vertex x. The next step is to determine the set of vertices that are adjacent to x. Following the algorithm, we determine the N_i's, where all vertices in a set N_j can be reached using exactly $j + 1$ edges, starting from x. We get the following sets: $N(x) = N_0 = \{a, c, d, e\}$, $N_1 = \{b, f, g, h, i, j, k, l, m, n, p, q\}$ and $N_2 = \{r\}$. The algorithm proceeds by calculating the MD trees of these sets recursively. At the moment, we will not dive into that recursive calculation, but rather continue the algorithm by taking these recursive calculations as given. Figure 4.6 shows the result.

4.2.2.2 Step 2: Refinement

This step aims to refine the resulting forest from Step 1 by checking the connections of every vertex v other than x. If there are two vertices m and n in G, where n is connected to v, but m is not, then it is not possible for m and n to be in the same module and precautions must be taken. The algorithm deals with the example graph in the following way:

1. Vertex c (alternatively vertex e or d) of MD tree (a) has a connection to vertex i in the right neighbor tree of (a), but there is no connection from c to the vertices $\{g, h, f\}$. As a result, these four vertices cannot form a module and should be separated. The algorithm works as follows:

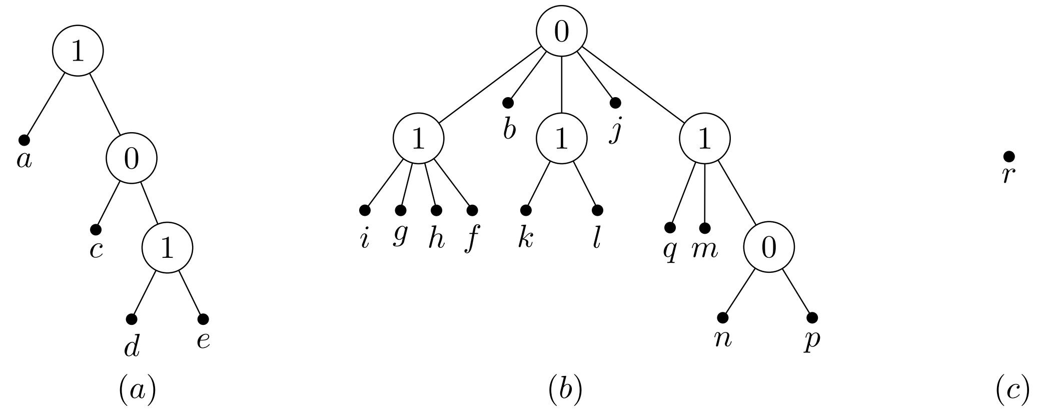

Figure 4.6: The recursively computed MD-trees, after x has been chosen as pivot-element: (a) T(N$_0$); (b) T(N$_1$); (c) T(N$_2$)

a) The incident active edges of c are determined. These active edges can be seen in Figure 4.7. and Table 4.1. The searched edges here are from c to $\{i, x\}$.

b) The set of maximal sub-trees, where the leaves are all in the set $\{i, x\}$, contains only the sub-tree that consists of one node, i. This sub-tree has the parent node P_1, that is labeled with "1" and has the children i, g, h, f. As P_1 is labeled with "1", it is not prime.

c) The next step is to determine T_a and T_b. Therefore, the sets A and B have to be computed. In this case, set A only contains the leaf-node i, whereas B contains the leaf-nodes g, h and f. So, T_a only consists of one node (i), and T_b gets a new root that is labeled with "1" and has the child-nodes g, h, f.

d) P_1 is not a root. Therefore, we replace the nodes i, g, h, f with T_a and T_b.

e) Finally, we mark the roots of T_a and T_b, as well as all of their ancestors up to the root of MD-tree (b) with "left", as c is placed to the left of x.

2. Vertex r in MD-tree (c) is connected to vertex q in the left neighboring tree, but not to any other vertex. This must be considered in the modular decomposition. The algorithm works as follows:

   a) Determine the incident active edges of r. In this case, there is exactly one edge from r to q.

   b) The set of maximal sub-trees, where the leafs are all in the set q, contains only the sub-tree that consists of the node q. This sub-tree has the parent node P_1, that is labeled with "1" and has the child nodes q, m and "0" (with "0" referring to the node labeled "0" with children n and p).

   c) Now, we need to determine T_a and T_b. To do this, compute the sets A and B. In this case, set A contains the leaf node q, and set B contains the nodes m, "0". Hence, T_a only consists of one node (q), and T_b gets a new root, labeled with "1," which has the child nodes m and "0."

   d) P_1 is not a root. As a following, we replace the node q with T_a and the nodes m, "0" with T_b.

   e) Finally, we mark the roots of T_a and T_b, as well as all of their ancestors up to the root of MD-tree (b), with "right" (The vertex r is placed to the right of x, and the tree being refined is tree (b), which is also to the right of x).

3. Vertex b (alternatively, the vertices g, h, f, k, l, j, q, m, n, p) in MD-tree (b) is connected to vertex a in the left neighboring tree, but not to the vertices c, d, e. This needs to be considered in the modular decomposition. The algorithm works as follows:

    a) Determine the incident active edges of b. In this case, there is exactly one edge from b to a.

    b) The set of maximal sub-trees, where the leafs are all in a, contains only the sub-tree that consists of the node a (T_1). This sub-tree has the parent node (P_1) labeled with "1" and has the child nodes a and "0."

    c) The set A contains only the leaf node a, set B contains the node "0." Thus, T_a and T_b are trivially determined.

    d) In this case, P_1 is a root (of MD-tree (a)). Additionally, a tree to the left of x is being refined. Thus, we replace the root of MD-tree (a) with T_a, T_b (in this exact order).

    e) Finally, we mark the roots of T_a and T_b with "left."

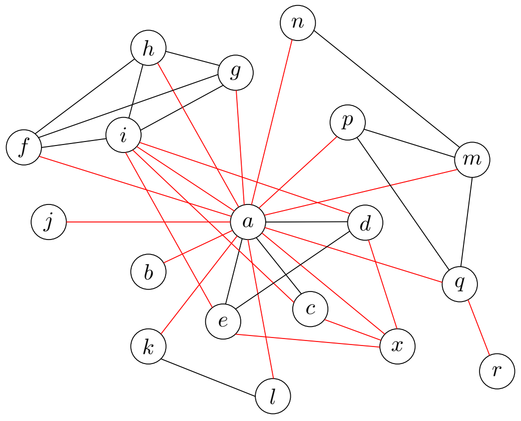

Figure 4.7: The graph with its active edges (colored red), after x has been chosen as the pivot-element

Vertex	Active Edges
a	$x, b, j, f, g, h, i, k, l, m, n, p, q$
c, d, e	x, i
$b, j, f, g, h, k, l, m, n, p$	a
i	a, c, d, e
q	a, r
r	q

Table 4.1: The active edges for each vertex of the graph after x has been chosen as the pivot-element

After refinement is done, we get the forest shown in Figure 4.8.

4.2.2.3 Step 3: Promotion

The task now is to separate the strong modules from each other. This is done by using the markings from Step 2. Here, the algorithm works as follows:

1. The root of the MD-tree (b) is marked with "left" and has a child node (Label "1", with child nodes i and "1"), that is marked with "left" as well. Hence, the connection between these two nodes is removed, and the new tree with root "1" gets placed directly to the left of tree (b). The root of tree (b) is also marked with "right" and has a child-node with that marking. Therefore, this connection gets removed as well. The node "1" (with children q and "1") becomes the root of a new tree that is placed directly to the right of (b).

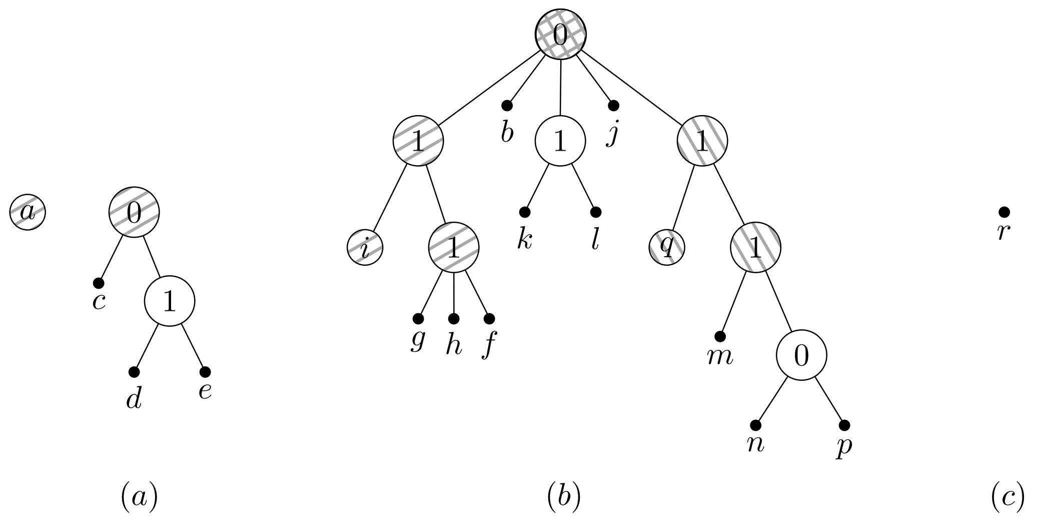

Figure 4.8: The MD-trees after step 2: Refinement. A rising shading corresponds to a marking with "left", a falling shading corresponds to a marking with "right". The crossed shading (at the root of (b)) shows, that this node has been marked with "left" and "right"

2. The newly inserted tree to the left of (b) has a root marked with "left," and this root has a child node (label "1") also marked with "left", with children g, h, f. Consequently, the connection between the child node "1" and its root gets removed, and the former child node is inserted as the root of a new tree to the left of the old one.
3. Similarly, the node with label "1" (children q and "1") in the right-inserted sub-tree has a root marked with "right" and a child node marked with "right" as well. Thus, the connection between them is severed, and the child node (label "1" with children m and "0") becomes the root of a new tree, that is inserted directly to the right.
4. Now, there are two trees, each consisting of a single root with a single child ("1" with child i, to the left of (b); "1" with child q, to the right of (b)). The roots "1" are removed, and the child nodes (i and q) are placed in their positions.
5. There is no root without a child node that needs to be removed (Except for the roots containing vertex data (a, i, q, r), and those roots can stay).
6. Finally, all markings are cleared

Figure 4.9 shows the forest we are left with after this step is completed.

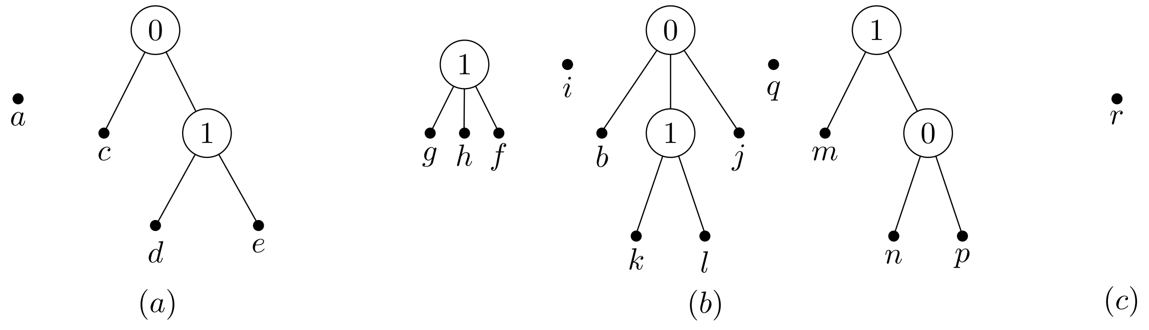

Figure 4.9: The forest after step 3: Promotion

Notice that the algorithm has refined and promoted the trees in a way, that a pre-order of their leaves (in order) results in a factorizing permutation.

Factorizing permutation: a, c, d, e, x, g, h, f, i, b, k, l, j, q, m, n, p, r

4.2.2.4 Step 4: Assembly

To start the assembly process, we first have to determine which of the remaining trees from Figure 4.9 are adjacent. Two trees T_1 and T_2 are adjacent if and only if at least one element of T_1 is connected to at least one element of T_2. Table 4.2 shows the adjacencies between the trees.

	$\{a\}$	$\{c, d, e\}$	$\{g, h, f\}$	$\{i\}$	$\{b, k, l, j\}$	$\{q\}$	$\{m, n, p\}$	$\{r\}$
$\{a\}$	X	X	X	X	X	X	X	
$\{c, d, e\}$	X	X		X				
$\{g, h, f\}$	X		X	X				
$\{i\}$	X	X	X	X				
$\{b, k, l, j\}$	X				X			
$\{q\}$	X					X	X	X
$\{m, n, p\}$	X					X	X	
$\{r\}$						X		X

Table 4.2: The connection matrix of the maximal modules (trees from Figure 4.9), based on the original graph. The values on the diagonal are not important for the algorithm

As a next step, the left- and right-pointers of each tree can be computed. These pointers always point between two maximal modules. Figure 4.10 shows all possible pointer locations and assigns indices to each of them.

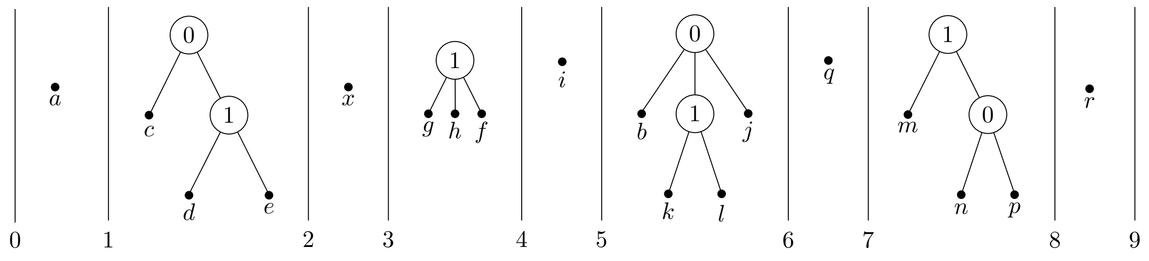

Figure 4.10: The possible left- and right-pointer locations and their indices

The calculation of the pointers for each maximal module can be demonstrated by a few examples:

1. The maximal module containing $\{a\}$ has its left pointer at index 0, as this is the only possible location to the left of it. As only $\{r\}$ is not adjacent to $\{a\}$, the right pointer of this module is at index 8.
2. The maximal module containing $\{c, d, e\}$ is adjacent to $\{a\}$. Therefore, its left pointer must be at index 1. This module has no connections to $\{r\}$, to $\{m, n, p\}$, to $\{q\}$ or to $\{b, k, l, j\}$. However, there is a connection to $\{i\}$. As a following, its right pointer is at index 5.

3. The maximal module containing $\{g, h, f\}$ is adjacent to $\{a\}$, but not to $\{c,\ d,\ e\}$. As a consequence, its left pointer must be at index 1 as well. Regarding its right pointer, one can see that all modules to the right of $\{i\}$ are non-adjacent, whereas $\{i\}$ is adjacent to it. So, the module's right pointer can be set to index 5.

   ...

Table 4.3 shows the calculated left- and right-pointers for all maximal modules.

Max. Module	**Left-Pointer**	**Right-Pointer**
$\{a\}$	0	8
$\{c,\ d,\ e\}$	1	5
$\{g, h, f\}$	1	5
$\{i\}$	4	5
$\{b,\ k,\ l,\ j\}$	1	6
$\{q\}$	1	9
$\{m,\ n,\ p\}$	1	8
$\{r\}$	0	9

Table 4.3: Left and right pointers of the maximal modules

Now, we can start with the construction of the final tree:

1. The algorithm starts at the position of x, in the middle of the list. First, it determines if a PARALLEL module can be formed. Including $\{g,\ h,\ f\}$ to the right of x would lead to the inclusion of $\{c,\ d,\ e\}$ as well, due to the modules left-pointer being 1. As a consequence, it starts by going left and including $\{c,\ d,\ e\}$. The new left-pointer is still 1, which means that (currently) no module to the left of $\{c,\ d,\ e\}$ must be included. The new right-pointer however, is 5. This infers that $\{g,\ h,\ f\}$ and $\{i\}$ have to be included in the current module. As both left pointers of the new trees are not smaller than 1 and both their right pointers not higher than 5, the module is complete. Note that trees to the left and to the right of x were included. Thus, a node labeled PRIME is created with the root-nodes of $\{c,\ d,\ e\}$, $\{g,\ h,\ f\}$, $\{i\}$ and x as its children.

2. In a next step, the algorithm once again tries to form a PARALLEL module by including the tree to its right, $\{b,\ k,\ l,\ j\}$. As this maximal module has the left- and right-pointers 1 and 6, no additional trees have to be included in the current module. As only trees on the right of the last module were incorporated in this step, the new module will be represented by a PARALLEL node, having the PRIME node of the last formed module and the root node of $\{b,\ k,\ l,\ j\}$ as its children. As $\{b,\ k,\ l,\ j\}$ has a PARALLEL node as its root as well, it is possible to combine it with the newly created node.

3. In the third step, the algorithm cannot form a PARALLEL module, as including $\{q\}$ would lead to the inclusion of $\{r\}$ and therefore including $\{a\}$. Hence, the algorithm starts with the integration of $\{a\}$ into the new module. As $\{a\}$'s right-pointer is 8, $\{q\}$ and $\{m,\ n,\ p\}$ and therefore $\{r\}$ (as the right-pointer of $\{q\}$ is 9) have to be included too. The new module is labeled PRIME and has the root nodes of the listed trees, as well as the PARALLEL node that was created in step 2 as its children. Now,

there are no more trees left outside the module. This means, that the algorithm has finished.

After this is done, we are left with the final modular decomposition tree. This tree is displayed in Figure 4.11.

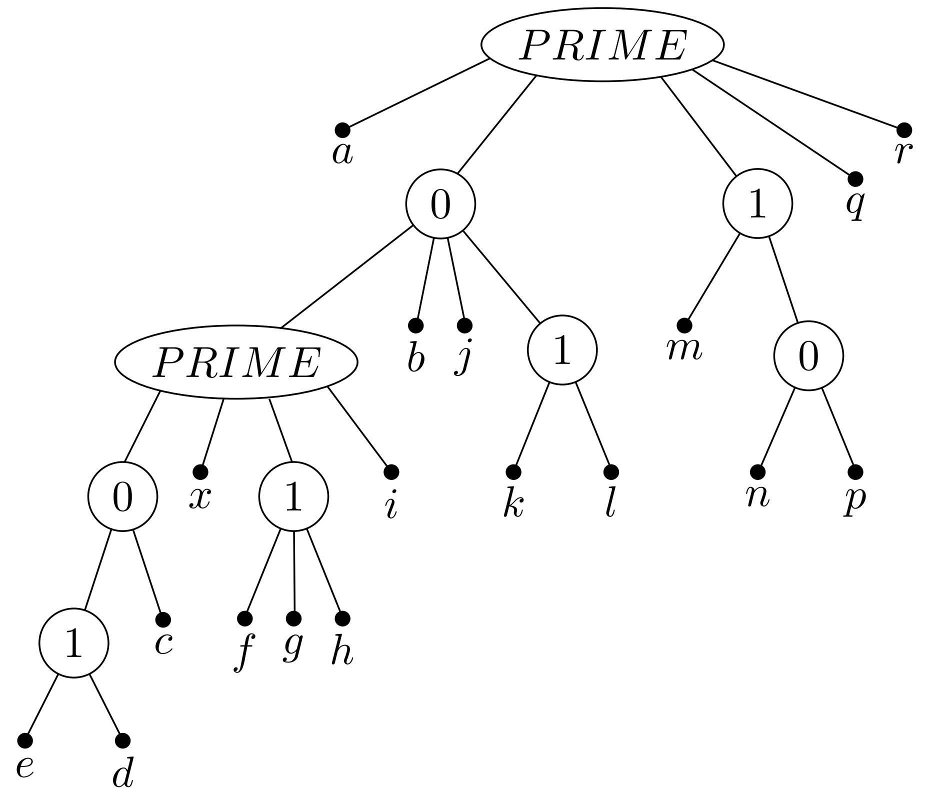

Figure 4.11: The final modular decomposition tree of the graph. PARALLEL nodes are labeled "0", SERIES nodes are labeled "1". PRIME nodes are marked with "PRIME"

5. Algorithm Implementation and Evaluation

This chapter will focus on implementing the algorithm by Marc Tedder, Derek Corneil, Michel Habib and Christophe Paul [TCHP08] that was analysed in Chapter 4. In Section 5.1, we will describe important parts of our implementation that were not present in the algorithm's description. Then, in Section 5.2, we will proceed to evaluate that implementation. Section 5.2.1 will focus on assuring the implementation's correctness by using a two-step procedure and several input graphs of different size and types. Afterwards, Section 5.2.2 will investigate the implementation's time-complexity.

5.1 Implementing a Modular Decomposition Algorithm

Implementing a complex algorithm can take a long time and multiple iterations of bug-fixing. Fortunately, we won't have to worry about that now, as the implementation is already done. A link to this implementation is provided here:

https://github.com/LuisGoeppel/ModularDecomposition_v4

However, there are still several concepts and details in the implementation that are worth talking about. This will be the content of this section.

5.1.1 The Programming Language

Choosing the right programming language can be of great importance for the development of an algorithm. There are multiple languages to choose from, with some being more, and some being less optimal for the problem. As our algorithm centers around objects like graphs, modular decomposition trees or ordered forests, an object oriented programming language seems like the best choice.
Out of the various options, we selected *C++* for the algorithm-development. One reason that supports this choice is the pointer-arithmetic that languages like *C* or *C++* offer. In the algorithm, several elements are linked to each other. These links can be between nodes in a modular decomposition tree or between modular decomposition trees, just to name a few. Representing such links with pointers is the easiest and fastest way. Additionally, *C++* is known for its performance and fast calculations. Due to being "closer" to the computer than other object oriented programming languages like *Java* or *Python*, it is

often considered as the best option for high-performance code. As our algorithm aims to calculate the modular decomposition with speed as a relevant factor, *C++* seems like the obvious choice. Considering algorithm-testing, *C++* offers the use of the *Open Graph Drawing Framework (OGDF)* as well. Section 5.2 will contain more information on that topic. Referring to the implementation of the algorithm, no additional libraries were used. This has the benefit that running this algorithm is as simple as possible and that understanding the code requires only knowledge in *C++*.

5.1.2 Data - Structures

With the choice of the programming language out of the way, we can jump into the implementation of the algorithm. Looking at the data structures of the program seems to be a good start for presenting the algorithm.

The data structure that is undoubtedly at the core of the algorithm, is the modular decomposition tree. In our program, this tree is represented through the class *MDTree*. As every modular decomposition tree consists of multiple linked tree nodes, *MDTree* has a *TreeNode* class as inner class.
The nodes of a modular decomposition tree can be grouped into LEAF-nodes, that contain a value corresponding to a vertex in the input graph and INNER-nodes. INNER-nodes can either be PARALLEL, SERIES or PRIME. Our program uses an enumeration to separate all nodes into these four different types (PARALLEL, SERIES, PRIME or LEAF) and an integer to store the value of the node. This value is only set if the node is of type LEAF, otherwise it is left uninitialized. To make operations on the tree nodes more efficient, every node stores the number of its child nodes as well. Furthermore, a tree node can be marked with either "left" or "right" during refinement. This information gets stored using two bool - variables.
As already mentioned, the connections between the nodes are stored using pointers. As the amount of a node's children can vary, it is not possible to store every child in one variable as, e.g. a left-child and right-child pointer in a binary tree. One possibility to store the child nodes is using a *vector* (an expandable array). However, this possibility is not efficient enough for our algorithm. Storing all child nodes in a vector takes a lot amount of space and brings the problem, that removing a child node can cost a remarkable amount of time, as a vector has to shift multiple elements to the left in case of a deletion. Therefore, our program uses one *child* and one *sibling* pointer in every tree node. Following this approach, an infinite amount of children can be stored for every node, by using only two pointers. Here, all children of a node are stored in a linked list using the sibling-pointers and the start of the list is stored in the actual node using its child-pointer. Figure 5.1 displays this method of managing a node's children. In addition, every node has a third pointer to its parent-node. Many procedures during refinement rely on this pointer to work efficiently.

The ordered forest is represented by a double linked list of *MDTrees*'s. Thus, every *MDTree* has one pointer to its left neighbor in the list and one pointer to its right neighbor. This representation has the advantage, that deleting a tree and replacing it with two new trees, as needed during refinement, can be achieved in $O(1)$ time. As explained earlier, a vector of *MDTree*'s would not allow this. The first and last element of this list is referenced in a *TreeList* class, that is also responsible for simple tasks considering this list, like inserting a new tree at the end.
Additional attributes of the *MDTree* class are a pointer to its root-node and two integers for its left- and right index, which are used in the assembly step of the algorithm.

Graphs are also represented in their own class, which contains - in addition to the graph representation - methods that are responsible for graph-specific tasks like providing a list of the graph's connected components or creating subgraphs based on the vertices distances to a pivot element. The graph itself is represented using an adjacency list.

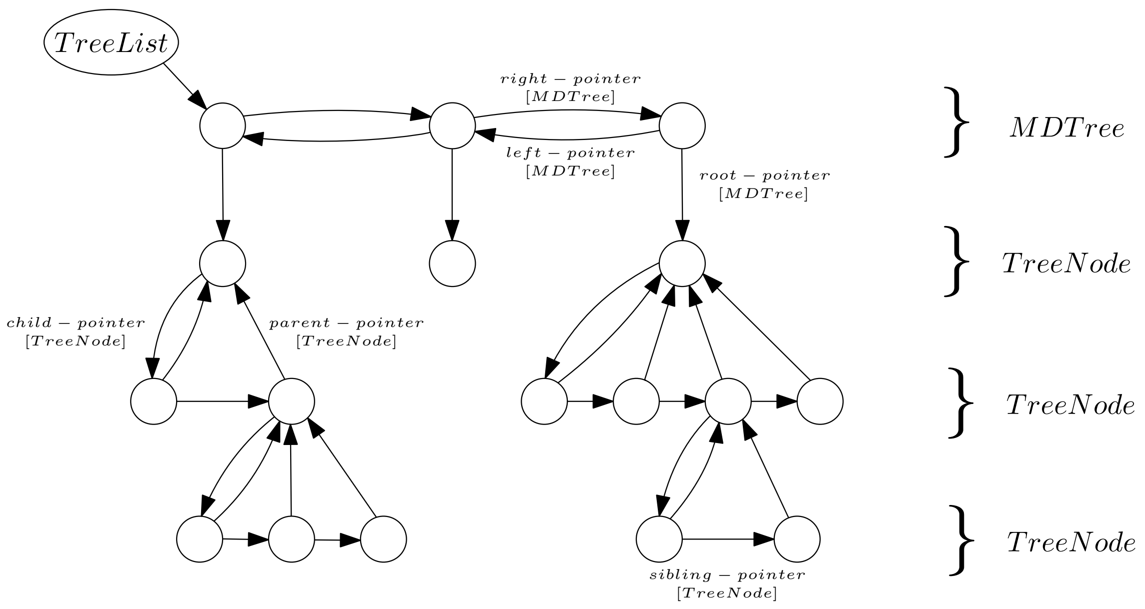
$TreeList$
$right - pointer$ $[MDTree]$
$left - pointer$ $[MDTree]$
$root - pointer$ $[MDTree]$
$child - pointer$ $[TreeNode]$
$parent - pointer$ $[TreeNode]$
$sibling - pointer$ $[TreeNode]$
$MDTree$
$TreeNode$
$TreeNode$
$TreeNode$

Figure 5.1: The overall structure of the implementation: The modular decomposition trees are in a double linked list to represent the forest. Every modular decomposition tree has a pointer to its root node. The tree nodes have a pointer to their parent node and a pointer to one child node. The other children are attached to that child node using their sibling pointers

5.1.3 Methods and Concepts

After discussing the data-structures used for the algorithm, this section will provide information on how certain parts of the algorithm were implemented. Chapter 4 already provided information on how to implement the four main steps of the algorithm: Recursion, Refinement, Promotion and Assembly. Thus, we will not discuss these elements of the implementation again. However, some smaller processes of the algorithm are not that straight-forward to implement efficiently and will therefore be analysed here.

5.1.3.1 Constructing Subgraphs

In our algorithm, the construction of subgraphs is of huge importance. During recursion, sets with vertices are created and the procedure uses these sets to build subgraphs and then recursively call itself. Hence, the code for creating subgraphs must work efficiently to contribute to an overall efficient time bound.
Keep in mind, that graphs are represented using adjacency lists. However, this representation requires that the vertices contain values from 0 to $n - 1$, with n being the number of vertices in the graph. A trivial approach for creating a subgraph cannot guarantee this. Let's consider an example: Let G be a graph with five vertices, labeled a, b, c, d, e. These values can be represented using the numbers 0 to 4. Now, recursion wants to create a subgraph based on the vertices a, d and e. As keeping the values of the vertices unchanged is important, the recursive call would have to work with the values $0, 3$ and 4 to calculate its modular decomposition. This contradicts with the representation of an adjacency list and the overall efficiency of the procedure. As a solution to this problem, our implementation

uses an index mapping whenever a subgraph is created. This implies, that - considering the previous example - the values $0, 3$ and 4 get mapped to the values $0, 1$ and 2 in the new subgraph. In addition to this subgraph, recursion gets a mapping containing information on how to change the values after the recursive procedure has finished. In our example, this mapping would contain the entries $0 \to 0, 1 \to 3$ and $2 \to 4$. By using this approach, the recursive procedure can work efficiently and no information about the values of the subgraph's vertices is lost.
Additionally, it is important to notice that the construction of a subgraph requires traversing all vertices. If done too often, this can have a negative influence on the algorithm's time-bound. To solve this problem, our implementation constructs multiple subgraphs at once. In our implementation, recursion does not require a subgraph to be computed immediately after a set N_i has been calculated. Therefore, the algorithm can wait until all vertices have been partitioned into the sets and then construct all subgraphs using only one traversal of the current graph's vertices.

5.1.3.2 Maximal Containing Subtrees

Refinement requires a procedure that can take an ordered list of modular decomposition trees and a set X of values and return the maximal containing subtrees for this set. This implies the following: A node in the forest is returned if and only if all of its leaf - descendants have values that are in X and if this condition is not true for its parent. Figure 5.2 displays an example for this. To be efficient, this work has to be done in an amount of time that is proportional to the values in X. This narrow time bound makes this problem somewhat difficult to solve. An initial approach could start at the root of every tree and recursively call itself on the node's children and siblings. Whenever a leaf node is reached, it could be marked to be included according to its appearance in X and - based on that information - every inner node could include itself if all of its children are marked, or its marked children, if there is at least one unmarked child. However, such an approach would traverse each node of the tree, making it not suitable for our purposes.

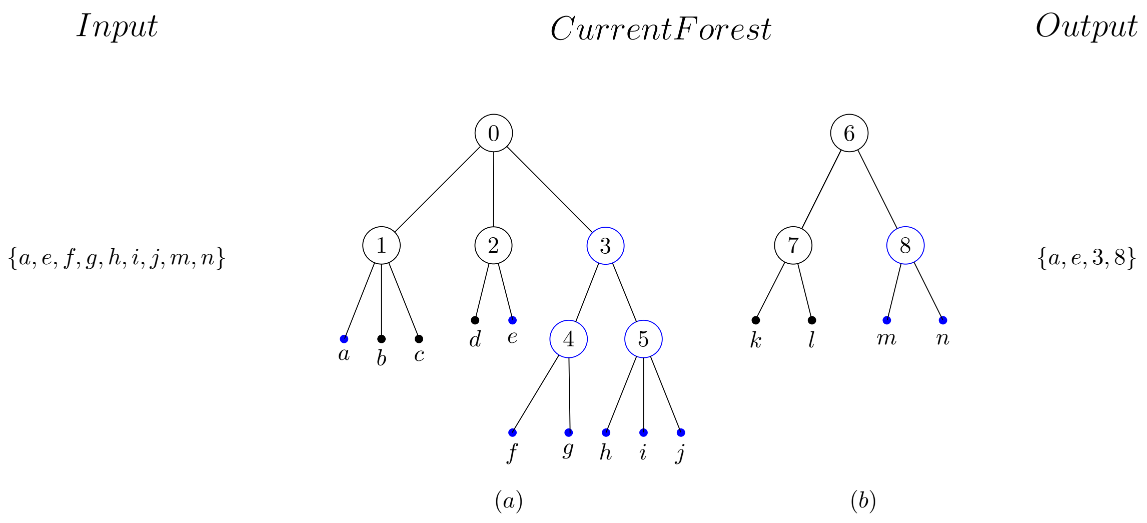

Figure 5.2: An Example for getting the maximal containing subtrees, based on set of node values. Here, the numbers inside of the inner nodes represent indices for these nodes, to be able to distinguish them in the output. Every node that is colored blue is marked to be included in the output

The only way to be able to achieve the desired time bound is to know the location of the leaf nodes that contain the values of X in advance. Thus, our implementation constructs a mapping, that contains the location of the node with the value v for every value v.

This mapping can be seen as global, as it does not belong to one specific recursive step. Every recursive procedure needs access to this mapping to be able to read it and update it according to its progress. On one side, this adds complexity to the algorithm, but on the other side, it allows the set X to be passed as a set of tree nodes, instead of a set of values, as the corresponding node for every value can be found in $O(1)$ time. Hence, the following approach for finding the maximal containing subtrees is possible: Receiving a set of tree nodes as parameter, the method can mark every node in this set as included. Additionally, it increases an inclusion counter in every included nodes parent. A set is used to keep track of every parent node that has been updated. In a recursive manner, these parent nodes can then compare their updated counters to the number of their children (this information has to be known in advance for every node). Based on that comparison, these nodes can mark themselves to be included and increase their parent nodes counter. A second traversal of every updated node can include all nodes, which are marked to be included and have a parent node without that marking. This traversal is also responsible for resetting the inclusion marks and inclusion counters of the nodes.
Using this approach, the implementation manages to find all maximal containing subtrees in the desired time bound and is overall consistent with an efficient time-bound.

5.1.3.3 Determining left- and right-Indices Pointer

The first step in the Assembly process of the algorithm is to determine the left- and right-pointers for every modular decomposition tree in the forest. This requires the trees to know to which other trees they are connected, and to which they are not. An initial approach is to construct a connection matrix, that could be used to look up the connections in a single statement. However, this matrix has a size of t^2, with t referring to the amount of trees in the forest. It is possible for a tree to only consist of one node, which is representing one vertex of the graph. Hence, a worst-case scenario is possible in which the number of trees corresponds to the number of the graph's vertices, making it a quadratic effort to assign a value to each entry in the matrix.
Our solution to this problem, is to only store one array of size t, that is initialized with 'false' values. Every tree of the forest uses this array by setting values to 'true' based on its connections. In addition, it keeps track of the indices that are set to 'true' and the maximal index that is set. Then, the left-pointer of this tree is determined by looping only over 'true' values and the right-pointer is determined using the maximal 'true' index in $O(1)$. After these pointers have been established, the array gets reset using the 'true' indices that were determined earlier.
This approach manages to keep the work that is done for every tree proportional to the amount of 'true' values. These values are limited by the connections of the trees, which makes the method as efficient as possible.

Overall, this section covered some details of the implementations. Even though there are multiple other methods and concepts involved, mentioning them all would take way too long and not be the purpose of this work. The three given examples should emphasize that a lot of the work for developing a modular decomposition algorithm goes into making sub-algorithms keeping up with the time-bound and the overall algorithm as efficient as possible.

5.2 Evaluation

In the following, we evaluate the performance of the previously described implementation. We split the evaluation process into two sub-parts. In Section 5.2.1, we ensure the implementation's correctness and in Section 5.2.2, we will analyse the implementation's running time and time-complexity.

5.2.1 Correctness

Guaranteeing, that the implemented algorithm works correctly is an essential task. If a user cannot be assured, that the calculated output is the correct modular decomposition tree for a given input graph, the usability of the algorithm suffers drastically. Hence, we have placed significant emphasis on examining the correctness of the algorithm through the implementation of a two-step process designed for precisely this purpose.

Our first step is the more complicated one, as it involves several new algorithms, which are placed in a *Util*-class of our implementation. The overall process of this step starts by using the method *generateRandomModularDecompositionTree*. This method takes a number of vertices as parameter and proceeds to generate a random modular decomposition tree with that amount of leaf-nodes. Here, it is important to assure that the generated modular decomposition tree is valid. This infers, i.e. that the tree does not contain any SERIES or PARALLEL nodes with less than two children or any PRIME nodes with less than four children. Furthermore, there cannot be a pair of a parent- and a child node with the same label (PARALLEL, SERIES). This tree-generation contains sub-steps as well, as it starts by generating co-graphs and then proceeds to generate graphs without any limitations. Co-graphs are those, which have only PARALLEL and SERIES nodes in their modular decomposition tree. Let's call this generated tree $T_{Initial}$.
After a random tree has been generated, a second algorithm, called *convertToAdjacencyList*, takes the generated tree and converts it into a graph G_T, represented by its adjacency list. This is achieved by considering the inner node $N_{u,v}$ in the modular decomposition tree for every pair (u, v) of vertices, which has both, u and v as descendants. This property must not be true for any child node of $N_{u,v}$. If this node is PARALLEL, u and v are disconnected in the graph, if it is SERIES, u and v are connected. If the node is PRIME, u and v might be connected or disconnected. Our algorithm handles PRIME nodes by determining that u and v are connected if and only if the two child nodes of $N_{u,v}$, N_u and N_v, which are either u or v, or have u or v as descendant, are placed directly next to each other in the list of $N_{u,v}$'s child nodes. This infers that the connections in a PRIME node P are seen as a simple path, going from P's left-most to its right-most child-node. Using this definition, the algorithm can handle PRIME nodes as well.
In a third step of the process, our implementation of the modular decomposition algorithm by Tedder, Corneil, Habib and Paul computes a modular decomposition tree, $T_{Computed}$, based on the previously generated graph. Afterwards, a fourth algorithm, *sortTree*, is used to sort both modular decomposition trees, $T_{Initial}$ and $T_{Computed}$. Sorting a modular decomposition tree infers, that the children of every node are sorted ascendingly. This is trivial for leafs, as their value is used for representation. Inner nodes are represented by the smallest value of any of their leave-descendants. After this sorting process has finished, the string representations of $T_{Initial}$ and $T_{Computed}$ can be compared for equality, to determine if the modular decomposition algorithm has worked correctly. This is possible, based on the fact that the modular decomposition tree for any graph G is unambiguous, up to isomorphism. That means, that any two correct modular decomposition trees T_1 and T_2 of G can only differ in the order of each of their inner nodes' children.
Figure 5.3 shows the steps of this procedure in a visually simpler way.

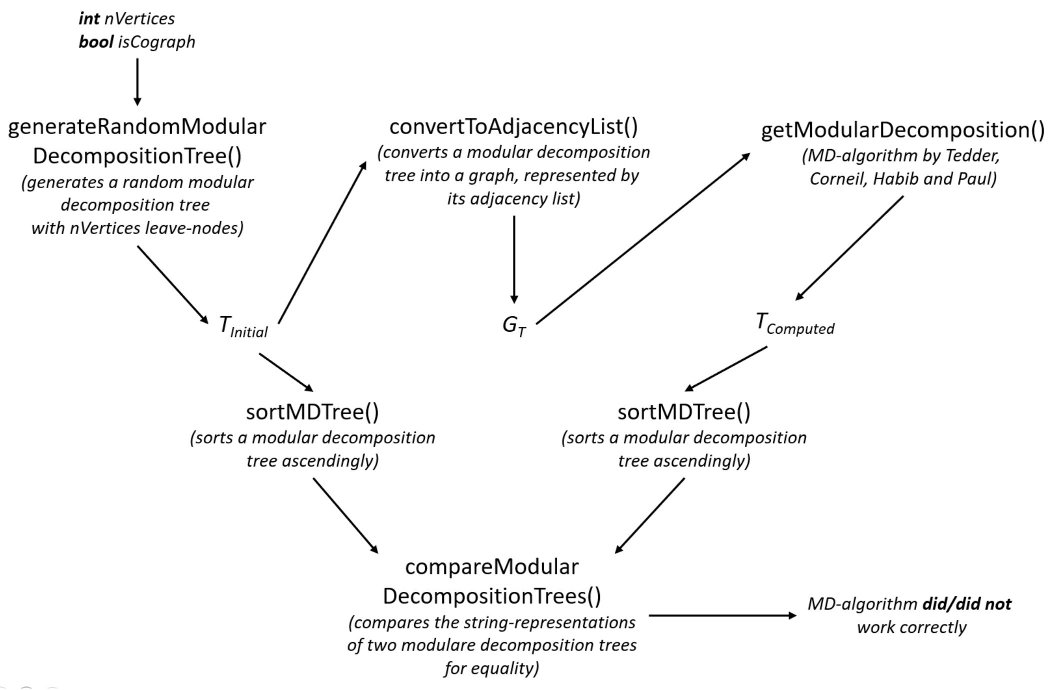
int nVertices
bool isCograph
generateRandomModular DecompositionTree()
(generates a random modular decomposition tree with nVertices leave-nodes)
convertToAdjacencyList()
(converts a modular decomposition tree into a graph, represented by its adjacency list)
getModularDecomposition()
(MD-algorithm by Tedder, Corneil, Habib and Paul)
$T_{Initial}$
G_T
$T_{Computed}$
sortMDTree()
(sorts a modular decomposition tree ascendingly)
sortMDTree()
(sorts a modular decomposition tree ascendingly)
compareModular DecompositionTrees()
(compares the string-representations of two modulare decomposition trees for equality)
MD-algorithm did/did not work correctly

Figure 5.3: A visualization on how the previously described procedure for ensuring the algorithms correctness works

Our implementation has been tested using at least 1000 repetitions for every $n \in \{10, ..., 100\}$, 100 repetitions for every $n \in \{100, ..., 500\}$ and 10 repetitions for every $n \in \{500, ..., 1000\}$. Here, n represents the number of leaf-nodes in the generated modular decomposition tree, and therefore the number of vertices in the graphs.

The second procedure that ensures the correctness of the implementation is based on *OGDF*. The *Open Graph Drawing Framework (OGDF)* is a comprehensive C++ library designed to facilitate the development and implementation of graph-related algorithms. OGDF offers efficient algorithms for generating simple, connected graphs with n vertices and m edges.
In our testing environment, several graphs with $n + m \in \{10, ..., 10000\}$ and $m \in \{1.5n, 2n, 4n\}$ were generated using this framework. In each test, one of these graphs was used as input for our implementation of the modular decomposition algorithm. To ensure the outputs correctness, a function *testModularDecompositionTree*, which takes the original graph and the computed modular decomposition tree as inputs was used. Similar to the function generating a graph based on a given tree in the previous procedure, this function determines the inner node $N_{u,v}$ in the modular decomposition tree for every pair (u, v) of vertices, which has both, u and v as descendants and which has no child node with this property. The function establishes the generated tree to be wrong, if any pair (u, v) is connected in the graph and $N_{u,v}$ is of type PARALLEL, or if any pair (u, v) is disconnected in the graph and $N_{u,v}$ is of type SERIES. To ensure the correctness in the case of $N_{u,v}$ being a PRIME node, a slightly more complicated process has to be taken, as the algorithm cannot know, whether u and v are connected in the modular decomposition tree or not. Therefore, our testing algorithm starts by assuming that u and v are either connected or disconnected, based on their adjacency in the graph. However, if the child of $N_{u,v}$ with u as descendant or the child of $N_{u,v}$ with v as descendent is not the LEAF node with u/v itself, then all other vertices that are in the subtree induced by those children of $N_{u,v}$ must be connected/disconnected to u/v as well. Thus, our algorithm uses a two-dimensional matrix

M to save this information for any vertices that can be found in any of those subtrees. Afterwards, if the connections of another pair (u', v') of vertices are checked, and $N_{u',v'}$ turns out to be the same PRIME node as $N_{u,v}$, the connection of (u', v') in the graph is not allowed to be different as the saved connection-information for u' and v' in M, as this would indicate a wrongly calculated modular decomposition tree. Using this, our testing algorithm can determine the tree to be wrong in the case of $N_{u,v}$ being a PRIME-node for two vertices u and v as well.

As neither this function, nor the previously described procedure was able to detect a wrong output for any of the given inputs, we assume our algorithm to work correctly for all input graphs.

5.2.2 Running Time and Time-Complexity

To evaluate the running time of our implementation, we used the cluster *Zeus* at the *University of Passau*. This cluster has the following hardware:

- Number of nodes: 11 (of 24)
- CPUs per node: 2x Intel Xeon E5-2650v2 @ 2.60 GHz
- CPU cores (hyperthreads) per node: 2x8 (2x16)
- GiB RAM per Node: 256
- CPU market release: 2013

Similar to the second procedure in the previous section, we used *ogdf* to generate random simple, connected graphs with n vertices and m edges. In our evaluation, it applies that $n \in \{10, ..., 10000\}$ and $m \in \{1.5n, 2n, 4n, \frac{n^2}{3}\}$. Figure 5.4, Figure 5.5, Figure 5.6 and Figure 5.7 show the results of the evaluations respectively.

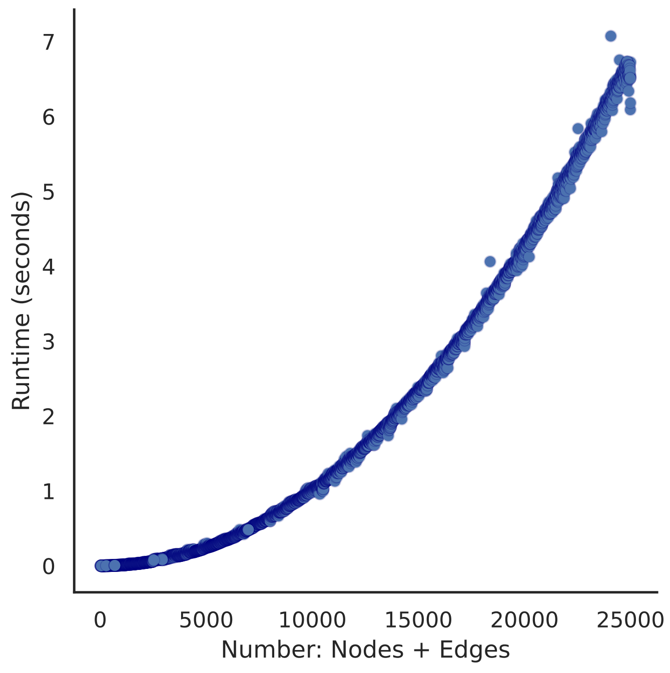

Figure 5.4: The algorithms performance on graphs with $n \in \{10, ..., 10000\}$ vertices and $m = 1.5n$ edges

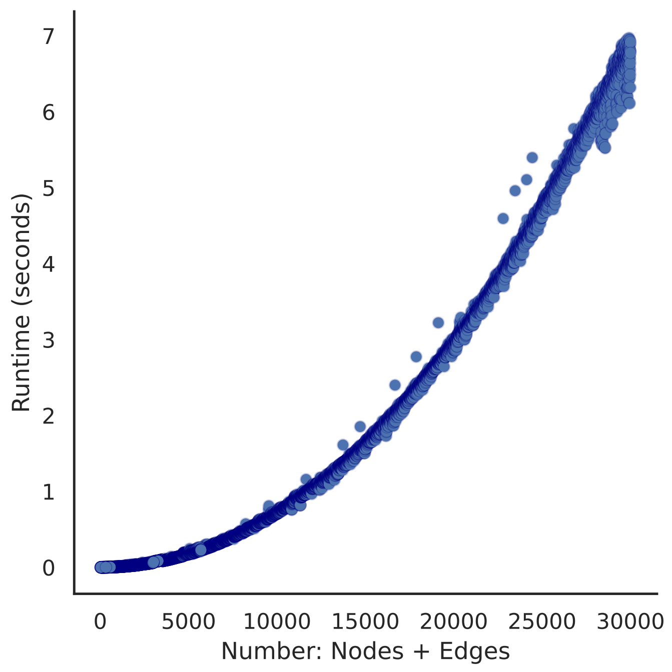

Figure 5.5: The algorithms performance on graphs with $n \in \{10, ..., 10000\}$ vertices and $m = 2n$ edges

These results demonstrate that, unfortunately, our implementation cannot keep up with the linear time-bound promised by Tedder, Corneil, Habib and Paul [TCHP08]. With the exception of $m = \frac{n^2}{3}$, the time-complexity of our implementation currently shows to be quadratic in the number of vertices and edges of the graph. Therefore, the rest of this chapter will analyse the running time in our implementation and prove, that the implementation's time-complexity is actually within quadratic bounds. However, as we - as well as the algorithm's authors [TCHP08] - are confident, that a linear-time implementation of the described algorithm is possible, we will use the next chapter to share ideas and thoughts on how the implementation could be improved to achieve this time complexity.

5.2.2.1 Recursion and Active Edges

To be able to distinguish a graph's vertices by the distance to the pivot-element, our implementation uses a breadth-first search at every recursive step. A breadth-first search is well known to take $O(n + m)$ time, as it processes every vertex once and every edge twice. The active edges, that are used in refinement can be calculated without any additional costs, as the distance of two vertices u and v to the pivot can be checked, when the edge (u, v) is processed. An additional effort comes in the calculation of the subgraphs. Here, our implementation processes each vertex (except for the pivot) and puts it into a new subgraph, based on its distance to the pivot. Placing a vertex into a new subgraph requires the traversal of its adjacency list, therefore this process takes $O(n + m)$ time as well. This additional effort happens in every recursive substep.
The first step of the recursion has to consider all vertices (and edges). In the next step, several recursive process are created, for all vertices with different distances to the pivot. Let's call these recursive processes $p_1, ...p_i$, with i being the maximal distance to the previous pivot-element x. So, let v be a vertex, that is not the pivot element from the previous recursive step. Then v has exactly one distance to x and can therefore be found at exactly one of the p_k. The previous pivot x however, cannot be found in $\{p_1, ...p_i\}$. Therefore, each recursive substep has to consider at most $h - 1$ vertices, with h being the

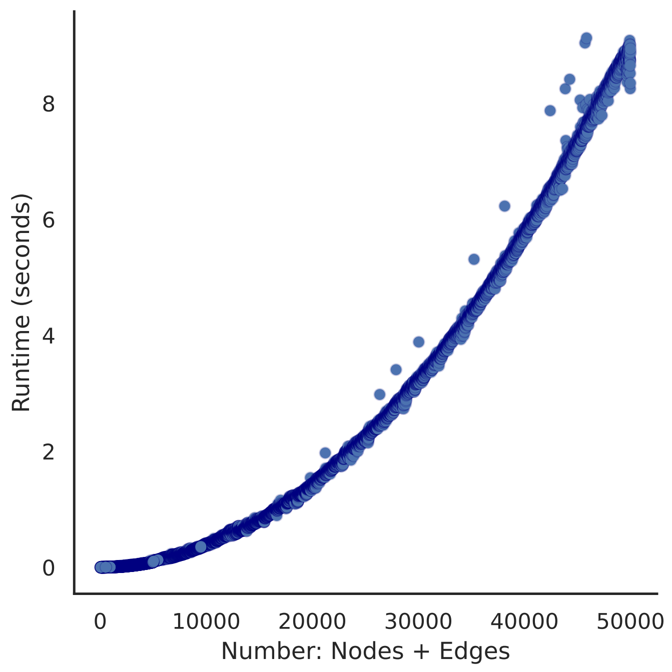

Figure 5.6: The algorithms performance on graphs with $n \in \{10, ..., 10000\}$ vertices and $m = 4n$ edges

amount of vertices of the previous recursive substep. The edges are used along with the vertices they are connected to, therefore m reduces in size in the same manner as n in every recursive substep.
Hence, there are overall n recursive substeps, where step i has to consider at most $n - i$ vertices, along with their edges. This averages out to a worst-case scenario of $\frac{n}{2}$ vertices and $\frac{m}{2}$ edges per substep. This combines to an overall quadratic running time of $O(n \cdot \frac{n+m}{2})$, equal to $O(n \cdot (n + m))$ for the process of recursion and the calculation of the active edges in our current implementation of the algorithm.

5.2.2.2 Refinement, Promotion and Assembly

As described in the previous section, our algorithm is bottle-necked by the step of recursion requiring a quadratic amount of time. This infers, that the time-complexities of Refinement, Promotion and Assembly do not have to be inspected that strictly. As long, as none of them exceeds a quadratic time-bound, the overall time-complexity of the implementation will not change. Thus, we will show that none of these steps surpasses a quadratic running time, even though all of these steps are likely to have a much better time-complexity.

As already discussed in the previous section, the overall size of the graph decreases every recursive substep by at least one vertex. Therefore, there are at most n recursive substeps during the whole calculation. Each of these substeps might consist of multiple recursive calls, but the sum of the vertices in these calls can never exceed n. Thus, to prove a quadratic running time for Refinement, Promotion and Assembly, it is sufficient to show that each of their recursive substeps has a linear time-bound.

Refinement takes the recursively calculated modular decomposition trees and refines them using the previously calculated active edges and left-flags. This step involves finding the maximal containing subtrees, based on a set X and their parents $p_1, ..., p_n$. After this is done, the algorithm partitions the parent node's children into two sets A and B, based on their appearance in the set of maximal containing subtrees and constructs new trees T_a

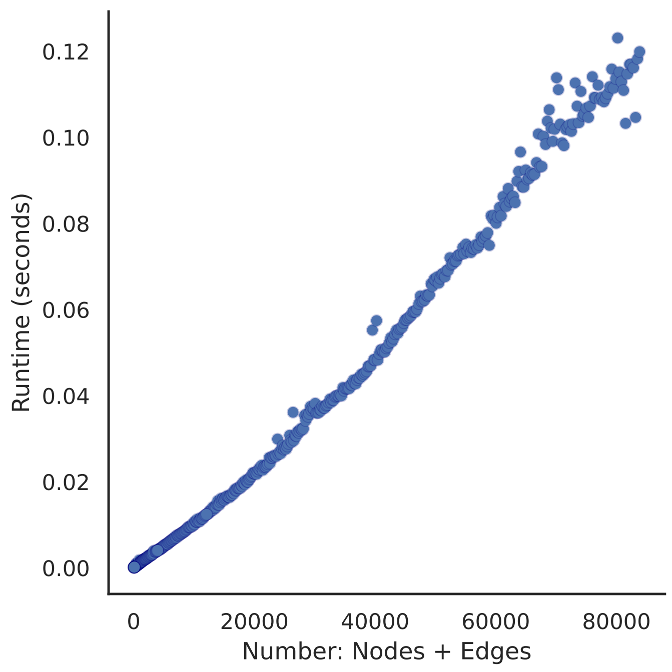

Figure 5.7: The algorithms performance on graphs with $n \in \{10, ..., 500\}$ vertices and $m = \frac{n^2}{3}$ edges

and T_b accordingly. In a next step, refinement either inserts nodes to separate children from different modules or splits a tree into two new trees. Finally, nodes along with their children and ancestors get marked with either "left" or "right".
Let's consider one recursive substep of Refinement. This substep processes each vertex v and uses time proportional to the number of v's active edges. Each edge $e = (u, v)$ can be reached at most twice (once from u, once from v), thus the work done in this iteration is bounded from above by $2m$. Hence, one substep of Refinement uses h (active) edges, with $h \leq 2m$. These h edges are grouped into sets, based on the vertices they are adjacent to. Let's consider one of these sets S, which contains k (with $k \leq h$) edges: Finding the maximal containing subtrees based on the edges in S takes a time proportional to k, due to every inner node in our forest having at least two children and due to our approach that starts at the corresponding leaf-nodes in the tree and progresses "upwards" to the common ancestors. Finding the parent nodes can be achieved in $O(1)$ time by using parent-pointers at every node. Partitioning a node's children into sets A and B requires time proportional to the amount of these children, which is overall linear in the number of vertices in the graph. Using a list-structure of trees, replacing a tree with two new trees or inserting two new children for an inner node can be achieved within a constant amount of time. Considering the process of marking, notice that every node can be marked at most twice (once for "left" and once for "right"). The number of nodes in all trees is proportional to n. Hence, the work done in one recursive substep is in $O(n + m)$. Having at most n substeps, the time needed for Refinement is within quadratic time-bounds.

Promotion is responsible for further refining the forest, based on the "left" and "right" markings of the node. In a first step, promotion detects pairs of parent- and child nodes with the same kind of marking and then separates the child node from its parent and creates a new tree with the child node as its root. Finally, the new tree has to be inserted into the forest either before, or after its original tree, based on the marking. After this is done, promotion has to remove nodes with no children, promote nodes if their parent has

only one child and delete markings.
During this process, only the roots of all trees, along with their children, have to be considered. The number of roots and their children in the forest of one recursive substep is bounded from above by n, as there can be at most $\frac{n}{2}$ inner nodes in a tree with n leave nodes. Thus, it is easy to see that the work needed for one recursive substep of Promotion is bounded from above by a time proportional to n. Having at most n substeps, the time needed for Promotion is within quadratic time-bounds as well.

Assembly is the final step of the modular decomposition process. It starts by indexing left- and right-pointers to every maximal module, which is represented by a tree in our ordered forest. The final task is to assemble all maximal modules of the ordered forest into a modular decomposition tree, by starting at the pivot x and working "upwards" in the tree.
Let's consider one recursive substep of Assembly. Due to the process explained in Section 5.1.3.3, we use a time proportional to a tree's connections for indexing its left- and right-pointers. As these connections originated from the adjacencies of a tree's vertices, the amount of time used for indexing the pointers of all trees is at most proportional to m. Considering the assembly of the final modular decomposition trees of the current recursive substep, notice that the time spent for this process is proportional to the amount of trees in the current forests. This number is bounded from above by n. As a following, the time to perform one recursive substep of Assembly is in $O(n+m)$. Having at most n substeps, the time needed for Assembly is also within quadratic time-bounds.

As none of the steps surpasses a running time of $O(n \cdot (n+m))$, our implementation has a quadratic time-complexity. This is the result of both, theoretical analysis and practical evaluation. As this implementation is not optimal however, we will use the next chapter to share our thoughts on how a linear implementation of this algorithm can be achieved.

6. Towards Linear Time

In the previous chapter, we showed that our implementation for the algorithm by Tedder, Corneil, Habib and Paul [TCHP08] needs a quadratic amount of time to complete. However, this implementation is not optimal in regards to its time-complexity. Thus, we want to use this chapter to share our thoughts on how a linear-time implementation of the algorithm could be achieved. We will put our focus on Step 1: Recursion, as this step served as a bottleneck in the implementation shown in Chapter 5.

In the following, we will show that it is possible to execute the step of recursion in $O(n+m)$ time, by presenting an approach that does exactly that:
Instead of performing a breadth-first search at every recursive substep, it is possible to achieve the same results by only performing one breadth-first search over all recursive levels. In this more complicated version of the algorithm, only the adjacency list of the pivot element must be checked at every recursive substep. In addition, no subgraphs must be computed, instead every recursive step receives the whole graph and a vector of bool-values of size n, representing which of the vertices are in the current subgraph. Furthermore, a *pullForward*-set of vertices has to be passed between the recursive steps. A global *previousPivots* bool-vector of size n has to be accessible from every recursive substep as well. Then, each recursive step does the following: It traverses the adjacency-list of the pivot. For every adjacent vertex v in this list it checks, whether v can be found in the current subgraph. In the positive case, v is put in a new list, containing the elements that the pivot is connected to. Otherwise, v is put in the current *pullForward*-set, but only if it has not been a pivot in a previous step. This information can be obtained using the *previousPivots*-vector. Of course, each recursive step has to add its pivot to this vector before making any recursive calls. As soon as the pivot's adjacency list has been fully traversed, the modular decomposition tree of the elements to the pivot's left can be computed recursively.
Using this approach, every recursive step can obtain the modular decomposition tree of the elements that are connected to its pivot (Recall, that the recursion becomes trivial in case of a subgraph with a single vertex), along with a list of all remaining vertices that are connected to any of the vertices in the computed tree, due to the *pullForward*-procedure. Now, this list can be used to compute the next modular decomposition, containing the elements with distance 2 to the pivot, pulling the next elements forward in a similar fashion. By continuing this procedure, all further modular decomposition trees are calculated likewise.
This approach, even though being more complicated to implement, manages to achieve the

process of recursion in $O(n+m)$ time, as its work over all recursive levels is similar to one breadth-first search. Algorithm 6.1 sketches a recursive function that could be used for this approach. This function gets called at every recursive step.

Algorithm 6.1: Recursion

```
Data: The full input graph $G$,
A *currentSubgraph*-set of vertices representing the current subgraph,
A global *previousPivots*-set of vertices containing the pivot-elements of all
previous recursive steps
Result: The modular decomposition tree of the given subgraph,
A *pullForward*-set containing all vertices that are connected to any element in
the current subgraph
1  Select an arbitrary vertex $x$ from the current subgraph as pivot
2  Insert $x$ in the *previousPivots*-set
3  foreach adjacency adj in x's adjacency list do
4      if adj is in the current subgraph then
5          Insert *adj* into a *neighbor*-set $N$
6      else
7          if adj is not in the previousPivots-set then
8              Insert *adj* in the *pullForward*-set $P$
9  if $N$ is empty (the current subgraph contains only the pivot) then
10     return a trivial modular decomposition tree with $x$ as root and $P$ as
         *pullForward*-set
11 else
12     Make a recursive call to this function, passing $G$ and the *previousPivots*-set,
         as well as $N$ as new subgraph. Receive the modular decomposition tree $T_0$
         and the *pullForward*-set $P_i$
13     while $P_i$ is not empty do
14         Add the vertices of $P_i$ to $P$
15         Make a recursive call to this function, passing $G$ and the
             *previousPivots*-set, as well as $P_i$ as new subgraph. Receive the modular
             decomposition tree $T_i$ and the *pullForward*-set $P_{i+1}$
16         Set $P_{i+1}$ as new $P_i$
17     Use Refinement, Promotion and Assembly to calculate the current modular
         decomposition tree $T_{MD}$, based on the list $T_0, x, T_1, ...T_k$, where $T_1, ...T_k$
         represent the calculated $T_i$
18     return $T_{MD}$ as modular decomposition tree and $P$ as *pullForward*-set
```

This algorithm assumes that the given graph G, as well as the current subgraph is connected. Handling disconnected graphs however adds only a constant amount of work to each stage. Furthermore, this approach also relies on Refinement, Promotion and Assembly to have an upper time-bound of $O(n+m)$ over all recursive stages to work in linear time.
Overall, however - based on the fact that each vertex v becomes the pivot-element exactly once throughout the whole procedure and that each recursive step takes time proportional to the number of the current pivot's adjacencies - this approach manages to run within a linear time-bound.

Calculating the active edges for each recursive call adds a constant amount of work as well: Remember that these edges connect vertices with different distances to a pivot x. So,

let $e = (u, v)$ be an edge of the input graph G. During recursion, e is processed exactly twice, once from the adjacency list of u, and once from the adjacency list of v. During the recursive substeps that are centered around u or v being the pivot, it is possible to determine whether e is active or not in a constant amount of time: Let v be the pivot element of a current recursive substep: If u is present in the current subgraph, then e is an active edge in the current substep. Otherwise, e can be added as active edge to the previous substep.
Notice, that e is active exactly once during the algorithm. If e is active in a recursive substep S, then it is not present in any of the subgraphs created in S, as u and v cannot be placed in the same.
Hence, the active edges can be determined during recursion while adding only a constant amount of effort to each stage. Therefore, the calculation of the active edges is consistent with linear time overall.

As already mentioned, the algorithm can only stay within a linear time-bound, if Refinement, Promotion and Assembly stay within $O(n+m)$ as well. The previous chapter showed, that these processes finish within quadratic time-constraints, however, we are confident that they are in fact a lot faster. The rest of the chapter will present thoughts on that topic.

Refinement contains a loop over all vertices of the current subgraph at every recursive substep. On a first thought, the sum of all these loops is way to large for our time-constraint. However, the vertices in the loops are only used to gather a set of their incident active edges. We already established, that every edge is active only once throughout the whole procedure. Once an edge $e = (u, v)$ is active, it appears exactly twice in further calculations: First within the set of u's active edges, and then within the set of v's active edges. If we manage to skip all vertices that contain no incident active edges in the loops of Refinement, the effort of the sum of all these loops is proportional to the number of edges in the graph. Progressing further, a set X has to be filled with the common ancestors of n active edges. By keeping track of the placement of all leaf nodes, the algorithm can start by considering the corresponding leaf-nodes and progress "upwards" in the tree. This makes the amount of work for finding the set X proportional to n. Due to every node in the trees having at least two children, the size of each subtree is linear in the number of its leafs. The number of leafs is equal to the number of incident active edges of the vertex being refined, thus this effort is consistent with linear time.
Finding the parent node p of a subtree can be achieved in $O(1)$, using parent-pointers for every node. Splitting p's children into the sets A and B takes time proportional to the amount of p's children. Due to the tree-structure, the amount of the children of all parent nodes combined is bound by the overall number of vertices in the current (sub)graph, which keeps this effort in linear time as well.
By using a linked-list to represent the forest, replacing a tree with two other trees can be achieved in $O(1)$ time as well. Furthermore, it is important to notice that the children of a prime node need only to be marked once throughout the whole procedure, and the ancestors of a node are marked at most twice (one marking for "left", and one for "right"). Thus, the time for marking is proportional to the size of the forest, which is linear in the number of its leafs. As a conclusion, the process of refinement can be implemented to be consistent within the aim of linear time.

At every recursive substep, a single traversal of the current ordered forest is sufficient for the task of promotion. During this traversal, only the root nodes of each tree have to be checked, which makes the effort of the process consistent in the number of trees

(after Promotion) in the forest of the current substep. Using a list-structure, the process of removing a connection and inserting a new tree can be achieved in $O(1)$ time. As the amount of trees in all recursive substeps combined is proportional to the amount of vertices in the input graph, this effort is consistent with linear time.
A second traversal of the ordered forest can delete roots with no children and promote children of roots having only one child by exclusively considering the roots of every tree. Using the same arguments as before, this traversal is consistent with overall linear time as well.
Considering the process of mark-deletion, notice that a node can only be marked if its parent node is marked as well, making it possible to stop with the process whenever the root of a tree is not marked. Children of unmarked nodes can be excluded from mark-deletion likewise. Therefore, this process is constant in the number of marked nodes, which is consistent with linear time.
As every traversal of the forest Promotion uses is consistent with linear time, and there is only a constant amount of traversals, the whole process of Promotion can be implemented to be consistent within linear time as well.

Regarding Assembly, the procedure of checking each maximal module's connections uses a time proportional to the graph's vertices by only keeping track of the connected maximal modules. As keeping track of the maximum index of all connected maximal modules is sufficient for indexing the right-pointer, the algorithm does not have to loop over disconnected modules at any step of the process.
Considering the left- and right-pointers of all maximal modules to determine which of them are to be placed in the current module takes time proportional to the amount of trees at a single recursive step. Using the same arguments as before, the work of this process and therefore assembly is overall achievable in linear time.

This chapter hopes to show how a linear-time implementation of the algorithm by Tedder, Corneil, Habib, and Paul could look like. We are optimistic that future work will provide such an implementation, thereby advancing the efficiency and applicability of the algorithm in diverse computational contexts.

7. Conclusion

This thesis provided a working implementation of the modular decomposition algorithm by Tedder, Corneil, Habib and Paul [TCHP08]. To be able to understand this algorithm, we started by introducing the reader to modular graph decomposition and describing its functionality. We proceeded to show several applications of this decomposition in and out of graph theory, like its contribution in handling some NP-hard problems [Utk17], its usage in transitively orienting undirected graphs [MS00], its familiarity with permutation graphs [PLE71], or its applications in graph drawing [PV07] and biological network analysis [SPH14]. We gave an overview of established modular-decomposition algorithms, using the first MD-algorithm by James, Stanton and Cowan [JSC72], the incremental MD-algorithm by Muller and Spinrad [MS89], the early linear-time approach by Cournier and Habib[CH94] or the sequential MD-algorithm by Dahlhaus, Gustedt and McConnell [DGM01] as representative examples. As a center of this work, we provided a larger-scale analysis of the simple, linear time algorithm by Tedder, Corneil, Habib, and Paul [TCHP08], along with the calculation this algorithm does on an example input graph. In the following, we introduced our implementation of this algorithm, which was analysed and tested to run within quadratic time-bounds. The correctness of this implementation was evaluated using several procedures and several classes of graphs, with graphs containing up to 10000 vertices and edges. In the end, we shared our thoughts on an approach to implement the algorithm with a linear time-constraint, which will hopefully see further developments in future research.

As a summary, efficient algorithms for modular graph decomposition, exemplified by Tedder, Corneil, Habib, and Paul's work [TCHP08], have broad applications. As we move forward, these algorithms can contribute to large-scale graph analysis, network modeling, biological network research, graph visualization, and algorithmic innovations. The scalability and efficiency of such algorithms are crucial for addressing the increasing complexity of real-world network data. In essence, the continued development of modular decomposition algorithms can unlock new possibilities and extend their impact across diverse domains.

Bibliography

[CE80] William H. Cunningham and Jack Edmonds. A combinatorial decomposition theory. *Canadian Journal of Mathematics*, 32(3):734–765, 1980.

[CH94] Alain Cournier and Michel Habib. A new linear algorithm for modular decomposition. In Sophie Tison, editor, *Trees in Algebra and Programming - CAAP'94, 19th International Colloquium, Edinburgh, UK, April 11-13, 1994, Proceedings*, volume 787 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 1994.

[CHM81] M. Chein, M. Habib, and M.C. Maurer. Partitive hypergraphs. *Discrete Mathematics*, 37(1):35–50, 1981.

[CPS85] D. G. Corneil, Y. Perl, and L. K. Stewart. A linear recognition algorithm for cographs. *SIAM Journal on Computing*, 14(4):926–934, 1985.

[Dah95] Elias Dahlhaus. Efficient parallel recognition algorithms of cographs and distance hereditary graphs. *Discrete Applied Mathematics*, 57(1):29–44, 1995.

[DGM01] Elias Dahlhaus, Jens Gustedt, and Ross M. McConnell. Efficient and practical algorithms for sequential modular decomposition. *J. Algorithms*, 41(2):360–387, 2001.

[EGMS94] Andrzej Ehrenfeucht, Harold N. Gabow, Ross M. McConnell, and Stephen J. Sullivan. An $O(n^2)$ divide-and-conquer algorithm for the prime tree decomposition of two-structures and modular decomposition of graphs. *J. Algorithms*, 16:283–294, 1994.

[Gal67] T. Gallai. Transitiv orientierbare graphen. *Acta Mathematica Academiae Scientiarum Hungarica*, 18:25–66, 1967.

[GKBC04] Julien Gagneur, Roland Krause, Tewis Bouwmeester, and Georg Casari. Open access method modular decomposition of protein-protein interaction networks. *Genome biology*, 5:R57, 02 2004.

[HM79] M. Habib and M.C. Maurer. On the x-join decomposition for undirected graphs. *Discrete Applied Mathematics*, 1(3):201–207, 1979.

[JSC72] L.O. James, R.G. Stanton, and Donald Cowan. Graph decomposition for undirected graphs. *Utilitas Mathematica*, 01 1972.

[Möh85] Rolf H. Möhring. Algorithmic aspects of comparability graphs and interval graphs. In Ivan Rival, editor, *Graphs and Order: The Role of Graphs in the Theory of Ordered Sets and Its Applications*, pages 41–101. Springer Netherlands, Dordrecht, 1985.

[MR84] Rolf Möhring and Franz Radermacher. Substitution decomposition for discrete structures and connections with combinatorial optimization. *North-Holland Mathematics Studies*, 95:257–355, 12 1984.

[MS89] John H. Muller and Jeremy P. Spinrad. Incremental modular decomposition. *J. ACM*, 36(1):1–19, 1989.

[MS94] Ross M. McConnell and Jeremy P. Spinrad. Linear-time modular decomposition and efficient transitive orientation of comparability graphs. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, page 536–545. Society for Industrial and Applied Mathematics, 1994.

[MS00] Ross M. McConnell and Jeremy P. Spinrad. Ordered vertex partitioning. *Discret. Math. Theor. Comput. Sci.*, 4(1):45–60, 2000.

[Pau23] Christophe Paul. Personal Communication, 2023.

[PLE71] A. Pnueli, A. Lempel, and S. Even. Transitive orientation of graphs and identification of permutation graphs. *Canadian Journal of Mathematics*, 23:160–175, 1971.

[PV07] Charis Papadopoulos and Constantinos Voglis. Drawing graphs using modular decomposition. *J. Graph Algorithms Appl.*, 11(2):481–511, 2007.

[SPH14] Hari Sivakumar, Stephen R. Proulx, and João P. Hespanha. Modular decomposition and analysis of biological networks, 2014.

[TCHP08] Marc Tedder, Derek G. Corneil, Michel Habib, and Christophe Paul. Simpler linear-time modular decomposition via recursive factorizing permutations. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Proceedings of 35th International Colloqium on Automata, Languages and Programming (ICALP'08), Part I: Tack A: Algorithms, Automata, Complexity, and Games*, volume 5125 of *Lecture Notes in Computer Science*, pages 634–645. Springer, 2008.

[Utk17] Irina Utkina. Using modular decomposition technique to solve the maximum clique problem. *CoRR*, abs/1710.04040, 2017.