A Practical Evaluation of Modular Decomposition Algorithms

Master's Thesis of

Jonas Spinner

At the Department of Informatics
Institute of Theoretical Informatics (ITI)

Reviewer:	T.T.-Prof. Dr. Thomas Bläsius
Second reviewer:	PD Dr. Torsten Ueckerdt
Advisors:	Dr. Maximilian Katzmann
	Dr. Christopher Weyand

August 23rd 2023 – February 23rd 2024

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, February 23rd 2024

. . . . . .

(Jonas Spinner)

Abstract

Modular decomposition is an essential preprocessing step for many algorithmic problems, like recognizing graph classes, parameterized graph problems and optimization problems. The modular decomposition tree hierarchically decomposes the graph into modules, sets of vertices with the same neighborhood outside the set. There are linear time algorithms to find the modular decomposition of a graph.

Although several implementations exist, it is not easy to compare their performance. The implementations are in different programming languages, hard to use or implemented inefficiently.

This thesis presents three established algorithms with linear or almost linear running time and discusses similarities in a common framework. Moreover, we provide efficient and comparable implementations for these algorithms, improve upon existing implementations, and share important implementation details.

Finally, we evaluate the algorithms on real-world and generated graphs. We find that an algorithm with a non-optimal $O(n+m \log n)$ running time performs best in practice, because it avoids large constant factors and practical instances are rarely the worst-case. This algorithm computes a factorizing permutation and converts it to a modular decomposition via a fracture tree. Surprisingly, the implementations of the theoretical optimal linear algorithm are often the slowest and sometimes do not follow the expected linear scaling behavior.

Zusammenfassung

Die *Modular Decomposition* ist ein wichtiger Bestandteil vieler Algorithmen, beispielsweise für das Erkennen von Graph-Klassen, parametrisierte Graph-Algorithmen und Optimierungsprobleme. Der Modular Decomposition Baum teilt den Graphen hierarchisch in sogenannte Module auf. Das sind Knotenmengen, für die jeder Knoten die gleiche Nachbarschaft außerhalb der Menge hat. Mittlerweile existieren Linearzeitalgorithmen für die Berechnung der Modular Decomposition.

Zwar gibt es auch praktische Implementierungen, aber es ist schwierig ihre Laufzeit miteinander zu vergleichen. Das liegt unter anderem daran, dass sie in unterschiedlichen Programmiersprachen geschrieben und ineffizient implementiert wurden, oder es kompliziert ist, sie zu nutzen.

In dieser Abschlussarbeit werden drei etablierte Algorithmen vorgestellt und in einem gemeinsamen Kontext miteinander verglichen. Wir stellen effiziente und vergleichbare Implementierungen dieser Algorithmen bereit, diskutieren wichtige Implementierungsdetails und verbessern existierende Implementierungen.

Die Algorithmen wurden auf Echtwelt und generierten Graphen evaluiert. Eine Erkenntnis ist, dass ein Algorithmus mit einer nicht-optimalen $O(n+m \log n)$ Zeitkomplexität in der Praxis am besten abschneidet. Das liegt unter anderem an der Vermeidung von großen konstanten Faktoren und daran, dass es sich bei der Laufzeitanalyse um eine obere Schranke handelt und dass der Logfaktor kaum zum tragen kommt. Der besagte Algorithmus berechnet erst eine *Factorizing Permutation* und mit der Hilfe eines *Fracture Tree*s wird mit dieser dann die Modular Decomposition berechnet. Ein weiterer interessanter Aspekt ist, dass die Implementierungen eines Linearzeit-Algorithmus, meistens am schlechtesten abschneiden. Außerdem zeigen sich in den Experimenten Verhaltensmuster, die nicht mit der Erwartungen einer linearen Laufzeit übereinstimmen.

Contents

1 Introduction

The modular decomposition hierarchically decomposes a graph from very coarse, the whole graph, to very fine, a single vertex. Gallai has first introduced the concept [Gal67]. Informally, modules are sets of vertices with the same neighborhood outside the set. Modules are a generalization of connected components, co-components, and twins.

Modular decompositions have been studied in a variety of different settings, like undirected and directed graphs [CHM02 | MdM05], 2-structures [EGMS94] or permutations[BCdMR05]. In the past, there have been several names for modules, e.g., *autonomous sets*, *closed sets*, or *clans*. We refer to a survey paper for a more exhaustive list [HP10]. In this thesis, we focus on efficient algorithms for the case of undirected graphs.

There is a wide variety of combinatorial problems where modular decomposition is an important preprocessing step [Möh85]. It has been used for adaptive parameterized algorithms for maximum matching, triangle counting, edge-disjoint $s-t$ paths, minimum cut, maximum flow, and more [KN18]. Many graph classes can be recognized with the help of modular decompositions [Gol80], for example, interval graphs [Möh85], permutation graphs [PLE71], or cographs [CPS85]. Algorithms for transitive orientation are closely related as well [MS99 | Ted15]. The modular decomposition has also been used in fields like bioinformatics to study the structure of protein-protein interaction networks [GKBC04]. Additionally, there are applications in pattern matching, computational biology, and parameterized complexity [HP10].

Due to its broad applicability, efficient and practical algorithms for modular decomposition are essential. Moreover, high-performing implementations are crucial for profiting from the usefulness of modular decompositions in practice. The first polynomial algorithm by James, Stanton, and Cowan, in 1972, achieved a worst-case running time of $O(n^4)$. The running time was later improved to $O(n^3)$ by Habib and Maurer [HM79] and $O(n^2)$ by Muller and Spinrad [MS89]. The first two linear-time algorithms have been independently proposed by McConnell and Spinrad [MS94] and Cournier and Habib [CH94]. Later, efforts went into creating simplified algorithms which achieved linear-time [MS99 | DGM01 | TCHP08], or almost linear time $O(n+m\alpha(m,n))$ [DGM01] and $O(n+m\log n)$ [MS00 | HPV99], with $\alpha(m,n)$ being the inverse Ackermann function. For an overview of some additional algorithmic ideas not covered in this thesis, we refer to the survey paper by Habib and Paul [HP10].

Practical implementation efforts and a desire for simplicity accompanied the steady theoretical progress. However, most efficient algorithms are difficult to understand and implement.

The authors of a more recent linear-time algorithm [TCHP08 | Ted11] provided a preliminary Java implementation, which was accessible until recently[1]. In order to analyze protein-protein interaction networks, another algorithm [MS00] has been implemented, but the implementation is no longer available [GKBC04]. There are implementations in Perl[2] [Sal04], C++[3] [FL15], Julia[4] [Kar19], and also a Java implementation used in software for structuring

[1] *https://web.archive.org/web/20231117180242/http://www.cs.toronto.edu/~mtedder/*
[2] *https://metacpan.org/release/AZS/Graph-ModularDecomposition-0.15*
[3] *https://github.com/LyteFM/modular-decomposition*
[4] *https://github.com/StefanKarpinski/GraphModularDecomposition.jl*

process models[5] [Art11]. Unfortunately, they are either not easily usable or need to be implemented more efficiently. More recently, a usable C++ implementation[6] [Miz23] was made available.

Although there has been much theoretical work to produce simple and efficient algorithms and several implementations are available, it has been challenging to compare "efficient and practical" modular decomposition algorithms on actual real-world data from practical contexts. We aim to bridge that gap and provide comparable implementations of multiple algorithms to evaluate their performance.

1.1 Contributions

This thesis presents the theoretical background of three popular modular decomposition algorithms, discusses similarities, and provides details on their implementation. More specifically, we implement the FRACTURE [HPV99 | CHM02], SKELETON [MS00], and LINEAR [TCHP08] algorithms. We improved the performances by several orders of magnitude compared to naive and reference implementations. The code is freely available online[7]. Furthermore, we evaluate the algorithms on a wide variety of real-world and generated data and their scaling behavior for instances with growing size. Overall, the theoretical non-optimal FRACTURE algorithm consistently performs best in practice.

1.2 Outline

In Chapter 2, we introduce notation and concepts and present some of the theoretical background. Then, in Chapter 3, we present three modular decomposition algorithms, FRACTURE, SKELETON, and LINEAR, in Section 3.1, Section 3.2, and Section 3.3, respectively. At the end of each section, we discuss their efficient implementation. The FRACTURE and SKELETON algorithms have a worst-case running time of $O(n + m \log n)$ and are based on [HPV99 | CHM02] and [MS00], respectively. The LINEAR algorithm is based on [TCHP08]. Next, we evaluate our implementations of the algorithms in Chapter 4. Finally, we conclude our findings in Chapter 5.

[5] *https://code.google.com/archive/p/bpstruct/*
[6] *https://github.com/mogproject/modular-decomposition*
[7] *https://github.com/jonasspinner/modular-decomposition*

2 Preliminaries

We consider undirected graphs $G = (V, E)$ with no self-loops and multi-edges unless otherwise noted. We use V and E to denote the vertex set and edge set, respectively, and use $V(G)$ and $E(G)$, when we want to be explicit about which graph these sets belong to. We use uv to denote an edge between $u, v \in V$. An induced subgraph on the vertex set $X \subseteq V$ is denoted with $G[X]$, with $V(G[X]) = X$ and edges $E(G[X]) = \{uv \in E(G) \mid u, v \in X\}$. For a graph G and a partition $\mathcal{P}$ of $V(G)$, we use $G/\mathcal{P}$ to denote the *quotient graph* with vertex set $\mathcal{P}$ and edges $\{XY \mid X, Y \in \mathcal{P}, x \in X, y \in Y, xy \in E(G)\}$. We use $\overline{G}$ to denote the complement of a graph G, with $V(\overline{G}) = V(G)$ and $E(\overline{G}) = \{uv \mid u, v \in V(G), uv \notin E(G)\}$. We say that two sets A and B *overlap*, if $A \cap B \neq \emptyset$, $A \setminus B \neq \emptyset$ and $B \setminus A \neq \emptyset$. We use $A \triangle B$ to denote the symmetric difference between sets A and B.

2.1 Modules

Definition 2.1: *Let $G = (V, E)$ be a graph. A vertex set $M \subseteq V$ is called a* module *if for every vertex $x \in V \setminus M$ either $M \subseteq N(x)$ or $M \cap N(x) = \emptyset$.*

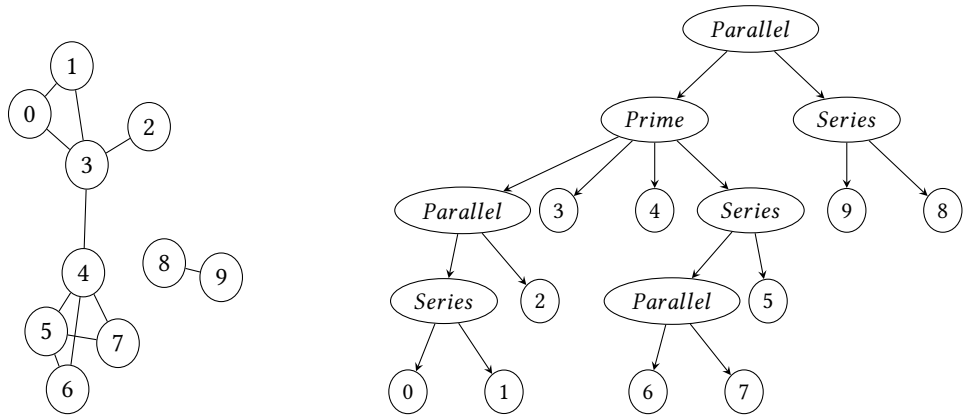
An equivalent definition of a module M is that all vertices in M have the same neighborhood in $V \setminus M$. The sets $\emptyset, V$ and $\{x\}$, for $x \in V$ are modules of G. These are called *trivial* modules. If a graph consists of only trivial modules, it is called *prime*. The family of modules of a graph is partitive. This means that any two overlapping modules M and M', the sets $M \setminus M'$, $M' \setminus M$, $M \cap M'$, $M \cup M'$ and $M \triangle M'$ are also modules.

Definition 2.2: *Let $G = (V, E)$ be a graph and $X \subseteq V$ be a vertex set. A vertex $z \in V \setminus X$ is called a* splitter *of X if there exist vertices x and y, such that $xz \in E$ and $yz \notin E$.*

If a set X has a splitter z, then any module containing X also contains z. This leads to the following additional characterization of modules. A vertex set M is a module if and only if it has no splitters.

Note that there might be an exponential number of modules in a graph. For example, in the complete graph, any vertex set is a module. Fortunately, because the family of modules is partitive, that the set of all modules can be completely described by a $O(n)$ set of modules. A module is called *strong* if it does not overlap any other modules. Strong modules either do not share any vertices, or are subsets of each other. This subset relation results in a hierarchical structure on the set of strong modules: a tree with the set V at its root, $\{v\}$, $v \in V$ as the leaves and size $O(n)$. The tree defined by the strong modules of G and their inclusion relation is called the *modular decomposition tree* $MD(G)$ of G. We use the term *modular decomposition* to refer to that tree and use the term *nodes* for the vertices of the tree, i.e. the strong modules. We use the term *children* for outgoing neighbors of a node. An example for the modular decomposition can be seen in Figure 2.1.

A module is *maximal* with respect to some other set S, if it is the largest subset of S that is a module.



(a) A graph G. **(b)** The modular decomposition $MD(G)$ of graph G.

Figure 2.1: An example of a graph, on the left, and its modular decomposition to its right. The root node is "parallel" and its children correspond to the connected components of the graph.

Definition 2.3: *Let $\mathcal{P}$ be a partition of the vertex set of a graph G. Then, $\mathcal{P}$ is called a modular partition if every part is a module of G.*

A useful modular partition is the following.

Definition 2.4: *A v-modular partition $\mathcal{M}(G, v)$ is the set $\{v\}$ and all maximal modules not containing v.*

The non-trivial modular partition that consists of maximal modules of G is called the *maximal modular partition*. Next, we formulate the following theorem.

Theorem 2.5 (Modular Decomposition Theorem [Gal67])**:** *Let $G = (V, E)$ be a graph. One of the following holds:*

1. *G is connected (series),*
2. *$\overline{G}$ is connected (parallel), or*
3. *G and $\overline{G}$ are disconnected and the graph $G/\mathcal{P}$, with the maximal modular partition $\mathcal{P}$, is prime (prime).*

This classification can also be applied to strong modules by classifying the induced subgraph of the strong module.

To reiterate, the family of modules of a graph is fully defined by its strong modules. The strong modules are structured in a modular decomposition tree $MD(G)$ and the nodes, corresponding to the strong modules, are either series, parallel or prime.

The following results are crucial for algorithms computing the modular decomposition.

Theorem 2.6 ([Möh85])**:** *Let $\mathcal{P}$ be a modular partition of G, then $\mathcal{X} \subseteq \mathcal{P}$ is a module of $G/\mathcal{P}$ if and only if $\bigcup_{M \in \mathcal{X}} M$ is a module of G.*

Theorem 2.7 ([Möh85])**:** *Let X be a module of G. The modules of G that are a subset of X are the modules of $G[X]$.*

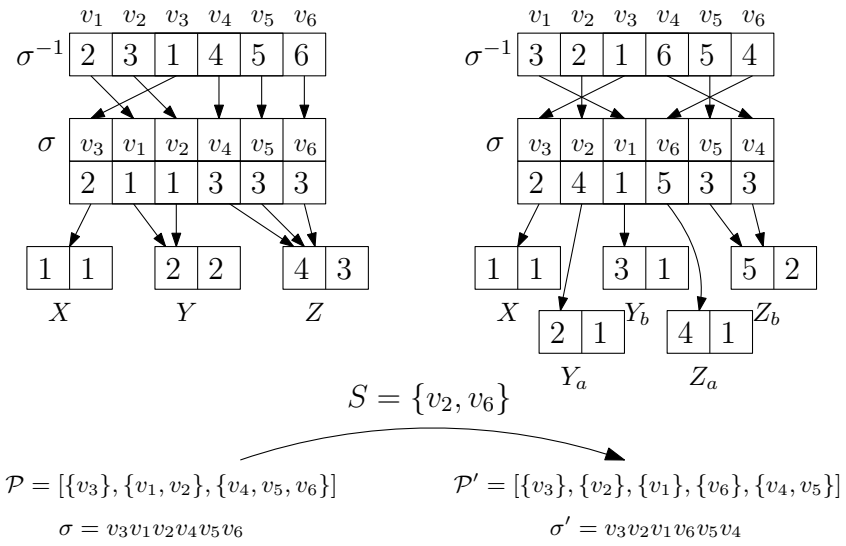


Figure 2.2: The ordered partition data structures. Parts are stored as (index, length) pairs. The permutation and its inverse are stored explicitly. The right side is the result $\mathcal{P}' = \textsc{Refine}(\mathcal{P}, S)$, with the choice to place $X \cap S$ to the left of $X \setminus S$.

Definition 2.8: *Let σ be a permutation of the vertex set V. A set $X \subseteq V$ is called a* factor *if all vertices appear consecutively in σ. The permutation σ is a* factorizing permutation *of G if every strong module M of $MD(G)$ is a factor of σ.*

The concept of factorizing permutation is closely related to modular decomposition [Cap97]. More precisely, one can compute a factorizing permutation in linear time, by doing a preorder traversal of the modular decomposition tree. The reverse is also possible in linear time [CHM02]. Note, that a factorizing permutation might not be unique for a given graph G. For example, any permutation is factorizing for a prime graph.

2.2 An Ordered Partition Data Structure

A central technique is *partition refinement*, which originated from algorithms related to state minimization in a finite automaton [Hop71]. For an overview of applications of this technique, we refer to [HPV99]. In this section we describe the data structure we use in our implementations.

Definition 2.9: *An ordered partition $\mathcal{P} = [P_1, \ldots, P_k]$ is a partition with parts $P_1, \ldots, P_k$ and an order $\leq_{\mathcal{P}}$, i.e. $P_i \leq_{\mathcal{P}} P_j$ iff $i \leq j$.*

Let $\mathcal{P}$ be a partition of the set V. A *refinement* of $\mathcal{P}$ with a pivot set $S \subseteq V$ is the partition $\mathcal{P}'$ such that each part $X' \in \mathcal{P}'$ is a subset of some part $X \in \mathcal{P}$ and either $X' = X \cap S$ or $X' = X \setminus S$. We equip the partition with additional structure by ordering the parts. We use $\mathcal{P} = [X_1 \ldots X_k]$ to denote an *ordered partition* with parts $X_1, \ldots, X_k$ and the ordering $X_i \leq X_j$ iff $i \leq j$. To refine a partition $\mathcal{P}' = \textsc{Refine}(\mathcal{P}, S)$, we replace a part X with $A := X \cap S$ and $B := X \setminus S$, such that A and B are ordered the same as X and make a choice whether $A \leq_{\mathcal{P}'} B$ or $B \leq_{\mathcal{P}'} A$. We use the ability to choose the order of the new parts for the algorithms discussed in Section 3.1 and Section 3.2.

We implement this data structure by storing a permutation σ of the vertex set V and its inverse σ^{-1}. Additionally, we store the parts $X_i \in \mathcal{P}$ and the corresponding part for each vertex. An example can be seen in Figure 2.2.

The following Lemma is crucial for the time complexity of the algorithms using partition refinement.

Lemma 2.10: *(Lemma 1 [HPV99]) Computing* $\textsc{Refine}(\mathcal{P}, S)$ *can be done in* $O(|S|)$ *time.*

We implement the Refine operation by swapping nodes in the permutation, updating the inverse permutation and the parts. In short, the vertices of the pivot set are moved from the part X to $X \cap S$ in $O(1)$ time each.

In the next chapter, we describe we modular decomposition algorithms implemented in this thesis using the concepts and data structures we just discussed.

3 Algorithms

In this chapter, we present three modular decomposition algorithms. The choice of algorithms is based on existing work and a preliminary study. The FRACTURE algorithm already has an usable implementation [Kar19] and the algorithm description of the SKELETON algorithm is not overly complicated. The more complex parts of the algorithm are conceptually quite simple and more accessible to debugging methods, for example, the ordered partition data structure discussed in Section 2.2. The authors of the LINEAR algorithm claim that it positively answers the search for a "simple" linear algorithm formulated in earlier literature [Spi03 | HP10], and its implementation only uses one data structure, an ordered list of trees. We considered additional algorithms and attempted to implement them but ultimately did not proceed because we deemed them too inefficient or they require complicated implementation details [JSC72 | MS89 | DGM01].

The FRACTURE and SKELETON algorithms have a $O(n + m \log n)$ running time and are discussed in Sections 3.1 and 3.2, respectively. Then, we present an algorithm with linear running time in Section 3.3. For each algorithm, we describe implementation details and improvements at the end of their respective sections. Finally, we provide a short comparison of the algorithms and discuss common concepts in Section 3.4

3.1 Fracture Algorithm

The algorithm described in this section works in two steps. First, a factorizing permutation is computed in $O(n + m \log n)$ (Section 3.1.1, [HPV99]) and then the modular decomposition is computed from it in $O(n + m)$ [CHM02].

3.1.1 Computing a Factorizing Permutation

Here, we discuss the first step of the algorithm, computing a factorizing permutation. Recall Definition 2.8: a permutation on the vertex set is *factorizing* if all strong modules appear consecutively. The algorithm for computing the factorizing permutation repeatedly refines an ordered partition while maintaining an ordering on the parts. Specifically, we extend the meaning of factor to ordered partitions.

Definition 3.1: *A set is a* factor of an ordered partition $\mathcal{P}$ *if all parts it overlaps with are consecutive in* $\mathcal{P}$.

Subsequently, in a factorizing permutation, the strong modules containing a vertex x are factors and are nested around x. Motivated by this, we start by ordering the vertices such that the strong modules containing x are a factor. The strategy is illustrated in Figure 3.1. The vertices are ordered such that all neighbors of x are to its right and all non-neighbors are to its left. This results in the strong modules containing x to be factors. Additionally, every strong module not containing x remains as a part. They are ordered with respect to x and each other, but the order of the vertices within the parts is not specified. The algorithm can be applied to the remaining parts until only singletons remain, resulting in a permutation of the vertices.

Algorithm 3.1: Factorizing Permutation [HPV99]

Input: A graph $G = (V, E)$
Output: A factorizing permutation
begin
  $\mathcal{P} \leftarrow [V]$, `Center` $\leftarrow \bot, K \leftarrow [], L \leftarrow \{\}$
  while Init($\mathcal{P}$) **do**
    while $\exists Y \in L$ **do**
      remove Y from L
      for each $y \in Y$ **do**
        Refine($\mathcal{P}, S = N(y) \setminus Y$)
  return $[v \mid \{v\} \in \mathcal{P}]$
Function *Init*($\mathcal{P}$)
  `// L is empty when Init is called`
  if $\forall X \in \mathcal{P} : |X| = 1$ **then return** false
  if K is empty **then**
    `// all parts` $X \in \mathcal{P}$ `are modules`
    `// choose` $x \in X$ `and start to compute` $\mathcal{M}(G[X], x)$
    $X \leftarrow$ any $X \in \mathcal{P}$ with $|X| > 1$
13 $x \leftarrow$ `FirstPivot`(X) if defined or else any $x \in X$
14 `Center` $\leftarrow x$
15 $X_a, X_b \leftarrow X \cap N(x), X \setminus (N(x) \cup \{x\})$
16 replace X with $[X_b, \{x\}, X_a]$
17 $Z, Z' \leftarrow$ OrderBySize(X_a, X_b)
    add Z to L and add Z' to K
  else
20 remove X from the front of K `// X is a module of G`
    $x \leftarrow$ any $x \in X$ and add $\{x\}$ to L
    `FirstPivot`$(X) \leftarrow x$
  return true
Function *Refine*($\mathcal{P}, S = N(y) \setminus Y$)
  for each part $X \in \mathcal{P}$, such that $X \cap S \neq \emptyset$ and $X \setminus S \neq \emptyset$ **do**
    $X_a, X_b \leftarrow X \cap S, X \setminus S$
27 **if** X is between `Center` and Y **then**
      replace X by $[X_a, X_b]$ in $\mathcal{P}$
    else replace X by $[X_b, X_a]$ in $\mathcal{P}$
    if $X \in L$ **then** replace X by $\{X_a, X_b\}$ in L
    else
32 $Z, Z' \leftarrow$ OrderBySize(X_a, X_b)
      add Z to L `// smallest half rule`
      if $X \in K$ **then** replace X by Z' in K
      else add Z' to the end of K

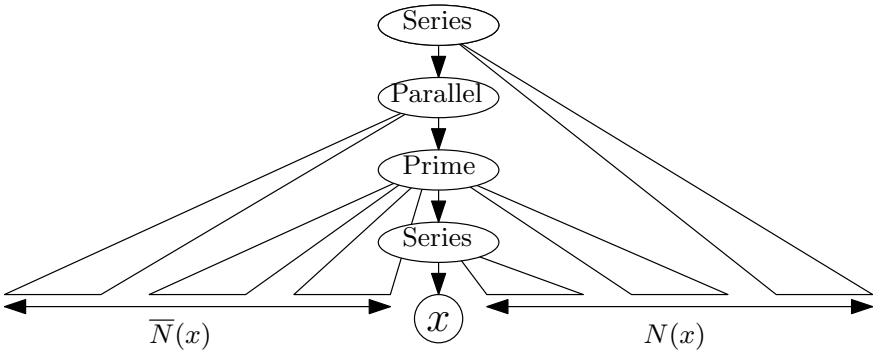


Figure 3.1: It is possible to order the leaves of $MD(G)$, such that $\overline{N}(x)$ and $N(x)$ are to the left and right oft x respectively. The parts are $\{x\}$ and the modules not containing x, i.e. $\mathcal{M}(G, x)$. Adapted from [HPV99].

Algorithm 3.1 keeps two disjoint sets of parts, L and K. We use the neighborhood of vertices of the parts in L to refine the ordered partition $\mathcal{P}$. Both L and K are initially empty, and we treat the set K as a FIFO queue. When L and K are empty and non-singleton parts remain, we choose a vertex as the *center*. When the center vertex is chosen, the algorithm refines the ordered partition until the parts equal the v-modular partition $\mathcal{M}(G, x)$. Then, a new center vertex x' is chosen from a non-singleton part X, and the algorithm refines the part X into the parts of $\mathcal{M}(G[X], x')$.

3.1.2 Invariants

For the algorithm to work, we maintain the strong modules of G as factors of the ordered partition. Additionally, the algorithm refines the partition until all parts are singletons. The following invariants are used to prove the algorithm's correctness.

1. Let $x \in V$ be the latest vertex chosen as the center in Line 13 and X_a, X_b the sets in Line 15. If a module M is a subset of a part in $[X_b, \{x\}, X_a]$, then there exists a part in the current partition containing M.

2. If $L = \emptyset$, then the first part of K is a module.

3. If the current partition contains a part $X \in \mathcal{P}$ that is not a module, then there exists a part $Y \in L \cup K$, such that $Y \neq X$ and Y contains a splitter y for X (see Definition 2.2).

4. Every strong module is a factor of the current ordered partition $\mathcal{P}$.

Invariant 1 implies that modules are preserved during refinement, and only choosing a center vertex and partitioning its part can split a module (Line 16). Invariant 3 implies that when $L = K = \emptyset$, then every part is a module. We do not prove invariants 1, 2, 3, but we use them to prove 4. We refer to [HP10 | HPV99] for proof of the other invariants.

The essence of proving invariant 4 is that the invariant holds for strong modules containing the center vertex x. The algorithm computes $\mathcal{M}(G, x)$. Every module not containing x is a subset of some module in $\mathcal{M}(G, x)$. The proof for a single vertex can then be applied to the remaining parts, for which a new center vertex is chosen. We show that the following invariant holds.

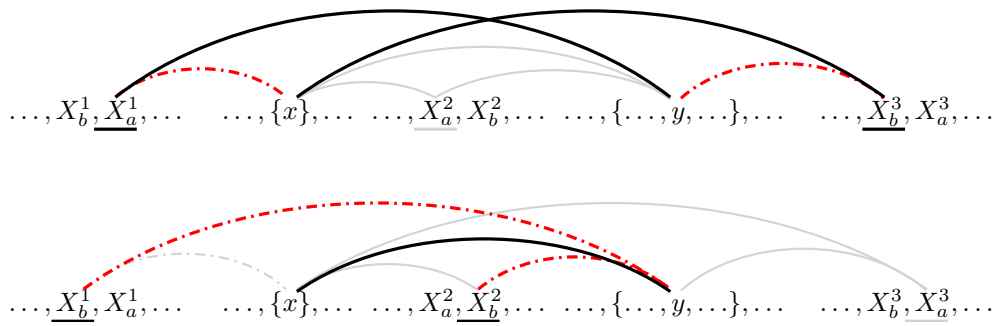


Figure 3.2: This shows $\mathcal{P}' = \textsc{Refine}(\mathcal{P}, S)$, with $S = N(y) \setminus Y$ for some part Y, vertex $y \in Y$ and x as the center vertex. The parts $X^i \in \mathcal{P}$, $i = \{1, 2, 3\}$, are split into $X_a^i = X^i \cap S$ and $X_b^i = X^i \setminus S$, $X_a^i, X_b^i \in \mathcal{P}'$, and ordered according to the rule in Line 27. Edges are lines and non-edges are dotted. Both upper and lower illustrations show the same ordered partition, but highlight the parts closer to x and further from x, respectively. At the top, elements of X_a^1 and X_b^3 are splitters of $\{x, y\}$. At the bottom, y is a splitter of $\{x\} \cup X_b^1$ and $\{x\} \cup X_b^2$.

Lemma 3.2: *Every strong module containing the center vertex x is a factor of the current ordered partition.*

Proof. We need to show that the strong modules containing x are a factor of the first non-trivial ordered partition and that any changes maintain that invariant. Only calls to Refine modify the ordered partition until a new center is chosen. When that happens, the set of parts is $\mathcal{M}(G, x)$, and any module containing x is a disjoint union of parts, and further refinement cannot change the fact that it is a factor of the ordered partition.

For the first non-trivial ordered partition, $[\overline{N}(x), \{x\}, N(x)]$, all strong modules containing x overlap $\{x\}$ and either $\overline{N}(x)$, $N(x)$, or both. Therefore, the modules are a factor of that partition.

Now assume that the invariant holds for the current partition before a call to Refine. Let Y be a part, $y \in Y$ a vertex, and $S = N(y) \setminus Y$ the pivot set for a call to Refine. Let $\mathcal{P}$ and $\mathcal{P}'$ be the partition before and after the refinement. Let M be a strong module containing x, and $X \in \mathcal{P}$ be a part that is split into $X_a, X_b \in \mathcal{P}'$, with $X_a = X \cap S \neq \emptyset$ and $X_b = X \setminus S \neq \emptyset$.

We now do a case analysis to show that the invariant is maintained by the ordering of the new parts, X_a and X_b, chosen in Line 27. The cases are illustrated in Figure 3.2. Assume that $y \in N(x)$, i.e. y is to the right of x. The case $y \in \overline{N}(x)$ is symmetrical.

Assume both x and y are a member of a strong module M. We show that the part X_a or X_b that is closer to x, is a subset of M. This corresponds to the upper part of Figure 3.2.

- If X is to the left of x and y, then every vertex z in $X_a = X \cap S$ belongs to M, otherwise z is a splitter for x and y.
- If X is between x and y, then $X \subseteq M$, because M was a factor of $\mathcal{P}$.
- If X is to the right of x and y, then every vertex z in $X_b = X \setminus S$ belongs to M, otherwise z is a splitter for x and y.

Now assume y is not a member of M. We show that the part X_a or X_b that is further from x, is not a subset of M. This corresponds to the lower part of Figure 3.2.

- If X is to the left of x and y, then no vertex z of $X_a = X \cap S$ belongs to M, otherwise y would be a splitter for x and z.
- If X is between x and y, then no vertex z of $X_b = X \cap S$ belongs to M. Otherwise, y would be a splitter for x and z.
- If X is to the right of x and y, then $X \cap M = \emptyset$, because M was a factor of $\mathcal{P}$.

We have shown that any module containing x and y contains the split half that is nearer to x. And, any module containing x but not y does not contain the split half that is further from x. Let M be a strong module containing x. If it was a factor before the call to Refine, then it still is a factor of the ordered partition.

Therefore, every strong module containing the center vertex x is a factor of the current ordered partition. ∎

3.1.3 Running time

The algorithm's total running time depends on how many times an element is used in a pivot set. The authors of [CHM02] use "Hopcroft's rule" to bound the total runtime. This rule is also known as "process the smallest half rule" [Hop71]. More formally, the rule is the following. Let X be a part that is split into X_a and X_b. If the ordered partition $\mathcal{P}$ is stable with respect to refinement with the neighbors of the elements in X, then choose the smaller part of X_a and X_b to refine $\mathcal{P}$.

Theorem 3.3: *Algorithm 3.1 has a running time of* $O(n + m \log n)$.

Proof Sketch. Let Y be a part, and $y \in Y$ be a vertex such that the pivot set $S = N(y) \setminus Y$ is used to refine the ordered partition. The call to Refine takes $O(|S|)$ time. Lines 17 and 32 ensure that when Y is split, only the smaller part is added to L. When a part X is removed from K, then a vertex $x \in X$ is chosen, and only $\{x\}$ is added to L. The vertex x and every other vertex in X are only ever used again when X is split by choosing a new center. By keeping track of `FirstPivot`, when x is chosen from a module from K, it is used as a center and then never again. Therefore, every neighborhood is used at most $O(\log n)$ times as a pivot set. We obtain a total running time of $O(n + \sum_y \log n |N(y)|) = O(n + m \log n)$. □

3.1.4 From Factorizing Permutation to Modular Decomposition

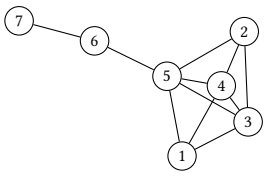
The next step is to use the factorizing permutation to compute the modular decomposition. We present the algorithm from Capelle, Habib, and Montgolfier [CHM02].

Recall Definition 2.2 of a splitter. Let $G = (V, E)$ be a graph and $M \subseteq V$ be a vertex set. A vertex $x \in V \setminus M$ is called a *splitter* of M, if there exists $y, z \in M$ such that $xy \in E$, but $xz \notin E$. Then, M is a module of G exactly if it has no splitter. By starting with any pair of vertices $\{x, y\} \in \binom{V}{2}$ and iteratively adding splitters, we find the minimal strong module containing x and y. This leads to a naive algorithm for computing the modular decomposition by finding a non-trivial module M and computing the modular decomposition of the induced subgraph $G[M]$ and of the graph where M is contracted to a single vertex [HP10].

Given the factorizing permutation σ, only the splitters of consecutive pairs of vertices x, y, $x = \sigma(i)$ and $y = \sigma(i+1)$, $i \in [1 \dots n-1]$, are needed. The algorithm described in [CHM02] computes the leftmost and rightmost splitter of x and y. Let $L[(x, y)]$ be the leftmost splitter of x and y in $[\sigma(1) \dots x]$ if it exists and x otherwise. Similarly, let $R[(x, y)]$

Algorithm 3.2: Modular Decomposition ([CHM02])

```
σ ← factorizing permutation of G                                  // Algorithm 3.1
σ̂ ← (σ(1) ... σ(n))          // a parenthesized σ with initial '(' ')' pair.
for each x, y adjacent in σ do
    L[(x, y)] ← left most splitter of {x, y} in [σ(1) ... x] or x if it does not exists
    R[(x, y)] ← right most splitter of {x, y} in [y ... σ(n)] or y if it does not exists
    if L[(x, y)] ≠ x then insert '(' ')' around [L[(x, y)] ... x]
    if R[(x, y)] ≠ y then insert '(' ')' around [y ... R[(x, y)]]
T ← FractureTree(σ̂)
T ← remove non-module nodes from T using L[·] and R[·]
T ← recover merged modules from T using L[·] and R[·]
T ← remove repeated nodes from T
T ← add node types to nodes from T
return T
```



(a) A graph G with $\sigma = 1234567$ as factorizing permutation.

x, y	$S(x, y)$	$[L[(x, y)] \dots x]$	$[y \dots R[(x, y)]]$
1,2	$\emptyset$	-	-
2,3	{1}	$[1 \dots 2]$	-
3,4	$\emptyset$	-	-
4,5	{6}	-	$[5 \dots 6]$
5,6	{1, 2, 3, 4, 7}	$[1 \dots 5]$	$[6 \dots 7]$
6,7	{5}	$[5 \dots 6]$	-

(b) The consecutive pairs of vertices in σ, the set of splitters $S(x, y)$ and their left and right fractures, $[L[(x, y)] \dots x]$ and $[y \dots R[(x, y)]]$, if $L[(x, y)] \neq x$ and $R[(x, y)] \neq y$.

$$\hat{\sigma} = (((12)34((5)(6))7))$$
$$\overset{52}{}\ \overset{2}{}\ \overset{64}{}\ \overset{55}{}\ \overset{46}{}\ \overset{5}{}$$

(c) The factorizing permutation σ is parenthesized by the left and right fractures. The numbers on top are the vertex x of the x, y pair responsible for the parenthesis.

Figure 3.3: A small example graph, its left and right fractures and the resulting parenthesized factorizing permutation.

be the rightmost splitter of x and y in $[y \dots \sigma(n)]$ if it exists and y otherwise. If $L[(x,y)] \neq x$, then $[L[(x,y)] \dots x]$ is the *left fracture* of (x,y) and if $R[(x,y)] \neq y$, then $[y \dots R[(x,y)]]$ is the *right fracture* of (x,y). An example can be seen in Figure 3.3. Note that vertices that are twins, i.e. $N(x) \setminus \{y\} = N(y) \setminus \{x\}$ do not have any splitters.

The permutation with an initial pair of outer parenthesis and additional parenthesis inserted around the intervals defined by the left and right fractures define a parenthesis system. This *parenthesized factorizing permutation* $\hat{\sigma}$ is balanced, i.e. every prefix contains more opening than closing parenthesis. Figure 3.3 shows an example. The parenthesized factorizing permutation can be transformed into a tree called *fracture tree* $FT(G)$. The leaves correspond to the vertices of the original graph, and every node corresponds to a matching pair of parenthesis. This tree is a good approximation for the modular decomposition $MD(G)$. This can be formalized with the following theorem.

Theorem 3.4: *(Theorem 2 in [CHM02]). Let σ be a factorizing permutation of a graph G. Let M be a node of $MD(G)$ representing a strong module. If M is a prime node or M has series or parallel node as a parent, then there exists a node N in the fracture tree of σ that represents M.*

Note that each pair adds at most 4 parenthesis to $\hat{\sigma}$. Therefore, $FT(G)$ has n leaves and at most $2(n-1)+1$ inner nodes. The tree can be computed in linear time from the factorizing permutation. The fractures can be computed by using radix sort to order the neighborhoods of the vertices by the location of the vertex in the permutation and traversing the neighborhoods once from both directions until the left- or rightmost splitter is found. The tree can be built in linear time by starting with a root node for the outer pair of vertices, adding a new child to the current node, and moving down to that child for each '(', adding a leaf when a vertex is encountered and moving up the tree for each ')'.

As a step to compute $MD(G)$ we remove all nodes in $FT(G)$ that do not correspond to a strong module. This can be done by traversing $FT(G)$ with a postorder DFS. Let N be a node in $FT(G)$ with children nodes $C_1 \dots C_k$ ordered by σ. We utilize the following property.

Lemma 3.5: *(Property 4 in [CHM02]) If all splitters of consecutive pairs of N belong to N, then N represents a module.*

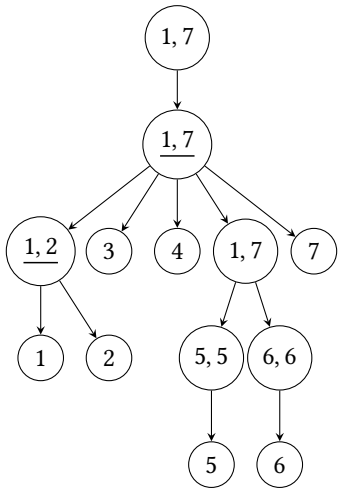
We use the arrays L and R as defined earlier. We denote a node's first and last vertex with $\text{first}(N)$ and $\text{last}(N)$, respectively. Additionally, we use fs and ls to define the first and last splitter for the nodes of the tree. The leave nodes $\{v\}$, $v \in V$ are initialized with $fs(\{v\}) = \text{first}(\{v\}) = v = \text{last}(\{v\}) = ls(\{v\})$. The values for $fs(C_i), \text{first}(C_i), \text{last}(C_i)$, and $ls(C_i)$ are already computed as we traverse the tree with a postorder DFS.

We combine the results of the child nodes by additionally considering the adjacent vertex pairs between children. We use the following recursive definition.

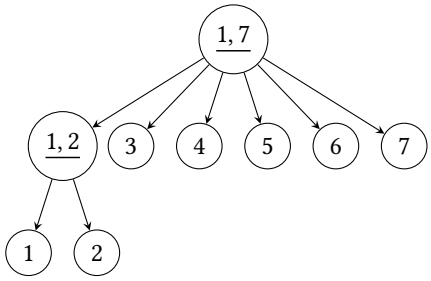
$$\textit{first splitter } fs(N) = \min\left(\text{first}(N), \min_{i \in 1 \dots k-1} \{L[(\text{last}(C_i), \text{first}(C_{i+1}))]\}, \min_{i \in 1 \dots k} fs(C_i)\right)$$

$$\textit{last splitter } ls(N) = \max\left(\text{last}(N), \max_{i \in 1 \dots k-1} \{R[(\text{last}(C_i), \text{first}(C_{i+1}))]\}, \max_{i \in 1 \dots k} ls(C_i)\right)$$

Because of Lemma 3.5 we can determine if a node we encounter in the tree represents a module. A non-leaf node N represents a module exactly if $\text{first}(N) \leq fs(N) < ls(N) \leq \text{last}(N)$. All non-module nodes are removed, and all children are assigned to the parent of the removed node. Figure 3.4 shows an example tree.



(a) Before removal of non-module nodes.



(b) The fracture tree with only strong-module nodes an non-repeating nodes.

Figure 3.4: An example example continuing from the graph in Figure 3.3. The fracture tree corresponding to $\hat{\sigma}$ is shown on the left side. Inner nodes are marked with $fs(N), ls(N)$ and underlined, if they correspond to a strong module. For example, the least common ancestor of 5 and 6 has $fs(N) = 1$ and $ls(N) = 7$ and therefore does not correspond to a module. Notice that the strong module $\{1, 2, 3, 4\}$ is missing, and its children $\{1, 2\}, 3, 4$ appear consecutively.

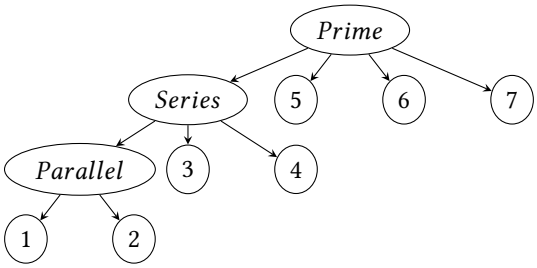


Figure 3.5: The final modular decomposition tree for the example in Figure 3.3 and Figure 3.4.

Note that the fracture tree might not represent series and parallel nodes with a prime node as a parent. This is true for the module $\{1, 2, 3, 4\}$ of the example in Figure 3.4. Such nodes are called *merged strong modules*, which must be recovered to appear in the tree. The children of a merged strong module appear consecutively as the children of a node in the fracture tree. The merged modules can be recovered by inserting nodes for consecutive pairs that are twins. After determining the module types, the result is the modular decomposition tree. The result of examples in this section is shown in Figure 3.5.

3.1.5 Implementation Details

The combination of the algorithm for the factorizing permutation and the modular decomposition algorithms has been previously implemented in `Julia`[1] [Kar19] and `C++`[2] [FL15]. Initially, we ported the `Julia` implementation to Rust. Then, we added several improvements to the code and achieved a speedup of several orders of magnitude. Still, we use some of the ideas of the reference implementation. Next, we list some of the implementation details.

Using an Ordered Partition Data Structure. Certainly, an important step was to replace a vector of vector representation of an ordered partition with the data structure discussed in Section 2.2. The data structure allows a $O(|S|)$ refine operation instead of an $O(n^2)$ worst-case. Additionally, this allows the efficient representation of parts and consecutive parts as intervals of the underlying permutation.

Working on Parenthesized Permutation Directly. One useful idea from the `Julia` implementation [Kar19] is to avoid explicitly building the fracture tree $FT(G)$ but instead, work on the parenthesized factorizing permutation directly. The parenthesized factorizing permutation is represented with a vector of vertices, representing the permutation, and two vectors counting the number of opening and closing parenthesis directly before and after a vertex. A postorder traversal of the fracture tree is simulated by iterating over those arrays: i) initializing a new child node for each opening parenthesis, ii) handling the leaf vertex and iii) leaving a node and going up the tree for each closing parenthesis.

Algorithm 3.3: Simulated postorder traversal of a tree

Input: A factorizing permutation σ, number of opening and closing brackets, op and cl, describing a tree T.
Output: A postorder traversal of T calling LEAF_NODE and INNER_NODE.

```
S ← empty stack <>
for each j ∈ {1, ..., n} do
    for i ∈ {1, ..., op[j]} do S.PUSH(∅)
    LAST(S).PUSH(LEAF_NODE(σ(j), j))
    for i ∈ {1, ..., cl[j]} do
        S_{-1} ← S.POP()
        LAST(S).PUSH(INNER_NODE(S_{-1}))
```

We use a segmented stack $S = S_0, \ldots, S_k$, where S_i is a stack corresponding to the depth i in the tree, to store information during such a postorder traversal. We append an empty stack when entering a subtree. A leaf node appends some data to S_k and leaving a subtree removes the last stack S_k and appends data to S_{k-1}. Algorithm 3.3 illustrates the postorder traversal in pseudo code. For example, to remove the non-module nodes of the tree, we maintain the first and last vertex and the first and last splitter for a node. Any parenthesis before the current one is not visited again during this postorder traversal. A node can be deleted by decrementing the number of opening and closing parenthesis by one at the first and last vertex, respectively.

[1] *https://github.com/StefanKarpinski/GraphModularDecomposition.jl*
[2] *https://github.com/LyteFM/modular-decomposition*

Determining Module Types. Let M be a module and $C_1, \ldots, C_k$ its children in the modular decomposition tree. The original algorithm description and the reference implementation compute the quotient graph $G_M = G[M]/\{C_1, \ldots, C_k\}$ either explicitly or implicitly and determine the module type by counting its edges. We avoid that and use the neighborhood of one vertex, the vertex in each subtree with minimum degree in G.

For each C_i, let $x_i \in C_i$ be the vertex with minimal degree in G and j such that x_j has minimal degree among the x_i. We use $O(n)$ extra space to be able to mark vertices. We mark all x_i and iterate over $N(x_j)$ to determine $d_{G_M}(C_j) = |\{x_1, \ldots, x_k\} \cap N(x_j)|$, the degree of the vertex C_j in the quotient graph $G_M = G[M]/\{C_1, \ldots, C_k\}$. This takes $O(k + |N(x_j)|)$ time. We know that the tree only contains strong modules. If the quotient has only two vertices, it is either series or parallel and we can distinguish these cases by whether or not $d_{G_M}(C_j) = 0$. Now assume there are at least three vertices. If $0 < d_{G_M}(C_j) < k - 1$, the graph cannot be series or parallel and is prime. If $d_{G_M}(C_j) = 0$, then C_j is the child of a parallel node, and if $d_{G_M}(C_j) = k - 1$, then it is the child of a series node, determining the module type of M.

The total time can be bounded by $O(m+n)$ by using the fact that there are $O(n)$ inner nodes, charging the children $O(1)$ when their representative vertex is marked and noticing that the total cost of iterating over each $N(x_j)$ to compute $d_{G_M}(C_j)$ is bounded by $O(\sum_v (d(v) + 1)) = O(m + n)$.

Putting it all together, we used useful existing ideas and added further improvements to implement the algorithm described in this section efficiently. Next, we cover another $O(n + m \log n)$ algorithm.

3.2 Skeleton Algorithm

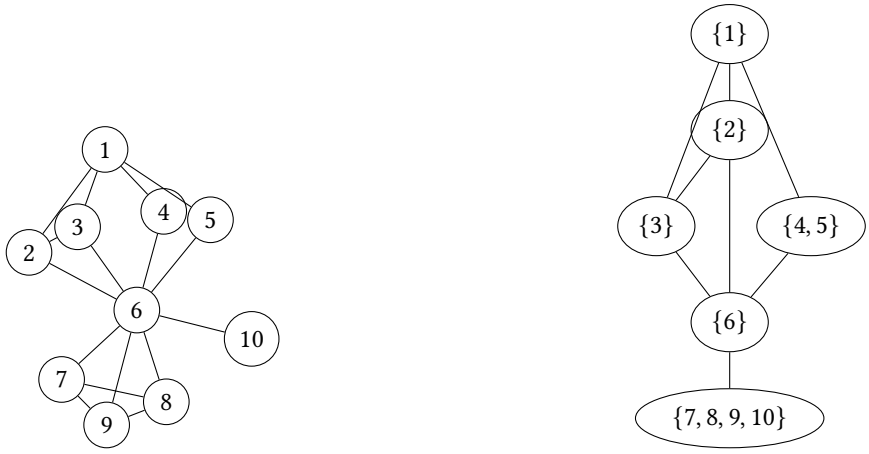
Previously, an $O(n^2)$ algorithm for modular decomposition introduced a divide-and-conquer strategy [EGMS94]. It is referred to as the SKELETON algorithm [HP10] and is the basis for several algorithms that achieve running times $O(n + m)$ or $O(n + m\alpha(n, m))$ [DGM01], with $\alpha(n, m)$ being the inverse Ackermann function. We describe a comparatively simple variant with running time $O(n + m \log n)$ [MS00].

Recall Theorem 2.6 and Theorem 2.7 from Section 2.1. Theorem 2.6 relates the modular decomposition of the graph $MD(G)$, with the modular decomposition of the quotient graph $MD(G/\mathcal{P})$ for a modular partition $\mathcal{P}$. Theorem 2.7 states that whether or not a set $Y \subseteq X$ is a module is "local" to the subgraph $G[X]$.

The SKELETON algorithms use the v-modular partition $\mathcal{M}(G, v)$ to divide G. Recall that $\mathcal{M}(G, v)$ consists of $\{v\}$ and all maximal modules not containing v. Figure 3.6 shows an example. The modular decomposition tree $MD(G/\mathcal{M}(G, v))$ is called spine(G, v), giving the algorithm its name. As every part $X \in \mathcal{M}(G, v)$ is a module of G, Theorem 2.6 is applicable.

We give a high-level description of the SKELETON algorithm using Algorithm 3.4. Afterward, we present Algorithms 3.5, 3.6, and 3.7, which compute the v-modular partition $\mathcal{M}(G, v)$, the quotient $G/\mathcal{P}$, and $MD(Q)$, respectively.

Before continuing with the algorithm details, we look at the spine(G, v). Figure 3.7 shows the modular decomposition tree of a small example graph. The following properties hold [HP10 | EGMS94]:



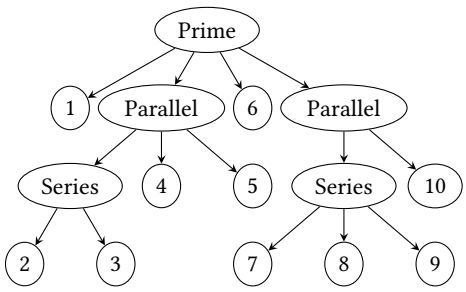
(a) A graph G.

(b) The quotient graph $G/\mathcal{M}(G,3)$.

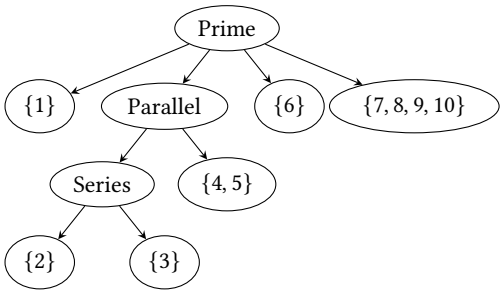
Figure 3.6: An example graph and its quotient in respect to the 3-modular partition.

Algorithm 3.4: Skeleton Algorithm [EGMS94 | MS00]

Input: A graph $G = (V, E)$
Output: The modular decomposition tree $MD(G)$
$v \leftarrow$ any vertex of G
if G has only one vertex **then return** $\{\{v\}\}$
else
  $\mathcal{P} \leftarrow \mathcal{M}(G, v)$ `// Algorithm 3.5`
  $Q \leftarrow G/\mathcal{P}$ `// Algorithm 3.6`
  $T \leftarrow MD(Q)$ `// =` $spine(G, v)$`, Algorithm 3.7`
  for each $Y \in \mathcal{M}(G, v)$ **do**
    $T_Y \leftarrow MD(G[Y])$ `// recurse, Algorithm 3.4`
    replace Y with T_Y in T
    merge T_Y with its parent in T if they are both series or parallel
  return T



(a) The modular decomposition tree $MD(G)$ of graph G from Figure 3.6.



(b) The tree spine$(G, 3)$. We have $\mathcal{M}(G, 3) = \{\{3\}\} \cup \{\{1\}, \{2\}, \{4, 5\}, \{6\}, \{7, 8, 9, 10\}\}$.

Figure 3.7: An example modular decomposition

- The inner modules of $G/\mathcal{M}(G, v)$ correspond to ancestors of v in $MD(G)$. More precisely, a set $\mathcal{X} \subseteq \mathcal{M}(G, v)$ is a module of $G/\mathcal{M}(G, v)$ if and only if $\bigcup_{M \in \mathcal{X}}$ is an ancestor of v in $MD(G)$.
- Every inner module of $G/\mathcal{M}(G, v)$ contains v.
- Every module of G that does not contain v is a subset of some part $M \in \mathcal{M}(G, v)$.

To complete Algorithm 3.4, we need to know how to compute $\mathcal{M}(G, v)$, $G/\mathcal{M}(G, v)$, and $MD(G/\mathcal{M}(G, v))$. The algorithms are described in the following sections

3.2.1 Computing $\mathcal{M}(G, v)$

Algorithm 3.5, called ordered vertex partition (OVP), takes an ordered partition $\mathcal{P}$ of the vertex set and produces an ordered partition $\mathcal{P}'$ such that every part $X \in \mathcal{P}'$ is a maximal module and a subset of some part in $\mathcal{P}$. Calling the algorithm with the initial parameter $\mathcal{P} = [\{v\}, V \setminus \{v\}]$ results in the parts $\{v\}$ and the maximal modules not containing v. Therefore, we have $\mathcal{M}(G, v) = \text{OVP}(G, [\{v\}, V \setminus \{v\}])$.

The edges crossing from X to $V \setminus X$ can be used to compute the pivot sets $N(x) \setminus X$ for each $x \in X$ and $N(y) \cap X$ for each $y \in V \setminus X$. This is done by grouping the edges first by their endpoint in X and then by their endpoint in $V \setminus X$. After removing the crossing edges, the recursive calls are independent of each other, and one can keep using the graph G instead of explicitly building $G[X]$ and $G[V \setminus X]$. This is an important detail when implementing the algorithm. Note that after the call to Split, for every vertex $x \in X$ and every part Y that is a subset of $V \setminus X$, either $Y \subseteq N(x)$ or $Y \cap N(x) = \emptyset$. The same holds for every vertex $y \in V \setminus X$ and every part Z that is a subset of X. This fact is useful for proof of correctness. In order to facilitate understanding of the algorithm, we reproduce this proof.

Theorem 3.6: *([MS00]) Let $G = (V, E)$ be a graph, and $\mathcal{P}$ be an ordered partition of V. Algorithm 3.5 computes an ordered partition of G such that every part is a maximal module of G that is a subset of some part of $\mathcal{P}$ in $O(n + m \log n)$.*

Proof. We show that every part X of the resulting ordered partition is a module. The algorithm only splits parts that are not a module, as no pivot can split a module.

In the base case $|\mathcal{P}| = 1$, the only member of $\mathcal{P}$ is V, which is a trivial module of G.

Algorithm 3.5: Ordered Vertex Partition [MS00]

Input: A graph $G = (V, E)$ and an ordered partition $\mathcal{P}$ of V
Output: An ordered partition such that every part is a maximal module that is a subset of some part in $\mathcal{P}$

if $|\mathcal{P}| = 1$ **then return** $\mathcal{P}$
else
  $X \leftarrow$ any $X \in \mathcal{P}$ such that $|X| \leq |V|/2$
  $E' \leftarrow \{xy \in E \mid x \in X \text{ and } y \in N(x) \setminus X\}$
  $\mathcal{P}' \leftarrow \textsc{Split}(X, \mathcal{P})$
  $G \leftarrow (V, E \setminus E')$ `// remove edges between` X `and` $V \setminus X$
  $Q_1 \leftarrow [Y \in \mathcal{P}' \mid Y \subseteq X]$
  $Q_2 \leftarrow [Y \in \mathcal{P}' \mid Y \subseteq V \setminus X]$
  $\mathcal{P}_1 \leftarrow \text{OVP}(G[X], Q_1)$ `// first recursive call`
  $\mathcal{P}_2 \leftarrow \text{OVP}(G[V \setminus X], Q_2)$ `// second recursive call`
  return $\mathcal{P}_1 \cup \mathcal{P}_2$
Function *Split*$(X, \mathcal{P})$
13 When $p \in Y \in \mathcal{P}$ splits X into adjacent and non-adjacent vertices (X_a and X_n), place X_n nearer to Y in the order
  foreach $x \in X$ **do** $\mathcal{P} \leftarrow \textsc{Refine}(\mathcal{P}, N(x) \setminus X)$
  foreach $y \in V \setminus X$ **do** $\mathcal{P} \leftarrow \textsc{Refine}(\mathcal{P}, N(y) \cap X)$
  return $\mathcal{P}$

For a non-trivial partition, we Q_1 and Q_1 to denote the partitions that will be used in the subsequent recursive calls with the graphs $G[X]$ and $G[V \setminus X]$, respectively. The results of these calls are denoted with $\mathcal{P}_1$ and $\mathcal{P}_2$.

Assume that the parts of $\mathcal{P}_1$ are modules of $G[X]$ and subsets of parts of Q_1. Let Y be a part of $\mathcal{P}_1$. Recall that Y is a module if and only if every vertex in $V \setminus Y$ is either adjacent to every vertex in Y or non-adjacent to every vertex in Y. As Y is a module of $G[X]$, the statement holds for all vertices in $X \setminus Y$. We now prove the same for $V \setminus X$. Let Z be the part of Q_1, such that $Y \subseteq Z$. Because of the call to Split, every vertex of $V \setminus X$ is either a neighbor to every element of Z or a non-neighbor to every element of Z. As Y is a subset of Z, the same is true for Y. Therefore, Y is a module of G. The case for each part of $\mathcal{P}_2$ is symmetric.

Subsequently, only the time bound is left to prove. The call to Split takes $O(|X|+\sum_{x \in X} d(x))$ to compute the crossing edges from X to $V \setminus X$. The cost is charged with $O(1)$ to every vertex in X and every edge incident to X. As we guaranteed that $|X| \leq |V|/2$, a vertex and its edges are only charged if its part is at most half of the vertices. The next time a vertex $x \in X$ is charged, its part has size at most $|X|/2$. Therefore, every vertex (and its incident edges) is charged at most $\log n$ times. This results in a $O((n + m) \log n)$ time bound.

The running time can be improved to $O(n + m \log n)$ by running the algorithm on each connected component, which can be computed in linear time, and combining their results. ■

Note that we did not use the ordering of the partition and the ordering rule in Line 13. This only becomes relevant in later parts of the algorithm. We now have covered the computation of $\mathcal{M}(G, v)$. The missing pieces of Algorithm 3.4 are the computation of $G/\mathcal{M}(G, v)$ and its modular decomposition $MD(G/\mathcal{M}(G, v))$.

3.2.2 Computing spine(G, v)

Computing the quotient graph $G/\mathcal{M}(G, v)$ is relatively straightforward, as described in Algorithm 3.6.

Algorithm 3.6: Quotient [MS00]

1 Algorithm 3.5 removes all edges between the parts of $\mathcal{M}(G, v)$. We keep track of those edges and map the vertices of the edges to their part in $\mathcal{M}(G, v)$. We de-duplicate the edge set using radix sort and build the quotient graph.

The most complicated part of the SKELETON algorithm is the computation of spine(G, v), the modular decomposition of $G/\mathcal{M}(G, v)$. Recall that spine(G, v) has a special structure. All graphs for which there exists a vertex v such that all non-trivial modules contain v are called *nested*, and v is called an *innermost* vertex. The graph $G/\mathcal{M}(G, v)$ is nested, and v is an innermost vertex, and Algorithm 3.7 computes the modular decomposition for such graphs.

Algorithm 3.7: Chain [MS00]

Input: A nested graph $G = (V, E)$ and an innermost vertex v of G
Output: The modular decomposition tree $MD(G)$

1 $\mathcal{P} \leftarrow \text{OVP}(G, [\{v\}, V \setminus \{v\}])$
Number the vertices V in order of their appearance in $\mathcal{P}$
return ROP(G)
`// Recursive OVP`
Function *ROP*$(G = (V, E))$
  $w \leftarrow$ an isolated vertex if it exists **or** the highest numbered vertex
  if $|V| = 1$ **then return** $\{w\}$
  else
8 $\mathcal{P} \leftarrow \text{OVP}(G, [\{w\}, V \setminus \{w\}])$
    $T \leftarrow$ tree with one node V `// determine module type from` $\mathcal{P}$
    `// There is at most one` $X \in \mathcal{P}$ `with` $|X| > 1$
    foreach $X \in \mathcal{P}$ **do** Add ROP$(G[X])$ as child to V in T
    return T

The core idea of this variant of the SKELETON algorithm lies in Algorithm 3.7 [MS00]. The algorithm computes the children of the root node and then works recursively. The children are determined by two calls to ordered vertex partition, using the resulting ordering of the vertices from the first one to initialize the input for the second one. The following lemma states that in the context of the algorithm, instead of computing the vertex ordering for every recursive call, the vertex ordering of the first call can be used.

Lemma 3.7: *(Lemma 6.4 [MS00]) If v is in all non-trivial modules of G, X is a module such that $v \in X$ and σ is the ordering of the vertices resulting from OVP$(G, [\{v\}, V \setminus \{v\}])$, then there exists a sequence of pivot choices such that $\sigma[X]$ is the result of OVP$(G[X], [\{v\}, X \setminus \{v\}])$.*

We prove the correctness of Algorithm 3.7 by induction and show that the algorithm is correct for the root node.

When the root node is prime, we use the following Lemma 3.8.

Lemma 3.8: *Let $G = (V, E)$ be a nested graph, $v \in V$ an innermost vertex of G, and $\mathcal{P} = OVP(G, [\{v\}, V \setminus \{v\}])$. Let $M_1, \ldots, M_k$ be the children of V in $MD(G)$. If V is prime in $MD(G)$, then the rightmost part of $\mathcal{P}$ is a singleton $\{w\}$ and the parts of $OVP(G, [\{w\}, V \setminus \{w\}])$ are $M_1, \ldots, M_k$.*

This result is based on an algorithm for the transitive orientation of prime graphs that can be used to check whether an arbitrary graph is prime [MS00]. This algorithm uses the two calls to OVP in the exact same way as described in Lemma 3.8. We refer to the original paper for proofs of Lemma 3.7 and Lemma 3.8.

Using these lemmas, we reproduce the correctness proof of Algorithm 3.7 in order to facilitate understandability.

Theorem 3.9: *Let G be a nested graph and v an innermost vertex. Algorithm 3.7 computes the modular decomposition tree $MD(G)$.*

Proof. To prove the correctness of Algorithm 3.7, we first show that it is correct for the root of $MD(G)$. Then, we argue that calling ROP recursively has the same result as calling Chain recursively.

As v is an innermost vertex of G, every non-trivial module of G contains v. Therefore, the call to OVP in Line 1 returns an ordered partition of singleton sets. Let σ be the ordering of the vertices based on the ordered partition $\mathcal{P}$.

We now enter the call to ROP. If the graph has only one vertex, we are finished and return. We now assume that the graph has at least two vertices. Let w' be the choice of w in this first call to ROP. We show that OVP$(G, [\{w'\}, V \setminus \{w'\}])$ returns the children of the root.

For that we use the following properties. Because G is nested, its modular decomposition has additional structure. Every inner node of $MD(G)$ has at most one non-singleton child, the one containing v, otherwise, there would be a non-trivial module not containing v. Furthermore, every series or parallel node has exactly two children. Otherwise, the union of two children that do not contain v would be a module.

Using those properties, we show that the choice of w' results in the computation of the children of the root.

- If the root is a parallel node, it has two children, one of which is a singleton isolated vertex. The vertex w' is that vertex, as ROP chooses an isolated vertex if it exists.
- If the root is a series node, it has two children, one of which is a singleton isolated vertex in the complement of G. The ordering rule in OVP ensures that w' is that vertex, as w' is adjacent to all other vertices.
- Otherwise, the root is a prime node, and Lemma 3.8 applies.

We now use induction to complete the proof. Chain is correct for $|V| = 1$. Assume that Chain is correct for nested graphs with fewer vertices than G.

We have shown that OVP$(G, [\{w'\}, V \setminus \{w'\}])$ computes the children. Any child not containing v is a singleton and can be added as a child to the root node. Let X the child of V that contains v. To compute $MD(G)$, only $MD(G[X])$ is left to compute. The graph $G[X]$ is nested with v as the innermost vertices. By induction hypothesis, a call to Chain with $G[X]$ and v computes $MD(G[X])$. Such a call would first call OVP to get an ordering and then call ROP on $G[X]$. We can apply Lemma 3.7, and we can use ordering $\sigma[X]$ instead of computing the ordering new. Therefore calling ROP(G[X]) has the same result as calling Chain$(G[X])$: the modular decomposition tree $MD(G[X])$. This tree is added as a child of the root node, resulting in $MD(G)$. ∎

This concludes the description of the SKELETON Algorithm 3.4. The analysis of the total running time follows from the running time of OVP, resulting in $O(n + m \log n)$.

3.2.3 Implementation Details

In the following, we discuss some of the details of implementing the SKELETON algorithm. Although the algorithm has been implemented before, its implementation is no longer available [GKBC04]. The algorithm heavily relies on recursion on induced subgraphs. This needs to be done efficiently.

Efficiently Representing Subgraphs and Sub-Partitions. We start by discussing OVP (Algorithm 3.5). Recall that OVP chooses a part $X \in \mathcal{P}$, refines the partition by calling SPLIT$(X, \mathcal{P})$, removes any edges between X and $V \setminus X$ from G and recurses on $G[X]$ and $G[V \setminus X]$. Building $G[X]$ and $G[V \setminus X]$ explicitly would be very costly. As any edges between X and $V \setminus X$ are removed, we can reuse the data structures for G and $\mathcal{P}$ and only need to keep track of the vertex sets X and $V \setminus X$. This can be done by using an additional parameter $\overline{V}$, initialized with V for the first call, and setting it to be X and $V \setminus X$ for the recursive calls for $G[X]$ and $G[V \setminus X]$, respectively.

We also want to represent the vertex sets X and $V \setminus X$ efficiently. As X was a part of $\mathcal{P}$ at the start of the call, its vertices are consecutive in our ordered partition data structure. For that to be also true for $V \setminus X$ we restrict the choice of X. We choose the first or the last part of $\mathcal{P}$, such that $|X| \leq |V|/2$. This allows all recursive calls to work on the same graph and ordered partition and let their subproblem be defined by a range of consecutive vertices in the ordered partition data structure. This also works for the recursive steps on induced subgraphs in the SKELETON algorithm and CHAIN (Algorithm 3.4 and Algorithm 3.7). We only have to be careful with the CHAIN algorithm. The first call to OVP removes all edges from G and leaves the ordered partition $\mathcal{P}$ with only singleton parts. For that, all removed edges are added again, and the parts of $\mathcal{P}$ are merged into a single part, while numbering the vertices with the order of their appearance in $\mathcal{P}$.

A Suitable Graph Data Structure. The algorithm requires that we can efficiently remove a set of edges and, with the previous optimization, the ability to add them to the graph again. To do that, we adapt the adjacency array representation of graphs. All edges are stored in an array. Each node keeps indices to two consecutive ranges of that array, its current, and removed edges. Every edge (u, v) is identified by its position in the edge array, and every edge stores its endpoint v, the index of its reverse edge (v, u), and whether it has been removed. Given the set of indices of the edges we want to remove, we can iterate over them once to mark them as removed and a second time to swap their position in the edge array with an edge next to the range of removed edges. The range of current and removed edges of the nodes need to be adjusted. This takes linear time in the number of removed edges. Note that after the call to remove a set of edges, the edges move, and the edge indices are invalidated. Adding all removed edges to the graph again can be done in linear time by iterating over all edges, setting the removed bit to false and iterating over all vertices, adjusting their ranges of current edges and removed edges.

Avoiding Recursion. One very important step while implementing the algorithm is to convert any recursion to iteration to avoid stack overflow. This is pretty straightforward. In CHAIN (Algorithm 3.7) we can even avoid a stack. Recall that in that context, every node in $MD(G)$ only has at most one non-singleton child. By covering the case of a singleton child early, we can keep iterating over the non-singleton children until we finish.

Special Cases for Hard Instances. Unfortunately, some input graphs lead to a lot of unnecessary work that can be avoided. Consider the graph with vertices $V = \{1, \dots, n\}$ and no edges. The SKELETON algorithm first computes $\mathcal{M}(G, 1) = \{\{1\}, \{2, \dots, n\}\}$ and the spine$(G, 1)$, which is the tree with a parallel node and two children. The algorithm recurses on both and many calls to the other algorithms are done. The same applies to the complete graph and the tree with a series node and two children. Note that the theoretical running time is not affected by this. Before computing $\mathcal{M}(G, v)$ in a call to the SKELETON algorithm, we first check whether all vertices have degrees 0 or $n - 1$ and handle that case appropriately. This leads to significant improvements, especially for graphs with many twins.

This concludes the description of the SKELETON algorithm and its implementation details. Next, we discuss the LINEAR algorithm.

3.3 Linear Algorithm

One of the latest modular decomposition algorithms combines several ideas from previous algorithms to achieve linear time [TCHP08]. It combines a divide-and-conquer approach [EGMS94|DGM97] and factorizing permutations [CHM02]. Additionally, the concept of partition refinement is generalized from sets to trees. The algorithm is regarded as simple compared to other linear time algorithms. We only give an overview of the algorithm without discussing its details.

The algorithm relies on an ordered list of trees as the basic data structure. The leaves correspond to the vertices of the input graph. During this exposition, sets of vertices are also called trees. This can be represented in the data structure by grouping the vertices under a common dummy node. Let $G = (V, E)$ be the input graph. The algorithm works recursively and starts by choosing an arbitrary vertex $x \in V$ as a pivot. Its neighbors and non-neighbors are placed left and right, respectively, resulting in the ordered list of trees $N(x), x, \overline{N}(x)$. With this starting point, the modular decomposition tree of $G[N(x)]$ is calculated recursively. During its computation $\overline{N}(x)$ is refined into $\overline{N}_A(x), \overline{N}_N(x)$, the sets of vertices in $\overline{N}(x)$ with at least one neighbor and no neighbor in $N(x)$, respectively. Let $T(N(x))$ be the modular decomposition tree of $G[N(x)]$. Then we have the ordered list of trees $T(N(x)), x, \overline{N}_A(x), \overline{N}_N(x)$, and the algorithm is applied recursively for $\overline{N}_A(x)$, refining $\overline{N}_N(x)$. This continues and results in the following list of modular decomposition trees of *layers* N_i.

$$\underbrace{T(N_0)}_{N(x)}, x, \underbrace{T(N_1), \dots, T(N_k)}_{\overline{N}(x)}$$

The next step, *refinement*, aims to transform the list into something close to a factorizing permutation. The procedure works on edges incident to pivots and edges between layers. For a strong module M of G not containing x, it holds that $M \subseteq N_i$ for some N_i. Either M is also a strong module of $G[N_i]$, in which case $T(N_i)$ already has a node corresponding to M, or it corresponds to the union of siblings in $T(N_i)$, in which case they are grouped under a new internal node during refinement. After the refinement, the strong modules not containing x appear consecutively (Lemma 1 [TCHP08]). Let $T_k, \dots, T_1, x, T'_1, \dots, T'_\ell$ be the resulting ordered forest. For a strong module M' of G containing x, there exist trees T_i, T'_j, such that $M' \subseteq T_i, \dots, T_1, x, T'_1, \dots, T'_j$, with i and j being maximal (Lemma 2 [TCHP08]). The trees T_i and T_j are called bounding trees of M'. This interval becomes more precise and is made exact by the next step.

The *promotion* step deletes nodes such that the remaining nodes correspond to the strong modules not containing x. When a node is deleted, its children are *promoted* upwards, and any portion of a bounding tree that belongs to a module M' containing x is placed closer to x. This leads to a factorizing permutation (Lemma 3 [TCHP08]).

The ordered list of trees corresponds to $\mathcal{M}(G, x)$, $\{x\}$ and the maximal modules not containing x. The modular decomposition of the trees is already computed. It remains to identify the strong modules containing x and combine them with the modular decomposition tree of G in the *assembly* step. As we computed a factorizing permutation, the strong modules are nested $[\cdots[\cdots[\cdots x \cdots]\cdots]\cdots]$. Similar to the skeleton algorithm, a *spine* is computed, and the previously computed trees are placed accordingly. The remaining tree is the modular decomposition tree of G.

3.3.1 Implementation Details

The authors of the original paper provided a provisional Java implementation of the algorithm [TCHP08 | Ted11].[3] Unfortunately, the website was taken offline during the writing of this thesis, and the implementation is no longer readily available.

Fortunately, a recent implementation of the algorithm in `C++` and Python3 exists[4] [Miz23]. The authors of this implementation used it as a component of an algorithm to compute the twin-width [BKTW20]. They won the exact track of the PACE 2023 algorithm challenge.[5] We used the foreign function interface to call the `C++` implementation directly from Rust. Additionally, we translated the `C++` code to provide a pure Rust implementation.

Our Rust implementation has some small improvements. Instead of iterating over the tree, collecting the node indices, and then iterating over those indices, we use iterators or callbacks to traverse the tree directly. Other than those changes, the reference implementation is already well-optimized. Nevertheless, we uncovered a memory leak and problems with missing headers and could provide fixes as pull requests to the C++ implementation.

3.4 Comparison of the Algorithms

In the previous sections, we saw a description of the FRACTURE, SKELETON, and LINEAR algorithms. Although they are quite different, they share some similarities.

All three algorithms use refinement techniques to computing the modules, either for partitions or on trees, to separate the vertices into sets.

The concept of a v-modular partition $\mathcal{M}(G, v)$ can be seen as a central component of all three algorithms. During the computation of factorizing permutations in the FRACTURE algorithm, the algorithm can be interpreted as recursively calculating $\mathcal{M}(G, v)$ and placing its parts in the ordered partition, such that the modules of v are a factor. The SKELETON algorithm works recursively on the parts of $\mathcal{M}(G, v)$ and computes a modular decomposition of $G/\mathcal{M}(G, v)$, the "spine". In contrast to the other algorithms, it orders the parts to compute the children of the root node of the spine and does not produce a factorizing permutation. The ordering is motivated by an algorithm for computing the transitive order of a prime graph. The LINEAR algorithm recursively computes modular decomposition trees in $N(x)$ and

[3] *https://web.archive.org/web/20231117180242/http://www.cs.toronto.edu/~mtedder/*
[4] *https://github.com/mogproject/modular-decomposition*
[5] *https://pacechallenge.org/2023/*

$\overline{N(x)}$ for some pivot vertex x. It then builds a factorizing permutation in the refinement and promotion steps. At this point, the ordered list of trees corresponds to the parts of $\mathcal{M}(G, x)$ and, due to the factorizing permutation, the modules containing x are nested (or a "factor").

As discussed, the FRACTURE and the LINEAR algorithm use a factorizing permutation to compute the modular decomposition. The FRACTURE algorithm is special in that it fully computes the factorizing permutation and then uses that to compute the modular decomposition. This approach separates the implementation into two distinct phases, while the SKELETON and the LINEAR algorithm recursively compute the modular decomposition instead.

4 Evaluation

In this chapter, we compare the implementations of the algorithms discussed in Chapter 3. As one of the goals of this thesis is to find an algorithm that is efficient in practice, and we evaluate them on real-world and generated graphs.

In particular, we are interested in the following questions.

1. How do the algorithms perform on practical data?
2. Does the algorithm's performance depend on instance properties regarding their decomposition?
3. How do the algorithm's theoretical running times relate to their implementation's' runtimes?
4. Are some algorithms better suited for certain instances than others?
5. How do the algorithms perform on simple graph structures?

In the rest of this chapter, we describe the experiment setup, give an overview of the datasets we used and then attempt to answer the questions.

4.1 Experimental Setup

This section gives a short overview of the hardware and software used in the experiments.

We take several steps to ensure that the results are comparable between algorithms. The Rust implementation of the algorithms of Sections 3.1, 3.2, and 3.3 are called FRACTURE, SKELETON and LINEAR, respectively. We also evaluate the reference implementation in C++ of the LINEAR algorithm and call it LINEAR (REF). We use the Rust foreign function interface to call the C++ code. In the remainder of this chapter, we use algorithm and implementation interchangeably. All algorithms provide the same interface and are divided into *preparation*, *computation*, and a *finalize* step. The first and last steps cover any building and transformation of graph and tree data structures and copying of data between Rust and C++. Only the computation step is measured for the experiments. While the reference implementation provides its execution time through its API, we measure all runtimes the same way to be consistent across algorithms. A comparison has shown that the results of our setup differ by less than a percent for almost all instances.

All experiments are performed on a single core of an Intel Xeon E5-2670 CPU with 8 cores clocked at a base frequency of 2.60 GHz and 16 threads. At most 8 experiments were executed at the same time. 64 GiB RAM is available, and the system runs on Ubuntu 22.04.3. The project is compiled with Rust 1.72.1, target `x86_64-unknown-linux-gnu` and `--release` flag. The wrapper for the C++ code is compiled with the `cc` crate at version 1.0, gcc at version 12.3.0 and optimization level -O3. Our implementation of the algorithms and code to reproduce the experiments is available[1].

[1] *https://github.com/jonasspinner/modular-decomposition*

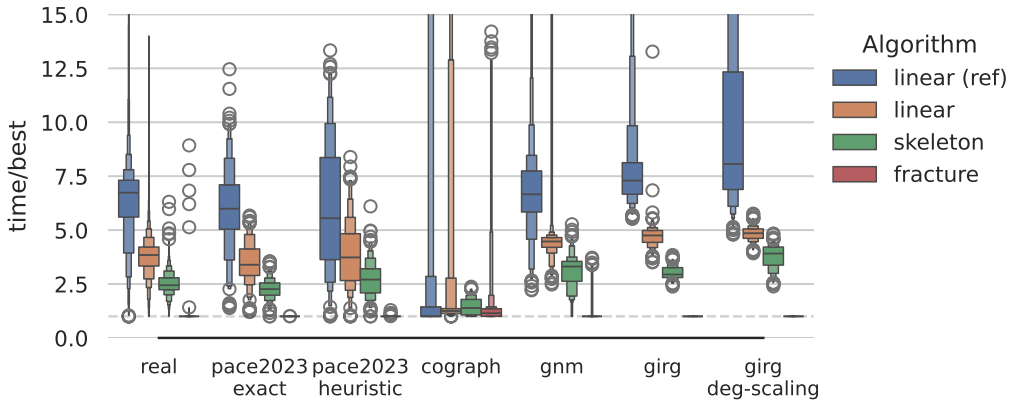


Figure 4.1: A comparison of the algorithm performances for different datasets. The average of three runs for each algorithm and instance is taken. The time for an algorithm is divided by the time of the best algorithm for that instance. The median and quantiles are shown. Outliers are marked as circles.

4.2 Instances

We used several types of real-world and generated graphs to evaluate the algorithms. The following is a short description of each dataset.

- **real**[2]. 2977 real-world networks. These instances are originally from networkrepository.com [RA15] and have previously been used for analyzing graph algorithms [BF23]. Both this and the next dataset contain graphs from many different contexts, and we use them as practical data in Section 4.3.
- **pace2023 exact/heuristic**[3]. 200 instances each. The public and private instances for the exact and heuristic track of the PACE 2023 algorithm challenge. They are based on various real-world and generated graphs and are chosen to present instances with varying difficulty for computing the twin-width [BB23].
- **gnm**. 395 instances. Graphs from the $G(n, m)$ model, i.e., uniformly randomly chosen graphs with n vertices and m edges. The graphs are prime with high probability for a large enough average degree. They also easily allow adjusting the number of edges. They are used in Section 4.4.3 for being prime and in Section 4.5 to investigate the scaling behavior of the algorithms.
- **girg/girg-deg-scaling**.[4] 500/900 instances. Graphs generated from the geometric inhomogeneous and random graph (GIRG) model [BKL19] that have previously been used for analyzing graph algorithms [BF23]. They are used for analyzing the scaling behavior in Section 4.5.

[2] *https://zenodo.org/records/8058432.* `edge_lists_real`. 111 of the 2977 instances were identified as being generated, but we still include all instances.
[3] *https://pacechallenge.org/2023/*
[4] *https://zenodo.org/records/8058432.* `edge_lists_girg/edge_lists_girg_deg_scaling`

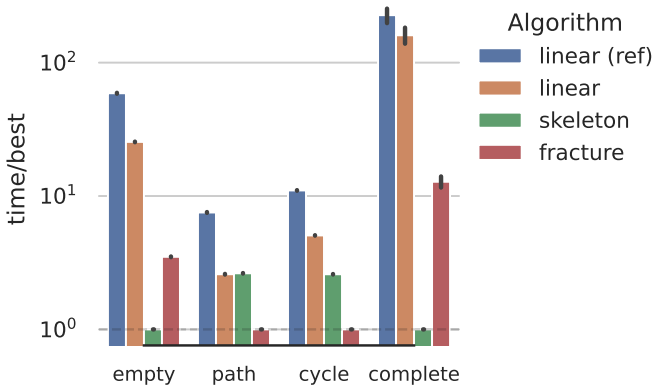


Figure 4.2: Algorithm performance on empty, path, cycle, and complete graphs. Note, that the vertical axis has a logarithmic scale.

- **cograph**. 320 instances. Cographs generated by first sampling a modular decomposition tree with only series and parallel nodes and converting it into a graph. They represent graphs with no prime modules. Details on how the graphs are generated are in Section 4.4.2.
- **empty/path/cycle/complete**. 256/256/256/64 instances. Generated empty, path, cycle, and complete graphs. They have a simple structure, and we use them as a baseline in Section 4.4.1.

4.3 Algorithms for Practical Data

We evaluate the performance of the algorithms for practical data by using the real and pace2023 datasets. They contain graphs from various sources and were previously used to analyze and compare algorithms. The FRACTURE algorithm performs best, as seen in Figure 4.1. The next best algorithm is the SKELETON algorithm, which is about 2.5 times slower than the FRACTURE algorithm. The LINEAR and LINEAR (REF) implementation need about 3.5 and 6.5 times as much time as the FRACTURE algorithm. The algorithms with the non-optimal theoretical running time are faster in practice for these instances. Interestingly, the port of the reference implementation with only a few improvements is already significantly faster than the original.

More detailed results for the instances can be seen in Figure 4.3. The instances in the real and pace2023 datasets are colored blue in the plot. We see that the real dataset provides a wide range of instances, with at most one million vertices and edges. The exact and heuristic instances are present as two and one clusters, respectively. We see the same general result as in Figure 4.1. Additionally, we see an initial overhead for smaller graphs. The algorithms are generally more efficient per vertex and edge for larger instances.

Another interesting detail is that the running time is only partially determined by the number of vertices and edges. There are instances with very similar $n + m$, that result in different runtimes, even for the same algorithm. The individual influence of n and m independently is ignored when just looking at their sum, but the other datasets' results also suggest that the graph's structure influences the runtime.

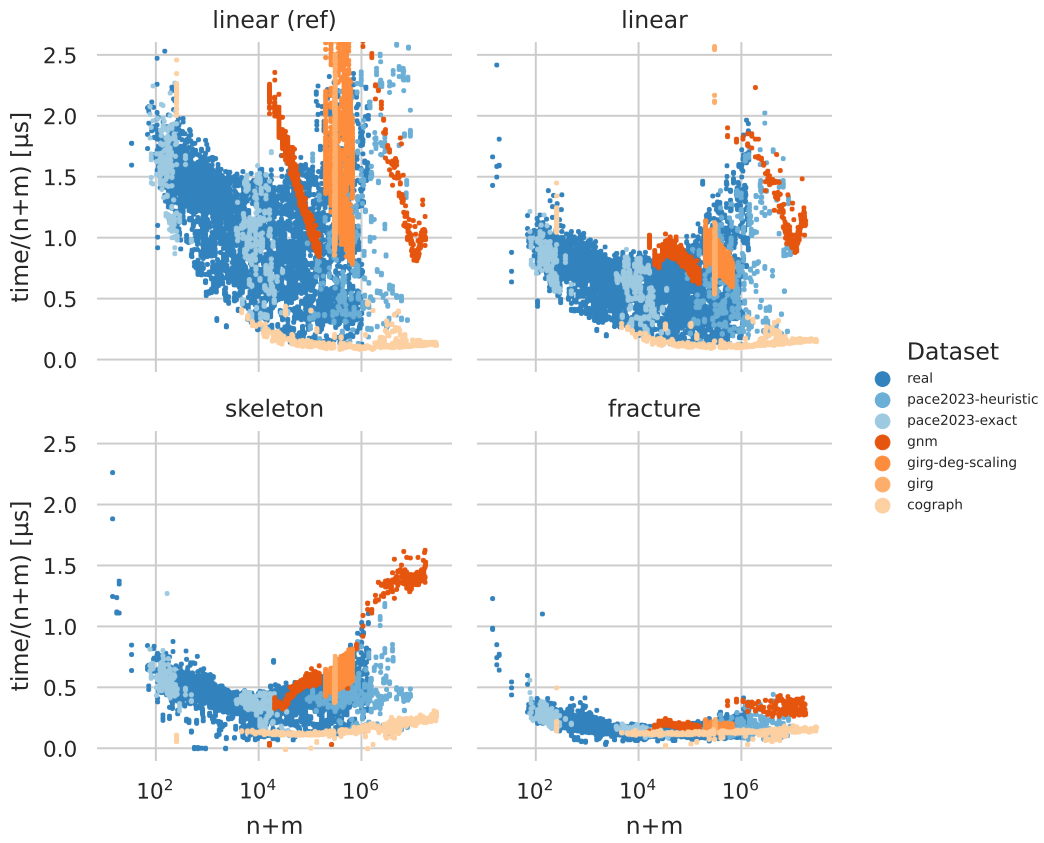


Figure 4.3: Algorithm performance with respect to $n + m$. Note that the horizontal axis is scaled logarithmically. For each algorithm 5492 instances with 3 runs each are plotted. The LINEAR and LINEAR (REF) implementation have 10 and 537 runs ($\approx 3\%$) with a value of more than 2.5 µs, respectively.

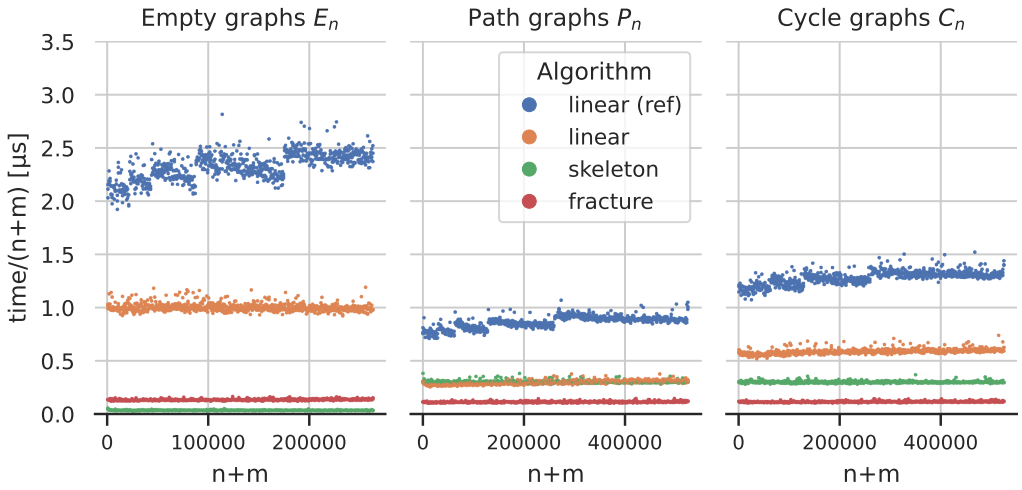


Figure 4.4: Simple graph classes: empty graphs, path graphs and the cycle paths for n up to 2^{18}. The graphs have $0, n-1$ and n edges respectively.

The FRACTURE algorithm performs best for the real and pace2023 datasets. For almost all instances, it is the best algorithm and is 2.5 times faster on average than the next best algorithm.

4.4 Influence of Graph Structure on Runtimes

In this section, we use graphs with different structures to determine if the properties of the modular decomposition influence the algorithm's performance. We start with very simple graphs and then continue with graphs that do not contain any prime modules, and finally, graphs that are prime with high probability.

4.4.1 Simple Graphs

Although not very practical with respect to real-world applications, simple graphs are a good starting point to evaluate the algorithms.

We look at the following graph classes: empty graphs E_n with n vertices and no edges, the path graphs P_n, and the cycle graphs C_n. Note that the path graph and cycle paths are prime. Any graph with less than $n-1$ edges is not prime. We vary the number of vertices. Note that empty graphs have a single parallel module and that the path graphs are a minimal prime graph. The results are shown in Figure 4.4. The FRACTURE and SKELETON algorithms are consistent, and their performances on paths and cycles are very similar. The FRACTURE algorithm is the fastest, and the SKELETON algorithm is faster for the empty graph. An optimization introduced in our implementation that tries to handle series and parallel subgraphs without repeatedly computing the spine, explains this. The linear algorithms are significantly slower. The reference implementation is about twice as slow as the Rust implementation.

Surprisingly, both LINEAR and LINEAR (REF) are slowest for the empty graphs and fastest for the path graphs. This might be explained by looking at the algorithm. Recall that the algorithm works recursively with layers N_i, $i \in [0, k]$, where N_i are the vertices with distance $i+1$ to the pivot vertex. A possible exception is the last layer N_k, which might hold the

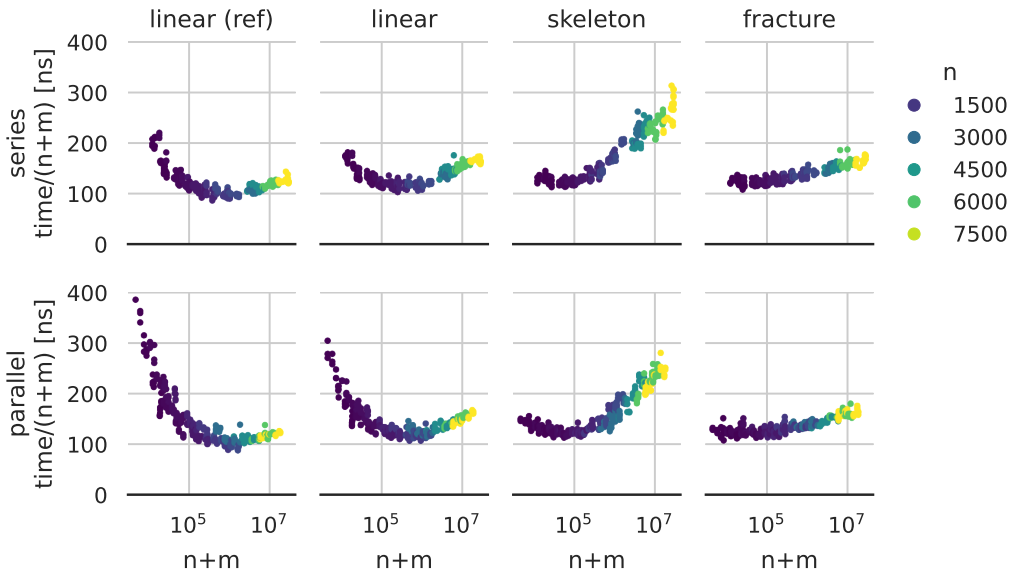


Figure 4.5: Random cographs with parameters $a = 2$, $b = 8$, and n ranging from 2^8 to 2^{13}. Graphs with a series and parallel root node are shown at the top and bottom, respectively. Note that the horizontal axis has a logarithmic scale, and the runtime is normalized. For this choice of parameters, the modular decompositions of the instances have approximately $n/4$ inner nodes.

vertices that are not reachable from the pivot vertex. For the empty graph, all vertices are unreachable from the first pivot vertex. For the path graphs, every layer consists of exactly one vertex. For the cycle graphs, almost all layers have two vertices with the same distance to the pivot vertex. This might degrade the performance for empty and cycle graphs.

Another weird result is the step pattern for LINEAR (REF). This might be the result of some data structure that grows exponentially. Unfortunately, we were unable to track down the source of that effect. Interestingly, the Rust port does not behave that way. More alarmingly, the normalized running time of the LINEAR (REF) algorithm does not seem to stay constant for a growing number of vertices, and the experiment results suggest an additional logarithmic factor.

In conclusion, the FRACTURE and SKELETON algorithm performs best for simple graphs. For the SKELETON algorithm, the empty graphs are handled by an optimization we introduced in the implementation. The LINEAR and LINEAR (REF) performance degrades with an additional edge added to a path graph.

4.4.2 Cographs

The graphs with only series and parallel nodes in their modular decomposition are called cographs. They are fully defined by the tree structure of their modular decomposition and the type of its root node.

We generate random cographs, by first generating a random tree with a given number of leaves and then assigning the node types, starting from the root. The random model is parameterized by three parameters, $n, a, b \in \mathbb{N}$, $2 \leq a \leq b \leq n$. The number of leaves is determined by n, and a and b define the minimum and maximum number of children of inner

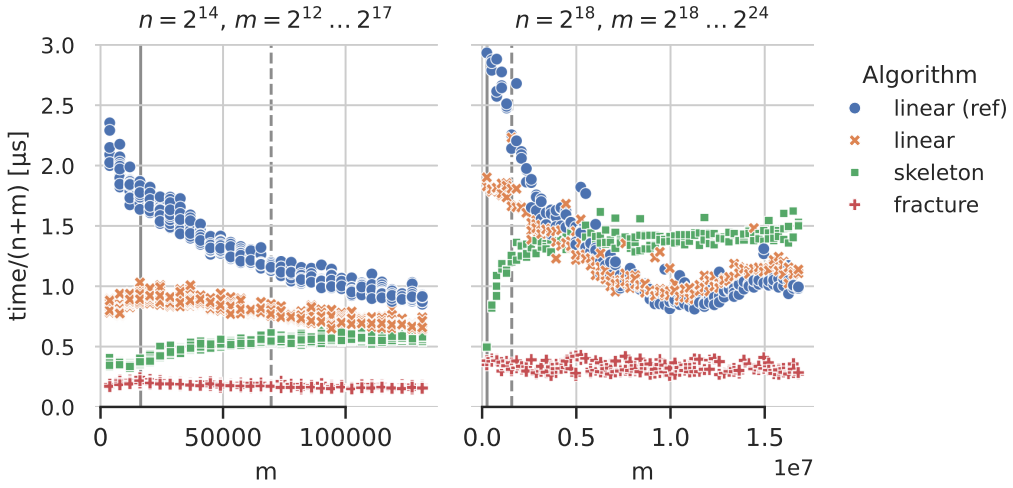


Figure 4.6: Random $G(n, m)$ graphs with fixed n and growing m. Note that the horizontal axis has a linear scale. Plotting $n + m$ instead of m on the horizontal axis only shifts the values by the fixed value of n. A constant normalized runtime implies a $O(m)$ scaling behavior. The first vertical line marks n, and the second vertical line marks the point, where all instances to its right are prime graphs.

nodes. To generate the tree, a set of nodes without parents is maintained and initialized with the leaves. Then, an out-degree d is sampled from the uniform distribution $U([a, b])$, and a new inner node is added with d children from the set of parent-less nodes. When only a single node remains, the algorithm stops.

The algorithms are compared for 220 such graphs, as seen in Figure 4.5, with parameters $a = 2, b = 8$ and n ranging from 2^8 to 2^{13}. Note that all algorithms perform very well for cographs. The normalized runtime is measured in *ns* instead of µs. This can also be seen in Figure 4.3, where cographs have the lowest normalized runtime compared to the other datasets. All algorithms perform approximately as well on cographs as the FRACTURE algorithm for the simple graphs in Section 4.4.1. The FRACTURE and SKELETON algorithms perform best for a smaller number of vertices. This changes for larger choices for n. For $n = 2^{13}$, the reference implementation of the LINEAR algorithm in C++ performs the best.

Interestingly, the results do not match the scaling behavior we would expect for a linear algorithm. A linear scaling behavior, with maybe some initial overhead, would show up as the normalized time approaching a constant function for larger $n + m$. This might be caused by the fact that we are far away from the worst case. All algorithms are a lot more efficient for cographs than for other kinds of graphs.

Nevertheless, the LINEAR (REF) algorithm performs best for cographs with at $n + m$ at least 10^5. The LINEAR and FRACTURE algorithms are not far off, but the SKELETON algorithm performs the worst by taking approximately twice as long for larger instances.

4.4.3 Prime Graphs

In the previous section, we investigated instances with no prime module. The other extreme are prime graphs. Note that the property of being prime restricts the number of edges. Any prime graph has at least $n - 1$ edges and a P_4 as an induced subgraph.

We do not use a dataset of prime graphs directly but use a random graph model that results in graphs with a giant prime module most of the time. A preliminary study showed that the $G(n, m)$ random graph model, uniformly randomly chosen graphs with a fixed number of vertices and edges, can be used for that when the right parameters are used. For $m = 0$ the graphs are the empty graph with a single parallel module. When m is lower than a small multiple of n, there are many and varied types of module in the modular decomposition tree. Note that a prime graph with a minimal number of edges is the path on n vertices with $m = n - 1$ edges. For most values of m, the modular decomposition is almost always one giant prime module. Another reason for investigating $G(n, m)$ random graphs is the ability to vary the number of vertices and edges precisely. This allows the comparison of the theoretical running times with the actual performance of the algorithms. We will do that in Section 4.5.

We chose $n = 2^{14}$ and $n = 2^{18}$ and varied the number of edges. The normalized running times can be seen in Figure 4.6. Both implementations of the LINEAR algorithm start out worst because of some overhead. All algorithms approach a constant normalized runtime. Recall the theoretical running time of $O(n + m)$ for the LINEAR algorithm and $O(n + m \log n)$ for the SKELETON and FRACTURE algorithm. When n is fixed, the worst-case runtime is linear with respect to m. This is supported by Figure 4.6. The number of vertices should affect the scaling with the number of edges for the SKELETON and FRACTURE algorithm. The FRACTURE algorithm has only a slightly larger normalized runtime for $n = 2^{18}$. The SKELETON algorithm takes almost three times the amount of time per vertex and edge for $n = 2^{18}$ than for $n = 2^{14}$. This makes the linear algorithms better than the SKELETON algorithm for $n = 2^{18}$. The FRACTURE algorithm performs the best for both choices for n and all number of edges investigated in this experiment.

For $G(n, m)$ graphs, which are mostly prime for a larger number of edges, the FRACTURE algorithm performs best and scales consistently with the number of edges. The LINEAR algorithms become more efficient for more edges but still take at least twice as much time as the FRACTURE algorithm. The SKELETON algorithm performs well for the smaller number of vertices but is heavily affected when the n is larger.

4.5 Scaling Experiments

In this section, we want to evaluate the algorithms on datasets of varying sizes to look at their scaling behavior and to see if the theoretical running times influence the practical runtimes.

The girg-deg-scaling dataset is useful for analyzing the scaling behavior of the algorithms. All instances have approximately the same number of vertices, and the average number of edges takes on a range of values. The reference implementation of the LINEAR algorithm performs worst and has a high variance in runtime for graphs with a similar number of edges. In contrast, the FRACTURE algorithm is consistent and almost three times as fast as any other algorithm. All algorithms, except the SKELETON algorithms, are more efficient for a larger number of edges.

Another graph model useful for investigating the algorithm performance for a range of instance sizes is the $G(n, m)$ model. We looked at the case for constant n and varying m in Section 4.4.3 and Figure 4.6. We predicted and observed the $O(m)$ scaling behavior.

For the $O(n + m \log n)$ algorithms, SKELETON, and FRACTURE, we would like to know if we can see the $\log n$ factor. Fixing m and varying n is insufficient, as the $O(n)$ part would dominate the $O(\log n)$ term. We choose to let m grow linearly with n. Then, the theoretical running time is $O(n \log n)$, and normalizing with $n + m$, which is $O(n)$, leaves the $\log n$ factor.

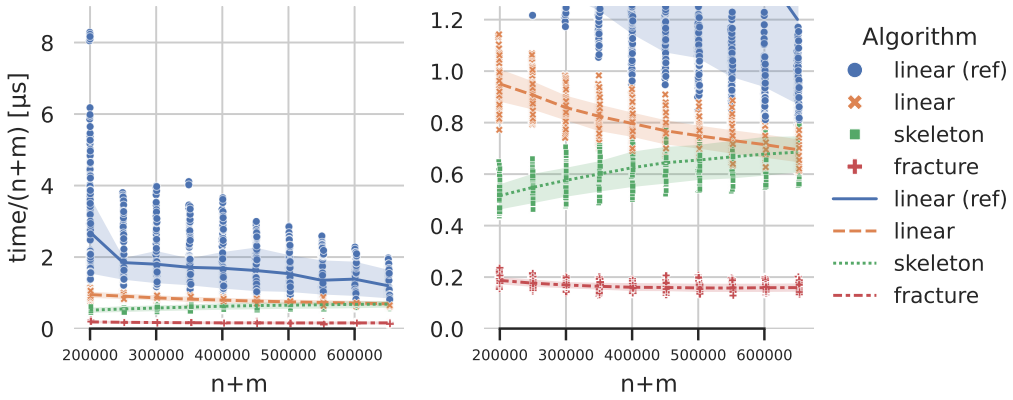


Figure 4.7: GIRG graphs from the girg-deg-scaling dataset. $n \approx 5 \cdot 10^4$ and $m \approx 15 \cdot 10^4 \ldots 60 \cdot 10^4$. Both plots show the same data points at different scales of the y-axis. The bands are between the 25% and 75% quantile, and the mean value is shown as a line.

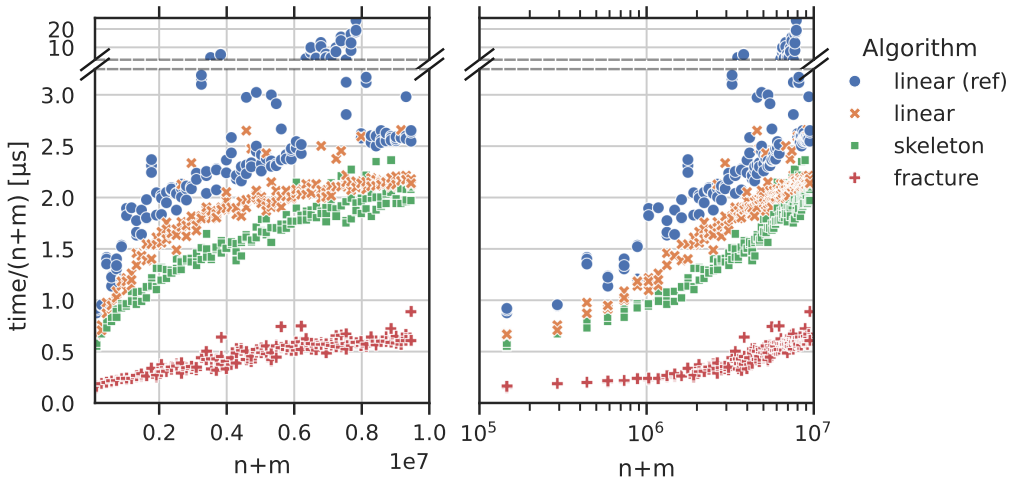


Figure 4.8: Random $G(n, m)$ graphs with $m = 8n$ and n ranging from 2^{14} to 2^{20}. Both plots show the same data points, only the scaling of the horizontal axis is different. The plot is divided into $[0, 3.25]$ and $(3.25, 26]$ on the vertical axis to accommodate outliers for the LINEAR (REF) algorithm. For each instance, three runs are plotted.

We choose $m = 8n$ and let n vary. Figure 4.8 shows the experiments. The FRACTURE algorithm has the best performance and is more than twice as fast as the other algorithms for almost all values of n. The next best algorithm is SKELETON, followed by LINEAR, and finally LINEAR (REF). The LINEAR (REF) algorithm has heavy outliers that persist over multiple runs.

An $O(n \log n)$ scaling behavior would end up as a linear function in the plot of normalized runtimes and a logarithmic horizontal axis. This matches the data for larger instances. This is expected for the FRACTURE and SKELETON algorithms but very surprising for the LINEAR and LINEAR (REF) algorithms.

To summarize, the actual runtimes behavior of the FRACTURE and SKELETON algorithms can be predicted by their theoretical running times, with the log factor being less pronounced for the FRACTURE algorithm. Unfortunately, the linear algorithms behave similarly. We are unsure if the scaling behavior of all algorithms is a result of the dataset or if this is an indicator of a performance bug in the linear implementations.

5 Conclusion

In this thesis, we presented three modular decomposition algorithms, FRACTURE, SKELETON, and LINEAR, and showed how to implement them efficiently. The algorithms share common concepts, like partition refinement, factorizing permutations, and divide-and-conquer strategies based on the ν-modular partition.

Overall, the FRACTURE algorithm was nearly 2.5 times faster than the next best algorithm for almost all datasets. The effect of the log factor is minimal. We improved upon the implementation of LINEAR (REF) by porting it to Rust and only applying minor changes to avoid allocations and multiple traversals of the same tree nodes.

However, both linear implementations showed worse results than we would have predicted. The LINEAR (REF) behaves weirdly for simple graph classes, such as empty graphs, path graphs, and cycle graphs. Furthermore, a scaling experiment with an increasing number of edges and a constant average vertex degree showed that the linear algorithms are still worse than the $O(n + m \log n)$ algorithms. Additionally, their runtime increases similarly to the non-linear algorithms we evaluated. It remains open, whether that is the result of the algorithm, the implementations, or the choice of graphs.

Future Work

In terms of future work, although the thesis only covered modular decomposition on undirected graphs, the algorithmic ideas and implementation details might transfer to similar decomposition algorithms, such as modular decomposition on hypergraphs or directed graphs. One can study the effects of choices throughout the algorithm, such as the choice of pivot elements. Another would be the choice of part X in the ordered vertex partition algorithm of the skeleton. Those choices might influence the runtime performance.

Finally, one could use the implementations in this thesis as a starting point for efficient and practical implementations of several algorithms that use modular decomposition as a preprocessing step. For example, the parameterized algorithms for maximum matching, triangle counting, minimum cut, maximum flow, and more from [KN18].

Bibliography

[Art11] Artem Polyvyanyy. *BPStruct*. 2011. URL: *https://code.google.com/archive/p/bpstruct/* (visited on 06/16/2023).

[BB23] Max Bannach and Sebastian Berndt. "PACE Solver Description: The PACE 2023 Parameterized Algorithms and Computational Experiments Challenge: Twin-width". In: *18th International Symposium on Parameterized and Exact Computation (IPEC 2023)*. Vol. 285. Schloss-Dagstuhl - Leibniz Zentrum für Informatik, 2023, 35:1–35:14. ISBN: 978-3-95977-305-8. DOI: *10.4230/LIPIcs.IPEC.2023.35*.

[BCdMR05] Anne Bergeron, Cedric Chauve, Fabien de Montgolfier, and Mathieu Raffinot. "Computing Common Intervals of K Permutations, with Applications to Modular Decomposition of Graphs". In: *Algorithms – ESA 2005*. Berlin, Heidelberg: Springer, 2005, pp. 779–790. ISBN: 978-3-540-31951-1. DOI: *10.1007/11561071_69*.

[BF23] Thomas Bläsius and Philipp Fischbeck. *On the External Validity of Average-Case Analyses of Graph Algorithms (Data, Docker, and Code)*. Zenodo, June 20, 2023. DOI: *10.5281/zenodo.8058432*. URL: *https://zenodo.org/records/8058432* (visited on 02/17/2024).

[BKL19] Karl Bringmann, Ralph Keusch, and Johannes Lengler. "Geometric Inhomogeneous Random Graphs". In: *Theoretical Computer Science* Volume 760 (Feb. 14, 2019), pp. 35–54. ISSN: 0304-3975. DOI: *10.1016/j.tcs.2018.08.014*.

[BKTW20] Edouard Bonnet, Eun Jung Kim, Stephan Thomasse, and Remi Watrigant. "Twin-Width I: Tractable FO Model Checking". In: *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. 2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS). Durham, NC, USA: IEEE, Nov. 2020, pp. 601–612. ISBN: 978-1-72819-621-3. DOI: *10.1109/FOCS46700.2020.00062*.

[Cap97] Christian Capelle. "Décomposition de Graphes et Permutations Factorisantes". PhD thesis. Univ. de Montpellier II, 1997.

[CH94] Alain Cournier and Michel Habib. "A New Linear Algorithm for Modular Decomposition". In: *Trees in Algebra and Programming — CAAP'94*. Berlin, Heidelberg: Springer, 1994, pp. 68–84. ISBN: 978-3-540-48373-1. DOI: *10.1007/BFb0017474*.

[CHM02] Christian Capelle, Michel Habib, and Fabien Montgolfier. "Graph Decompositions and Factorizing Permutations". In: *Discrete Mathematics & Theoretical Computer Science* Volume Vol. 5 (Jan. 1, 2002). ISSN: 1365-8050. DOI: *10.46298/dmtcs.298*.

[CPS85] D. G. Corneil, Y. Perl, and L. K. Stewart. "A Linear Recognition Algorithm for Cographs". In: *SIAM Journal on Computing* Volume 14 (Nov. 1985), pp. 926–934. ISSN: 0097-5397. DOI: *10.1137/0214065*.

[DGM01] Elias Dahlhaus, Jens Gustedt, and Ross M McConnell. "Efficient and Practical Algorithms for Sequential Modular Decomposition". In: *Journal of Algorithms* Volume 41 (Nov. 1, 2001), pp. 360–387. ISSN: 0196-6774. DOI: *10.1006/jagm.2001.1185*.

[DGM97] Elias Dahlhaus, Jens Gustedt, and Ross M. McConnell. "Efficient and Practical Modular Decomposition". In: *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. USA: Society for Industrial and Applied Mathematics, Jan. 5, 1997, pp. 26–35. ISBN: 978-0-89871-390-9.

[EGMS94] A. Ehrenfeucht, H. N. Gabow, R. M. Mcconnell, and S. J. Sullivan. "An O(N2) Divide-and-Conquer Algorithm for the Prime Tree Decomposition of Two-Structures and Modular Decomposition of Graphs". In: *Journal of Algorithms* Volume 16 (Mar. 1, 1994), pp. 283–294. ISSN: 0196-6774. DOI: *10.1006/jagm.1994.1013*.

[FL15] Adrian Fritz and Fynn Lyte. *Modular-Decomposition*. 2015. URL: *https://github.com/LyteFM/modular-decomposition* (visited on 06/12/2023).

[Gal67] T. Gallai. "Transitiv orientierbare Graphen". In: *Acta Mathematica Academiae Scientiarum Hungarica* Volume 18 (Mar. 1, 1967), pp. 25–66. ISSN: 1588-2632. DOI: *10.1007/BF02020961*.

[GKBC04] Julien Gagneur, Roland Krause, Tewis Bouwmeester, and Georg Casari. "Modular Decomposition of Protein-Protein Interaction Networks". In: *Genome Biology* Volume 5 (July 21, 2004), R57. ISSN: 1474-760X. DOI: *10.1186/gb-2004-5-8-r57*.

[Gol80] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. 1980. ISBN: 978-0-12-289260-8.

[HM79] M. Habib and M. C. Maurer. "On the X-join Decomposition for Undirected Graphs". In: *Discrete Applied Mathematics* Volume 1 (Nov. 1, 1979), pp. 201–207. ISSN: 0166-218X. DOI: *10.1016/0166-218X(79)90043-X*.

[Hop71] John Hopcroft. "An n Log n Algorithm for Minimizing States in a Finite Automaton". In: *Theory of Machines and Computations*. Elsevier, 1971, pp. 189–196. ISBN: 978-0-12-417750-5. DOI: *10.1016/B978-0-12-417750-5.50022-1*.

[HP10] Michel Habib and Christophe Paul. "A Survey of the Algorithmic Aspects of Modular Decomposition". In: *Computer Science Review* Volume 4 (Feb. 1, 2010), pp. 41–59. ISSN: 1574-0137. DOI: *10.1016/j.cosrev.2010.01.001*.

[HPV99] Michel Habib, Christophe Paul, and Laurent Viennot. "Partition Refinement Techniques: An Interesting Algorithmic Tool Kit". In: *International Journal of Foundations of Computer Science* Volume 10 (June 1999), pp. 147–170. ISSN: 0129-0541. DOI: *10.1142/S0129054199000125*.

[JSC72] L.O. James, R.G. Stanton, and Donald Cowan. "Graph Decomposition for Undirected Graphs". In: *Utilitas Mathematica*. Third Southeastern Conference On Combinatorics, Graph Theory, And Computing. Florida Atlantic University, 1972, pp. 281–290.

[Kar19] Stefan Karpinski. "GraphModularDecomposition.Jl". 2019. URL: *https://github.com/StefanKarpinski/GraphModularDecomposition.jl* (visited on 06/16/2023).

[KN18] Stefan Kratsch and Florian Nelles. “Efficient and Adaptive Parameterized Algorithms on Modular Decompositions”. In: *26th Annual European Symposium on Algorithms (ESA 2018)*. Vol. 112. Schloss-Dagstuhl - Leibniz Zentrum für Informatik, 2018, 55:1–55:15. ISBN: 978-3-95977-081-1. DOI: *10.4230/LIPIcs.ESA.2018.55*.

[MdM05] Ross M. McConnell and Fabien de Montgolfier. “Linear-Time Modular Decomposition of Directed Graphs”. In: *Discrete Applied Mathematics* Volume 145 (Jan. 15, 2005), pp. 198–209. ISSN: 0166-218X. DOI: *10.1016/j.dam.2004.02.017*.

[Miz23] Yosuke Mizutani. *Mogproject/Modular-Decomposition*. 2023. URL: *https://github.com/mogproject/modular-decomposition*.

[Möh85] Rolf H. Möhring. “Algorithmic Aspects of Comparability Graphs and Interval Graphs”. In: *Graphs and Order: The Role of Graphs in the Theory of Ordered Sets and Its Applications*. Dordrecht: Springer Netherlands, 1985, pp. 41–101. ISBN: 978-94-009-5315-4. DOI: *10.1007/978-94-009-5315-4_2*.

[MS00] Ross M. Mcconnell and Jeremy P. Spinrad. “Ordered Vertex Partitioning”. In: *Discrete Mathematics & Theoretical Computer Science* Volume Vol. 4 no. 1 (Jan. 1, 2000). ISSN: 1365-8050. DOI: *10.46298/dmtcs.274*.

[MS89] John H. Muller and Jeremy Spinrad. “Incremental Modular Decomposition”. In: *Journal of the ACM* Volume 36 (Jan. 1, 1989), pp. 1–19. ISSN: 0004-5411. DOI: *10.1145/58562.59300*.

[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*. USA: Society for Industrial and Applied Mathematics, Jan. 23, 1994, pp. 536–545. ISBN: 978-0-89871-329-9.

[MS99] Ross M. McConnell and Jeremy P. Spinrad. “Modular Decomposition and Transitive Orientation”. In: *Discrete Mathematics* Volume 201 (Apr. 28, 1999), pp. 189–241. ISSN: 0012-365X. DOI: *10.1016/S0012-365X(98)00319-7*.

[PLE71] A. Pnueli, A. Lempel, and S. Even. “Transitive Orientation of Graphs and Identification of Permutation Graphs”. In: *Canadian Journal of Mathematics* Volume 23 (Feb. 1971), pp. 160–175. ISSN: 0008-414X, 1496-4279. DOI: *10.4153/CJM-1971-016-5*.

[RA15] Ryan Rossi and Nesreen Ahmed. “The Network Data Repository with Interactive Graph Analytics and Visualization”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* Volume 29 (Mar. 4, 2015). ISSN: 2374-3468, 2159-5399. DOI: *10.1609/aaai.v29i1.9277*.

[Sal04] Andras Salamon. *Graph-ModularDecomposition-0.15 - Modular Decomposition of Directed Graphs - Metacpan.Org*. 2004. URL: *https://metacpan.org/release/AZS/Graph-ModularDecomposition-0.15* (visited on 06/16/2023).

[Spi03] Jeremy P. Spinrad. *Efficient Graph Representations*. Providence, R.I: American Mathematical Society, 2003. ISBN: 978-0-8218-2815-1.

[Spi24] Jonas Spinner. “Modular Decomposition. Thesis Repository.” GitHub. 2024. URL: *https://github.com/jonasspinner/modular-decomposition* (visited on 02/21/2024).

[TCHP08] Marc Tedder, Derek Corneil, Michel Habib, and Christophe Paul. "Simpler Linear-Time Modular Decomposition Via Recursive Factorizing Permutations". In: *Automata, Languages and Programming*. Berlin, Heidelberg: Springer, 2008, pp. 634–645. ISBN: 978-3-540-70575-8. DOI: *10.1007/978-3-540-70575-8_52*.

[Ted] Marc Tedder. "Simpler, Linear-Time Modular Decomposition via Recursive Factorizing Permutations (Code)". URL: *https://web.archive.org/web/20231117180242/http://www.cs.toronto.edu/~mtedder/* (visited on 02/06/2024).

[Ted11] Marc Tedder. "Applications of Lexicographic Breadth-first Search to Modular Decomposition, Split Decomposition, and Circle Graphs". PhD thesis. Aug. 31, 2011. URL: *https://tspace.library.utoronto.ca/handle/1807/29888*.

[Ted15] Marc Tedder. "Simpler, Linear-Time Transitive Orientation via Lexicographic Breadth-First Search". Mar. 9, 2015. arXiv: *1503.02773*.