Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Die Berechnung der modularen Zerlegung eines ungerichteten Graphen in linearer Zeit entsprechend dem Algorithmus von Tedder et al.

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Malte Quiter

Gutachter/innen: Prof. Dr. Stefan Kratsch
Prof. Dr. Johannes Köbler

Inhaltsverzeichnis

Einleitung

Bei dem Entwurf von Algorithmen stellt sich die abstrakte Struktur des Graphen oft als ein geeignetes Mittel heraus, um viele real auftretende Probleme zu modellieren. Das Erforschen von Graphalgorithmen und die Suche nach effizienten Algorithmen stellt demnach schon lange ein zentrales Feld der theoretischen Informatik dar. Für viele der bekannten Graphprobleme existieren allgemeine algorithmische Lösungen, deren worst-case-Laufzeiten seit vielen Jahren entweder nicht geschlagen werden konnten oder welche als weitestgehend optimal angesehen werden. Hierzu sei exemplarisch der Algorithmus von Silvio Micali and Vijay V. Vazirani für Maximum Matching erwähnt [44], welcher mit einer Laufzeit in $\mathcal{O}(m \cdot \sqrt{n})$ seit nun mehr als 40 Jahren die beste, allgemeine Lösung für das Problem darstellt.

Dennoch gibt es Ansätze, diese Laufzeiten in speziellen Fällen zu schlagen. Eine Möglichkeit besteht darin, die Struktur von Probleminstanzen zu berücksichtigen, um mit maßgeschneiderten Algorithmen effizientere Lösungsverfahren anbieten zu können. Ist im Vorfeld bekannt, dass eine strukturelle Eigenschaft für alle Probleminstanzen garantiert ist, so ist es im Fall eines Graphproblems sinnvoll, die zu den Probleminstanzen korrespondierende Graphklasse zu betrachten und deren Eigenschaft algorithmisch auszunutzen. Oftmals gelingen auf diese Weise Algorithmen, welche – für den speziellen Anwendungsfall – mit deutlich schnelleren Laufzeiten aufwarten können. Lineare Algorithmen für Maximum Matching belegen dies gut: [53] für Cographen, [23] für P_4-tidy Graphen oder [9] für chordal bipartite Graphen. Vielfach kann die modulare Zerlegung von Graphen dazu genutzt werden, Instanzen einer Graphklasse zu erkennen. Ein einfaches Beispiel bildet die Klasse der Cographen, deren Mitglieder als P_4-freie Graphen [26] durch modulare Zerlegungsbäume ohne prime Knoten gekennzeichnet sind. Weitere Beispiele finden sich in [27, 48, 41]. Somit bildet die modulare Zerlegung ein nützliches Werkzeug bei dem Kategorisieren von Graphen sowie bei der Entwicklung spezialisierter Algorithmen.

Parametrisierte Algorithmen bilden eine ähnliche Möglichkeit für potenzielle Laufzeitverbesserungen. Hierbei werden strukturelle Aspekte von Probleminstanzen genutzt, deren Grad der Ausprägung – mittels Parametern – numerisch bemessen werden kann. Entsprechend erfolgt die Bewertung der Komplexität diesbezüglich ausgelegter Algorithmen, neben der Größe der Eingabe, anhand des untersuchten Parameters[1].

[1] Typischerweise wird dieses Vorgehen gewählt, um die nicht-polynomielle Anteile der Komplexität NP-harter Probleme durch den Parameter zu beschränken. *Fixed-Parameter Tractability (FPT)* ist hier der zentrale Begriff [16]. Gegenstand vieler Veröffentlichungen ist das Entscheidungsproblem Vertex Cover. [22] gibt hierzu einen Überblick.

Ein Parameter, welcher dabei in jüngerer Vergangenheit Aufmerksamkeit erfahren hat (z.B. in [13, 37, 38]), ist der Graphparameter der modularen Weite $mw(G)$ eines Graphen G. Dieser beschreibt die maximale Anzahl von Kindern eines primen Knotens in dem modularen Zerlegungsbaum T_G von G. Oftmals können (Teil-)Probleme anhand von Quotientengraphen gelöst werden, wobei deren Größe durch die modulare Weite beschränkt ist. Auf diese Weise ergeben sich kürzere Berechnungszeiten, bei kleiner modularer Weite [13]. Da die modulare Zerlegung eines Graphen eindeutig bestimmt ist [25] und zudem in $\mathcal{O}(n+m)$ ermittelt werden kann [40, 14, 42, 50] eignet sie sich entsprechend gut für Entwicklung parametrisierter Algorithmen[2]. Beispielsweise konnte für Maximum Bi-Matching, unter Nutzung der modulare Zerlegung, ein Algorithmus [37] gefunden werden, welcher bereits im ungünstigsten Fall, d.h. $mw(G) = \Theta(n)$, die Laufzeit des besten bekannten unparametrisierten Algorithmus [24] erreicht und sie bei geringerer modularer Weite sogar schlägt.

Sei es die Erkennung von Instanzen einer Graphklasse oder die Berechnung der modularen Zerlegung als Vorverarbeitungsschritt in parametrisierten Algorithmen – die Fülle an struktureller Information über einen Graphen, welche mittels modularer Zerlegung kostengünstig gewonnen werden kann, motiviert zahlreiche Anwendungsfälle[3].

Grundsätzlich handelt es sich bei der modularen Zerlegung um eine kombinatorische Zerlegungtechnik, welche auf viele diskrete Strukturen wie Graphen, gerichtete Graphen, 2-Strukturen, Hypergraphen, u.a. anwendbar ist [4]. Im Kontext von Graphen, bezeichnet der Begriff des Moduls eine Menge $M \subseteq V(G)$ der Knoten eines Graphen G, sodass alle Knoten innerhalb von M, die gleiche Nachbarschaft zu den Knoten außerhalb von M haben. Die Grundidee der modularen Zerlegung ist es, die Module eines Graphen zu identifizieren und deren Inklusionsbeziehung zueinander zu darzustellen. Bereits 1967 wurde diese Technik erstmals von Tibor Gallai im Rahmen der Erforschung transitiv orientierbarer Graphen angewendet [25]. Gallai zeigte dabei ebenfalls, dass die Familie der Module eines Graphen G durch einen gewurzelten Baum repräsentiert werden kann. Dieser definiert den modularen Zerlegungsbaum T_G.

Für die Berechnung der modularen Zerlegung eines Graphen wurden im Verlauf der Jahre viele Algorithmen vorgeschlagen[4]. Der Algorithmus von Cowan et al. [36] aus dem Jahr 1972 benötigt hierfür eine Laufzeit in $\mathcal{O}(n^4)$ und gilt gemeinhin als erster polynomieller Algorithmus zur Berechnung der modularen Zerlegung. Seitdem konnten

[2] Insbesondere für Probleme in P (*FPT in P*) [13].
[3] Weitere umfassen etwa das Graphzeichnen [46] oder die Graphkompression [2].
[4] Die Darstellung der geschichtlichen Entwicklung der Algorithmen, orientiert sich an [31] sowie [50].

diverse Arbeiten Verbesserungen der Laufzeit erzielen. So erreichte ein Algorithmus von Habib und Maurer [30] bereits im Jahr 1979 eine kubische, ein Algorithmus von Müller und Spinrad [45] im Jahr 1989 eine quadratische Laufzeit. Im Jahr 1994 wurden schließlich mit zwei unabhängigen Ergebnissen von McConnell und Spinrad [40] beziehungsweise Cournier und Habib [14] Algorithmen mit linearen Laufzeiten vorgeschlagen. Aufgrund ihrer hohen technischen Komplexität gelten diese vorrangig als theoretische Meilensteine.

Ein weiterer Algorithmus des Jahres 1994 stammt von Ehrenfeucht et al. [21]. Obwohl dieser nur eine quadratische Laufzeit erreichen konnte und für 2-Strukturen formuliert ist, bildete er die Grundlage für viele schnellere Algorithmen der späten 1990er Jahre. Hierzu sei auf die Algorithmen von McConnell und Spinnrad [43] mit einer Laufzeit in $\mathcal{O}(n + m \log n)$, sowie auf gleich zwei Algorithmen von Dahlhaus et al. [15] in $\mathcal{O}(n + \alpha(n, m)m)$ beziehungsweise $\mathcal{O}(n + m)$ verwiesen, welche alle als Verbesserungen des Algorithmus von Ehrenfeucht gesehen werden können[5].

In den Algorithmen von Dahlhaus et al. besteht die grundlegende Idee darin, einen beliebigen Knoten $x \in V(G)$ zu wählen und zunächst die modularen Zerlegungsbäume der induzierten Subgraphen $G[N(x)]$ sowie $G[\overline{N(x)}]$ rekursiv zu berechnen. Da für alle Module ohne den Knoten x gilt, dass sie entweder ein Modul in $G[N(x)]$ oder $G[\overline{N(x)}]$ sein müssen, können die starken Module ohne x anhand der modularen Zerlegungsbäume $T_{G[N(x)]}$ und $T_{G[\overline{N(x)}]}$ ermittelt werden. Diese bilden zusammen mit dem Knoten x eine Partition $\mathcal{P}$ der Knotenmenge $V(G)$, wobei nun der Quotientengraph $G_{/\mathcal{P}}$ betrachtet wird. Basierend auf Beobachtung, dass alle nicht-trivialen Module von $G_{/\mathcal{P}}$ den Knoten x enthalten, wird der modulare Zerlegungsbaum $T_{G_{/\mathcal{P}}}$ - die „Spine“ - gebildet. Anschließend lassen sich, mittels der Spine, die starken Module mit dem Knoten x ebenfalls ermitteln. Schließlich werden die Ergebnisse zu dem modularen Zerlegungsbaum T_G kombiniert.

Obwohl der rekursive Ansatz nach Ehrenfeucht konzeptuell einfach ist, dokumentieren die auf ihm aufbauenden Algorithmen, dass eine besondere Schwierigkeit in der Berechnung der Spine, der Ermittelung der starken Module mit x und der sich anschließenden Konstruktion des modularen Zerlegungsbaums liegt. Hierbei erreichen die vorgeschlagenen Algorithmen nur mit hohem technischen Aufwand eine lineare Laufzeit oder sie opfern Linearität für Praktikabilität und erreichen so bestenfalls eine Laufzeit in $\mathcal{O}(n + m \log n)$.

Im Jahr 1997 wurde von Capelle und Habib der Nutzen einer faktorisierenden Permutation zum Berechnen der modularen Zerlegung eines Graphens untersucht [8]. Eine faktorisierende Permutation ist eine geordnete Auflistung der Knoten eines Graphen, sodass die Knoten jedes starken Moduls aufeinanderfolgend gelistet sind. Capelle und

[5] Es gilt $\alpha(n, m) \leq 4$ für praktisch relevante Werte.

Habib konnten zeigen, dass ein modularer Zerlegungsbaum in eine faktorisierende Permutation und ebenso eine faktorisierende Permutation in den korrespondierenden modularen Zerlegungsbaum eines Graphen, in jeweils linearer Zeit, überführt werden kann. Das Berechnen eines modularen Zerlegungsbaums und das Ermitteln einer faktorisierenden Permutation stellen demnach äquivalente Aufgaben dar [7]. Der sich aus diesen Überlegungen ergebende Algorithmus berechnet daher zuerst eine faktorisierende Permutation zu einem gegebenen Graphen und transformiert diese anschließend in den entsprechenden modularen Zerlegungsbaum des Graphen. Diesem alternativen Ansatz folgend, wurde im Jahr 2004 von Habib et al. [29] ein linearer Algorithmus vorgeschlagen. Leider enthielt dieser Algorithmus einen Fehler, während ein früherer Algorithmus [33] von Habib et al. bereits zum Berechnen der faktorisierenden Permutation eine Laufzeit in $\mathcal{O}(n + m \log n)$ benötigte. Die Hoffnung, die technischen Schwierigkeiten des rekursiven Ansatzes durch faktorisierende Permutationen gänzlich umgehen zu können, wurde enttäuscht, da sich neue technische Schwierigkeiten bei der Berechnung der faktorisierenden Permutation offenbarten.

Der in dieser Arbeit behandelte Algorithmus [50] kombiniert die Vorteile der beide Ansätze – dem nach Ehrenfeucht bzw. dem via faktorisierender Permutationen – wobei die Autoren[6] den Anspruch erheben, den ersten, einfachen linearen Algorithmus zur Berechnung der modularen Zerlegung eines Graphen entwickelt zu haben:

> „*By combining the advantages of the direct approach with those of the indirect approach we achieve the first simple, linear-time modular decomposition algorithm. The algorithm is conceptually straightforward and easy to implement, requiring only trees and linked-lists, and nothing more than elementary traversals of these structures. We thereby achieve a long-standing goal in the area of modular decomposition.*“

Zwei überarbeitete Versionen des Algorithmus finden sich in [51] sowie in Marc Tedder's Dissertation „Applications of Lexicographic Breadth-First Search to Modular Decomposition, Split Decomposition, and Circle Graphs“ [49]. Aufgrund der vertiefenden Darstellung in der letztgenannten Veröffentlichung, bildet diese explizit die Grundlage dieser Arbeit.

Ziel dieser Arbeit ist es, den Algorithmus zur Berechnung der modularen Zerlegung eines ungerichteten Graphen im Detail vorzustellen. Hierzu werden in Kapitel 1 zunächst grundlegende mathematische Begriffe sowie deren Notation eingeführt. Darüber hinaus

[6] Marc Tedder, Derek Corneil, Michel Habib, Christophe Paul.

werden die für den Algorithmus wesentlichen Techniken der Partitionsverfeinerung als auch der lexikographischen Breitensuche vorgestellt. Kapitel 2 erläutert den Begriff der modularen Zerlegung eines Graphen. Ausgehend von einer Einführung in die Theorie um partitive Familien, wird die Darstellung der modularen Zerlegung mittels des modularen Zerlegungsbaums konstruiert. Abschließend wird eine Generalisierung der modularen Zerlegung knapp vorgestellt. Kapitel 3 ist dem Algorithmus zur Berechnung der modularen Zerlegung eines ungerichteten Graphen gewidmet. Dieser Abschnitt bildet den umfangreichsten Teil der Arbeit und orientiert sich dabei an der Darstellung des Algorithmus in [49]. Daher besitzen die hier gemachten Aussagen und deren formale Beweise jeweils eine Entsprechung in [49]. Die Beweise sind bezüglich der Notation in dieser Arbeit angepasst, ggf. um Zwischenschritte erweitert und in eigenen Worten wiedergegeben. Auf Quellenverweise zu [49] wird in diesem Kapitel verzichtet. Im Anschluss werden in Kapitel 4 die theoretischen Argumente für die lineare Laufzeit des Algorithmus gegeben. Es werden die nötigen Datenstrukturen vorgestellt und die Berechnungsschritte anhand dieser erklärt. Abschließend wird in Kapitel 5 die, im Rahmen dieser Arbeit angefertigte, Implementierung des Algorithmus mit verschiedenartigen, zufällig erzeugten Graphen getestet und die praktisch erreichten Laufzeiten diskutiert.

1 Grundlagen

1.1 Elementare Begriffe

Der nachstehende Abschnitt widmet sich der Einführung der elementaren Begriffe, welche für die Beschreibung des Begriffs der modularen Zerlegung nötig sind. Den gängigen Bezeichnungskonventionen wird dabei weitestgehend gefolgt. Für eine detailliertere Betrachtung des hier vorgestellten Inhalts sei auf [3] und [52] verwiesen.

Eine Relation R auf einer Menge X, d.h. $R \subseteq X \times X$ heißt ***Halbordnung*** auf X, wenn für jedes $x \in X$ gilt, dass $(x, x) \in R$ (Reflexivität); für alle $x, y, z \in X$ aus $(x, y) \in R$ und $(y, z) \in R$ folgt, dass $(x, z) \in R$ (Transitivität); für alle $x, y \in X$ aus $(x, y) \in R$ und $(y, x) \in R$ folgt, dass $x = y$ (Antisymmetrie). Falls zusätzlich für alle $x, y \in X$ gilt, dass $(x, y) \in R$ oder $(y, x) \in R$ (Totalität), dann wird R eine ***lineare Ordnung*** auf X genannt. In der Infixnotation, wird $(x, y) \in R$ auch als xRy geschrieben. Das Paar $X_R = (X, R)$ heißt ***geordnete Menge***, wenn R eine Ordnung auf einer Menge $X = \{x_1, \ldots, x_k\}$ ist. Die Schreibweise $X = [x_1, \ldots, x_k]$ zeigt an, dass X eine linear geordnet ist.

Eine Menge von Teilmengen einer Menge X, d.h. $\mathcal{F} = \{A_1, \ldots, A_k\}$ mit $A_1, \ldots, A_k \subseteq 2^X$ heißt ***Familie***. Hierbei werden die Elemente $A_1, \ldots, A_k \in \mathcal{F}$ die ***Mitglieder*** der Familie genannt. Sofern für eine Familie $\mathcal{F}$ einer Menge X gilt, dass $X \in \mathcal{F}$ und $\{x\} \in \mathcal{F}$ für jedes $x \in X$ sowie $\{\varnothing\} \notin \mathcal{F}$, so heißt die Familie ***richtige und verbundene*** Familie, wobei die einelementigen Mengen $\{x\} \in \mathcal{F}$ für alle $x \in X$ sowie die Menge $X \in \mathcal{F}$ die ***trivialen*** Mitglieder der Familie genannt werden. Falls $\mathcal{F}$ und $\mathcal{F}'$ Familien über einer Menge X sind und $\mathcal{F}' \subseteq \mathcal{F}$, so wird $\mathcal{F}'$ eine ***Unterfamilie*** von $\mathcal{F}$ genannt. Zwei Mengen ***überlappen***, falls $A \cap B \neq \varnothing$, $A \setminus B \neq \varnothing$ und $B \setminus A \neq \varnothing$ gilt. Die Schreibweise hierfür ist $A \perp B$, beziehungsweise $A \not\perp B$ für den nicht überlappenden Fall. Die ***symmetrische Differenz*** zweier Mengen A, B wird mit $A \triangle B \coloneqq (A \setminus B) \cup (B \setminus A)$ bezeichnet.

Eine Menge disjunkter Teilmengen $\mathcal{P} = \{P_1, \ldots, P_k\}$ einer Menge X ist eine ***Partition*** von X, wenn $P_1 \cup \ldots \cup P_k = X$ und $P_i \neq \varnothing$ für alle $i \in \{1, \ldots, k\}$ gilt. Die Mengen $P_i \in \mathcal{P}$ werden hierbei ***Partitionsklassen*** genannt. Eine Partition $\mathcal{P}$ ist eine ***geordnete Partition***, falls $\mathcal{P}$ linear geordnet ist (geschrieben: $\mathcal{P} = [P_1, \ldots, P_k]$). Im Fall von zwei Partitionen $\mathcal{P}, \mathcal{P}'$ einer Menge X, wird $\mathcal{P}$ die ***gröbere*** Partition genannt, wenn es für jede Partitionsklasse $P' \in \mathcal{P}'$ eine Partitionsklasse $P \in \mathcal{P}$ gibt, sodass $P' \subseteq P$. Die Partition $\mathcal{P}'$ wird in diesem Zusammenhang als ***Verfeinerung*** von $\mathcal{P}$ oder als die ***feinere*** Partition bezeichnet. Wenn für eine Partition $\mathcal{P} = \{P_1, \ldots, P_k\}$ und eine Menge S gilt, dass $P_i \not\perp S$ für alle $i \in \{1, \ldots, k\}$, dann wird die Partition $\mathcal{P}$ als ***stabil*** bezüglich

der Menge S bezeichnet.

Ein ***(ungerichteter) Graph*** ist ein Paar $G = (V, E)$, wobei V eine endliche Menge von ***Knoten*** und $E \subseteq \binom{V}{2} = \{\{u, v\} \in V \mid u \neq v\}$ die Menge der ***Kanten*** ist. Ist im weiteren Verlauf von einem Graphen die Rede, so ist stets ein ungerichteter Graph gemeint. Sofern hierbei die Menge der Knoten eines Graphen G gemeint und G nicht weiter spezifiziert ist, so wird sie durch $V(G)$ angegeben. Analog dazu wird die Menge der Kanten eines Graphen G mit $E(G)$ bezeichnet. Die ***Anzahl der Knoten*** eines Graphen G wird mit $n \coloneqq |V(G)|$, die ***Anzahl der Kanten*** eines Graphen mit $m \coloneqq |E(G)|$ angegeben. Eine Kante $\{u, v\} \in E(G)$ wird auch durch uv oder den äquivalenten Ausdruck vu beschrieben. Zwei Knoten $u, v \in V(G), u \neq v$ heißen ***benachbart*** in G, falls $\{u, v\} \in E(G)$. Die ***(offene) Nachbarschaft*** eines Knotens $v \in V(G)$ in einem Graphen G bezeichnet die Menge $N(v) \coloneqq \{u \in V(G) \mid \{u, v\} \in E(G)\}$. Der Parameter $deg_G(v) \coloneqq |N(v)|$ wird hierbei der ***Grad*** des Knoten genannt. Der Begriff der ***geschlossenen Nachbarschaft*** eines Knoten $v \in V(G)$ beschreibt die Menge $N[v] \coloneqq N(v) \cup \{v\}$. Sofern für einen Knoten $v \in V(G)$ gilt, dass $N[v] = V(G)$, so wird der Knoten ***universal*** in G bezeichnet. Gilt für einen Knoten $v \in V(G)$ und eine beliebige Menge von Knoten $V' \subseteq V(G)$, dass $V' \subseteq N(v)$, so heißt v universal zu V'. Ein Knoten $v \in V(G)$ wird als ***isoliert*** von einer Menge von Knoten $V' \subseteq V(G)$ bezeichnet, falls $N(v) \cap V' = \varnothing$. Ein Knoten $v \in V(G)$, der entweder isoliert von oder universal zu einer Menge von Knoten $V' \subseteq V$ ist, wird als ***homogen*** bezüglich V' bezeichnet. Ein Knoten $v \in V(G)$, der nicht homogen bezüglich einer Menge von Knoten $V' \subseteq V(G)$ ist, wird als ***inhomogen*** bezüglich V' bezeichnet. Ein ***uv-Pfad*** der Länge k ist eine Folge paarweise verschiedener Knoten $v_0, \ldots, v_k$ mit $\{v_i, v_{i+1}\} \in E(G)$ für $i = 0, \ldots, k-1$ mit $(v_0, v_k) = (u, v)$. Eine Folge von Knoten $v_0, \ldots, v_{k-1}, v_0$ ist ein ***Kreis*** der Länge $k \geq 3$ in G, falls $v_0, \ldots, v_{k-1}$ ein v_0v_{k-1}-Pfad in G ist und $\{v_{k-1}, v_0\} \in E(G)$ gilt. Sofern ein Graph G keinen Kreis enthält, so heißt G ***kreisfrei***. Ein Graph G heißt ***zusammenhängend***, falls für alle paarweise verschiedenen Knoten $u, v \in V$ ein uv-Pfad in G existiert. Ein Graph $G' = (V', E')$ wird als ***Subgraph*** eines Graphen $G = (V, E)$ bezeichnet, wenn $V' \subseteq V$ und $E' \subseteq E$. Falls für den Subgraphen $G' = (V', E')$ gilt, dass $E' = E \cap \binom{V'}{2}$, so wird der Subgraph G' als der durch V' in G ***induzierte Graph*** bezeichnet und als $G[V']$ geschrieben. Ein Graph $\overline{G} = (V, \overline{E})$ ist der ***Komplementärgraph*** eines Graphen $G = (V, E)$, wenn $\overline{E} = \binom{V}{2} \setminus E$. Ein Subgraph C eines Graphen G ist eine ***Komponente*** von G, falls $G[V(C)]$ zusammenhängend ist und es keinen Knoten $v \in V(G) \setminus V(C)$ gibt, sodass $G[V(C) \cup \{v\}]$ zusammenhängend ist. Falls C' eine Komponente von $\overline{G}$ ist, so wird C' als ***Ko-Komponente*** von G bezeichnet. Eine Teilmenge der Knoten $U \subseteq V(G)$ eines Graphen G heißt ***Clique***, falls für paarweise verschiedene Knoten $u, v \in U$ gilt, dass

$\{u, v\} \in E(G)$. Gilt für eine Teilmenge $W \subseteq V(G)$ eines Graphen G und für paarweise verschiedene Knoten $u, v \in W$, dass $\{u, v\} \notin E(G)$, so wird W eine ***stabile Menge*** genannt.

Ein Graph G wird ***Baum*** genannt, falls G zusammenhängend und kreisfrei ist. Falls hierbei ein beliebiger Knoten $r \in V(G)$ als ***Wurzel*** ausgezeichnet ist, dann ist G ein ***gewurzelter Baum***. Ist fortan von einem Baum die Rede, so ist stets ein gewurzelter Baum gemeint, wobei dieser mit T und die Wurzel mit r bezeichnet ist. Gilt für zwei verschiedene Knoten $u, v \in V(T)$, dass es einen rv-Pfad der Form $r, \ldots, u, \ldots, v$ oder einen uv-Pfad mit $u = r$ in T gibt, so wird u als ein ***Vorgänger*** von v bezeichnet und v ein ***Nachfolger*** von u genannt. Gilt zusätzlich, dass $\{u, v\} \in E(T)$, so ist u der ***Vater*** von v sowie v ein ***Kind*** von u. Ein Knoten $v \in V(T)$ eines Baums T mit $deg(v) = 1$ und $v \neq r$ wird ein ***Blatt*** von T genannt. Ein Knoten $v \in V(T)$ eines Baums T mit $deg(v) > 1$ heißt ***innerer Knoten*** von T. Die ***Menge der Blätter*** eines Baums T wird mit $L(T)$ (oder einfach L) bezeichnet. Falls T ein Baum ist, $u \in V(T)$ und $S \coloneqq \{v \in V(T) \mid \text{v ist ein Nachfolger von u}\} \cup \{u\}$, so wird der induzierte Graph $T[S]$ mit dem als Wurzel ausgezeichneten Knoten u, als der in u gewurzelte ***Teilbaum*** von T bezeichnet und hierfür wird die Schreibweise $T[u]$ verwendet. Ein Baum T wird ein ***voller k-ärer Baum*** genannt, falls alle Knoten in T entweder k-viele oder keine Kinder haben.

Ein ***gerichteter Graph*** ist ein Paar $D = (V, E)$, wobei V eine endliche Menge von Knoten und $E \subseteq V \times V$ die Menge der gerichteten Kanten ist. Ein ***gerichteter uv-Pfad*** der Länge k ist eine Folge paarweise verschiedener Knoten $v_0, ..., v_k$ mit $(v_i, v_{i+1}) \in E(D)$ für $i = 1, ..., k-1$ und $(v_0, v_k) = (u, v)$. Die Menge aller von einem Knoten $u \in V(D)$ aus erreichbaren Knoten wird mit $R(u) = \{v \in V(D) \mid$ Es existiert ein gerichteter uv-Pfad in D$\}$ bezeichnet. Ein ***gerichteter Kreis*** der Länge $k \geq 1$ ist eine Folge von paarweise verschiedener Knoten $v_0, ..., v_{k-1}, v_0$ mit $(v_{k-1}, v_0) \in E(D)$ und $(v_i, v_{i+1}) \in E(D)$ für $i = 1, ..., k-1$. Wenn kein gerichteter Kreis in D vorliegt, so ist D ***kreisfrei bezüglich gerichteter Kreise***. Eine Kante $(v, v) \in E(D)$ heißt ***Schlinge*** von D. Falls ein gerichteter Graph D keine Schlingen enthält, so wird D ein ***einfacher gerichteter Graph*** genannt. Bezüglich eines gerichteten Graphen $D = (V, E)$ bezeichnet ein Graph $D' = (V, E')$ mit $E' \subseteq E$ eine ***transitive Reduktion*** von D, falls (1.) für alle Paare $u, v \in V$ gilt, dass es einen gerichteten uv-Pfad in D gibt gdw. es einen gerichteten uv-Pfad in D' gibt und (2.) kein Graph $D'' = (V, E'')$ mit $|E''| < |E'|$ existiert, sodass (1.) erfüllt ist.

1.2 Partitionsverfeinerung

Die algorithmische Technik der Berechnung einer Verfeinerung einer Partition $\mathcal{P} = \{P_1, \ldots, P_k\}$ einer Menge X anhand einer als ***Pivotmenge*** bezeichneten Menge $S \subseteq X$ wird ***Partitionsverfeinerung*** genannt.

In einem Schritt der Verfeinerung werden diejenigen Partitionsklassen $P_i \in \mathcal{P}$ durch die Klassen $P_{i_A} = P_i \cap S$ sowie $P_{i_B} = P_i \setminus S$ ersetzt, welche mit S überlappen d.h. für welche $P_i \perp S$ gilt. Offensichtlich gilt für die so entstehende, feinere Partition $\mathcal{P}' = \{P'_1, \ldots, P'_l\}$, dass keine ihrer Partitionsklassen $P'_i \in \mathcal{P}'$ mit der Pivotmenge S überlappt, weswegen die Partition $\mathcal{P}'$ stabil bezüglich der Pivotmenge S ist.

Es existieren zahlreiche Anwendungsfälle dieser recht einfachen, algorithmischen Technik, wobei die erstmalige Nutzung John Hopcroft zugesprochen wird, welcher diese Technik im Jahr 1971 in seinem Algorithmus zur DFA-Minimierung anwendete [35]. Weitere der zahlreichen Beispiele umfassen Varianten der lexikographischen Breitensuche [34], als auch den bekannten Algorithmus Quick-Sort. Auch im Zusammenhang mit der Berechnung der modularen Zerlegung eines Graphens kommt der Partitionsverfeinerung eine zentrale Rolle zu [49], [34], [43], [29]. Einen Überblick zur Partitionsverfeinerung gibt [32].

Algorithmus 1 : `PartitionRefinement`$(\mathcal{P}, S)$

Input : An ordered partition $\mathcal{P} = [P_1, \ldots, P_k]$ of a set X and a pivot set $S \subseteq X$.

Output : An ordered partition $\mathcal{P}' = [P'_1, \ldots, P'_{k'}]$ that is a refinement of $\mathcal{P}$ and stable with respect to S.

```
1 foreach class P ∈ 𝒫 do
2     if P ⊥ S then
3         P_A ← P ∩ S;
4         P_B ← P \ S;
5         replace P in 𝒫 with P_A, P_B in this order;
```

Bei Verwendung der geeigneten Datenstruktur zur Partitionsverfeinerung, lässt sich eine Partition $\mathcal{P}$ anhand einer Pivotmenge S in $\mathcal{O}(|S|)$ verfeinern [32]. Hierfür werden alle Elemente $x_i \in X$ in einer doppelt verketteten Liste verwaltet, wobei die Elemente einer Partitionsklasse jeweils aufeinanderfolgend gelistet sind. Jede Partitionsklassen $P_i \in \mathcal{P}$ wird durch ein Feld realisiert, welches einen Zeiger auf den Anfang und einen Zeiger auf das Ende des Bereiches der Liste besitzt, in dem die zu der Partitionsklasse gehörenden Elemente angesiedelt sind. Außerdem besitzen die Elemente $x_i \in X$ jeweils einen Zeiger zu dem Feld, welches die Partitionsklasse P_i repräsentiert, in der das Element enthalten

ist, d.h. falls $x_i \in P_i$. Ein Feld mit Zeigern auf die Elemente der Liste, welche zu S gehören, repräsentiert die Pivotmenge $S \subseteq X$.

Während eines Verfeinerungsschrittes wird jedes Element in S in konstanter Zeit an den Anfang (oder das Ende) des zu einer Partitionsklasse P_i korrespondierenden Bereichs in der Liste bewegt. Hierbei wird für eine Partitionsklasse P_i gezählt, wie viele Elemente insgesamt bewegt werden. Hierdurch können die Zeiger eines, wiederum in konstanter Zeit generierten, Feldes für die Repräsentation der Partitionsklasse $P_{i_A} = P_i \cap S$ gesetzt und die Zeiger der alten Partitionsklasse P_i entsprechend aktualisiert werden. Auf diese Weise lässt sich das Feld der Klasse P_i, nun für die Repräsentation der Klasse $P_{i_B} = P_i \setminus S$ verwenden. Da insgesamt genau $|S|$ Elemente in jeweils konstanter Zeit verschoben und höchstens $|S|$ neue Felder generiert werden, sind Kosten für das Verfeinern in $\mathcal{O}(|S|)$.

Es sei abschließend darauf hingewiesen, dass durch die vorgestellte Datenstruktur eine geordnete Partition realisiert wird. Je nach Anwendungsfall, kann dies entweder vernachlässigt oder ausgenutzt werden. Sofern eine lexikographischen Breitensuche mittels der Technik der Partitionsverfeinerung implementiert wird, ist die Eigenschaft der geordneten Partition grundlegend. Eine detailliertere Betrachtung dieser Zusammenhänge findet sich in Abschnitt 1.3.

1.3 Lexikographische Breitensuche (LBFS)

Die ***lexikographische Breitensuche (LBFS)*** ist eine Verfeinerung der traditionellen ***Breitensuche (BFS)*** und stellt ebenso eine iterative Methode zum Durchsuchen eines Graphen[7] dar. Erstmalig wurde die lexikographische Breitensuche im Kontext der Erkennung chordaler Graphen verwendet [47]. Heute existieren diverse LBFS-basierte Erkennungsalgorithmen für weitere Graphenklassen [34, 11].

Ausgehend von einem ausgewählten Startknoten $x \in V$, beginnt eine BFS in einem Graphen $G = (V, E)$ damit, dass zunächst alle zu x benachbarten Knoten entdeckt und entsprechend gekennzeichnet werden. In den Folgeschritten wird untersucht, welche noch unentdeckten Knoten, von den in der vorhergegangenen Stufe entdeckten Knoten aus, direkt erreichbar sind. Auf diese Weise werden schließlich alle von x aus erreichbaren Knoten des Graphen G entdeckt. Hierbei gilt, dass die Menge der Knoten, die während der i-ten Iteration entdeckt werden, genau der Menge von Knoten mit einem minimalen Pfad der Länge i zum Startknoten entspricht.

Technisch wird eine BFS gewöhnlich durch eine Warteschlange realisiert, in der sich zu Beginn nur der Startknoten x befindet. Während der BFS wird die Nachbarschaft

[7] In diesem Abschnitt werden zusammenhängende Graphen vorausgesetzt.

eines in der Warteschlange enthaltenen Knotens in dem Graphen G betrachtet und die benachbarten Knoten der Warteschlange in beliebiger Reihenfolge hinzugefügt, sofern diese noch nicht in der Warteschlange enthalten sind. Ist die Abarbeitung eines Knotens beendet, so wird der nächste Knoten in der Warteschlange untersucht. Falls beispielsweise durch Labels der Knoten festgehalten wird, ob ein Knoten noch unentdeckt, gerade entdeckt oder bereits entdeckt und untersucht ist, kann eine BFS mit einer Laufzeit in $\mathcal{O}(n+m)$ implementiert werden. ($\mathcal{O}(n)$ für die Betrachtung jedes Knotens; $\mathcal{O}(m)$ für die Betrachtung der Nachbarschaft.)

Eine LBFS ist im Vergleich zu einer BFS determinierter. Ist die Abarbeitungsreihenfolge von Knoten mit der gleichen minimalen Entfernung zum Startknoten im Kontext einer BFS nicht weiter spezifiziert, so wird diese im Rahmen einer LBFS, durch die Verwendung von ***lexikographischen Labels***, weiter konkretisiert. Effektiv erhalten die Knoten mit mehr bereits besuchten Nachbarn, gegenüber Knoten mit gleicher minimaler Entfernung zum Startknoten, aber mit weniger bereits besuchten Nachbarn, eine bevorzugte Behandlung, was sich in einem größeren lexikographischen Label widerspiegelt.

Algorithmus 2 : `LBFS`(G)

Input : A graph G with n vertices.

Output : A numbering σ of $V(G)$ inducing the ordering: $\sigma^{-1}(1), \ldots, \sigma^{-1}(n)$.

```
1 for x ∈ V(G) do
2     label(x) ← null;
3 for i ∈ [1, n] do
4     pick an unnumbered vertex x with lexicographically largest label;
      // assign x the number i
5     σ(x) ← i;
6     foreach unnumbered vertex y ∈ N(x) do
7         append n − i + 1 to label(y);
```

Definition 1. *Sei $G = (V, E)$ ein ungerichteter Graph und sei $n = |V|$. Eine Anordnung der Knotenmenge V, d.h. $\sigma = x_1, \ldots, x_n$ mit $x_i \in V$ heißt* ***LBFS-Anordnung*** *von G, wenn sie durch eine Ausführung von `LBFS`(G) entstanden ist.*

Eine zentrale Charakterisierung[8] von LBFS-Anordnungen ist durch das folgende Lemma gegeben:

Lemma 1.1. *[17] Eine Anordnung σ der Knotenmenge V eines Graphen $G = (V, E)$ ist genau dann eine LBFS-Anordnung, wenn für jedes Knotentripel $a, b, c \in V$ mit $a <_\sigma b <_\sigma c$ und $(a, c) \in E$ sowie $(a, b) \notin E$ gilt, dass ein Knoten $d \in V$ mit $d <_\sigma a$ existiert, sodass $(d, b) \in E$ und $(d, c) \notin E$.*

$y \in V(G)$	$\sigma(y)$	$label(y)$	$S(y)$
x	1	$\varnothing$	$V(G)$
a	2	15	$\{a, b, c, d, e\}$
b	3	15, 14	$\{b, c\}$
c	4	15, 14, 13	$\{c\}$
d	5	15, 13, 12	$\{d, e\}$
e	6	15, 13, 12	$\{e\}$
f	7	13, 12	$\{f, g, h, i, j\}$
g	8	13, 12, 9	$\{g, h, i\}$
h	9	13, 12, 9, 8	$\{h, i\}$
i	10	13, 12, 9, 8	$\{i\}$
j	11	13, 12	$\{j\}$
k	12	6	$\{k, l, m, n, o\}$
l	13	6, 4	$\{l\}$
m	14	6, 3	$\{m, n\}$
n	15	6, 3	$\{n\}$
o	16	6, 2	$\{o\}$

Tabelle 1: Eine mögliche Ausführung von `LBFS`(G) (Für den Graphen G siehe Abbildung 3a). Es wird die LBFS-Anordnung $\sigma = x, a, b, c, d, e, f, g, h, i, j, k, l, n, m, o$ von G berechnet. Die Partition der maximalen Slices von G bezüglich σ ist durch $\mathcal{P}_S = [\{x\}, \{a, b, c, d, e\}, \{f, g, h, i, j\}, \{k, l, m, n, o\}]$ bestimmt.

[8] Eine ähnliche Charakterisierung existiert für BFS-Anordnungen eines Graphen $G = (V, E)$. Hierbei wird lediglich gefordert, dass es im Fall eines Knotentripels $a < b < c$ mit $(a, c) \in E$ und $(a, b) \notin E$, einen Knoten $d < a$ geben muss, sodass $(d, b) \in E$. Da es sich bei dieser Charakterisierung um eine Generalisierung der entsprechenden Aussage über LBFS-Anordnungen handelt, ist jede LBFS-Anordnung auch eine BFS-Anordnung [12].

Definition 2. *Sei $G = (V, E)$ ein ungerichteter Graph und bezeichne σ eine LBFS-Anordnung von G. Eine Menge von Knoten $S(y) \subseteq V$ mit dem lexikographisch größten Label zum Zeitpunkt der Nummerierung des Knotens y heißt* ***Slice*** *von σ. Ein Slice $S \subsetneq V$ wird als* ***maximaler Slice*** *bezeichnet, wenn kein Slice $S' \subsetneq V$ existiert, sodass $S \subsetneq S'$. Die Menge der maximalen Slices wird mit $\mathcal{S} = \{S_1, \ldots, S_k\}$ bezeichnet.*

Veranschaulicht beschreibt ein Slice $S(y) \subseteq V$ eine Menge unnummerierter Knoten, welche aufgrund ihrer (gleichen) Labels als potenzielle Kandidaten für eine Nummerierung in Frage kommen. Jeder bereits nummerierte Knoten ist homogen bezüglich $S(y)$, was sich in den gleichen Labels der Knoten in $S(y)$ widerspiegelt. Aus einem Slice $S(y)$ wird der Knoten $y \in S(y)$ für die Nummerierung ausgewählt. Jeder Knoten in V wird im Verlauf der LBFS nummeriert, wobei jedem Nummerierungsschritt $i \in [1, n]$ ein Slice S_i zugeordnet werden kann. Entsprechend definiert eine LBFS-Anordnung σ, eine Anordnung der Slices $S_1, \ldots, S_n$ mit $n = |V|$. Insbesondere werden die maximalen Slices $\mathcal{S}$ geordnet. Außerdem lässt sich Folgendes beobachten: Für paarweise verschiedene Slices S_i, S_j mit $i < j$ gilt, dass entweder $S_i \cap S_j = \varnothing$ oder $S_j \subsetneq S_i$. Hiermit lässt sich jedem Knoten $\sigma^{-1}(i) \in V$ mit $i > 1$ – d.h. mit Ausnahme des zuerst nummerierten Knotens $x \coloneqq \sigma^{-1}(1)$ – ein maximaler Slice zuordnen. Daher bildet die Menge der maximalen Slices $\mathcal{S}$ zusammen mit dem zuerst nummerierten Knoten x eine geordnete Partition $\mathcal{P}_S = [\{x\}, S_1, \ldots, S_k]$ der Knotenmenge V. Diese Überlegungen motivieren die folgende Definition und das damit verbundene Lemma:

Definition 3. *Sei σ eine LBFS-Anordnung eines Graphens $G = (V, E)$, sei $x \in V$ der zuerst nummerierte Knoten und sei $S_1, \ldots, S_k$ eine Anordnung der maximalen Slices bezüglich σ. Die Menge $\mathcal{P}_S = [\{x\}, S_1, \ldots, S_k]$ mit $S_i \in \mathcal{S}$ heißt* ***(geordnete) Partition der maximalen Slices von*** *G* ***bezüglich*** *σ.*

Lemma 1.2. *[49] Sei $G = (V, E)$ ein Graph und sei $\mathcal{P}_S = [P_1, \ldots, P_k]$ eine geordnete Partition maximaler Slices von G bezüglich einer LBFS-Anordnung. Es gilt:*

1. *$|P_1| = 1$;*
2. *für jedes Paar P_i, P_j mit $i < j$ gilt, dass jeder Knoten $y \in P_i$ entweder universal zu P_j oder isoliert von P_j ist;*
3. *zu jedem Paar P_i, P_j mit $1 < i < j$ gibt es eine Klasse P_l mit $1 \leq l < i$ und ein $y \in P_l$, sodass gilt:*
   a) *y ist universal zu P_i und isoliert von P_j;*
   b) *für jeden Knoten $z \in P_m$ mit $m < l$ gilt, dass z entweder universal zu P_i und P_j oder isoliert von P_i und P_j ist.*

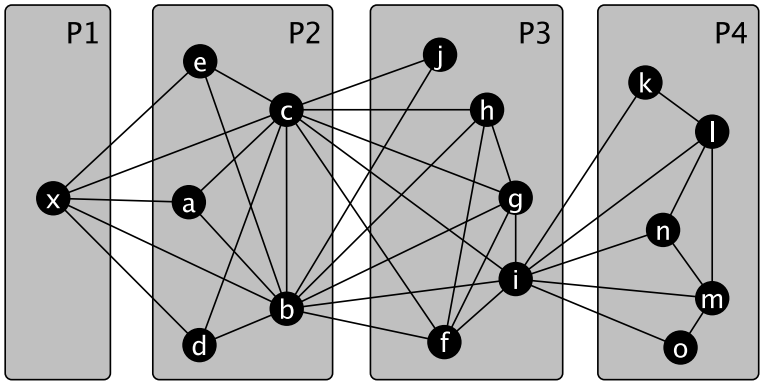


Abbildung 1: Die Partition der maximalen Slices $\mathcal{P}_S = [P_1, \ldots, P_4]$ von G bezüglich σ (Tabelle 3). Es gelten die Eigenschaften gemäß Lemma 1.2.

Abschließend sei eine besondere, rekursive Eigenschaft von LBFS-Anordnungen vorgestellt:

Lemma 1.3. *[34] Sei $G = (V, E)$ ein Graph, sei σ eine LBFS-Anordnung von G und sei $S \subseteq V$ ein Slice von σ. Es gilt:*

Die LBFS-Anordnung σ induziert eine LBFS-Anordnung σ' von $G[S]$.[9]

Beweis. Seien $a, b, c \in S$ mit $a <_\sigma b <_\sigma c$ und $(a, c) \in E$ sowie $(a, b) \notin E$ beliebig gewählt. Da σ eine LBFS-Anordnung von G ist, gibt es einen Knoten $d \in V$ mit $d <_\sigma a$ mit $(d, b) \in E$ und $(d, c) \notin E$ (Lemma 1.1). Da alle Knoten in S die gleichen lexikographischen Labels haben, ist jeder bereits nummerierte Knoten homogen bezüglich S, weswegen $d \in S$ gelten muss. Entsprechend existiert eine LBFS-Anordnung σ' von $G[S]$. □

Lemma 1.2 und Lemma 1.3 sind grundlegend für den Algorithmus zur Berechnung der modularen Zerlegung eines Graphen. Kapitel 3 erläutert die Bedeutung der Lemmata ausführlich.

Wie schon in Abschnitt 1.2 erwähnt, besteht die Möglichkeit eine LBFS durch die Technik der Partitionsverfeinerung zu implementieren. Da sich die dem Algorithmus von Tedder et al. zugrunde liegende LBFS stark an dieser Art der Umsetzung orientiert, sei diese Variante an dieser Stelle kurz vorgestellt.

Die grundsätzliche Idee hierbei ist es, die unnummerierten Knoten eines Graphen $G = (V, E)$ bis zu dem Zeitpunkt ihrer Nummerierung innerhalb einer geordneten Partition $\mathcal{P} = [P_1, \ldots, P_k]$ zu verwalten. Eine Verwaltung von Labels der Knoten ist hierbei nicht

[9] Die Aussage des Lemmas in [34] wurde verändert.

nötig, da eine diesbezügliche Entsprechung durch die Partitionsklassen gegeben ist. Im übertragenen Sinn bedeutet dies, dass Knoten innerhalb einer Partitionsklasse das gleiche Label tragen und die Labels in einer Partitionsklasse $P_i \in \mathcal{P}$ lexikographisch größer sind als die Labels in einer Partitionsklasse $P_j \in \mathcal{P}$, wenn $i < j$.

Zu Beginn der LBFS befindet sich genau eine Partitionsklasse P_1 in $\mathcal{P}$ und es gilt $V = P_1$. Die Partition $\mathcal{P}$ wird nun mehrfach verfeinert, wobei jedes Mal zunächst ein Knoten x aus der ersten Partitionsklasse P_1 entfernt und nummeriert wird. Im direkten Anschluss werden alle verbleibenden Partitionsklassen $P_i \in \mathcal{P}$ anhand der Nachbarschaft des Knotens $N(x)$ verfeinert, d.h. jeweils durch $P_i \cap N(x)$ und $P_i \setminus N(x)$ – in dieser Reihenfolge – ersetzt. Sobald die Partition nur noch einelementige Partitionsklassen enthält, ist ein Zustand erreicht, welcher analog als ein Zustand gelten kann, in dem alle Knoten $x \in V(G)$ paarweise verschiedene Labels tragen. Jetzt sind keine weiteren Partitionsverfeinerungsschritte mehr notwendig, da die Reihenfolge der Knoten innerhalb der Partition bereits eindeutig bestimmt, wie die noch unnummerierten Knoten zu nummerieren sind.

Sofern eine LBFS – unter Nutzung der entsprechenden Datenstrukturen – mittels der Technik der Partitionsverfeinerung implementiert wird, ergibt sich die Laufzeit einer LBFS zu $\mathcal{O}(n + m)$. Hierzu beachte man, dass jeder Knoten betrachtet wird, wofür $\mathcal{O}(n)$ anfallen. Ein Verfeinerungsschritt ist in $\mathcal{O}(|S|)$ realisierbar, wobei S die Pivotmenge bezeichnet. Im Fall der LBFS wird bezüglich der Nachbarschaft jedes Knotens verfeinert, wozu insgesamt $\mathcal{O}(m)$ benötigt werden.

2 Modulare Zerlegung

Die Absicht des folgenden Kapitels ist es, den Begriff der modularen Zerlegung einzuführen. Um eine bessere theoretische Einbettung zu geben und aufgrund der Tatsache, dass die Theorie der modularen Zerlegung seit den 1980er Jahren als Spezialfall der Theorie um partitive Mengenfamilien verstanden werden kann [10, 4], wird hierzu zunächst eine Einführung gegeben. Im Anschluss daran erfolgt eine detaillierte Darstellung der Begriffe bezüglich der modularen Zerlegung von Graphen. Hierbei werden die notwendigen Bezüge zur Theorie der partitiven Familien herausgestellt. Abschließend wird ein kleiner Einblick in eine nah verwandte Verallgemeinerung des Begriffs der modularen Zerlegung gegeben. Für einen alternativen Überblick zur modularen Zerlegung, sei auf [31] verwiesen.

2.1 Darstellung einer partitiven Familie: Der Zerlegungsbaum

Schwach partitive Familien können durch einen ***Zerlegungsbaum*** effizient dargestellt werden [10]. Der folgende Abschnitt erklärt den Begriff des Zerlegungsbaums und erläutert diesen anhand partitiver Familien. Die Grundlage für diesen Abschnitt bilden [6, 4, 5, 10].

Definition 4. *Sei X eine Menge und $\mathcal{F} \subseteq 2^X$ eine richtige, verbundene Familie. $\mathcal{F}$ wird eine **schwach partitive Familie** genannt, falls für paarweise verschiedene Mitglieder $A, B \in \mathcal{F}$ mit $A \perp B$ gilt, dass $A \cap B \in \mathcal{F}$, $A \cup B \in \mathcal{F}$, $A \setminus B \in \mathcal{F}$ und $B \setminus A \in \mathcal{F}$. Gilt zusätzlich, dass $A \triangle B \in \mathcal{F}$, so heißt $\mathcal{F}$ eine **partitive Familie**.*
(D.h. Eine partitive Familie ist bezüglich dem Schnitt, der Vereinigung, der symmetrischen Differenz und der Differenz abgeschlossen.)

Definition 5. *Sei $\mathcal{F}$ eine (schwach) partitive Familie. Ein Mitglied $F \in \mathcal{F}$ heißt **starkes Element** in $\mathcal{F}$, wenn für alle $F' \in \mathcal{F}$ gilt, dass $F \not\perp F'$. Die Unterfamilie der starken Elemente wird mit $\mathcal{F}_S := \{F \in \mathcal{F} \mid \forall F' \in \mathcal{F} : F \not\perp F'\}$ bezeichnet.*

Betrachtet man die Unterfamilie $\mathcal{F}_S \subseteq \mathcal{F}$ der starken Elemente einer partitiven Familie $\mathcal{F} \subseteq 2^X$ über einer Menge X, so enthält diese auch alle trivialen Mitglieder von $\mathcal{F}$ und ist entsprechend Definition 5 frei von Überlappung bezüglich ihrer Mitglieder. Diese Eigenschaft qualifiziert die Familie $\mathcal{F}_S$ als ***überlappungsfreie Familie*** und begünstigt so die Anwendbarkeit des folgenden Theorems:

Theorem 2.1. *[18] Eine Familie ist genau dann eine überlappungsfreie Familie, wenn sie durch einen gewurzelten Baum dargestellt werden kann.*

Im Folgenden bezeichne $T_S = (V, E)$ den gewurzelten Baum, welcher zur Darstellung der Familie $\mathcal{F}_S$ herangezogen werden soll. Der Baum T_S wird in diesem Zusammenhang der ***starke-Elemente-Baum*** der Familie $\mathcal{F}$ genannt und bildet das Grundgerüst für den Zerlegungsbaum $T_\mathcal{F}$ von $\mathcal{F}$. Es wird zunächst gezeigt, wie der Baum T_S konstruiert werden kann.

Die Grundlage für die Konstruktion von T_S bildet die Inklusionsrelation $\subseteq$ auf der Unterfamilie der starken Elemente $\mathcal{F}_S \subseteq \mathcal{F}$. Zunächst wird die halbgeordnete Menge $(\mathcal{F}_S, \subseteq)$ betrachtet und der hierzu isomorphe, gerichtete Graph $D = (V, E'')$ gebildet: Mittels einer Bijektion $f : \mathcal{F}_S \to V$ wird jedem Mitglied $A \in \mathcal{F}_S$ ein Knoten $v \in V$ zugewiesen, sodass für alle Paare $A, B \in \mathcal{F}_S$ gilt, dass $A \subseteq B$ gdw. $(f(A), f(B)) \in E''$. Da $A \subseteq X$ für alle $A \in \mathcal{F}_S$, ist D zusammenhängend. Außerdem folgt aus der Antisymmetrie und der Transitivität von $\subseteq$, dass D keinen gerichteten Kreis der Länge $l \geq 1$ enthält, sodass eine eindeutige transitive Reduktion $T = (V, E')$, $E' \subseteq E''$ von D gebildet werden kann [1]. Die Schlingen von D können hierbei vernachlässigt werden und werden im Folgenden weggelassen. Mit der Definition der transitiven Reduktion (siehe: Abschnitt 1.1) folgt, dass T ebenfalls zusammenhängend ist. Da für alle Paare $A, B \in \mathcal{F}_S$ gilt, dass $A \not\subset B$, ist außerdem gewährleistet, dass T auch, unter Vernachlässigung der Kantenrichtung, keinen Kreis der Länge $l \geq 3$ enthält. Somit definiert T einen Baum, wobei die Kantenrichtung durch das Festlegen einer Wurzel $r \coloneqq f(X)$ vernachlässigt wird. Dies definiert den starke-Elemente-Baum $T_S = (V, E)$ mit $E \coloneqq \{\{u, v\} \mid (u, v) \in E'\}$.

Der Baum T_S modelliert die Inklusionsordnung $\subseteq$ der Familie $\mathcal{F}_S$, sodass für alle paarweise verschiedenen Mitglieder $A, B \in \mathcal{F}_S$ gilt, dass $A \subseteq B$ gdw. $f(A)$ ein Nachfahre von $f(B)$ ist. Hiermit lässt sich die Familie $\mathcal{F}_S$ rekonstruieren. Um T_S zu einer Darstellung der gesamten partitiven Familie $\mathcal{F}$ – dem Zerlegungsbaum $T_\mathcal{F}$ – zu erweitern, ist es nötig die inneren Knoten von T_S weiter zu charakterisieren. Hierfür wird im Folgenden darauf verzichtet, den Isomorphismus f mit anzugeben.

Definition 6. *Sei $T_S = (V, E)$ der starke-Elemente-Baum einer (schwach) partitiven Familie $\mathcal{F} \subseteq 2^X$ über einer Menge X, sei $q \in V$ ein innerer Knoten von T_S und seien $f_1, \ldots, f_k$, $f_i \in V$ die Kinder von q, welche die starken Elemente $F_1, \ldots, F_k$, $F_i \in \mathcal{F}$ repräsentieren.*

*Die Familie $\mathcal{Q}(q) \subseteq 2^Y$ mit der **Grundmenge** $Y \coloneqq \{F_1, \ldots, F_k\}$ wird der **Quotient** von q genannt, wobei $\mathcal{Q}(q) \coloneqq \{Q \mid \exists I \subseteq \{1, \ldots, k\}, Q = \{F_i \in Y \mid i \in I\} \wedge \bigcup_{i \in I} F_i \in \mathcal{F}\}$.*

Theorem 2.2. *Sei $\mathcal{F} \subseteq 2^X$ eine (schwach) partitive Familie über einer Menge X und bezeichne $T_S = (V, E)$ den starke-Elemente-Baum von $\mathcal{F}$. Für einen beliebigen, inneren Knoten $q \in V$ in T_S und den Quotienten $\mathcal{Q}(q) \subseteq 2^Y$ über der dazugehörigen Grundmenge Y mit $|Y| \geq 3$ gilt genau eine, der folgenden Aussagen*[10]*:*

- $\mathcal{Q}(q) = \{Y\} \cup \{\{y\} \mid y \in Y\}$ *(Der Quotient heißt **prim**);*
- $\mathcal{Q}(q) = 2^Y \setminus \{\varnothing\}$ *(Der Quotient heißt **degeneriert (komplett)**);*
- *Es existiert eine Anordnung σ der Elemente von Y, sodass* $\mathcal{Q}(q) = \{\sigma^{-1}(i) \cup \ldots \cup \sigma^{-1}(j) \mid \forall i, j \in [1, |Y|], i \leq j\}$ *(Der Quotient heißt **linear**).*

Falls die Familie $\mathcal{F}$ partitiv ist, tritt der lineare Fall nicht auf.

Gemäß Theorem 2.2 lassen sich die inneren Knoten von T_S eindeutig charakterisieren. Entsprechend wird ein innerer Knoten q in T_S, in Abhängigkeit seines Quotienten $Q(q)$, entweder als ein ***primer*** oder ein ***degenerierter Knoten*** bezeichnet. Der Baum T_S kann nun mit diesbezüglichen *prim/degeneriert*-Labels ausgezeichnet werden, wodurch er den Zerlegungsbaum $T_\mathcal{F}$ definiert.

Anhand des Baums $T_\mathcal{F}$ lassen sich alle Mitglieder der Familie $\mathcal{F}$ aufzählen, sodass $T_\mathcal{F}$ die Familie $\mathcal{F}$ eindeutig repräsentiert. Das folgende, zentrale Theorem formalisiert diese Zusammenhänge:

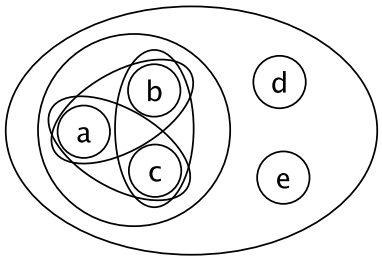
Theorem 2.3. *[10] Sei $\mathcal{F} \subseteq 2^X$ eine (schwach) partitive Familie über einer Menge X und sei $T_\mathcal{F}$ der Zerlegungsbaum von $\mathcal{F}$. Eine Teilmenge $F \subseteq X$ ist genau dann ein Mitglied der Familie $\mathcal{F}$ (d.h. $F \in \mathcal{F}$), wenn genau eine der folgenden Aussagen gilt:*

- *Es existiert ein Knoten q in $T_\mathcal{F}$, welcher F repräsentiert (d.h. $F \in \mathcal{F}_S$ ist ein starkes Element);*
- *es existiert ein innerer Knoten p in $T_\mathcal{F}$, sodass für ein nicht-triviales Mitglied $Q \in Q(p)$ gilt, dass $F = \bigcup_{q \in Q} q$.*

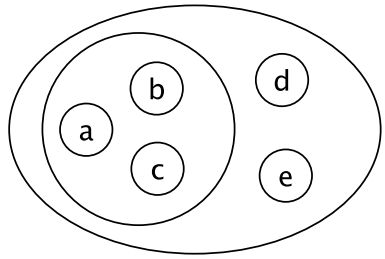
Das Bemerkenswerte an der Darstellung einer partitiven Familie via eines Zerlegungsbaums $T_\mathcal{F}$ ist deren Effizienz. Obwohl eine partitive Familie $\mathcal{F}$ über einer Menge X bis zu $2^{|X|} - 1$ Mitglieder haben kann[11], ist der Platz welcher für die Darstellung von $T_\mathcal{F}$ benötigt wird, linear zur Kardinalität der Grundmenge X, d.h. $|T_\mathcal{F}| \in \mathcal{O}(|X|)$, wobei $|T_\mathcal{F}| := |V| + |E|$. Diese Eigenschaft lässt sich leicht anhand der Beobachtung erklären, dass die Blätter des Baums $T_\mathcal{F}$ genau die einelementigen, starken Mitglieder der Familie

[10] Man beachte, dass ein Quotient in dem trivialen Fall $|Y| = 2$ gleichzeitig prim, degeneriert sowie linear ist.

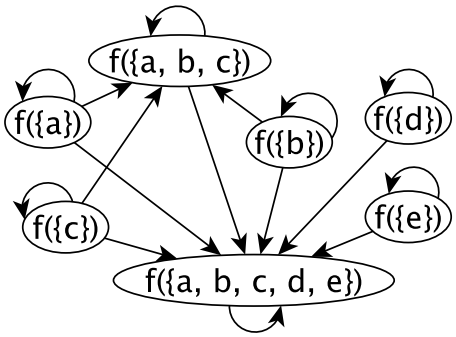
[11] Die Menge $2^X \setminus \{\varnothing\}$ über einer Menge X bildet eine partitive Familie.



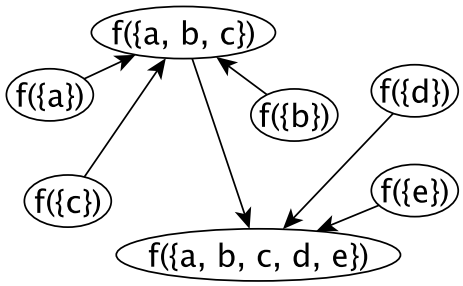
(a) Eine partitive Familie $\mathcal{F} = \{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}, \{a, b, c, d, e\}\}$.



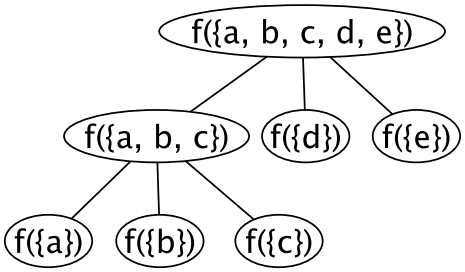
(b) Die Unterfamilie der starken Elemente $\mathcal{F}_S = \{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{a, b, c\}, \{a, b, c, d, e\}\}$.



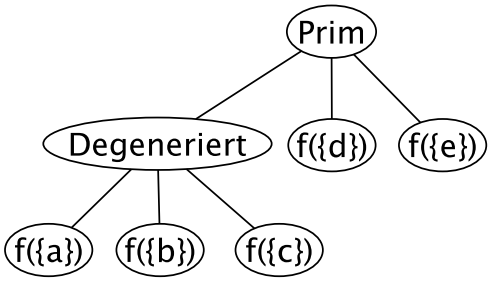
(c) Der zu $(\mathcal{F}_S, \subseteq)$ isomorphe, gerichtete Graph D.



(d) Die transitive Reduktion T von D.



(e) Durch das Festlegen der Wurzel $r := f(\{a, b, c, d, e\})$ in T kann die Kantenrichtung vernachlässigt werden. Es entsteht der starke-Elemente-Baum T_S.



(f) Mit zusätzlichen Labels definiert der starke-Elemente-Baum T_S, den Zerlegungsbaum $T_{\mathcal{F}}$. Dieser bildet eine eindeutige Darstellung der partitiven Familie $\mathcal{F}$.

Abbildung 2: Die Konstruktion des Zerlegungsbaums $T_{\mathcal{F}}$ einer partitiven Familie $\mathcal{F}$.

$\mathcal{F}$, d.h. $\{x\} \in \mathcal{F}$ für alle $x \in X$, repräsentieren und jeder innere Knoten in $T_\mathcal{F}$ mindestens zwei Kinder hat. Entsprechend ist die Gesamtanzahl $n = |V|$ der Knoten von $T_\mathcal{F} = (V, E)$ in Abhängigkeit der Anzahl der Blätter $l = |X|$ durch $n \leq 2l - 1$ bestimmt, wobei konstante Kosten für die Labels der inneren Knoten anfallen[12]. Da die Anzahl der Kanten m in einem Baum (in Abhängigkeit der Knotenanzahl n) durch $m = n - 1$ bestimmt ist, folgt $|T_\mathcal{F}| \in \mathcal{O}(|X|)$. Als eine äquivalente Randbeobachtung gilt für eine partitive Familie $\mathcal{F}$ über einer Menge X, dass die Anzahl der starken Elemente durch $|\mathcal{F}_S| \leq 2|X| - 1$ beschränkt ist.

Theorem 2.4. *[10, 5, 6] Eine (schwach) partitive Familie $\mathcal{F}$ über einer Menge X kann durch einen Zerlegungsbaum $T_\mathcal{F}$ in $\mathcal{O}(|X|)$ (Platz) dargestellt werden.*

Abschließen sei der Begriff der ***faktorisierenden Permutation*** erwähnt, da diesem in dem Algorithmus von Tedder et al. eine zentrale Rolle zukommt:

Definition 7. *Sei $\mathcal{F}_S \subseteq \mathcal{F}$ die Unterfamilie der starken Elemente einer partitiven Familie $\mathcal{F}$ über X. Eine Permutation σ der Elemente von X ist eine **faktorisierenden Permutation** bezüglich $\mathcal{F}$, falls es zu jedem Mitglied $F \in \mathcal{F}_S$ ein Paar $i, j \in [1, |X|]$ mit $i \leq j$ gibt, sodass $F = \{\sigma^{-1}(k) \mid i \leq k \leq j\}$.*

Vereinfacht ausgedrückt beschreibt eine faktorisierende Permutation σ, im Fall einer partitiven Familie $\mathcal{F}$ über einer Menge X, eine Anordnung der Elemente von X mit der Eigenschaft, dass die zu einem starken Mitglied $F_S \in \mathcal{F}$ korrespondierenden Elemente in σ aufeinanderfolgend gelistet sind. Falls der Zerlegungsbaum $T_\mathcal{F}$ einer partitiven Familie $\mathcal{F}$ vorliegt, kann eine faktorisierende Permutation beispielsweise durch das Aufzählen der Blätter von $T_\mathcal{F}$ gemäß einer, in der Wurzel begonnenen, Tiefensuche in $T_\mathcal{F}$ erzeugt werden. In dem Algorithmus von Tedder et al. wird der umgekehrte Ansatz verfolgt, einen (modularen) Zerlegungsbaum aus einer speziellen faktorisierenden Permutation zu gewinnen.

[12] Im Fall einer schwach partitiven Familie, werden lineare Knoten um eine Anordnung der Kinder ergänzt.

2.2 Module eines Graphen

Module eines Graphen sind Teilmengen der Knotenmenge, bei denen alle Knoten innerhalb einer solchen Menge die gleiche Nachbarschaftsbeziehung allen Knoten außerhalb dieser Menge haben:

Definition 8. *Sei $G = (V, E)$ ein Graph. Eine Menge von Knoten $M \subseteq V$ mit $M \neq \varnothing$ ist ein **Modul** von G, falls für paarweise verschiedene Knoten $x, y \in M$ gilt, dass $N(x) \setminus M = N(y) \setminus M$. Die Familie der Module eines Graphen G wird mit $\mathcal{M}$ bezeichnet.*[13]

Offensichtliche Beispiele für Module in einem Graphen $G = (V, E)$ sind die Knotenmenge V oder die einelementigen Mengen $\{v\}$ für alle $v \in V$. Analog zu der Terminologie um richtige, verbundene Familien werden diese Module die ***trivialen Module*** genannt. Enthält ein Graph ausschließlich triviale Module, so wird dieser als ***primer Graph***[14] bezeichnet. Weitere, einfache Beispiele sind die Komponenten eines Graphen. Zudem kann leicht nachvollzogen werden, dass auch die Vereinigung mehrerer Komponenten die Moduleigenschaft stets erfüllt und eine äquivalente Aussage über Ko-Komponenten gilt.

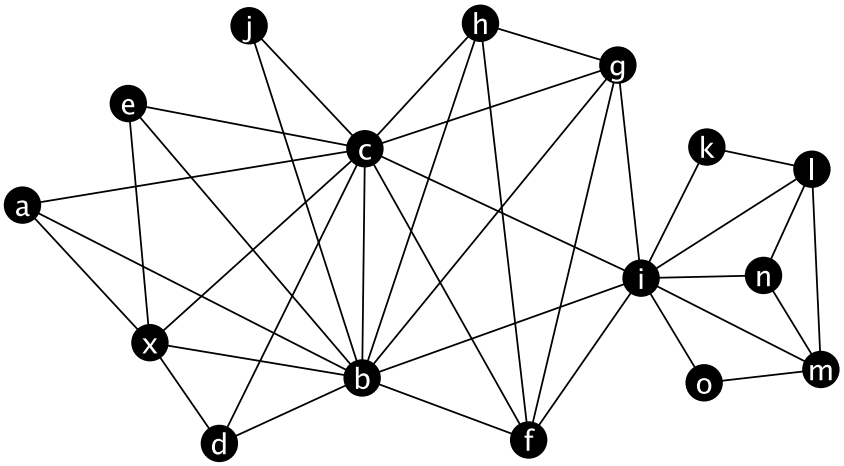
Potenziell kann für bis zu $|2^V \setminus \{\varnothing\}| = 2^{|V|} - 1$ Teilmengen der Knotenmenge V gelten, dass sie die Moduleigenschaft erfüllen. Zur Veranschaulichung stelle man sich einen beliebigen Graphen ohne Kanten vor, also $G = (V, E)$ mit $E = \varnothing$: Offensichtlich gilt für jeden Knoten $v \in V$, dass $N(v) = \varnothing$. Entsprechend gilt für jede Teilemenge $M \subseteq V$ und paarweise verschiedene Knoten $x, y \in M$, dass $N(x) \setminus M = N(y) \setminus M$. Da es $2^{|V|} - 1$ Möglichkeiten gibt, eine nicht-leere Teilmenge der Knoten von V zu definieren, besitzt der Graph G also genau $2^{|V|} - 1$ viele Module[15].

Neben einem Eindruck über die mögliche Anzahl von Modulen in einem Graphen, vermittelt dieses Beispiel ebenfalls, dass Module in verschiedenen Beziehungen zueinander auftreten können. So können Module wiederum andere Module enthalten, Module einander überlappen oder Module von anderen Modulen disjunkt sein. Als direkte Folge der Moduleigenschaft lässt sich der disjunkte Fall $M \cap M' = \varnothing$ weiter, in genau zwei Fälle, unterteilen: Entweder alle Knoten eines Moduls M sind zu allen Knoten eines anderen Moduls M' benachbart, oder das Gegenteil ist der Fall. D.h. entweder $uv \in E$ für alle Paare $u \in M, v \in M'$ oder $uv \notin E$ für alle Paare $u \in M, v \in M'$. Deswegen werden disjunkte

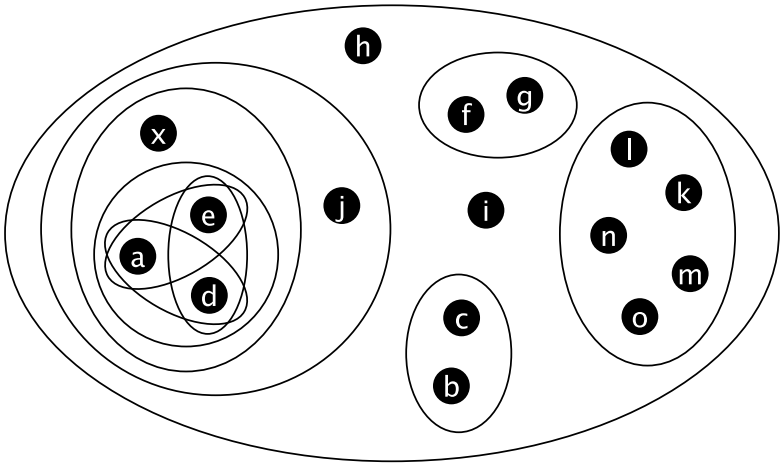
[13] Um Inkonsistenzen mit der Definition von richtigen, verbundenen Familien zu vermeiden und da die Darstellung der Familie der Module eines Graphen darunter leiden würde, sei die leere Menge als Modul ausgeschlossen.

[14] Der selbstkomplementäre Graph P_4 bildet den kleinsten primen Graphen.

[15] Die maximal mögliche Anzahl von Modulen in einem Graphen $G = (V, E)$ entspricht der maximalen möglichen Anzahl von Mitgliedern einer partitiven Familie $\mathcal{F}$ über einer Menge X, wenn $|V| = |X|$. Tatsächlich bildet die Familie der Module $\mathcal{M}$ eine partitive Familie, wie noch gezeigt wird.



(a) Ein Graph G.



(b) Die Familie der Module $\mathcal{M}$ von G.

Abbildung 3

Module entweder als zueinander ***benachbart*** oder als zueinander ***nicht benachbart*** bezeichnet. Einige der in diesem Kontext verwendeten Begriffe seinen in der folgenden Definition festgehalten:

Definition 9. *Sei $G = (V, E)$ ein Graph und bezeichne $\mathcal{M}$ die Familie der Module von G. Ein Modul $M \in \mathcal{M}$ wird ein **triviales Modul** genannt, falls $M = V$ oder $M = \{v\}$ für ein $v \in V$. Ein Modul $M \in \mathcal{M}$ heißt **starkes Modul**, falls für alle Module $M' \in \mathcal{M}$ gilt, dass $M \not\perp M'$. Die Familie der starken Module wird mit $\mathcal{M}_S \coloneqq \{M \in \mathcal{M} \mid \forall M' \in \mathcal{M} : M \not\perp M'\}$ angegeben. Ein Modul $M \in \mathcal{M}$ wird als **maximal** bezüglich einer Menge $S \subseteq V$ bezeichnet, falls $M \subsetneq S$ und kein Modul $M' \in \mathcal{M}$ existiert für das gilt, dass $M \subsetneq M' \subsetneq S$.*

2.3 Quotientengraphen

Richtet man den Fokus auf die Module eines Graphen $G = (V, E)$ welche stark und zugleich maximal bezüglich V sind, so ergibt sich, dass diese eine Partition $\mathcal{P} = \{M_1, \ldots, M_k\}$ der Knotenmenge V bilden. Wegen der Maximalität der Module gilt außerdem, dass $\mathcal{P}$ eindeutig bestimmt ist. Zusammen mit der Eigenschaft disjunkter Module, entweder zueinander benachbart oder nicht benachbart zu sein, eröffnet dies eine Möglichkeit der kompakteren Darstellung von G.

Hierzu definiert man einen Graphen $G_{/\mathcal{P}} \coloneqq (V_{\mathcal{P}}, E_{\mathcal{P}})$, dessen Knotenmenge $V_{\mathcal{P}}$ für jedes Modul $M_i \in \mathcal{P}$ einen eineindeutigen, stellvertretenden Knoten $q_{M_i} \in V_{\mathcal{P}}$ enthält, d.h. $V_{\mathcal{P}} \coloneqq \{q_{M_1}, \ldots, q_{M_k}\}$. Zusätzlich wird mittels der Kantenmenge $E_{\mathcal{P}}$ die Nachbarschaftsbeziehung der Module $M_i \in \mathcal{P}$ auf den Stellvertretern $q_{M_i} \in V_{\mathcal{P}}$ modelliert, also $E_{\mathcal{P}} \coloneqq \{\{q_{M_i}, q_{M_j}\} \mid \exists u \in M_i, v \in M_j : \{u, v\} \in E\}$. Kennt man die bijektive Entsprechung der Knoten $q_{M_i} \in V_{\mathcal{P}}$ zu den Modulen $M_i \in \mathcal{P}$, so ist es ausreichend lediglich den Graphen $G_{/\mathcal{P}}$ und die durch die Module induzierten Graphen $G[M_i]$ zu verwalten, da hieraus G jederzeit rekonstruiert werden kann. Ein Vorteil dieser kompakteren Darstellung ist, dass hierbei potenziell weniger Kanten verwaltet werden müssen. Die nachstehenden Definitionen widmen sich den diesbezüglichen Begriffen:

Definition 10. *Sei $G = (V, E)$ ein Graph und bezeichne $\mathcal{P} = \{M_1, \ldots, M_k\}$ eine Partition der Knotenmenge V. Falls für alle $i \in [1, k]$ gilt, dass M_i ein Modul in G ist, so wird $\mathcal{P}$ eine **modulare Partition** von G genannt. Gilt zusätzlich, dass für alle $i \in [1, k]$ gilt, dass M_i ein starkes und bezüglich V maximales Modul ist, dann heißt $\mathcal{P}$ eine **maximale modulare Partition**.*

Definition 11. *Sei $G = (V, E)$ ein Graph. Es bezeichne $\mathcal{P} = \{M_1, \ldots, M_k\}$ eine maximale modulare Partition von G und es sei $V_{\mathcal{P}} = \{q_{M_1}, \ldots, q_{M_k}\}$ eine Menge von Knoten, sodass für alle $i \in [1, k]$ ein Knoten $q_{M_i} \in V_{\mathcal{P}}$, das Modul $M_i \in \mathcal{P}$ eineindeutig repräsentiert. Ein Knoten $q_{M_i} \in V_{\mathcal{P}}$ heißt **Repräsentant** des Moduls $M_i \in \mathcal{P}$. Der Graph $G_{/\mathcal{P}} = (V_{\mathcal{P}}, E_{\mathcal{P}})$ mit $E_{\mathcal{P}} = \{\{q_{M_i}, q_{M_j}\} \mid \exists u \in M_i, v \in M_j : \{u, v\} \in E\}$ wird der **Quotientengraph** von G unter $\mathcal{P}$ genannt. Die durch die Module $M_i \in \mathcal{P}$ in G induzierten Teilgraphen $G[M_i]$ werden als die **Faktoren** von $G_{/\mathcal{P}}$ bezeichnet.*

Die Idee einen Graphen $G = (V, E)$ durch seinen Quotientengraphen $G_{/\mathcal{P}} = (V_{\mathcal{P}}, E_{\mathcal{P}})$ unter der maximale modulare Partition $\mathcal{P} = \{M_1, \ldots, M_k\}$ zusammen mit den zu den Knoten $q_{M_i} \in V_P$ korrespondierenden Faktoren $G[M_i]$ zu verwalten, lässt sich weiter auf die Faktoren übertragen: Verwaltet man anstelle der Faktoren $G[M_i]$ jeweils die Quotientengraphen $G[M_i]_{/\mathcal{P}_{M_i}}$, wobei $\mathcal{P}_{M_i} = \{M_{i_1}, \ldots, M_{i_l}\}$ eine maximale modulare

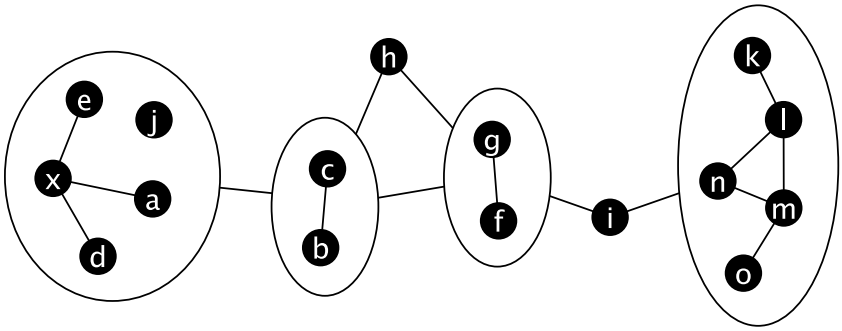


Abbildung 4: Der Quotientengraph $G_{/\mathcal{P}}$ von G (Abbildung 3a) unter der maximalen modularen Partition $\mathcal{P} = \{\{x, a, d, e, j\}, \{c, b\}, \{h\}, \{f, g\}, \{i\}, \{k, l, m, n, o\}\}$. Die Faktoren von $G_{/\mathcal{P}}$ sind in den Repräsentanten der Module dargestellt.

Partition von $G[M_i]$ ist und führt man dieses Vorgehen rekursiv fort, bis sich die Faktoren zu den trivialen Modulen $\{v\} \in V$ ergeben, so gelangt man auf natürliche Weise zu einer nochmals kompakteren, eineindeutigen Darstellung von G. Eine kompakte Darstellung von G zu erzeugen, beschreibt im Kern das Wesen der modularen Zerlegung, wobei die hier nur umrissene Form der Darstellung schließlich zur Formalisierung des ***modularen Zerlegungsbaums*** beiträgt.

2.4 Der modulare Zerlegungsbaum

Es ist klar, dass die Familie der Module $\mathcal{M}$ eines Graphen $G = (V, E)$ die trivialen Module von G enthält. Da die leere Menge als Modul von G ausgeschlossen ist, bildet $\mathcal{M}$ eine richtige, verbundene Familie. Tatsächlich gilt die folgende, stärkere Aussage:

Theorem 2.5. *[10, 5] Die Familie der Module $\mathcal{M}$ eines Graphen G ist partitiv.*

Beweis. Um zu zeigen, dass die Familie der Module $\mathcal{M}$ eines Graphen $G = (V, E)$ partitiv ist, muss für überlappende Module $M \perp M'$, $M, M' \in \mathcal{M}$ gezeigt werden, dass der Schnitt $M \cap M'$, die Vereinigung $M \cup M'$, die Differenz $M \setminus M'$ (und $M' \setminus M$) sowie die symmetrische Differenz $M \triangle M'$ ebenfalls Module von G sind.

Angenommen der Schnitt $X := M \cap M'$ ist kein Modul. Es seien $x, y \in X$ und $s \notin X$ so gewählt, dass $xs \in E$ und $ys \notin E$. Aus $M \in \mathcal{M}$ und $x, y \in M$ ergibt sich, dass (1) $s \in M \setminus M'$. Wegen (1) und da $M' \in \mathcal{M}$ sowie $x, y \in M'$ folgt, dass $xs \in E$ genau dann, wenn $ys \in E$. Dies steht im Widerspruch zu $xs \in E$ und $ys \notin E$.

Angenommen die Vereinigung $Y := M \cup M'$ ist kein Modul. Es seien $x, y \in Y$ sowie $s \notin Y$. Außerdem gelte $xs \in E$ und $ys \notin E$. Da $M, M' \in \mathcal{M}$ folgt, dass $x, y \notin M \cap M'$

beziehungsweise o.B.d.A. $x \in M \setminus M'$ und $y \in M' \setminus M$. Weil $M \perp M'$ existiert ein Knoten $z \in M \cap M'$. Wegen $M \in \mathcal{M}$ mit $x, z \in M$ gilt (1) $zs \in E$ genau dann, wenn $xs \in E$. Aus $M' \in \mathcal{M}$ mit $y, z \in M'$ ergibt sich, dass (2) $zs \in E$ genau dann, wenn $ys \in E$. Mit (1), (2) folgt, dass $xs \in E$ genau dann, wenn $ys \in E$. Dies steht im Widerspruch zu $xs \in E$ und $ys \notin E$.

Angenommen die Differenz $W := M \setminus M'$ ist kein Modul. Es seien $x, y \in Y$ und es sei $s \notin Y$ so gewählt, dass $xs \in E$ und $ys \notin E$. Da $M \in \mathcal{M}$ und $x, y \in M$ folgt, dass (1) $s \in M \cap M'$. Außerdem existiert ein (2) $t \in M' \setminus M$, da $M \perp M'$. Aus (1),(2) und $M' \in \mathcal{M}$ folgt, dass (3) $xs \in E$ genau dann, wenn $xt \in E$ und entsprechend, dass (4) $ys \in E$ genau dann, wenn $yt \in E$. Weil $M \in \mathcal{M}$ ist (5) $xt \in E$ genau dann, wenn $sy \in E$. Wegen (3), (4), (5) gilt, dass $xs \in E$ genau dann, wenn $ys \in E$. Dies steht im Widerspruch zu $xs \in E$ und $ys \notin E$. ($M' \setminus M \in \mathcal{M}$ analog.)

Angenommen $Z := M \triangle M'$ ist kein Modul. Es sei $x, y \in Z$ und $s \notin Z$ so, dass $xs \in E$ und $ys \notin E$. Da $M, M' \in \mathcal{M}$ folgt, dass o.B.d.A. $x \in M \setminus M'$ und $y \in M' \setminus M$, wobei außerdem $s \in M \cap M'$. Aus $M \in \mathcal{M}$ mit $x, s \in M$ folgt, dass (1) $xy \in E$ genau dann, wenn $xs \in E$. Analog ergibt $M' \in \mathcal{M}$ mit $y, s \in M'$, dass (2) $xy \in E$ genau dann, wenn $xs \in E$. Aus (1), (2) folgt, dass $ys \in E$ genau dann, wenn $xs \in E$. Dies widerspricht der Annahme, dass $xs \in E$ und $ys \notin E$. □

Neben Aussagen über die Moduleigenschaft von Schnitt, Vereinigung, Differenz und symmetrische Differenz von überlappenden Modulen, die sich aus der Partitivität von $\mathcal{M}$ ergeben, lassen sich die zu partitiven Familien getroffenen Aussagen nun auf den Kontext von Modulen übertragen. Die Eigenschaft der Partitivität der Familie der Module $\mathcal{M}$ eines Graphen $G = (V, E)$, erlaubt es insbesondere $\mathcal{M}$ durch einen Zerlegungsbaum $T_{\mathcal{M}}$ in $\mathcal{O}(|V|)$ darzustellen (Theorem 2.4). Dieser Zerlegungsbaum $T_{\mathcal{M}}$ bildet die Grundlage für den ***modularen Zerlegungsbaum*** T_G.

Wie schon bei der Darstellung einer partitiven Familie, so ist auch im Rahmen der Darstellung der Familie der Module $\mathcal{M}$ via $T_{\mathcal{M}}$, eine bijektive Entsprechung der starken Module von G zu den Knoten von $T_{\mathcal{M}}$ gegeben, sodass $T_{\mathcal{M}}$ die Inklusionsordnung $\subseteq$ von $\mathcal{M}_S$ modelliert. In diesem Sinne repräsentiert die Wurzel von $T_{\mathcal{M}}$ das zu der Knotenmenge V korrespondierende Modul $V \in \mathcal{M}_S$ und die Blätter von $T_{\mathcal{M}}$ genau die einelementigen, trivialen Module $\{v\} \in \mathcal{M}_S$ für alle $v \in V$. Grundsätzlich repräsentiert jeder innerer Knoten p_M in $T_{\mathcal{M}}$ jeweils ein eineindeutiges starkes Modul $M \in \mathcal{M}_S$ von G. Hierbei sind die Kinder von p_M genau die Knoten $\{q_{M_1}, \ldots, q_{M_k}\}$, welche die starken Module der maximalen modularen Partition $\mathcal{P} = \{M_1, \ldots, M_k\}$ von $G[M]$ repräsentieren, wobei für alle $i \in [1, k]$ ein Knoten q_{M_i} dem Modul M_i entspricht. Im Wesentlichen wird auch hierbei

durch die *prim/degeneriert*-Labels der inneren Knoten von $T_{\mathcal{M}}$ die nötige Information gegeben, welche es ermöglicht die Familie der Module $\mathcal{M}$ aufzuzählen (Theorem 2.3). Entsprechend bezeichnet der Begriff des ***primen Knotens*** einen inneren Knoten p_M mit dem primen Quotienten $Q(p_M) = \{\mathcal{P}\} \cup \{\{M_i\} \mid M_i \in \mathcal{P}\}$ über der maximalen modularen Partition $\mathcal{P} = \{M_1, \ldots, M_k\}$ von $G[M]$; der Begriff des ***degenerierten Knotens*** einen inneren Knoten p_M mit dem degenerierten Quotienten $Q(p_M) = 2^{\mathcal{P}} \setminus \{\varnothing\}$ über der maximalen modularen Partition $\mathcal{P} = \{M_1, \ldots, M_k\}$ von $G[M]$ (vgl. Theorem 2.2).

Die Verwendung der Labels *prim* und *degeneriert* ist zwar ausreichend um die Familie $\mathcal{M}$ aufzuzählen, jedoch ist deren Aussagekraft bezüglich der exakten Struktur des Graphen G nicht hinreichend stark. Aus diesem Grund, wird das Konzept des Quotientengraphen bezüglich der inneren Knoten von $T_{\mathcal{M}}$ herangezogen. Auf diese Weise gelingt es, $T_{\mathcal{M}}$ zu einer eineindeutigen Repräsentation des Graphen G sowie der dazugehörigen Familie der Module $\mathcal{M}$ zu erweitern. Diese definiert den modularen Zerlegungsbaum T_G von G.

Analog zu der Zuordnung des Quotienten $Q(p_M)$, kann einem beliebigen inneren Knoten p_M in $T_{\mathcal{M}}$ der Quotientengraph $G[M]_{/\mathcal{P}}$ unter der maximalen modularen Partition $\mathcal{P}$ von $G[M]$ zugeordnet werden. Der Zusammenhang zwischen dem Begriff des Quotienten bzw. dem des Quotientengraphen wird deutlich, ersetzt man ein (starkes) Modul $M' \in Q$ eines Mitglieds $Q \in Q(p_M)$, durch den Repräsentanten des Moduls $q_{M'} \in V(G[M]_{/\mathcal{P}})$, für alle $M' \in Q$ und alle $Q \in Q(p_M)$: Der so modifizierte Quotient bildet die Familie der Module des Quotientengraphen. Vergleichbar mit der Eigenschaft eines Quotienten entweder prim oder degeneriert zu sein, gilt für einen Quotientengraphen genau eine der Eigenschaften gemäß des folgenden, zentralen Theorems von Tibor Gallai.

Theorem 2.6. *[25] Für einen Graphen G gilt genau eine, der folgenden drei Aussagen:*

- *G ist nicht zusammenhängend;*
- *$\overline{G}$ ist nicht zusammenhängend;*
- *G und $\overline{G}$ sind zusammenhängend und der Quotientengraph $G_{/\mathcal{P}}$ unter der maximalen modularen Partition $\mathcal{P}$ von G ist ein primer Graph.*

Betrachtete man den Fall eines nicht zusammenhängenden Quotientengraphen $G[M]_{/\mathcal{P}} = (V_{/\mathcal{P}}, E_{/\mathcal{P}})$ zu einem inneren Knoten $p_M \in V(T_{\mathcal{M}})$, so repräsentiert ein Knoten $q_{C_i} \in V_{/\mathcal{P}}$ die Komponente C_i des Graphen $G[M]$ bzw. $\mathcal{P} = \{C_1, \ldots, C_k\}$, falls es k Komponenten in $G[M]$ gibt. Somit ist klar, dass die Knotenmenge $V_{/\mathcal{P}}$ eine stabile Menge in $G[M]_{/\mathcal{P}}$ bildet und jede Teilmenge $M' \subseteq V_{/\mathcal{P}}$ ein Modul in $G[M]_{/\mathcal{P}}$ ist. Entsprechend des Zusammenhangs zwischen Quotient und Quotientengraph, ist der Quotient $Q(p_M)$ degeneriert. Der Fall eines zusammenhängenden Quotientengraphen $G[M]_{/\mathcal{P}} = (V_{/\mathcal{P}}, E_{/\mathcal{P}})$ ist ähnlich.

Hier bilden die Ko-Komponenten $\overline{C}_1, \ldots, \overline{C}_k$ von $G[M]$ die maximale modulare Partition $\mathcal{P} = \{\overline{C}_1, \ldots, \overline{C}_k\}$, sodass deren Repräsentanten $V_{/\mathcal{P}} = \{q_{\overline{C}_1}, \ldots, q_{\overline{C}_k}\}$ in $G[M]_{/\mathcal{P}}$ eine Clique bilden. Folglich gilt auch hier, dass der Quotient $Q(p_M)$ degeneriert ist. Hiermit ergibt sich eine Verfeinerung des Begriffs des degenerierten Knotens: Im Fall der Clique, wird p_M ein ***serieller Knoten*** und im Fall der stabilen Menge, ein ***paralleler Knoten*** genannt. Ein *seriell/parallel*-Label eines Knotens p_M spezifiziert also stets, ob es sich bei dem korrespondierenden Quotientengraph $G[M]_{/\mathcal{P}}$ um einen vollständigen Graphen oder um einen Graphen ohne Kanten handelt.

Offensichtlich ist der Quotientengraph $G[M]_{/\mathcal{P}}$ zu einem Knoten p_M genau dann ein primer Graph, wenn der Quotient $Q(p_M)$ prim ist. Der Begriff des primen Knotens allein verfügt jedoch über keine hinreichende Aussagekraft bezüglich der Struktur von G. Deswegen ist es nötig, einen primen Knoten p_M in T_S direkt, durch das Angeben des dazugehörigen Quotientengraphen $G[M]_{/\mathcal{P}}$ zu spezifizieren, wobei M das dem Knoten p_M entsprechende Modul und $\mathcal{P}$ die maximale modulare Partition des Graphen $G[M]$ ist. Die Knoten des Quotientengraphen $G[M]_{/\mathcal{P}}$ entsprechen offenbar den Kindern von p_M. Wird diese Entsprechung im Fall von primen Knoten gekennzeichnet, so ist die sich ergebende Darstellung – der modulare Zerlegungsbaum T_G von G – eine Darstellung die es ermöglicht, alle Module aufzuzählen (Theorem 2.3) und die Struktur von G vollständig zu repräsentieren.

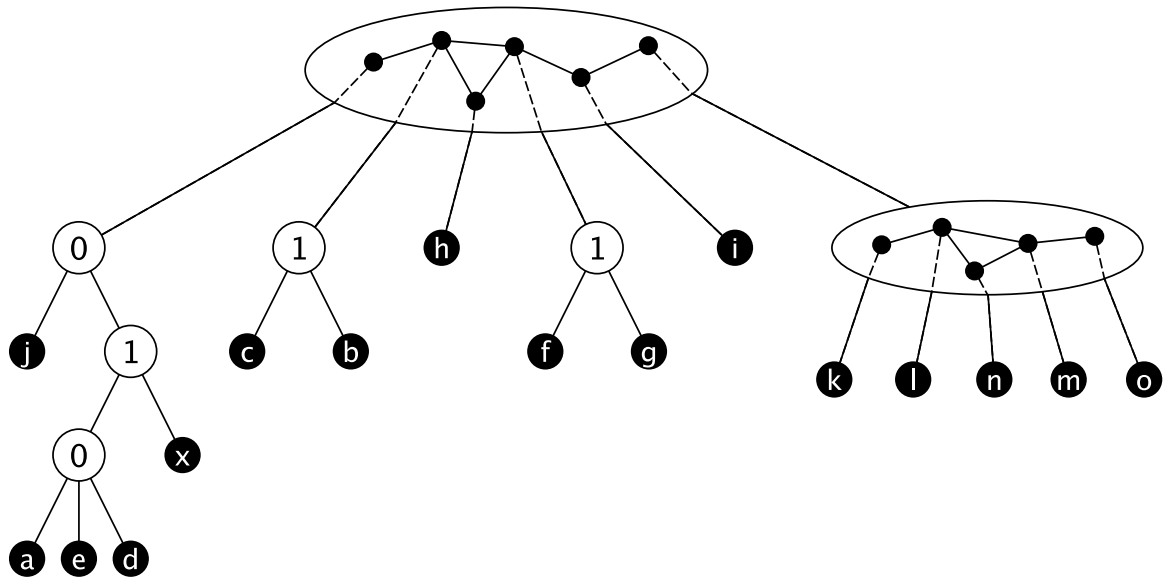


Abbildung 5: Der modulare Zerlegungsbaum T_G von G (Abbildung 3a).

2.5 Generalisierung der modularen Zerlegung von Graphen: Clan Zerlegung von 2-Strukturen

Zum Abschluss des Kapitels zum Begriff der modularen Zerlegung sei an dieser Stelle eine Verallgemeinerung des Begriffs der modularen Zerlegung von Graphen kurz vorgestellt. Die Grundlage für diesen Abschnitt bilden [20, 19, 5, 39].

Gerichtete Graphen stellen die wohl bekannteste Generalisierung ungerichteter Graphen dar. Hierbei ist der grundlegende Unterschied, dass die Kantenrelation $E(D)$ eines gerichteten Graphen D nicht notwendigerweise symmetrisch ist. Gewissermaßen lässt sich ein ungerichteter Graph als gerichteter Graph mit einer symmetrischen Kantenrelation auffassen. Eine mächtigere Generalisierung von sowohl ungerichteten als auch gerichteten Graphen sind 2-Strukturen. Naiv kann eine 2-Struktur als ein vollständiger, gerichteter Graph beschrieben werden, dessen Kanten mit einer endlichen Anzahl Farben gefärbt sind.

Definition 12. *Eine **2-Struktur** ist ein Tripel $G = (V, C, k)$, wobei V eine endliche Menge von Knoten, $k \in \mathbb{N}$ und $C : V \times V \to \{1, \ldots, k\}$ eine Färbfunktion bezeichnet. Eine 2-Struktur heißt **symmetrisch**, falls $C(x, y) = C(y, x)$ für alle $x, y \in V$.*

Ein gerichteter Graph lässt sich beispielsweise durch eine 2-Struktur $G = (V, C, k)$ mit $k = 2$ modellieren, indem man $C(x, y) = C(y, x) = 1$ als keine Kante, $C(x, y) = 2, C(y, x) = 1$ als eine Kante von einem Knoten $x \in V$ zu einem Knoten $y \in V$ interpretiert. Darüber hinaus sind Anwendungsfälle von 2-Strukturen als Modellierungsgrundlage für beispielsweise kantengewichtete Graphen eingeschränkt denkbar.

Der Begriff des Moduls findet seine Entsprechung im Rahmen von 2-Strukturen in dem Begriff des Clans:

Definition 13. *Sei $G = (V, C, k)$ eine 2-Struktur. Eine Menge von Knoten $M \subseteq V$ mit $M \neq \varnothing$ ist ein **Clan** von G, falls für paarweise verschiedene Knoten $x, y \in M$ und $s \notin M$ gilt, dass $C(s, x) = C(s, y)$ und $C(x, s) = C(y, s)$. Die Familie der Clans einer 2-Struktur G wird mit $\mathcal{C}$ bezeichnet. Die Clans $\{x\} \in \mathcal{C}$, für alle $x \in V$ sowie der Clan $V \in \mathcal{C}$ werden **triviale Clans** genannt.*

Die Clans einer symmetrischen 2-Struktur werden auch Module genannt. Im Fall von gerichteten Graphen, wird ebenfalls bevorzugt der Begriff des Moduls verwendet[16]. Mit dem folgenden Theorem und der in Abschnitt 2.1 geleisteten Vorarbeit, ist der Begriff der Clan Zerlegung schnell umrissen.

[16] Auch im nicht symmetrischen Fall.

Theorem 2.7. *[20]*

- *Die Familie der Clans $\mathcal{C}$ einer 2-Struktur G ist schwach partitiv.*
- *Die Familie der Clans $\mathcal{C}$ einer symmetrischen 2-Struktur G ist partitiv.*

Gemäß Theorem 2.7, Theorem 2.4, Theorem 2.3 sowie Theorem 2.2 kann die Familie der Clans $\mathcal{C}$ einer 2-Struktur $G = (V, C, k)$ durch einen Zerlegungsbaum $T_{\mathcal{C}}$ in $\mathcal{O}(|V|)$ dargestellt werden. Für den Quotienten eines inneren Knotens q in $T_{\mathcal{C}}$ gilt hierbei, dass dieser entweder prim, komplett oder linear ist. Analog zu dem modularen Zerlegungsbaum eines Graphen, können die inneren Knoten von $T_{\mathcal{C}}$ durch eine dem Quotienten zugeordnete 2-Struktur weiter spezifiziert werden. Auf diese Weise ist es möglich, die Familie der Clans $\mathcal{C}$ sowie die zugrundeliegende 2-Struktur durch den erweiterten Zerlegungsbaum darzustellen. Prime Knoten werden hierbei durch eine prime 2-Struktur charakterisiert. Komplette Knoten werden durch eine Farbe $c \in \{1, \ldots, k\}$ ausgezeichnet. Hierdurch wird angezeigt, dass es sich bei der mit dem Quotienten assoziierten 2-Struktur um eine c-Clique handelt. Lineare Knoten werden um ein Paar $c, c' \in \{1, \ldots, k\}$ sowie eine Anordnung der Knoten ergänzt. Diesbezüglich wird die 2-Struktur des Quotienten als eine (c, c')-Ordnung spezifiziert.

Definition 14. *Sei $G = (V, C, k)$ eine 2-Struktur. G heißt*

- ***prim**, falls G ausschließlich triviale Clans enthält;*
- *eine **c-Clique**, falls für paarweise verschiedene Knoten $x, y \in V$ und ein $c \in \{1, \ldots, k\}$ gilt, dass $C(x, y) = c$;*
- *eine **(c,c')-Ordnung**, falls für paarweise verschiedene Knoten $x, y \in V$ und ein Paar $c, c' \in \{1, \ldots, k\}$ gilt, dass $C(x, y) \in \{c, c'\}$ und die Relation xRy gdw. $C(x, y) = c$ eine lineare Ordnung ist.*

3 Berechnung der modularen Zerlegung

Grundsätzlich handelt es sich bei dem Algorithmus von Tedder et al. um einen rekursiven, dem Divide-and-Conquer-Prinzip folgenden Algorithmus. Um den modularen Zerlegungsbaum T_G eines Graphen G zu bestimmen, wird das Problem zunächst in Teilprobleme, d.h. die Berechnung von modularen Zerlegunsbäumen von Sub-Graphen aufgespalten. Die zu lösenden Teilprobleme sind hierbei durch die maximalen Slices einer LBFS-Anordnung von G bestimmt. Hierbei ist die rekursive Eigenschaft der lexikographischen Breitensuche gemäß Lemma 1.3 grundlegend, da die für die Teilprobleme zu lösenden (Teil-)Teilprobleme wieder rekursiv durch maximale Slices einer LBFS-Anordnung definiert sind. Sobald die nötigen Teilprobleme gelöst sind, wird der modulare Zerlegungsbaum T_G aus den berechneten Teillösungen konstruiert. Über den Verlauf der Conquer-Phase werden dazu die Eigenschaften einer maximalen Slice Partition (Lemma 1.2) systematisch genutzt.

Die diesbezüglich wesentlichen Datenstrukturen sind geordnete Partitionen für die Verwaltung der noch ungelösten Teilprobleme sowie geordnete Listen von Bäumen, für die Verwaltung der entsprechenden Lösungen. Insgesamt kann der Algorithmus gut durch fünf Haupt-Funktionen (Algorithmus 4: `DivideMDTree`, Algorithmus 5: `TreeRefinement`, Algorithmus 6: `Factorize`, Algorithmus 7: `BuildSpine`, Algorithmus 8: `ConquerMDTree`) und eine sechste, optionale Wrapper-Funktion (Algorithmus 3: `MDTree`) beschrieben werden. Das folgende Kapitel stellt die verschiedenen Berechnungsphasen im Detail vor und gibt formale Argumente für die Korrektheit der Berechnungsschritte. Die Argumente für die Laufzeit des Algorithmus finden sich in Kapitel 4.

Durch den Funktionsaufruf `MDTree`(G) wird der Algorithmus gestartet und bei Beendigung der zu G korrespondierende, modulare Zerlegungsbaum T_G zurückgegeben.

Algorithmus 3 : `MDTree`(G)

Input : A graph G.

Output : The modular decomposition tree T for G.

```
1 (T, P) ← DivideMDTree(V(G), ∅);
2 return T;
```

3.1 Rekursive LBFS: Berechnung der Teilprobleme

Nachdem der Algorithmus durch das Aufrufen von `MDTree` gestartet wurde, beginnt die eigentliche Berechnung des modularen Zerlegunsbaums T_G zu einem Graphen G, mit dem in `MDTree` enthaltenen Ruf der kaskadierend[17] rekursiven Funktion `DivideMDTree`. Die Parameter von `DivideMDTree` sind eine Knotenmenge $S \subseteq V(G)$ und eine geordnete Partition $\mathcal{P}$ einer Knotenmenge $S' \subseteq V(G)$ mit $S \cap S' = \varnothing$. Als Rückgabewert, wird der modulare Zerlegungsbaum $T_{G[S]}$ des Graphen $G[S]$ und eine Verfeinerung $\mathcal{P}'$ der Partition $\mathcal{P}$ zurückgegeben. Entsprechend gibt der durch `MDTree` ausgelöste, initiale Ruf `DivideMDTree`$(V(G), \varnothing)$ den modularen Zerlegungsbaum T_G sowie eine leere Partition zurück.

Wie eingangs erwähnt, wird das Problem der Berechnung des modularen Zerlegunsbaumes T_G gelöst, indem zunächst die modularen Zerlegunsbäume von Sub-Graphen von G berechnet werden. Diese Berechnungen werden durch die rekursiven Aufrufe von `DivideMDTree` realisiert und folgen dabei der gleichen rekursiven Vorgehensweise. Erst wenn die Lösungen zu diesen Teilproblemen vorliegen, werden diese während der Conquer-Phase zu dem modularen Zerlegungsbaum T_G kombiniert. Die zu lösenden Teilprobleme sowie deren Abarbeitungsreihenfolge sind dabei durch die geordnete Partition der maximalen Slices $\mathcal{P}_S = [\{x\}, S_1, \ldots, S_{k'}]$ bezüglich einer LBFS-Anordnung σ von G mit dem Startknoten x als die durch die Slices S_i in G induzierten Graphen $G[S_i]$ definiert. Die zu den Teilproblemen korrespondierenden, modularen Zerlegungsbäume $T_{G[S_i]}$ werden von der Funktion `DivideMDTree` via einer geordneten Liste gewurzelter Bäume $\mathcal{T}$ verwaltet.

Definition 15. *Sei $G = (V, E)$ ein Graph und sei $\mathcal{P}_S = [P_1, \ldots, P_k]$ die geordnete Partition der maximalen Slices bezüglich einer LBFS-Anordnung σ von G. Es bezeichne $L(T)$ die Menge der Blätter eines eines gewurzelten Baumes T.*

- *Eine geordnete Liste gewurzelter Bäumen $\mathcal{T} = [T_1, \ldots, T_{k'}]$ mit $\bigcup_{i \in [1,k']} L(T_i) = V$, heißt **geordneten Baum-Partition von** G.*
- *Falls $\mathcal{T}$ eine geordnete Baum-Partition von G ist und zudem gilt, dass $k = k'$ und $L(T_i) = P_i$ für alle $i \in [1, k]$, so wird $\mathcal{T}$ eine **geordneten Baum-Partition der maximalen Slices von** G genannt. Ein Baum $T_i \in \mathcal{T}$ bezeichnet hierbei den modularen Zerlegungsbaum zu einem induzierten Graphen $G[P_i]$ mit $P_i \in \mathcal{P}_S$. $T_1 \in \mathcal{T}$ wird im Fall einer geordneten Baum-Partition der maximalen Slices von G als **Pivot** (oder **Pivotknoten**) bezeichnet.*

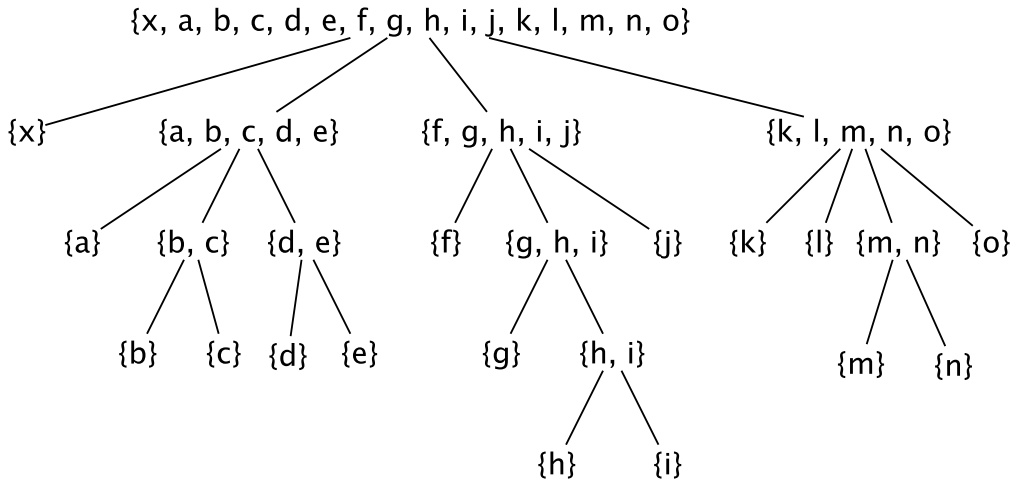


Abbildung 6: Die durch die LBFS-Anordnung $\sigma = x, a, b, c, d, e, f, g, h, i, j, k, l, n, m, o$ (vgl. Tabelle 3) definierten (Teil-)Probleme bei der Berechnung des modularen Zerlegungsbaums T_G. Jeder Knoten (der Abbildung) repräsentiert eine Menge $S \subseteq V(G)$, wobei dessen Kinder, eine Partition maximaler Slices bezüglich einer durch σ in $G[S]$ induzierten LBFS-Anordnung σ' repräsentieren.

Das Konzept der lexikographischen Breitensuche sowie der Begriff der Partition der maximalen Slices sind grundlegend für den gesamten Algorithmus. Insgesamt wird eine LBFS in dem Graphen G durch die Funktion `DivideMDTree` rekursiv realisiert.

Wie schon in dem Abschnitt 1.3 beschrieben, kann eine LBFS mittels der Technik der Partitionsverfeinerung umgesetzt werden. Hierbei wird eine geordnete Partition der Knotenmenge eines Graphen verwaltet und durch eine Folge von Verfeinerungsschritten – jeweils anhand der Nachbarschaft eines aus der ersten Partitionsklasse gewählten Knotens – so lange verfeinert, bis jede Partitionsklasse nur noch einen Knoten enthält. Jedes Mal, wenn dazu ein Knoten gewählt wird, wird dieser aus der Partition entfernt, was einer Nummerierung entspricht.

Die durch `DivideMDTree` induzierte LBFS nutzt diesen Ansatz, wobei sich die dazu nötigen Verfeinerungsschritte über die rekursiven Aufrufe von `DivideMDTree` – zu je einem Verfeinerungsschritt pro Aufruf – verteilen. In jedem Aufruf von `DivideMDTree` wird der Funktion eine Knotenmenge S und eine Partition $\mathcal{P}$ einer Knotenmenge S' als Argumente übergeben. Zusammengenommen, d.h. interpretiert man die Menge S als eine zusätzliche, erste Klasse in $\mathcal{P}$, stellt diese Partition die Partition der noch unnummerierten

[17] Es erfolgen mehrere rekursive Aufrufe innerhalb des Funktionsrumpfs.

Knoten dar, welche im Rahmen einer LBFS verwaltet wird. Je Aufruf von `DivideMDTree` wird ein Knoten $x \in S$ gewählt (Zeile 1) und die übergebene Partition $\mathcal{P}$ zunächst anhand der Nachbarschaft $N(x)$ verfeinert, also jede Partitionsklasse $P_i \in \mathcal{P}$ durch $P_i \cap N(x)$ und $P_i \setminus N(x)$ in dieser Reihenfolge ersetzt (Zeile 4-8). Es wird dann geprüft, ob $S = \{x\}$ (Zeile 9): Falls ja, wird der aus dem Knoten $x \in S$ bestehender Baum $T_1 = x$ und die verfeinerte Partition $\mathcal{P}$ an die aufrufende Funktion zurückgegeben (Zeile 10). Falls nein, so wird auch S in $S \cap N(x)$ und $S \setminus N[x]$ zerteilt und die entstehenden Teile der Partition – gemäß der Veranschaulichung, dass S die erste Klasse in $\mathcal{P}$ darstellt – hinzugefügt (Zeile 11-14). Die Liste $\mathcal{T}$ wird anschließend mit dem Baum $T_1 = x$ initialisiert (Zeile 15). In einer Schleife erfolgen dann weitere, rekursive Aufrufe von `DivideMDTree`, wobei die nun verfeinerte Partition $\mathcal{P}$ übergeben wird (Zeile 16-19).

Unter Vernachlässigung der exakten Rekursionsstruktur wird die LBFS also im Wesentlichen dadurch realisiert, dass die verwaltete Partition der Knoten zwischen verschiedenen Aufrufen von `DivideMDTree` durchgereicht wird. Jeder Knoten wird genau einmal als Pivotknoten genutzt, dabei aus der Partition entfernt und als Baumknoten initialisiert. Die Reihenfolge, nach welcher die Knoten als Pivotknoten auftreten, entspricht hierbei der LBFS-Anordnung σ.

Die exakte Sequenz der Aufrufe von `DivideMDTree`, durch welche die LBFS-Anordnung σ von G implizit produziert wird, ist im Detail allerdings von geringerem Interesse. Viel wichtiger ist die rekursive Struktur der Aufrufe. Diese gewährleistet, dass innerhalb eines Aufrufs der Form `DivideMDTree`$(S, \mathcal{P})$, eine geordneten Baum-Partition der maximalen Slices $\mathcal{P}_S$ von $G[S]$ bezüglich einer von σ in $G[S]$ induzierten LBFS-Anordnung σ' berechnet wird. Entsprechend wird die rekursive Eigenschaft einer LBFS gemäß Theorem 1.3 ausgenutzt. Zur Veranschaulichung der Rekursion kann Abbildung 6 herangezogen werden.

Konkret liegt die zu einem Aufruf `DivideMDTree`$(S, \mathcal{P})$ korrespondierende Partition der maximalen Slices $\mathcal{P}_S = [\{x\}, S_1, \ldots, S_k]$, als Auflistung aller noch zu bearbeitenden Teilprobleme, nie als Ganzes vor. Stattdessen lassen sich die einzelnen maximalen Slices $S_i \in \mathcal{P}_S$ zunächst als Wert der Variable P bei den rekursiven Aufrufen von `DivideMDTree`, während den Iterationen der While-Schleife beobachten (Zeile 18). In Gestalt der berechneten modularen Zerlegungsbäume $T_{G[S_i]}$, finden sich die Slices anschließend in der sukzessiv entstehenden, geordneten Baum-Partition der maximalen Slices $\mathcal{T}$ wieder. Die von der While-Schleife verwaltete Partition $\mathcal{P} = [P_1, \ldots, P_l]$, stellt zu Beginn der Schleife eine Partition der an den entsprechenden Aufruf von `DivideMDTree` übergebenen Knoten abzüglich dem Pivotknoten dar, d.h. $P_1 \cup \ldots \cup P_l = (S \cup S') \setminus x$. Zu diesem Zeitpunkt gilt für die erste Klasse P_1, dass $P_1 \subsetneq S$ bzw. $P_1 = N(x) \cap S$ falls $N(x) \cap S \neq \varnothing$. Bezüglich

der in $G[S]$ induzierten LBFS-Anordnung σ', stellt P_1 den ersten Slice von σ' dar und ist infolgedessen maximal in σ'[18]. Es gilt $P_1 = S_1$. Die Schleife läuft nun folgendermaßen ab: Sofern für die erste Klasse $P_1 \subseteq S$ gilt, wird diese jeweils, während den Iterationen aus der Partition entfernt und entsprechend der Parameterbelegung $S \leftarrow P_1$, $\mathcal{P} \leftarrow \{P_2, \ldots, P_l\}$ ein rekursiver Aufruf von `DivideMDTree` ausgelöst (Zeile 16-19). Hierdurch wird der modulare Zerlegungsbaum $T_{G[S_1]}$ berechnet und die übergebene Partition anhand der in P_1 enthaltenen Knoten rekursiv verfeinert. Die von der Schleife verwaltete Partition $\mathcal{P}$ wird dann entsprechend der berechneten Verfeinerung aktualisiert und der berechnete Baum der Liste $\mathcal{T}$ angehangen. Die jetzt erste Klasse P_1 entspricht genau dem nächsten maximalen Slice von σ', also $P_1 = S_2$. Dies ist dadurch begründet, dass P_1 der erste von S_1 disjunkte Slice hinsichtlich einer Auflistung aller Slices von σ' ist. Das Argument für die Maximalität der in Folgeiterationen übergebenen Slices ist analog. Für jeden bei der Berechnung von $T_{G[S_1]}$ zwischenzeitlich entstandenen Slice S' gilt, dass $S' \subsetneq S_1$. Diese Slices sind daher bezüglich σ' nicht maximal, wobei deren Bearbeitung ohnehin in tieferen rekursiven Aufrufen erfolgt.

Nach Beendigung der Schleife, ist die rekursive Bearbeitung der Teilprobleme abgeschlossen. Die Liste $\mathcal{T} = [T_1, \ldots, T_k]$ repräsentiert nun eine geordnete Baum-Partition maximaler Slices von $G[S]$. Diese enthält den, zu dem Pivotknoten x korrespondierenden, Baum $T_1(= x)$ sowie die modularen Zerlegungsbäume $T_{G[S_i]}$ für alle $S_i \in \mathcal{P}_S$. Für die anfangs übergebenen Partition $\mathcal{P}$ gilt jetzt, dass diese entsprechend aller Knoten $x \in S$ verfeinert wurde. Bezeichnet $\mathcal{P}$ die Partition bezüglich des Zustands vor den Verfeinerungsschritten und $\mathcal{P}'$ die Verfeinerung, so gilt für jedes Paar P'_i, P'_j mit $i < j$ und $P'_i \cup P'_j = P$ mit $P \in \mathcal{P}$, dass ein Knoten $x \in S$ existiert der universal zu P'_i und isoliert von P'_j ist.

Schließlich erfolgt die Berechnung des modularen Zerlegungsbaums $T_{G[S]}$ mittels dem Ruf `ConquerMDTree`($\mathcal{T}$) (Zeile 20). Wesentlich für die Konstruktion des modularen Zerlegungsbaums $T_{G[S]}$ aus den Teillösungen $T_i \in \mathcal{T}$ ist, dass $\mathcal{T}$ eine Baum-Partition maximaler Slices von $G[S]$ bezüglich σ' ist. Somit gelten die Eigenschaften gemäß Lemma 1.2, welche die Grundlage der Conquer-Phase bilden. Diesbezüglich kommt einer Kante $\{u, v\}$ in $G[S]$ eine besondere Rolle zu, falls $u \in P_i, v \in P_j$ wobei $i \neq j$. Jedes Mal, wenn ein Knoten als Pivotknoten x ausgewählt wird, werden derartige, ***aktive Kanten***, bezüglich x identifiziert und mittels einer, den Knoten zugeordneten, ***aktiven Liste*** repräsentiert (Zeile 2-3).

[18] Zur Veranschaulichung beachte man, dass im Rahmen einer LBFS via Partitionsverfeinerung ein – nicht notwendiger Weise maximaler – Slice, jeweils durch die erste Klasse der verwalteten Partition definiert wird.

Definition 16. *Sei $G = (V, E)$ ein Graph und bezeichne $\mathcal{P}_S = [P_1, \ldots, P_k]$ eine geordnete Partition maximaler Slices bezüglich LBFS-Anordnung σ von G.*

Eine Kante $uv \in E$ mit $u \in P_i$, $v \in P_j$ und $i \neq j$ heißt ***aktive Kante von*** *G. Bezüglich eines Knoten $v \in P_j$ bezeichnet $\alpha(v) = \{u \in P_i \mid uv \in E, i < j\}$ die* ***aktive Liste*** *von v.*

Sobald die Berechnung des modularen Zerlegungsbaums $T_{G[S]}$ abgeschlossen ist, wird $T_{G[S]}$ und die verbleibende Partition $\mathcal{P}$ zurückgegeben (Zeile 21).

Algorithmus 4 : `DivideMDTree`$(S, \mathcal{P})$

Input : A non-empty set $S \subseteq V(G)$, for some graph G, and an ordered partition $\mathcal{P}$ of a non-empty set $S' \subseteq V(G)$, where $S \cap S' = \varnothing$.

Output : A pair $(T, \mathcal{P}')$, where T is the MD tree for $G[S]$, and $\mathcal{P}' = P'_1, \ldots, P'_k$ is a refinement of $\mathcal{P}$ such that:
- each vertex x will have had its active list $\alpha(x)$ computed;
- every vertex in S is either universal to or isolated form every P'_i;
- for every pair $P'_i, P'_j, i < j$, such that $P'_i \cup P'_j \subseteq P$ for some $P \in \mathcal{P}$, there is a vertex in S that is universal to P'_i and isolated from P'_j.

```
1  choose some x ∈ S;
2  for y ∈ N(x) ∩ (S ∪ S') do
3      add x to α(y);
4  for P ∈ 𝒫 do
5      A ← P ∩ N(x);
6      B ← P \ A;
7      if A, B ≠ ∅ then
8          replace P in 𝒫 with A, B in this order;
9  if S = {x} then
10     return (x, 𝒫);
11 if S \ (N[x] ∩ S) ≠ ∅ then
12     prepend S \ (N[x] ∩ S) to 𝒫;
13 if N(x) ∩ S ≠ ∅ then
14     prepend N(x) ∩ S to 𝒫;
15 initialise the ordered list of trees 𝒯 with {x};
16 while P ⊆ S, where P is the first class in 𝒫 do
17     remove P from 𝒫;
18     (T, 𝒫) ← DivideMDTree(P, 𝒫);
19     append T to 𝒯;
20 𝒯 ← ConquerMDTree(𝒯)
21 return (T, 𝒫);
```

3.2 Baumverfeinerung: Module ohne Pivot

Während einer durch `ConquerMDTree`($\mathcal{T}$) ausgelösten Conquer-Phase des Algorithmus wird aus einer geordneten Baum-Partition der maximalen Slices $\mathcal{T} = [T_1, \ldots, T_k]$ eines Graphen G der modulare Zerlegungsbaum T_G berechnet. In einem Teil der Berechnung werden dazu die starken Module des Graphen G identifiziert, welche den Pivotknoten $T_1 = x$ nicht enthalten. Diese korrespondieren zu inneren Knoten der in $\mathcal{T}$ enthaltenen Bäume. Innere Knoten, welche keine solchen Module darstellen, werden markiert und die Reihenfolge der Kinder dieser Knoten im Hinblick auf eine faktorisierenden Permutation angepasst. Module ohne Pivot, welche keine starken Module von G sind, müssen hierbei nicht gesondert betrachtet werden, da es diesbezüglich ausreicht die starken, degenerierten Module ohne Pivot zu bestimmen (Theorem 2.3). Die Funktion `TreeRefinement` realisiert diese Berechnungen unter Benutzung der zuvor berechneten, aktiven Listen der Knoten.

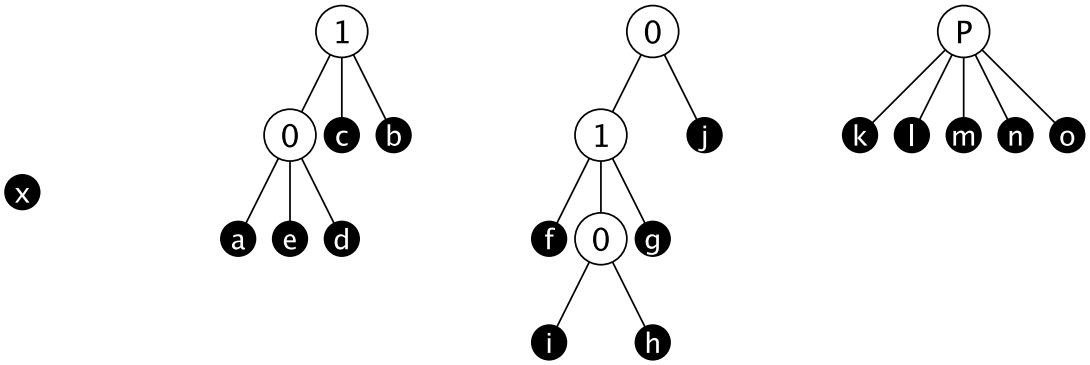


Abbildung 7: Die geordnete Baum-Partition der maximalen Slices $\mathcal{T}$ des Graphen G (Abbildung 3a) bezüglich der LBFS-Anordnung σ (vgl. Abbildung 6).

$x \in V(G)$	$\alpha(x)$
a, b, c, d, e	x
f, g, h, i, j	b, c
k, l, m, n, o	i

Tabelle 2: Die aktiven Listen der Knoten in $\mathcal{T}$ (Abbildung 7).

Lemma 3.1. *[49] Sei $\mathcal{P}_S = [P_1, \ldots, P_k]$ die geordnete Partition der maximalen Slices bezüglich einer LBFS-Anordnung σ von G. Falls M ein Modul von G ist, welches den*

Pivotknoten $x \in P_1$ nicht enthält, so existiert eine Klasse P_i, $i > 1$, wobei $M \subseteq P_i$ und M ein Modul in $G[P_i]$ ist.

Beweis. Falls M ein Modul in $G = (V, E)$ ist und $M \subseteq P_i$ folgt, dass M auch ein Modul in $G[P_i]$ ist. Angenommen M ist ein Modul in G und es existieren P_i, P_j mit $i < j$, wobei $M \cap P_i \neq \varnothing$ und $M \cap P_j \neq \varnothing$. Da $x \notin M$ ist $i > 1$. Wegen Lemma 1.2 gibt es eine Klasse P_l mit $l < i$ und einen Knoten $y \in P_l$, wobei y universal zu P_i und isoliert von P_j ist. Entsprechend gibt es einen Knoten $a \in M \cap P_i$ mit $ya \in E$ und einen Knoten $b \in M \cap P_j$ mit $yb \notin E$. Dies widerspricht der Annahme, dass M ein Modul in G ist. □

Offensichtlich repräsentiert jeder innere Knoten u eines $T_i \in \mathcal{T}$, $i > 1$ ein (starkes) Modul M in $G[P_i]$ mit $x \notin M$. Hierbei gilt jedoch nicht zwangsläufig, dass M in diesem Fall auch ein Modul in G darstellt. Falls eine solche Menge $M \subseteq P_i$ zwar ein Modul in $G[P_i]$, aber kein Modul in G ist, so gibt es einen Knoten $y \in V \setminus P_i$, der nicht homogen bezüglich M ist. Entsprechend genügt es einen derartigen Knoten y zu finden, um eine Menge $M \subseteq P_i$ als Modul in G auszuschließen.

Der naive Ansatz, alle Knoten in $V \setminus P_i$ diesbezüglich zu betrachten, ist hierbei vermeidbar, da für jeden Knoten $y \in P_l$ mit $l < i$ bereits gilt, dass dieser entweder universal zu oder isoliert von P_i bzw. $M \subseteq P_i$ ist (Lemma 1.2). Folglich müssen zur Prüfung, ob $M \subseteq P_i$ ein Modul in G ist, lediglich alle Knoten $y \in P_j$ mit $i < j$ bzw. deren aktive Kanten ausgewertet werden. Gilt in diesem Zusammenhang für die Menge M (d.h. für die Blätter $L(T_i[u])$ eines, durch einen inneren Knoten u in T_i bestimmten Teilbaums), dass es einen Knoten $y \in P_j$ gibt, sodass $M \perp \alpha(y)$, dann ist M kein – den Pivotknoten x nicht enthaltendes – Modul von G.

Für die technische Umsetzung[19] innerhalb der Funktion `TreeRefinement` wird jeweils ein Baum T_i bzw. dessen Blätter L_i und ein Knoten $y \in L_j$ mit $i < j$ untersucht. Zunächst werden alle Knoten in $\alpha(y) \cap L_i$ temporär markiert (Zeile 4). Im Anschluss werden die inneren Knoten des Baums T_i betrachtet und ebenfalls mit temporären Markierungen ausgestattet, falls jeweils all ihre Kinder markiert sind (Zeile 5-6). Anhand der temporären Markierungen lassen sich dann die Knoten in T_i identifizieren, welche aufgrund ihrer Inhomogenität bezüglich y keine Module in G repräsentieren. Technisch zeichnet sich ein derartiger Knoten u dadurch aus, dass dieser keine Markierung besitzt, während mindestens ein Kind von u markiert ist. Im Hinblick auf die weitere Verarbeitung von T_i bzw. $\mathcal{T}$ in der Funktion `Factorize`, wird diese Information mittels eines ***dead***-Labels in dem Knoten u kodiert (Zeile 13-14). Falls u ein degenerierter Knoten ist, erfolgt zudem eine Ersetzung der Kinder von u durch zwei neue, degenerierte Knoten des selben Typs.

[19] Für eine ausführlichere Betrachtung der technischen Details sei auf Abschnitt 4.3 verwiesen.

Entweder ersetzen zwei *seriell* oder zwei *parallel*-gelabelte Knoten die Kinder von u. Unter diesen neuen Knoten werden die alten Kinder von u, entsprechend markierten bzw. unmarkierten Knoten gruppiert. Hierbei wird der Vater der markierten Knoten ebenfalls temporär markiert (Zeile 8-12). In Abhängigkeit des Index i erfolgt außerdem eine Anpassung der Reihenfolge der neuen Kinder von u: Falls $i \neq 2$, so wird das temporär markierte Kind von u zum ***rechten*** und das unmarkierte Kind zum ***linken Kind*** von u. Falls $i = 2$ ist die Reihenfolge vertauscht (Zeile 15-20)[20]. Mit der Entfernung der temporären Markierungen, endet die Betrachtung eines Baums T_i und eines Knoten $y \in L_j$ mit $i < j$. Der Baum T_i ist nun bezüglich des Knoten y ***verfeinert***. In den Folgeschritten erfolgen weitere Verfeinerungsschritte, sodass sich mit Beendigung von `TreeRefinement`, eine ***Verfeinerung*** $\mathcal{T}'$ von $\mathcal{T}$ ergibt. Für die Verfeinerung $\mathcal{T}'$ gilt, dass jeder Knoten mit einem *dead*-Label genau ein linkes und ein rechtes Kind hat und jeder Knoten u in einem $T_i \in \mathcal{T}'$ eine Menge M repräsentiert, die genau dann homogen bezüglich jedem Knoten $y \in L(T_j)$ für alle $j \neq i$ ist, wenn weder u noch ein Nachfolger von u ein dead-Label trägt.

Definition 17. *Seien T, T' zwei gewurzelte Bäume mit $L(T) = L(T')$. T' bezeichnet eine* ***Verfeinerung*** *von T, falls es für jeden Knoten u in T einen Knoten v in T' gibt, sodass $L(T[u]) = L(T'[v])$. Seien $\mathcal{T} = [T_1, \ldots, T_k]$, $\mathcal{T}' = [T'_1, \ldots, T'_k]$ zwei geordnete Listen gewurzelter Bäume. $\mathcal{T}'$ wird eine* ***Verfeinerung*** *von $\mathcal{T}$ genannt, falls für alle $i \in [1, k]$ gilt, dass T'_i eine Verfeinerung von T_i ist.*

Durch die *dead*-Labels ist für alle Knoten in $\mathcal{T}'$ eindeutig bestimmt, ob diese Module von G (ohne Pivot) repräsentiert oder nicht. Die folgenden Aussagen widmen sich diesen Zusammenhängen:

Lemma 3.2. *[49] Sei $\mathcal{T}' = [T'_1, \ldots, T'_k]$ die durch `TreeRefinement` berechnete Verfeinerung einer geordneten Baum-Partition der maximalen Slices $\mathcal{T} = [T_1, \ldots, T_k]$ eines Graphen $G = (V, E)$.*

Ein Knoten u in T'_i repräsentiert eine Menge von Knoten $M \subseteq L(T'_i)$, die kein Modul in G ist, falls u oder ein Nachfolger von u ein dead-Label trägt.

Beweis. Falls u in T'_i ein *dead*-Label trägt, so gibt es einen Baum T'_j mit $i < j$ und $L(T'_i) \cap L(T'_j) = \varnothing$, einen Knoten $y \in L(T'_j)$ und zwei Knoten $a, b \in L(T'_i[u])$, sodass $ya \in E$ und $yb \neq E$. Der Knoten u repräsentiert deswegen kein Modul in G.

Sei v in T'_i ein Vorfahre von u ohne *dead*-Label. Da $L(T'_i[u]) \subsetneq L(T'_i[v])$, gilt $a, b \in L(T'_i[v])$. Der Knoten v entspricht daher keinem Modul in G. □

[20] Die Anpassung der Reihenfolge der Kinder der *dead*-gelabelten Knoten, erfolgte in der Fassung des Algorithmus in [49] erst in `Factorize`.

Lemma 3.3. *[49] Sei $\mathcal{T}' = [T'_1, \ldots, T'_k]$ die durch* `TreeRefinement` *berechnete Verfeinerung einer geordneten Baum-Partition der maximalen Slices $\mathcal{T} = [T_1, \ldots, T_k]$ eines Graphen $G = (V, E)$.*

Ein Knoten u in T'_i mit $i < 1$ repräsentiert eine Menge von Knoten $M \subseteq L(T'_i)$, die ein - den Pivotknoten $x = T'_1$ nicht enthaltendes - Modul in G ist, falls weder u noch ein Nachfolger von u ein dead-Label trägt.

Beweis. Sofern u eine Menge von Knoten $M \subsetneq L(T'_i)$ repräsentiert und $i < 1$, dann ist $x \notin M$, da $L(T'_i) \cap L(T'_j) = \varnothing$ für $i \neq j$ und $x \in L(T'_1)$.

Falls u in T'_i kein durch `TreeRefinement` neu erzeugter Knoten ist, so ist u auch ein Knoten in T_i. Entsprechend gilt, dass $M = L(T_i[u]) = L(T'_i[u])$, wobei M ein Modul in $G[L(T_i)]$ ist. Da T'_i eine Verfeinerung von T_i ist, ist $G[L(T_i)] = G[L(T'_i)]$ und daher M ein Modul in $G[L(T'_i)]$.

Falls u in T'_i ein, durch `TreeRefinement` neu erzeugter Knoten ist, so entspricht u einer Vereinigung M, der zu $k' > 1$ vielen Kindern eines degenerierten Knotens in T_i korrespondierenden Module in $G[L(T_i)]$. Aus diesem Grund ist M selbst ein Modul in $G[L(T_i)]$ (Theorem 2.3).

Wegen Lemma 1.2 gilt für alle Knoten v in T'_l, für alle $l < i$, dass v homogen bezüglich der durch u in T'_i repräsentierte Menge M ist. Da weder u, noch ein Nachfolger von u ein *dead*-Label trägt, gilt für alle Knoten w in T'_j und für alle j mit $i < j$, dass w homogen bezüglich M ist. Entsprechend ist M ein Modul in G, welches den Pivotknoten x nicht enthält. □

Lemma 3.4. *[49] Sei $\mathcal{T}' = [T'_1, \ldots, T'_k]$ die durch* `TreeRefinement` *berechnete Verfeinerung einer geordneten Baum-Partition der maximalen Slices $\mathcal{T} = [T_1, \ldots, T_k]$ eines Graphen $G = (V, E)$.*

Falls $M_S \subsetneq V$ ein - den Pivotknoten $x = T'_1$ nicht enthaltendes - starkes Modul in G ist, so gibt es einen Baum T'_i in $\mathcal{T}'$ und einen Knoten u in T'_i, wobei weder u noch ein Nachfolger von u ein dead-Label trägt. Der Knoten u repräsentiert in diesem Fall das Modul M_S und der Teilbaum $T'_i[u]$ entspricht dem modulare Zerlegungsbaum von $G[M_S]$.

Beweis. Es sei $L_i = L(T_i)$. Falls M_S ein Modul (ohne Pivot) in G ist, so gibt es ein $T_i \in \mathcal{T}$ mit $M_S \subseteq L_i$ (Lemma 3.1). Da T_i den modularen Zerlegungsbaum von $G[L_i]$ bezeichnet existiert entweder ein Knoten q in T_i, sodass q die Menge $L(T_i[q]) = M_S$ repräsentiert oder es gibt einen degenerierten Knoten p in T_i und eine echte Teilmenge $\{f_1, \ldots, f_l\}$ der Kinder von p, sodass $\bigcup_{j \in [1,l]} L(T_i[f_j]) = M_S$ (Theorem 2.3).

Falls q das Modul M_S repräsentiert, so gibt es einen Knoten q' in T'_i, wobei $L(T_i[q']) = M_S$, da $\mathcal{T}'$ eine Verfeinerung von $\mathcal{T}$ ist. Zusätzlich gilt wegen der Kontraposition von

Lemma 3.2, dass weder q' noch ein Nachfolger von q' ein *dead*-Label trägt, da M_S ein Modul ist. Hierbei ist $T_i[q']$ der modulare Zerlegunsbaum von $G[M_S]$.

Falls M_S durch die Vereinigung $\bigcup_{j\in[1,l]} L(T_i[f_j]) = M_S$ bezüglich der Kinder $\{f_1, \ldots, f_l\}$ eines Knoten p repräsentiert wird, so existieren die Knoten $f'_1, \ldots, f'_l$ in T'_i mit $\bigcup_{j\in[1,l]} L(T_i[f'_j]) = M_S$, da $\mathcal{T}'$ eine Verfeinerung von $\mathcal{T}$ ist. Da M_S ein Modul ist, gilt für jeden Knoten y in T_j für alle $j > i$, dass y homogen bezüglich M_S ist, sodass die Knoten $f'_1, \ldots, f'_l$ einen gemeinsamen Vater v in T'_i haben ($f_1, \ldots, f_l$ sind die Kinder von p in T_i und wurden gemäß ihrer Nachbarschaft zu allen Knoten y in T_j für alle $j > i$ gruppiert). Entsprechend repräsentiert v das Modul M_S, wobei weder v noch ein Nachfolger von v ein *dead*-Label besitzt (Kontraposition von Lemma 3.2).

Abschließend wird gezeigt, dass die Knoten $f'_1, \ldots, f'_l$ die einzigen Kinder von v sind. Dies zeigt, dass M_S ist ein starkes Modul ist. Angenommen v hat neben den Knoten $f'_1, \ldots, f'_l$ ein weiteres Kind c'. Der Knoten c' entspricht dann einem Knoten c, der ein Kind des degenerierten Knotens p ist, d.h. $L(T_i[c]) = L(T'_i[c'])$. Entsprechend repräsentiert c ein Modul M' in $G[L_i]$. Weil c' ein Nachfolger von v ist, hat weder c' noch ein Nachfolger von c' ein *dead*-Label, sodass M' auch ein Modul in G ist (Lemma 3.3). Da v ein degenerierter Knoten sein muss – v ist ein neu erzeugtes Kind von p – gibt es eine Menge M'' mit $M' \subsetneq M''$, die M_S überlappt, d.h. $M'' \perp M_S$. Dies widerspricht der Tatsache, dass M_S ein starkes Modul ist. Entsprechend ist M_S durch den Knoten v in T'_i mit den Kindern $f'_1, \ldots, f'_l$ repräsentiert, wobei weder v noch ein Nachfolger von v ein *dead*-Label trägt. Hierbei bezeichnet $T'_i[v]$ den modulare Zerlegunsbaum von $G[M_S]$

□

Wie Lemma 3.4 zeigt, sind bei Beendigung von `TreeRefinement` die starken Module, wechen Pivotknoten x nicht enthalten, durch die Teilbäume in $\mathcal{T}'$ bestimmt, welche frei von *dead*-Label sind. Genau diese Teilbäume finden sich schließlich in dem modularen Zerlegungsbaum des Graphen G wieder. In dem sich anschließenden Teil entsteht unter Nutzung der in `TreeRefinement` vergebenen *dead*-Labels eine faktorisierende Permutation. Mittels dieser lassen sich auch die starken Module in G ermitteln, welche den Pivotknoten x enthalten.

Algorithmus 5 : `TreeRefinement`$(\mathcal{T})$

Input : A maximal slice tree partition $\mathcal{T} = T_1, \ldots, T_k$ of some graph G, such that if $L_1, \ldots, L_k$ are the corresponding leaf sets, then T_i is the MD tree for $G[L_i]$. Moreover, each leaf $x \in L_j$ has an associated set $\alpha(x)$ consisting of its neighbors amongst the leaves of the T_i's, $i < j$. Finally, no node is labelled *dead* (see output).

Output : A refinement $\mathcal{T}' = T'_1, \ldots, T'_k$ of $\mathcal{T}$, in which some nodes are labelled *dead*, such that:

- each node labelled *dead* has exactly two children, one the *left* child, the other the *right* child;
- for each node u in T'_i, every leaf $x \in L_j$, $i < j$, is either universal to or isolated from u if and only if neither u nor any of its descendants are labelled *dead*.

```
1  for j ∈ [2, k] do
2      for y ∈ L_j do
3          for i ∈ [1, i − 1] do
4              mark each leaf in α(y) ∩ L_i and remove it from α(y);
5              while there is an unmarked node u, all of whose children are marked do
6                  mark u;
7              foreach unmarked node u with a marked child do
8                  let A be u's marked children and let B be u's unmarked children;
9                  if |A| > 1 and u is degenerate then
10                     replace the children in A with a new marked node inheriting u's type,
                       whose children are those nodes in A;
11                 if |B| > 1 and u is degenerate then
12                     replace the children in B with a new unmarked node inheriting u's
                       type, whose children are those nodes in B;
13                 if u is not labelled dead then
14                     label u as dead;
15                     if i = 2 then
16                         make u's marked child its left child;
17                         make u's unmarked child its right child;
18                     else
19                         make u's marked child its right child;
20                         make u's unmarked child its left child;
21             clear all marks off nodes (but not dead labels);
22 return T;
```

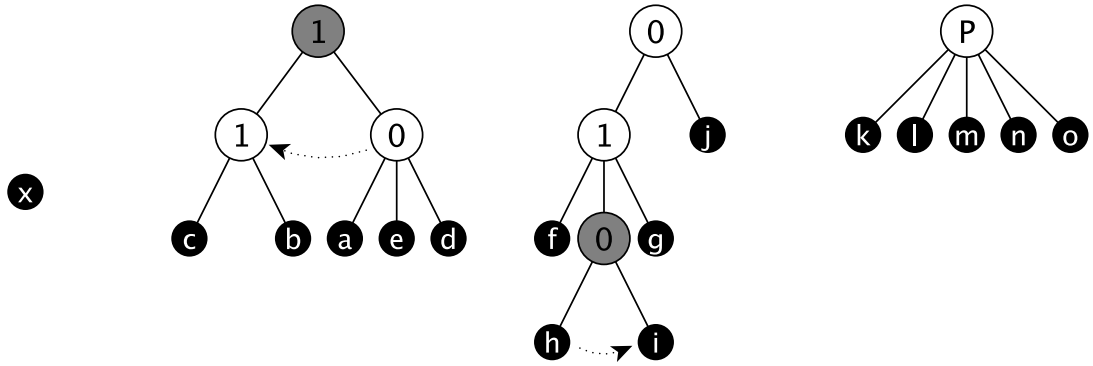


Abbildung 8: Die durch `TreeRefinement` berechnete Verfeinerung $\mathcal{T}'$ der Baum-Partition $\mathcal{T}$ (Abbildung 7). Ein neu erzeugter, serieller Knoten gruppiert die Blätter $c, d \in L(T_2)$. *Dead*-gelabelte Knoten sind grau dargestellt. Die Veränderungen der Reihenfolgen, der Kinder der *dead*-gelabelten Knoten, sind durch Pfeile dargestellt.

3.3 Faktorisierende Permutation

Nachdem die starken Module in $G = (V, E)$, welche den Pivotknoten x nicht enthalten, durch die Funktion `TreeRefinement` bestimmt wurden, erfolgt in der Funktion `Factorize` die Berechnung einer faktorisierenden Permutation σ. Hierzu wird der Funktion die durch `TreeRefinement` berechnete Verfeinerung einer geordneten Baum-Partition der maximalen Slices $\mathcal{T} = [T_1, \ldots, T_k]$ des Graphen G übergeben. Bei Beendigung von `Factorize` wird eine geordnete Baum-Partition $\mathcal{T}' = [T'_1, \ldots, T'_{k'}]$ zurückgegeben, wobei die darin enthaltenen Bäume ausschließlich Knoten enthalten, welche tatsächlich Module in G repräsentieren. Die Anordnung der Bäume in $\mathcal{T}'$ ist hierbei von zentralem Interesse und so gestaltet, dass eine Anordnung aller Blätter in $\mathcal{T}'$ in Verbindung mit einer Umpositionierung des Pivotknoten $T_1 = x$, eine faktorisierende Permutation σ der Knotenmenge V ergibt.

Definition 18. *Sei $\mathcal{T} = [T_1, \ldots, T_k]$ eine geordnete Baum-Partition eines Graphen $G = (V, E)$.*

*Eine **Anordnung der Blätter** eines Baumes $T_i \in \mathcal{T}$ gemäß der Reihenfolge einer **in-order Traversierung**[21] (einfach: Traversierung) von T_i wird mit $\sigma(T_i)$ bezeichnet. Eine entsprechende Anordnung der Blätter aller in $\mathcal{T}$ enthaltenen Bäume wird durch $\sigma(\mathcal{T}) = \sigma(T_1), \ldots, \sigma(T_k)$ beschrieben.*

[21] Eine in-order Traversierung eines Baumes T ist effektiv eine, in der Wurzel begonnene, Tiefensuche. Hierbei gilt die zusätzliche Beschränkung, dass ein linker Teilbaum zuerst untersucht wird.

Im Folgenden, wird zunächst eine Beschreibung des technischen Ablaufs der Funktion `Factorize` gegeben[22]. Im Anschluss daran erfolgt eine Auseinandersetzung mit den Eigenschaften, welche für die Anordnung der Blätter $\sigma(\mathcal{T}')$ gelten. Hierdurch lässt sich begründen, dass $\sigma(\mathcal{T}')$ in Verbindung mit einer Umpositionierung des Pivotknoten $T_1 = x$, die gewünschte Eigenschaft einer faktorisierenden Permutation erfüllt. Schließlich werden weitere nützliche Zusammenhänge vorgestellt, welche die Konstruktion des modularen Zerlegungsbaums T_G mittels der faktorisierenden Permutation σ begünstigen.

Die Funktion `Factorize` läuft folgendermaßen ab: Entsprechend der durch $\mathcal{T}$ gegebenen Reihenfolge, werden die darin enthaltenen Bäume nacheinander betrachtet und dabei Knoten entfernt, welche gemäß Lemma 3.2 keine Module in G repräsentieren. Bezüglich eines $T_i \in \mathcal{T}$ tragen die zu entfernenden Knoten entweder ein *dead*-Label oder sind ein Vorgänger, eines *dead*-gelabelten Knotens. Letztere werden mit einem ***zombie***-Label ausgestattet (Zeile 2-3). Was nun folgt, ist die Vorbereitung des Baums T_i auf das Entfernen der Knoten. Die Kinder jedes, mit einem *zombie*-Label versehenen, Knotens u werden entsprechend gelabelten (*dead*/*zombie*) und ungelabelten Knoten gruppiert (Zeile 5). Handelt es sich bei dem betrachteten Knoten u um einen degenerierten Knoten, so wird für die Gruppierung der ungelabelten Kinder von u zusätzlich ein neu erzeugter, degenerierter Knoten u' des selben Typs (*parallel*/*seriell*) herangezogen. Der Knoten u' ersetzt in diesem Fall die ungelabelten Kinder von u, wobei diese zu den Kindern von u' werden (Zeile 6-7)[23]. In Abhängigkeit des Index i erfolgt dann eine Anpassung der Reihenfolge der Kinder von u: Falls $i = 2$, erscheinen die gelabelten Kinder vor den ungelabelten Kindern. Falls $i \neq 2$, so ist die Reihenfolge vertauscht (Zeile 8-11).

Der eigentliche Löschvorgang wird mittels einer zu T_i korrespondierenden, temporären, geordneten Baum-Partition $\mathcal{T}_i$ realisiert. Hierfür wird $\mathcal{T}_i$ mit dem Baum T_i initialisiert (Zeile 12). Die Wurzel des Baumes T_i wird betrachtet: Handelt es sich hierbei um einen gelabelten Knoten (*dead*/*zombie*), so erfolgt eine Ersetzung von T_i durch die in den Kindern von T_i gewurzelten Teilbäume, gemäß der zuvor fixierten Reihenfolge. Für die nun in $\mathcal{T}_i$ enthaltenen Bäume erfolgen dann so lange entsprechende Substitutionen, bis alle Bäume in $\mathcal{T}_i$ frei von Labeln sind (Zeile 13-14). Abschließend wird T_i innerhalb von $\mathcal{T}$ durch die Bäume in $\mathcal{T}_i$, gemäß ihrer Reihenfolge, ersetzt. Die Betrachtung des Baums T_i ist hiermit beendet. Falls $i \neq k$, so erfolgt als Nächstes die Bearbeitung des Baums T_{i+1}.

[22] Die Betrachtung der Laufzeit von `Factorize` erfolgt in Abschnitt 4.4.

[23] In der Fassung des Algorithmus in [49] wird darauf verzichtet, die ungelabelten Kinder eines *zombie*-gelabelten, degenerierten Knotens unter einem neuen Knoten zu gruppieren. Dies vernachlässigt die Nachbarschaftsbeziehung der involvierten Knoten.

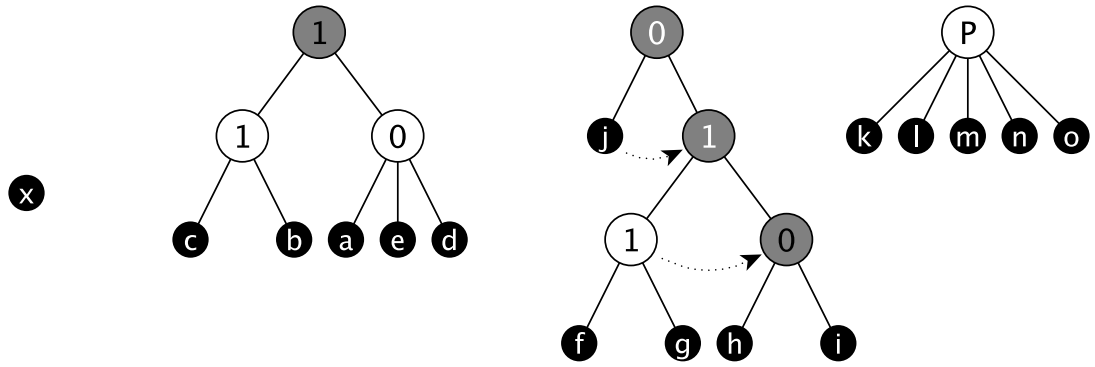


Abbildung 9: Während der Ausführung von `Factorize`, werden die Vorfahren der *dead*-gelabelten Knoten (grau/schwarz) in $\mathcal{T}$ (Abbildung 8) mit *zombie*-Labeln ausgestattet (grau/weiß). Ein neu erzeugter, serieller Knoten gruppiert die Blätter $f, g \in T_3$. Die Veränderungen der Reihenfolgen der Kinder der *zombie*-gelabelten Knoten sind durch Pfeile dargestellt.

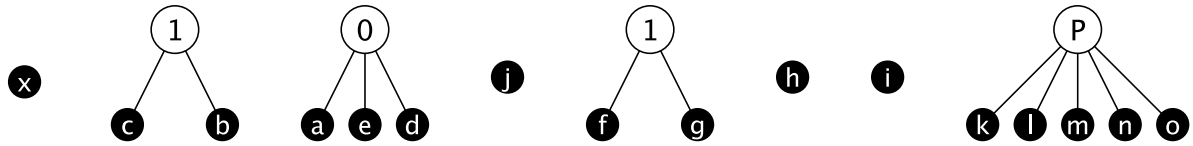


Abbildung 10: Die durch `Factorize` berechnete Baum-Partition $\mathcal{T}'$.

Es folgt die Untersuchung der Anordnung der Blätter $\sigma(\mathcal{T}')$. Die $\mathcal{T}$ zugrundeliegende Partition der maximalen Slices sei mit $\mathcal{P}_S = [P_1, \ldots, P_k]$ bezeichnet.

Vergleicht man die geordneten Baum-Partitionen $\mathcal{T}$ und $\mathcal{T}'$ miteinander, so lässt sich dabei eine grundsätzliche Beobachtung machen: Sofern ein Knoten u in einem $T_i \in \mathcal{T}$ kein *dead*-Label trägt und keinen *dead*-gelabelten Nachfahren hat, so bleibt der Teilbaum $T_i[u]$ erhalten. D.h. es gibt einen Baum $T'_j \in \mathcal{T}'$ und einen Knoten v in T'_j, sodass $T'_j[v]$ dem Baum $T_i[u]$ entspricht. Die Blätter $L(T_i[u])$ sind daher aufeinanderfolgend in $\sigma(\mathcal{T}')$. Falls u einen Knoten mit einem *dead*-Label oder einem *dead*-gelabelten Nachfahren bezeichnet, so gibt es keinen dem Baum $T_i[u]$ entsprechenden Baum in $\mathcal{T}'$. In diesem Fall kommt es zu einer Auflösung des Knotens u, wobei u zu diesem Zeitpunkt die Wurzel eines eigenständigen Baums $T_i[u]$ ist. Der Baum $T_i[u]$ wird dann durch eine aufeinanderfolgende Sequenz von Bäumen $[T''_1, \ldots, T''_l]$ ersetzt. Hierbei gilt, dass $\bigcup_{j \in [1,l]} L(T''_j) = L(T_i[u])$. Deswegen gilt auch für diesen Fall, dass die Knoten in $L(T_i[u])$ aufeinanderfolgend in $\sigma(\mathcal{T}')$ sind. Die Reihenfolge der Bäume $[T''_1, \ldots, T''_l]$ spielt in diesem Zusammenhang keine

Rolle. Entsprechend dieser zwei möglichen Fälle, ist die Eigenschaft für jeden Knoten u in einem beliebigen $T_i \in \mathcal{T}$ gewährleistet.

Lemma 3.5. *[49] Sei $\mathcal{T} = [T_1, \ldots, T_k]$ eine, durch* `TreeRefinement` *produzierte, geordnete Baum-Partition der maximalen Slices von $G = (V, E)$. Sei $\mathcal{T}' = [T'_1, \ldots, T'_{k'}]$ die durch* `Factorize`$(\mathcal{T})$ *berechnete, geordnete Baum-Partition von G. Sei $\sigma = \sigma(\mathcal{T}')$ die Anordnung der Blätter von $\mathcal{T}'$. Es gilt: Für jeden Knoten u in einem beliebigen Baum $T_i \in \mathcal{T}$ existiert ein Paar $a, b \in [1, |V|]$ mit $a \leq b$, sodass $L(T_i[u]) = \{\sigma^{-1}(k) \mid a \leq k \leq b\}$. (D.h. die Knoten $L(T_i[u])$ sind in σ aufeinanderfolgend .)*

Beweis. Gemäß der Argumentation im Haupttext. □

Als eine Konsequenz von Lemma 3.5 sowie Lemma 3.4 sind die zu einem starken Modul $M \subsetneq V$ (ohne Pivot) gehörenden Knoten aufeinanderfolgen in $\sigma(\mathcal{T}')$. Hinsichtlich den starken Modulen von G, welche den Pivotknoten x enthalten, ist die Argumentation etwas aufwendiger: Bezeichnet $M \subseteq V$ ein solches Modul, so gibt es offensichtlich keinen Knoten in $\mathcal{T}$, welcher M repräsentiert. Stattdessen beschreibt M – im nicht-trivialen Fall – eine Menge von Knoten, welche mit mindestens zwei Klassen $P_i, P_j \in \mathcal{P}_S$ überschneidet, also $M \cap P_i \neq \varnothing$ und $M \cap P_j \neq \varnothing$. Einer Teilmenge $C \subseteq P_i$ kommt in diesem Zusammenhang eine zentrale Rolle zu, falls C eine Komponente oder eine Ko-Komponente in einem Graphen $G[P_i]$ induziert[24].

Lemma 3.6. *[49] Sei $\mathcal{P}_S = [P_1, \ldots, P_k]$ die geordnete Partition der maximalen Slices bezüglich einer LBFS-Anordnung σ von G. Sei M ein Modul von $G = (V, E)$, welches den Pivotknoten $x \in P_1$ enthält. Es gilt:*

1. *Falls C' eine Ko-Komponente in $G[P_2]$ ist, so gilt entweder $C' \subseteq M$ oder $C' \cap M = \varnothing$;*
2. *Falls C eine Komponente in $G[P_i]$ mit $i > 2$ ist, so gilt entweder $C \subseteq M$ oder $C \cap M = \varnothing$.*

Beweis. (1.) Da $x \in M \setminus C'$, gilt $M \setminus C' \neq \varnothing$. Somit ist der Fall $M \subseteq C'$ ausgeschlossen. Angenommen es gilt, dass $C' \cap M \neq \varnothing$ und $C' \setminus M \neq \varnothing$. Es bezeichne $C'_1 = C' \cap M$ und $C'_2 = C' \setminus C'_1$. Da $x \in M$ und da $C' \subseteq P_2 = N(x)$, gilt für jedes Paar $a \in C'_1$, $b \in C'_2$, dass $ab \in E$. Dies widerspricht der Voraussetzung, dass C' eine Ko-Komponente in $G[P_2]$ ist. (2.) Analog zu (1.). □

[24] Zur Vereinfachung bezeichnet der Begriff der Komponente im Folgenden – abhängig vom Kontext – entweder eine Menge von Knoten, welche eine Komponente in einem Graphen induziert oder den entsprechenden Graphen. Gleiches gilt für den Begriff der Ko-Komponente.

Die zu einer Komponente/Ko-Komponente C in einem $G[P_i]$ gehörenden Knoten, bilden dabei eine Sequenz in $\sigma(\mathcal{T}')$. Dies ist dadurch begründet, dass es sich bei den Bäumen $T_i \in \mathcal{T}$, um Verfeinerungen der modularen Zerlegungsbäume $T_{G[P_i]}$ handelt. Folglich gibt es zu jedem Knoten q_C in einem beliebigen $T_{G[P_i]}$, welcher entweder eine Komponente oder eine Ko-Komponente C von $G[P_i]$ repräsentiert – Entweder die Wurzel oder ein Kind der Wurzel eines $T_{G[P_i]}$ (vgl. Theorem 2.6) – einen Knoten p in $T_i \in \mathcal{T}$ mit $L(T_i[p]) = L(T_{G[P_i]}[q_C])$. Mit der Existenz des Knoten p und Lemma 3.5 ergibt sich die Eigenschaft. Entsprechend ist es möglich, die Anordnung $\sigma(\mathcal{T}')$ als Folge von Komponenten/Ko-Komponenten C_i aufzufassen, d.h. $\sigma(\mathcal{T}') = x, C_1, \ldots, C_c$. Beabsichtigt man ein starkes Modul M (mit Pivot) durch eine Menge von Komponenten/Ko-Komponenten darzustellen, so ist es gemäß Lemma 3.6 günstig, im Fall der Klasse P_2, speziell die Ko-Komponenten von $G[P_2]$ zu betrachten. In den restlichen Fällen, erweisen sich die Komponenten in einem $G[P_i]$, $i > 2$ als hierfür geeignet. Hinsichtlich einer Klasse $P_i \in \mathcal{P}_S$ gilt ebenfalls, dass die darin enthaltenen Knoten in $\sigma(\mathcal{T}')$ aufeinanderfolgend sind. Diese Eigenschaft folgt direkt aus Definition 15 zusammen mit Lemma 3.5. Die Knoten innerhalb einer Klasse $P_i \in \mathcal{P}_S$, erscheinen dabei vor den Knoten in einer Klasse $P_j \in \mathcal{P}_S$, falls $i < j$. Außerdem lässt sich jede Klasse $P_i \in \mathcal{P}_S$ in Komponenten oder Ko-Komponenten $\{C_1, \ldots, C_l\}$ unterteilen, wobei $\bigcup_{i\in[1,l]} C_i = P_i$. In Verbindung mit der folgenden Aussage ergibt sich hiermit ein weiteres Argument für das sequenzielle Erscheinen der Knoten eines starken Moduls (mit Pivot) in $\sigma(\mathcal{T}')$.

Lemma 3.7. *[49] Sei $\mathcal{P}_S = [P_1, \ldots, P_k]$ die geordnete Partition der maximalen Slices bezüglich einer LBFS-Anordnung σ von G. Zusätzlich sei M ein starkes Modul von G, welches den Pivotknoten $x \in P_1$ enthält.*

Falls es eine Klasse P_j mit $j > 3$ und $P_j \cap M \neq \varnothing$ gibt, so gilt $P_2 \cap M \neq \varnothing$ und $P_3, \ldots, P_{j-1} \subsetneq M$.

Beweis. Im ersten Teil des Beweises wird gezeigt, dass $P_2 \cap M \neq \varnothing$, falls es eine Klasse P_j mit $j > 3$ und $P_j \cap M \neq \varnothing$ gibt: Wegen Lemma 1.2 (3.a) gilt für die Klassen P_3 und P_j, dass es einen Knoten $y \in P_2$ gibt, wobei y universal zu P_3 und isoliert von P_j ist. Angenommen es gilt $y \notin M$, so ist y universal zu M, da $y \in P_2 = N(x)$ und $x \in M$. Da $P_j \cap M \neq \varnothing$, gibt es einen Knoten $u \in P_j \cap M$, sodass $yu \in E$. Dies widerspricht der Tatsache, dass y isoliert von P_j ist. Deswegen gilt $y \in M$ d.h. $P_2 \cap M \neq \varnothing$.

Der zweite Teil des Beweises erfolgt per Induktion und zeigt, dass $P_3, \ldots, P_{j-1} \subsetneq M$, falls es eine Klasse P_j mit $j > 3$ und $P_j \cap M \neq \varnothing$ gibt. Für den Induktionsanfang wird gezeigt, dass $P_3 \subsetneq M$: Angenommen es gibt einen Knoten $z \in P_3$ mit $z \notin M$ dann gilt, dass z isoliert von M ist, da $x \in M$ und $xz \notin E$. Da aber $yz \in E$ ($y \in P_2$ ist universal

zu P_3; siehe erster Teil), ergibt sich ein Widerspruch. Deswegen gilt $z \in M$ d.h. $P_3 \subsetneq M$. Für den Induktionsschritt wird angenommen, dass $P_3, \ldots, P_i \subsetneq M$ für ein beliebiges $i \in [3, j-2]$. Es wird gezeigt, dass auch $P_{i+1} \subsetneq M$ gilt: Wegen Lemma 1.2 (3.a) gilt für eine Klasse P_{i+1}, dass es eine Klasse P_l mit $l < i+1$ und einen Knoten $v \in P_l$ gibt, sodass v universal zu P_{i+1} und isoliert von P_j ist. Falls $l = 2$, so gilt – wie zu Beginn gezeigt – $v \in M$. Sofern $l > 2$, so gilt wegen der Induktionsannahme, dass $v \in M$. Für einen Knoten $w \in P_{i+1}$ sei nun angenommen, dass $w \notin M$. Weil $xw \notin E$ (da $w \notin P_2 = N(x)$) und $vw \in E$ (v ist universal zu P_{i+1}), ergibt sich ein Widerspruch, da $x, v \in M$. Entsprechend gilt, dass $w \in M$ und somit $P_{i+1} \subsetneq M$. Es folgt, dass $P_3, \ldots, P_{j-1} \subsetneq M$. □

Wie zu Beginn des Abschnitts erwähnt, bedarf es schließlich einer Umpositionierung des Pivotknoten x innerhalb der Anordnung $\sigma(\mathcal{T}')$. Nur dann kann eine faktorisierende Permutation σ erreicht werden. Ein starkes Modul M (mit Pivot) welches lediglich einen Teil der Knoten aus P_2 und dazu Knoten aus einer Klasse P_i mit $i > 2$ enthält, kann die Anforderung einer Sequenz in $\sigma(\mathcal{T}')$ sonst nicht erfüllen. Entsprechend ist der Pivotknoten x hinter den Knoten aus P_2 zu platzieren, also $\sigma = P_2, x, P_3, \ldots, P_k$. Gemäß der Berücksichtigung von Ko-Komponenten C'_i im Fall der Klasse P_2, bzw. der Betrachtung von Komponenten C_i einer Klasse P_j mit $j > 2$, eröffnet sich die folgende Perspektive: $\sigma = C'_a, \ldots, C'_1, x, C_1, \ldots, C_b$. Hierbei gilt, dass $\bigcup_{i \in [1,a]} C'_i = P_2$ sowie $\bigcup_{i \in [1,b]} C_i = \bigcup_{i \in [3,k]} P_i$. Außerdem gibt es für jede Klasse P_j, $j > 2$ ein Paar $s, t \in [1, b]$ mit $s \leq t$, sodass $\bigcup_{i \in [s,t]} C_i = P_j$.

Für ein starkes Modul M (mit Pivot), verbleibt einzig die Klärung der Reihenfolge der Komponenten innerhalb einer Klasse P_i mit $i > 2$ in σ, falls $P_i \perp M$. Gleiches gilt für die Reihenfolge der Ko-Komponenten der Klasse P_2, falls $P_2 \perp M$. Für den Fall $P_i \perp M, i > 2$ ist bezüglich σ gefordert, dass die Knoten in $P_i \cap M$ vor den Knoten in $P_i \setminus M$ erscheinen. Der Fall $P_2 \perp M$ erfordert das Gegenteil, d.h. $P_2 \setminus M$ vor $P_2 \cap M$. Die Anpassung der Reihenfolge der Kinder eines *dead*-gelabelten Knotens, in `TreeRefinement` (Zeile 15-20) sowie die entsprechenden Schritte im Fall eines *zombie*-gelabelten Knotens in `Factorize` (Zeile 8-11), erreichen genau diese Ziel.

Lemma 3.8. *[49] Sei $\mathcal{T} = [T_1, \ldots, T_k]$ eine, durch* `TreeRefinement` *produzierte, geordnete Baum-Partition der maximalen Slices eines Graphen $G = (V, E)$. Sei $\mathcal{P}_S = [P_1, \ldots, P_k]$, die $\mathcal{T}$ zugrundeliegende, Partition der maximalen Slices von G. Sei $\mathcal{T}' = [T'_1, \ldots, T'_{k'}]$ die durch* `Factorize`*$(\mathcal{T})$ berechnete, geordnete Baum-Partition von G. Sei $\sigma(\mathcal{T}')$ die Anordnung der Blätter von $\mathcal{T}'$. Sei M ein starkes Modul von G, welches den Pivotknoten $T_1 = x$ enthält. Es gelten die folgenden Aussagen:*

1. *Falls für eine Klasse $P_i \in \mathcal{P}_S$, $i > 2$ gilt, dass $P_i \perp M$, so gilt für die Anordnung $\sigma(\mathcal{T}')$, dass die Knoten in $P_i \cap M$ vor den Knoten in $P_i \setminus M$ angeordnet sind.*
2. *Falls für die Klasse $P_2 \in \mathcal{P}_S$ gilt, dass $P_2 \perp M$, so gilt für die Anordnung $\sigma(\mathcal{T}')$, dass die Knoten in $P_2 \cap M$ hinter den Knoten in $P_2 \setminus M$ angeordnet sind.*

Beweis. (1.) Für eine beliebige Komponente C in $G[P_i]$, $i > 2$ gilt entweder $C \subsetneq M$ oder $C \cap M = \varnothing$ (Lemma 3.6). Da die Komponenten von $G[P_i]$ eine Partition von P_i bilden und weil $P_i \perp M$, existiert mindestens eine Komponente $C \subsetneq P_i \cap M$ sowie mindestens eine Komponente $C' \subsetneq P_i \setminus M$. Um zu zeigen, dass die Knoten in $P_i \cap M$ vor den Knoten in $P_i \setminus M$ gelistet sind, genügt es zu zeigen, dass die Knoten in C vor den Knoten in C' in $\sigma(\mathcal{T}')$ erscheinen, da C, C' beliebig wählbar sind.

Man betrachte den Baum $T_i \in \mathcal{T}$, während seiner Verarbeitung durch `Factorize`, nach Beendigung der zweiten inneren Schleife (nach Zeile 11, vor Zeile 12). Dieser sei mit T_i^* bezeichnet: Zu diesem Zeitpunkt entspricht T_i^* – gemäß `TreeRefinement` sowie den in `Factorize` ggf. erfolgten Verfeinerungsschritten (Zeile 6-7) – einer Verfeinerung des modularen Zerlegungsbaums $T_{G[P_i]}$. Da die Komponenten von $G[P_i]$ durch Kinder der Wurzel von $T_{G[P_i]}$ repräsentiert sind, existieren zwei Knoten c, c' in T_i^*, sodass $L(T_i^*[c]) = C$ und $L(T_i^*[c']) = C'$ (vgl. Definition 17). Außerdem ist die Reihenfolge der Knoten in P_i hinsichtlich $\sigma(\mathcal{T}')$ durch den Baum T_i^* bereits fixiert. Zusammen mit Lemma 3.5 folgt hieraus, dass $\sigma(T_i^*)$ eine Teilsequenz von $\sigma(\mathcal{T}')$ bildet.

Um zu zeigen, dass die Knoten in C vor den Knoten in C' in $\sigma(\mathcal{T}')$ erscheinen, genügt es nun zu zeigen, dass für einen gemeinsamen Vorfahre u von c, c' und zwei Kinder p, q von u gilt, dass c ein Knoten in $T_i^*[p]$ und c' ein Knoten in $T_i^*[q]$ ist. Hierbei muss bezüglich der Kinder von u gelten, dass p vor q. O.B.d.A. sei u der jüngste Vorfahre von c, c' (d.h. die Menge $L(T_i^*[u])$ soll minimal sein und $C, C' \subsetneq L(T_i^*[u])$). Für den Knoten u kann genau eine, der folgenden Aussagen zutreffen: (i.) u hat kein Label; (ii.) u hat ein *dead*-Label; (iii.) u hat ein *zombie*-Label.

(i.) Angenommen u hat kein Label. Da u kein Label hat, hat auch kein Nachfolger von u ein Label (vgl. `Factorize`, Zeile 2-3). Gemäß Lemma 3.3 repräsentiert u, deswegen ein Modul M' (ohne Pivot) von G. Hierbei gilt, dass $C, C' \subsetneq M'$ (weil c, c' in $T_i^*[u]$). Mit $C \subsetneq M$ sowie $C' \cap M = \varnothing$ folgt $M \perp M'$. Dies widerspricht der Forderung, dass M ein starkes Modul in G ist. Entsprechend muss u ein Label haben, sodass dieser Fall ausgeschlossen werden kann.

(ii.) Angenommen u hat ein *dead*-Label. Da u ein *dead*-Label besitzt, hat u ein linkes Kind p und ein rechtes Kind q. Hierbei gilt, dass der Knoten q zum rechten Kind von u wurde, da ein Knoten $y \in P_j$ für ein $j > i$ existiert, sodass y universal zu $L(T_i^*[q])$

ist (vgl. `TreeRefinement`). Angenommen c ist ein Knoten in $T_i^*[q]$. Weil $C' \subsetneq P_i \setminus M$ folgt mit Lemma 3.7, dass $y \notin M$. Deswegen gilt wegen Lemma 3.9, dass y isoliert von $C \subsetneq P_i \cap M$ ist. Dies widerspricht der Annahme, dass c ein Knoten in $T_i^*[q]$ ist. Folglich ist c ein Knoten in $T_i^*[p]$ bzw. c' ein Knoten in $T_i^*[q]$, da u der jüngste Vorfahre von c, c' ist.

(iii.) Ähnlich zu (ii.). □

Zusammenfassend ist die konkrete faktorisierende Permutation der Knotenmenge V, bezüglich aller starken Module von G durch die Anordnung $\sigma(T'_1, \ldots, T'_l, x, T_1, \ldots, T_r)$ gegeben. Die Bäume $T'_1, \ldots, T'_l \in \mathcal{T}'$ sind hierbei ehemalige Teilbäume des Baums $T_2 \in \mathcal{T}$. Die Bäume $T_1, \ldots, T_r \in \mathcal{T}'$ entsprechen Bäumen, welche aus den Bäumen $T_3, \ldots, T_k \in \mathcal{T}$ hervorgegangenen sind. Der Baum $x = T_1 \in \mathcal{T}$ verbleibt unverändert und wird hinter den Knoten aus P_2 platziert.

Für das weitere algorithmische Vorgehen ist speziell die Darstellung der faktorisierenden Permutation mittels Komponenten/Ko-Komponenten wichtig. Daher sei diese in der folgenden Definition festgehalten.

Definition 19. *Sei $\mathcal{P}_S = [P_1, \ldots, P_k]$ eine Partition der maximalen Slices eines Graphen G. Eine Anordnung $\sigma_x = C'_a, \ldots, C'_1, x, C_1, \ldots, C_b$ wird eine **Pivot-faktorisierende Permutation** von G genannt, falls:*

1. *Für $i \in [1, a]$ ist C'_i eine Ko-Komponente von $G[P_2]$;*
2. *Für $i \in [1, b]$ ist C_i eine Komponente eines $G[P_j]$ mit $j > 2$;*
3. *Falls $2 < i < j$, so sind die Komponenten von $G[P_i]$ vor den Komponenten in $G[P_j]$ in σ_x;*
4. *Für jedes starkes Modul M, welches den Pivotknoten x enthält, gibt es eine Sequenz $C'_c, \ldots, x, C_1, \ldots, C_d$ in σ_x, sodass $M = \bigcup_{i \in [1,c]} C'_i \cup \{x\} \cup \bigcup_{i \in [1,d]} C_i$.*

3.4 Die „Spine“: Module mit Pivot

Wie in dem letzten Abschnitt beschrieben, kann die von `Factorize` produzierte, geordnete Baum-Partition leicht als eine Pivot-faktorisierende Permutation $\sigma_x = C'_a, \ldots, C'_1, x, C_1, \ldots, C_b$ interpretiert werden. Die hierzu technisch nötigen Schritte werden durch die Funktion `ConquerMDTree` realisiert und sind nicht Gegenstand des vorliegenden Abschnitts. Stattdessen wird vorausgesetzt, dass die Pivot-faktorisierenden Permutation σ_x als Eingabe, der in diesem Abschnitt betrachteten Funktion `BuildSpine` vorliegt.

In Abschnitt 3.3 wurde ebenfalls gezeigt, dass die starken Module von $G = (V, E)$, welche den Pivotknoten x enthalten, durch Sequenzen in σ_x dargestellt werden können. Das

Algorithmus 6 : `Factorize`($\mathcal{T}$)

Input : A maximal slice tree partition $\mathcal{T} = T_1, \ldots, T_k$ of some graph G, as produced by `TreeRefinement`, where $T_1 = x$ is the pivot, and $L_1, \ldots, L_k$ are the corresponding leaf sets.

Output : An ordered tree partition $\mathcal{T}'$ of G such that if M is a strong module, then $M \setminus \{x\}$ appears consecutively in $\sigma(\mathcal{T}')$. Moreover, the vertices in each co-component of $G[L_2]$, and each component of $G[L_i], i > 2$, also appear consecutivly.

for $i \in [\,1, k\,]$ **do**
  foreach node $u \in T_i$ labelled *dead* **do**
    label all of u's ancestors as *zombie* unless they are labelled *dead*;
  foreach node $u \in T_i$ labelled *zombie* **do**
    let A be the children of u that are labelled *dead* or *zombie*, and let B its other children;
    if $|B| > 1$ and u is degenerate **then**
      replace the children in B with a new marked node inheriting u's type, whose children are those nodes in B;
    if $i = 2$ **then**
      order the children of u so that those in A appear first;
    else
      order the children of u so that those in A appear last;
  initialize an ordered list of trees $\mathcal{T}_i$ with T_i;
  while there exists a tree T in $\mathcal{T}_i$ whose root is labelled *dead* or *zombie* **do**
    replace T in $\mathcal{T}_i$ with the subtrees rooted at the children of its root, in the order fixed above;
  replace T_i in $\mathcal{T}$ with $\mathcal{T}_i$
return $\mathcal{T}$

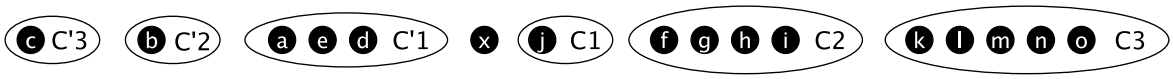


Abbildung 11: Die Pivot-faktorisierende Permutation $\sigma_x = C'_3, C'_2, C'_1, x, C_1, C_2, C_3$ von G.

Ermitteln dieser Sequenzen, d.h. das Bestimmen der zu einem starken Modul (mit Pivot) gehörenden Elemente aus σ_x, wird durch die Funktion `BuildSpine` erreicht. Gleichzeitig wird hierbei ein gewurzelter Baum T berechnet, welcher die Inklusionsordnung dieser Module modelliert und welcher als ein Grundgerüst – der „Spine“ – des modularen Zerlegungsbaums T_G fungiert.

Es folgt eine Einführung in die im folgenden Abschnitt verwendete Notation: Zur Veranschaulichung seien die starken Module (mit Pivot) mit $M_1, \ldots, M_k$ bezeichnet. O.B.d.A. gilt $\{x\} = M_1 \subsetneq \ldots \subsetneq M_k = V$. Gemäß Definition 19 ist ein Modul M_i durch eine Sequenz $C'_l, \ldots, C'_1, x, C_1, \ldots, C_r$ in σ_x durch $M_i = C'_l \cup \ldots \cup C'_1 \cup \{x\} \cup C_1 \cup \ldots \cup C_r$ bestimmt. Weil die zu den Modulen korrespondierenden Sequenzen ineinander verschachtelt sind, kann ein Modul M_{i+1} durch eine ***linke Erweiterung*** $C'_t, \ldots, C'_{l+1}$, $a \geq t \geq l$ oder eine ***rechte Erweiterung*** $C_{r+1}, \ldots, C_m$, $r \leq m \leq b$ des Moduls M_i entsprechend $M_{i+1} = C'_t \cup \ldots \cup C'_{l+1} \cup M_i \cup C_{r+1} \cup \ldots \cup C_m$ charakterisiert werden, falls $t > l$ oder $r < m$.

Grundsätzlich erfolgt die Ermittlung der Module iterativ, wobei mit dem innersten starken Modul M_1 begonnen und mit dem Modul M_k geendet wird. Hierbei wird der Baum T zunächst mit einem Repräsentanten für den Pivotknoten x – dem innersten Modul M_1 – initialisiert (Zeile 1). In den Folgeiteration wird jeweils versucht eine linke oder rechte Erweiterung eines starken Moduls M_i zu bilden, sodass hierdurch das starke Modul M_{i+1} definiert ist. Die Anforderungen die dabei an die Knoten einer Erweiterung, als Knoten des gesuchten Moduls M_{i+1} gestellt sind, ergeben sich indirekt aus der folgenden Aussage:

Lemma 3.9. *[49] Sei $\mathcal{P}_S = [P_1, \ldots, P_k]$ eine Partition der maximalen Slices eines Graphen G. Zusätzlich sei M ein Modul von G, welches den Pivotknoten $x \in P_1$ enthält. Es gilt:*

1. *Falls C' eine Ko-Komponente in $G[P_2]$ bezeichnet, so gilt für einen beliebigen Knoten $y \in C' \setminus M$, dass y universal zu M ist.*
2. *Falls C eine Komponente in $G[P_i]$ mit $i > 2$ bezeichnet, so gilt für einen beliebigen Knoten $y \in C \setminus M$, dass y isoliert von M ist.*

Beweis. (1.) Angenommen ein Knoten $y \in C' \setminus M$ ist nicht universal zu M. Da $y \notin M$ ist y isoliert von M. Weil aber $y \in P_2 = N(x)$ und $x \in M$ ergibt sich ein Widerspruch. Daher gilt, dass y universal zu M ist. (2.) Analog zu (1.). □

Gleichbedeutend muss für die Knoten der Erweiterungen $C'_t, \ldots, C'_{l+1}$ bzw. $C_{r+1}, \ldots, C_m$ gelten, dass diese universal zu allen Ko-Komponenten C'_i, $i > t$ sowie isoliert von allen Komponenten C_i, $i > m$ sind. Hieraus ergibt sich die Notwendigkeit die Erweiterungen ausreichend groß zu wählen, um diese Eigenschaft zu gewährleisten. Für einen Knoten einer linken Erweiterung muss dabei lediglich die Nachbarschaft zu den Komponenten C_i, $i > m$ überprüft werden, da sie ohnehin universal zu allen Ko-Komponenten C'_i, $i > t$ sind. Für einen Knoten einer rechten Erweiterung ist die Nachbarschaft zu den Ko-Komponenten C'_i, $i > t$ sowie die Nachbarschaft zu den Komponenten C_i, $i > m$ zu überprüfen. Hinsichtlich der technischen Ermittelung einer Erweiterung werden die durch die folgenden Definitionen bestimmten μ- und ρ-Werte herangezogen. Diese werden im Vorfeld durch die Funktion `ConquerMDTree` für alle Knoten in σ_x sowie für alle $C'_i, C_i \in \sigma_x$ berechnet.

Definition 20. *Sei $\sigma_x = C'_a, \ldots, C'_1, x, C_1, \ldots, C_b$ eine Pivot-faktorisierende Permutation eines Graphen $G = (V, E)$ und $\mathcal{P}_S = [P_1, \ldots, P_k]$ die zugrundeliegende Partition der maximalen Slices.*

- *Für $y \in P_2$ ist $\mu(y) \coloneqq \min\{j \in [0, b] \mid \forall l \in [j+1, b], \forall z \in C_l : yz \notin E\}$;*
- *Für $y \in P_i$, $i > 2$ ist $\mu(y) \coloneqq \min\{j \in [0, a] \mid \forall l \in [j+1, a], \forall z \in C'_l : yz \in E\}$;*
- *Für $C'_i \in \sigma_x$ ist $\mu(C'_i) \coloneqq \max\{\mu(y) \mid y \in C'_i\}$;*
- *Für $C_i \in \sigma_x$ ist $\mu(C_i) \coloneqq \max\{\mu(y) \mid y \in C_i\}$.*

Definition 21. *Sei $\sigma_x = C'_a, \ldots, C'_1, x, C_1, \ldots, C_b$ eine Pivot-faktorisierende Permutation eines Graphen $G = (V, E)$ und $\mathcal{P}_S = [P_1, \ldots, P_k]$ die zugrundeliegende Partition der maximalen Slices.*

- *Für $y \in C_i$ ist $\rho(y) \coloneqq \max(\{j \in [i+1, b] \mid \exists z \in C_j : yz \in E\} \cup \{0\})$;*
- *Für $C_i \in \sigma_x$ ist $\rho(C_i) \coloneqq \max\{\rho(y) \mid y \in C_i\}$.*[25]

Praktisch wird zuerst versucht, das Modul M_{i+1} durch eine maximale linke Erweiterung $C'_t, \ldots, C'_{l+1}$ von $M_i = C'_l \cup \ldots \cup C'_1 \cup \{x\} \cup C_1 \cup \ldots \cup C_r$ zu bilden, sodass $\mu(C'_i) = r$ für alle $C'_i, i \in [l+1, t]$ (Zeile 4-9). Existiert eine solche Erweiterung, so ist $M_{i+1} = C'_t \cup \ldots \cup C'_{l+1} \cup M_i$ das kleinste starke Modul von G, welches M_i enthält. Falls keine derartige

[25] Die Definition weicht inhaltlich von der entsprechenden Definition in [49] ab.

	μ		μ	ρ
C'_1	0	C_1	1	0
C'_2	2	C_2	1	3
C'_3	2	C_3	3	0

Tabelle 3: Die μ- und ρ-Werte der $C'_i, C_i \in \sigma_x$. Die Pivot-faktorisierende Permutation σ_x findet sich in Abbildung 11.

Erweiterung ermittelt werden kann, wird versucht eine maximale rechte Erweiterung $C_{r+1}, \ldots, C_m$ zu bestimmen, sodass $\mu(C_i) = l$ und $\rho(C_i) = 0$ für alle $C_i, i \in [r+1, m]$ (Zeile 10-15). Sofern diese existiert, bildet $M_{i+1} = M_i \cup C_{r+1} \cup \ldots \cup C_m$ das kleinste starke Modul von G, welches M_i enthält. In beiden Fällen ergibt sich für den modularen Zerlegungsbaum T_G, dass der Repräsentant q_{M_i} ein Kind des Repräsentanten $q_{M_{i+1}}$ ist. Das folgende Lemma zeigt, dass $q_{M_{i+1}}$ im Fall einer linken Erweiterung einen seriellen, im Fall der rechten Erweiterung einen parallelen Knoten bildet.

Lemma 3.10. *[49] Sei $\sigma_x = C'_a, \ldots, C'_1, x, C_1, \ldots, C_b$ eine Pivot-faktorisierende Permutation eines Graphen G. Seien M, M' mit $M' \subsetneq M$ zwei starke Module von G, welche den Pivotknoten x enthalten und seien $q_M, q_{M'}$ deren Repräsentanten in dem modularen Zerlegungsbaum T_G, wobei $q_{M'}$ ein Kind von q_M ist. Es gilt:*

1. *q_M ist ein serieller Knoten genau dann, wenn kein $C_i \in \sigma_x$ mit $C_i \cap (M \setminus M') \neq \varnothing$ existiert;*
2. *q_M ist ein paralleler Knoten genau dann, wenn kein $C'_i \in \sigma_x$ mit $C'_i \cap (M \setminus M') \neq \varnothing$ existiert und es kein $C_i \in \sigma_x$ mit $C_i \cap (M \setminus M') \neq \varnothing$ und $\rho(C_i) > 0$ gibt.*

Beweis. (1.) Vorausgesetzt q_M ist ein serieller Knoten und angenommen es gibt ein $C_i \in \sigma_x$ mit $C_i \cap (M \setminus M') \neq \varnothing$. Da $M \setminus M'$ ein Modul in G ist (Theorem 2.5) und $C_i \cap (M \setminus M') \neq \varnothing$ folgt mit Lemma 3.6, dass $C_i \subseteq M \setminus M'$. Weil $x \in M'$ und $N(x) \cap C_i = \varnothing$, gilt für jedes Paar $a \in M'$, $b \in C_i$, dass $ab \notin E$. Da q_M ein serieller Knoten ist, kann letzteres nur gelten, falls $C_i \cup M'$ in einer Ko-Komponente von $G[M]$ enthalten ist. Da $q_{M'}$ ein Kind von q_M ist, bildet M' bereits eine Ko-Komponente von $G[M]$. Mit $C_i \cup M' \neq M'$ ergibt sich ein Widerspruch. Entsprechend gibt es kein $C_i \in \sigma_x$ mit $C_i \cap (M \setminus M') \neq \varnothing$.

Für die Rückrichtung sei angenommen, dass kein $C_i \in \sigma_x$ mit $C_i \cap (M \setminus M') \neq \varnothing$ existiert. Wegen Lemma 3.6 zusammen mit Definition 19 gilt dann, dass $M = M' \cup C'_i \cup \ldots \cup C'_j$ für ein Paar i, j. Für jedes Paar $a \in M'$, $b \in C'_l$ für ein $l \in [i, j]$ gilt, dass $ab \in E$ (Lemma 3.9). Daher ist $\overline{G[M]}$ nicht zusammenhängend, weswegen q_M ein serieller Knoten ist.

(2.) Analog zur Hinrichtung des Beweises von (1.), kann gezeigt werden, dass kein $C'_i \in \sigma_x$ mit $C'_i \cap (M \setminus M') \neq \varnothing$ existiert, falls q_M ein paralleler Knoten ist.

Es wird nun gezeigt, dass es kein $C_i \in \sigma_x$ mit $C_i \cap (M \setminus M') \neq \varnothing$ und $\rho(C_i) > 0$ gibt, falls q_M ein paralleler Knoten ist: Vorausgesetzt q_M ist ein paralleler Knoten und angenommen es existiert ein $C_i \in \sigma_x$ mit $C_i \cap (M \setminus M') \neq \varnothing$ und $\rho(C_i) > 0$. O.B.d.A. sei C_i so gewählt, dass i minimal ist und es sei $\rho(C_i) = j$. Wegen Lemma 3.6 gilt $C_i \subsetneq M$. Zudem gibt es einen Knoten $y \in C_i$ sowie einen Knoten $z \in C_j$ mit $yz \in E$, da $\rho(C_i) = j$. Zusammen mit $x, y \in M$ und $xz \notin E$ folgt $z \in M$ bzw. mit Lemma 3.6, dass $C_j \subsetneq M$. Es sei $\mathcal{P}_S = [P_1, \ldots, P_k]$ die σ_x zugrundeliegende, Partition der maximalen Slices. Da C_i, C_j jeweils Komponenten in einem $G[P_i]$ mit $P_i \in \mathcal{P}_S$ induzieren und weil für das Paar $y \in C_i$, $z \in C_j$ gilt, dass $yz \in E$, gilt $C_i \subseteq P_l$ sowie $C_j \subseteq P_r$ für $P_l, P_r \in \mathcal{P}_S$ mit $l < r$. Gemäß Lemma 1.2 gibt es eine Klasse $P_{l'} \in \mathcal{P}_S$ mit $l' < l$ und einen Knoten $y' \in P_{l'}$, wobei y' universal zu P_l und isoliert von P_r ist. Mit $C_i, C_j \subsetneq M$ ergibt sich, dass $y' \in M$. Weil y' universal zu C_i ist und weil für alle Knoten in C_i gilt, dass diese isoliert von M' sind (Lemma 3.9) folgt außerdem, dass $y' \notin M'$. Entsprechend gilt $y' \in M \setminus M'$. Da $y' \notin C_i$ und $y' \notin C_j$ gibt es entweder ein $C_k \subsetneq M \setminus M'$ mit $k < i$, sodass $y' \in C_k$ oder ein $C'_{k'} \subsetneq M \setminus M'$ mit $y' \in C'_{k'}$. Falls $y' \in C_k$ so ist $\rho(C_k) > 0$, da y' universal zu C_i ist. Dies widerspricht der Wahl von C_i, sodass i minimal ist. Im Fall $y' \in C'_{k'}$ gilt $C'_{k'} \cap M \setminus M' \neq \varnothing$. Wie zuvor gezeigt, führt dies ebenfalls zu einem Widerspruch. Somit folgt, dass es kein $C_i \in \sigma_x$ mit $C_i \cap (M \setminus M') \neq \varnothing$ und $\rho(C_i) > 0$ gibt, falls q_M ein paralleler Knoten ist.

Für die Rückrichtung sei angenommen, dass kein $C'_i \in \sigma_x$ mit $C'_i \cap (M \setminus M') \neq \varnothing$ existiert und es kein $C_i \in \sigma_x$ mit $C_i \cap (M \setminus M') \neq \varnothing$ und $\rho(C_i) > 0$ gibt. Weil M ein starkes Modul mit Pivot x ist und weil es kein C'_i mit $C'_i \cap (M \setminus M') \neq \varnothing$ gibt, folgt wegen Lemma 3.6 zusammen mit Definition 19, dass $M = M' \cup C_i \cup \ldots \cup C_j$ für ein Paar i, j. Hierbei gilt für jedes Paar $a \in M'$, $b \in C_l$ mit $l \in [i, j]$ beliebig, dass $ab \notin E$ (Lemma 3.9). Weil es kein C_i mit $C_i \cap (M \setminus M') \neq \varnothing$ und $\rho(C_i) > 0$ gibt, gilt außerdem für jedes Paar $a' \in C_c$, $b' \in C_d$ mit $c, d \in [i, j]$, $c \neq d$, dass $a'b' \notin E$. Folglich ist $G[M]$ nicht zusammenhängend, weswegen q_M ein paralleler Knoten ist. □

Falls M_{i+1} weder durch eine linke, noch durch eine rechte Erweiterung ermittelt werden kann, so erfolgt schließlich die iterative Berechnung einer minimalen linken Erweiterung $C'_t, \ldots, C'_{l+1}$ sowie einer minimalen rechten Erweiterung $C_{r+1}, \ldots, C_m$ des Moduls $M_i = C'_l \cup \ldots \cup C'_1 \cup \{x\} \cup C_1 \cup \ldots \cup C_r$ (Zeile 16-33).

Dieser Fall gestaltet sich als der technisch aufwendigste, wobei für die Grenzen der minimalen Erweiterungen sichergestellt wird, dass $r + 1 \leq m = \max(\{\mu(C'_i) \mid i \in [l + 1, t]\} \cup \{\rho(C_i) \mid i \in [r + 1, m]\})$ und $l + 1 \leq t = \max(\{\mu(C_i) \mid i \in [r + 1, m]\})$.

Hierdurch ist das kleinste starke Modul M_{i+1}, welches das Modul M_i enthält, durch $M_{i+1} = C'_t \cup \ldots \cup C'_{l+1} \cup M_i \cup C_{r+1} \cup \ldots \cup C_m$ bestimmt[26]. Wiederrum ergibt sich für den modularen Zerlegungsbaum T_G, dass der Repräsentant q_{M_i} ein Kind des Repräsentanten $q_{M_{i+1}}$ ist. Die folgende Aussage qualifiziert den Knoten $q_{M_{i+1}}$ als einen primen Knoten.

Lemma 3.11. *[49] Sei $\sigma_x = C'_a, \ldots, C'_1, x, C_1, \ldots, C_b$ eine Pivot-faktorisierende Permutation eines Graphen G. Seien M, M' mit $M' \subsetneq M$ zwei starke Module von G, welche den Pivotknoten x enthalten und seien $q_M, q_{M'}$ deren Repräsentanten in dem modularen Zerlegungsbaum T_G, wobei $q_{M'}$ ein Kind von q_M ist. Zudem bezeichne l den größten Index mit $C'_l \cap M' \neq \varnothing$ und bezeichne r den größten Index, sodass $C_r \cap M' \neq \varnothing$. Falls q_M ein primer Knoten ist, so gilt:*

1. *$l \neq a$ und $r \neq b$;*
2. *$C'_{l+1} \cap M \neq \varnothing$ und $C_{r+1} \cap M \neq \varnothing$;*
3. *$\mu(C'_{l+1}) > r$;*
4. *$\mu(C_{r+1}) > l$ oder $\rho(C_{r+1}) > 0$.*

Beweis. Es gelten jeweils die Voraussetzungen gemäß dem Lemma und es sei q_M ein primer Knoten.

(1.) Falls $l \neq a$ und $r = b$, so ist q_M ein serieller Knoten (Lemma 3.10). Sofern $l = a$ und $r \neq b$, so ist q_M ein paralleler Knoten (Lemma 3.10.). Falls $l = a$ und $r = b$, so gilt $M' \not\subseteq M$. Da diese Fälle jeweils zu einem Widerspruch führen, gilt $l \neq a$ und $r \neq b$, wenn q_M ein primer Knoten ist.

(2.) Angenommen es gilt $C'_{l+1} \cap M = \varnothing$ und $C_{r+1} \cap M \neq \varnothing$. Da $C'_{l+1} \cap M = \varnothing$ und σ_x eine Pivot-faktorisierende Permutation ist, gilt für alle $C'_j \in \sigma_x$ mit $j \geq l + 1$, dass $C'_j \cap M = \varnothing$. Entsprechend gilt $M = M' \cup C_{r+1} \cup \ldots \cup C_{r+k}$ für ein $k \leq b - r$. Weil für alle $i \in [1, k]$ gilt, dass $C_{r+i} \cap M' = \varnothing$, folgt mit Lemma 3.9, dass $G[M]$ nicht zusammenhängend ist. Folglich muss q_M ein paralleler Knoten sein, was der Voraussetzung widerspricht. Analog hierzu kann die Annahme, dass $C'_{l+1} \cap M \neq \varnothing$ und $C_{r+1} \cap M = \varnothing$ gilt, zu einem Widerspruch geführt werden. Aus der Annahme, dass $C'_{l+1} \cap M = \varnothing$ und $C_{r+1} \cap M = \varnothing$ gilt, folgt der Widerspruch $M' \not\subseteq M$. Entsprechend gilt $C'_{l+1} \cap M \neq \varnothing$ und $C_{r+1} \cap M \neq \varnothing$, wenn q_M ein primer Knoten ist.

(3.) Angenommen es gilt $\mu(C'_{l+1}) \leq r$. $M'' = M' \cup C'_{l+1}$ bildet dann ein Modul von G. Aus $C'_{l+1} \cap M \neq \varnothing$ bzw. $C_{r+1} \cap M \neq \varnothing$ folgt mit Lemma 3.6, dass $C'_{l+1} \subseteq M$ sowie $C_{r+1} \subseteq M$. Da zudem $M' \subsetneq M$ und $C_{r+1} \cap M'' = \varnothing$ gilt, folgt $M' \subsetneq M'' \subsetneq M$. Weil aber

[26] Die Berechnung eines primen Moduls M_{i+1}, d.h. der Erweiterungen von M_i, ist in der Fassung des Algorithmus in [49] anders realisiert. Hierzu sei auf die Abbruchbedingung der While-Schleife in [49] verwiesen, welche lediglich die μ- und ρ-Werte der äußeren Grenzen berücksichtigt.

$q_{M'}$ ein Kind von q_M ist, kann das Modul M'' nur durch eine Vereinigung von Kindern von q_M repräsentiert sein. Dies impliziert den Widerspruch, dass q_M ein degenerierter Knoten ist. Es gilt daher, dass $\mu(C'_{l+1}) > r$, falls q_M ein primer Knoten ist.

(4.) Angenommen es gilt $\mu(C_{r+1}) \leq l$ und $\rho(C_{r+1}) = 0$, so ergibt sich - analog zu (3.) - der Widerspruch, dass q_M ein degenerierter Knoten ist. Entsprechend gilt, dass $\mu(C_{r+1}) > l$ oder $\rho(C_{r+1}) > 0$, falls q_M ein primer Knoten ist.

□

Sobald M_{i+1} identifiziert ist, wird der Baum T um einen entsprechenden Repräsentanten $q_{M_{i+1}}$ sowie um Repräsentanten für die Komponenten/Ko-Komponenten $C_i, C'_i \in M_{i+1} \setminus M_i$ erweitert. Letztere werden dabei zusammen mit dem zuvor erzeugten Repräsentant q_{M_i} zu Kindern von $q_{M_{i+1}}$. Der Typ des Moduls M_{i+1} ist gemäß Lemma 3.10 bzw. Lemma 3.11 durch die Art der Erweiterung bestimmt, sodass der Knoten $q_{M_{i+1}}$ mit einem entsprechenden Label ausgezeichnet wird (Zeile 34-41).

Mit Beendigung der Funktion `BuildSpine` sind die den Pivotknoten x enthaltenden, starken Module $M_1, \ldots, M_k$ von $G = (V, E)$ berechnet und durch die gelabelte Knoten $q_{M_1}, \ldots, q_{M_k}$ in dem Baum T repräsentiert. Diese bilden einen Pfad $q_{M_1}, \ldots, q_{M_k}$ in T, wobei $q_{M_k} = V$ die Wurzel und $q_{M_1} = x$ ein Blatt von T ist. Dies modelliert die Inklusionsordnung der Module $M_1, \ldots, M_k$. Außerdem sind durch die Labels der Knoten alle weiteren Module (mit Pivot) bestimmt. Die Repräsentanten der Komponenten/Ko-Komponenten $C_i, C'_i \in \sigma_x$ bilden zusammen mit q_{M_1} die Blätter von T. Hierdurch sind alle Knoten in V ebenfalls repräsentiert.

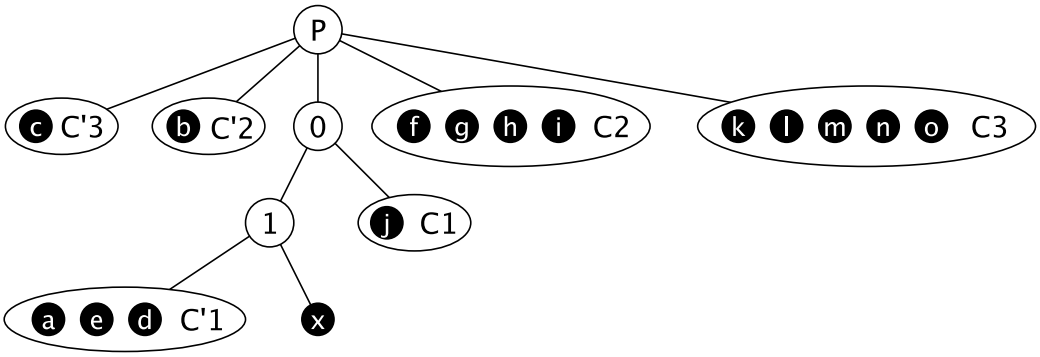


Abbildung 12: Der durch `BuildSpine` berechnete Baum T.

Obwohl die Module von G welche den Pivotknoten x nicht enthalten durch T nicht vollständig dargestellt sind, bildet der Baum T – gemäß seiner Konstruktion – ein Grundgerüst des modularen Zerlegungsbaums T_G. Nun bedarf es lediglich dem Ersetzen der

Repräsentanten der $C_i, C_i' \in \sigma_x$ in T durch die zuvor berechneten modularen Zerlegungsbäume der starken Module ohne Pivot, um den modularen Zerlegungsbaums T_G vollständig zu bestimmen.

3.5 Die Conquer-Phase

Wie schon in der Einleitung zu Kapitel 3 dargestellt, folgt der vorliegenden Algorithmus zur Bestimmung des modularen Zerlegungsbaums T_G eines Graphen G grundsätzlich dem Divide-and-Conquer-Prinzip. Hierbei bildet die lexikographische Breitensuche aufgrund ihrer Eigenschaften, einen fundamentalen Bestandteil des gesamten Algorithmus. Die Teilprobleme bei der Berechnung von T_G sind durch die Elemente einer Partition der maximalen Slices bezüglich einer LBFS-Anordnung von G definiert. Letztlich kann die Divide-Phase des Algorithmus als eine dahingehend optimierte, rekursive Variante einer LBFS beschrieben werden, welche die Slices einer LBFS in G als Partitionen maximaler Slices zusammenfasst und so die für ein (Teil-)Problem zu lösenden Teilprobleme definiert. Ein derartiges Vorgehen wird dabei durch die rekursive Eigenschaft einer LBFS gemäß Lemma 1.3 legitimiert (vgl. Abschnitt 3.1). Während einer Conquer-Phase wird aus einer maximale Baum-Partition $\mathcal{T} = [T_1, \ldots, T_k]$ eines Graphen G der modularen Zerlegungsbaum T_G berechnet. Hierbei sind die Eigenschaften der darunterliegenden Partition der maximalen Slices entsprechend Lemma 1.2 essentiell. Gegenstand dieses Abschnitts ist die Conquer-Phase des Algorithmus, welche durch die Funktion `ConquerMDTree` beschrieben ist. Der Aufruf der Funktion `ConquerMDTree` erfolgt innerhalb der Funktion `DivideMDTree`, sobald dort eine maximale Baum-Partition $\mathcal{T} = [T_1, \ldots, T_k]$ eines Graphen G berechnet wurde. $\mathcal{T}$ wird dabei als Argument an `ConquerMDTree` übergeben, wodurch die Berechnung des modulare Zerlegungsbaums T_G ausgelöst wird. Zu diesem Zeitpunkt sind die aktiven Listen, der in den $T_i \in \mathcal{T}$ enthaltenen Knoten, durch die Funktion `DivideMDTree` berechnet.

Das grundsätzliche Vorgehen während einer Conquer-Phase besteht darin, zunächst die starken Module von G zu identifizieren, welche den Pivotknoten $T_1 = x$ nicht enthalten. Diese Berechnungen werden durch die Funktion `TreeRefinement` realisiert und sind im Detail in Abschnitt 3.2 beschrieben. Im Anschluss werden die starken Module, welche den Pivotknoten x enthalten, ermittelt. Die hierfür wesentlichen Schritte erfolgen in den Funktionen `Factorize` und `BuildSpine`. Hierzu findet sich eine detaillierte Betrachtung in den Abschnitten 3.3 bzw. 3.4. Sobald alle starken Module berechnet sind, werden die Ergebnisse zu dem modularen Zerlegungsbaum T_G kombiniert. Zusätzlich anfallende Berechnungen, wie etwa die Berechnung der μ- und ρ-Werte werden ebenfalls durch

Algorithmus 7 : `BuildSpine`(σ_x)

Input : A pivot factorizing permutation $\sigma_x = C'_a, \ldots, C'_1, x, C_1, \ldots, C_b$ for which $\mu(C'_i)$ has been computed for each C'_i, and for which $\mu(C_i)$ and $\rho(C_i)$ have been computed for each C_i.

Output : A rooted tree T whose nodes correspond to the strong modules containing x, each one properly labelled by the modules's type, where the leaves of T are the elements of σ_x, and each node and leaf is descendent from all strong modules containing it.

```
1  T ← x;
2  l, r ← 0;
3  while l ≠ a or r ≠ b do
4      M ← ∅;
       // Locating series modules
5      l ← l + 1;
6      while μ(C'_l = r) and l ≤ a do
7          M ← M ∪ C'_l;
8          l ← l + 1;
9      l ← l − 1;
       // Locating parallel modules
10     if M = ∅ then
11         r ← r + 1;
12         while μ(C_r) = l and ρ(C_r) = 0 and r ≤ b do
13             M ← M ∪ C_r;
14             r ← r + 1;
15         r ← r − 1;
       // Locating prime modules
16     if M = ∅ then
17         l ← l + 1;
18         r ← r + 1;
19         l' ← l;
20         r' ← r;
21         t ← max({ μ(C_r) } ∪ { l });
22         m ← max({ μ(C'_l) } ∪ { ρ(C_r) } ∪ { r });
23         do
24             t' ← t;
25             m' ← m;
26             t ← max({ μ(C_i) | i ∈ [r', m] } ∪ { t });
27             m ← max({ μ(C'_i) | i ∈ [l', t] } ∪ { ρ(C_i) | i ∈ [r', m] } ∪ { m });
28             l' ← t';
29             r' ← m';
30         while t' ≠ t or m' ≠ m;
31         M ← M ∪ C'_l ∪ ... ∪ C'_t ∪ C_r ∪ ... ∪ C_m;
32         l ← t;
33         r ← m;
34     create a new node u with the elements of M as its children;
35     if there is no C_i ∈ M then
36         label u as series;
37     if there is no C'_i ∈ M then
38         label u as parallel;
39     else
40         label u as prime;
41     update T by making u the parent of T's root;
42 return T;
```

`ConquerMDTree` realisiert.

Es folgt eine detaillierte Betrachtung der Funktion `ConquerMDTree`. Die $\mathcal{T}$ zugrundeliegende Partition der maximalen Slices bezüglich einer LBFS-Anordnung von G ist mit $\mathcal{P}_S = [\{x\}, P_1, \ldots, P_k]$ bezeichnet. Direkt zu Beginn wird geprüft, ob der Graph G zusammenhängend ist (Zeile 2). Hierzu genügt es ein Blatt des Baums $T_k \in \mathcal{T}$ zu untersuchen. Gemäß Definition15 entspricht T_k dem modularen Zerlegungsbaum des Graphen $G[P_k]$, wobei P_k den letzten maximalen Slice in $\mathcal{P}_S$ darstellt. Falls G nicht zusammenhängend ist, so ist es mit Definition 3 leicht ersichtlich, dass die Knoten $C = P_1 \cup \ldots \cup P_{k-1}$ eine Komponente in G induzieren, während die restlichen Komponenten von G durch Teilmengen von P_k induziert werden. Wegen Lemma 1.2 gilt für paarweise verschiedene Knoten $y, y' \in P_k$, dass $\alpha(y) = \alpha(y')$. Daher genügt es die aktive Liste eines Knotens $y \in P_k$ zu überprüfen. Gilt für alle Knoten $u \in P_i$ für alle $i < k$, dass $u \notin \alpha(y)$, so ist G nicht zusammenhängend. Gibt es eine von P_k ausgehende aktive Kante, dann ist G zusammenhängend. In Abhängigkeit hiervon, ergibt sich ein unterschiedliches Vorgehen: Sofern G zusammenhängend ist, wird T_G direkt berechnet, wobei $\mathcal{T}$ die Grundlage der nachstehenden Konstruktionsschritte bildet. Ist G nicht zusammenhängend, so wird zuerst der modularen Zerlegungsbaum der Komponente $G[C]$ berechnet. In diesem Fall bildet die Baum-Partition $[T_1, \ldots, T_{k-1}]$ die nötige Berechnungsgrundlage. Schließlich wird der so berechnete modulare Zerlegungsbaum $T_{G[C]}$ mit dem Zerlegungsbaum T_k unter einer *parallel*-gelabelten Wurzel w vereinigt. Sind mehrere Komponenten von G durch T_k repräsentiert, d.h. falls die Wurzel w' von T_k ein *parallel*-gelabelter Knoten ist, so werden die Kinder von w' zusammen mit der Wurzel von $T_{G[C]}$, zu Kindern von w. Ansonsten werden die Wurzeln von T_k bzw. $T_{G[C]}$ zu den Kindern von w (Zeile 32-36). Nach der Bestimmung des Zusammenhangs von G erfolgt ein Vorverarbeitungsschritt hinsichtlich der Erzeugung der Pivot-faktorisierenden Permutation σ_x von G, welche das Argument des noch folgenden Aufrufs der Funktion `BuildSpine` darstellt. Gemäß Definition 19 entsprechen die Elemente einer Pivot-faktorisierenden Permutation von G, neben dem Pivotknoten x, den Ko-Komponenten in $G[P_2]$ sowie den Komponenten in den Graphen $G[P_i]$, für alle $i > 2$. Diesbezüglich werden die Blätter der $T_i \in \mathcal{T}$ mit entsprechenden Labels ausgezeichnet, wozu jeweils die Wurzel oder die Kinder der Wurzel der $T_i \in \mathcal{T}$ herangezogen werden können (Zeile 6-10). Im Anschluss erfolgt mit dem Aufruf der Funktion `TreeRefinement` die Bestimmung der starken Module von G, welche den Pivotknoten nicht enthalten (Zeile 11). Hierbei wird anhand der aktiven Listen der Blätter der $T_i \in \mathcal{T}$ eine Verfeinerung $\mathcal{T}'$ von $\mathcal{T}$ berechnet. Zusätzliche *dead*-Labels der inneren Knoten der $T_i' \in \mathcal{T}'$ zeigen an, welche Knoten keine starken Module von G (ohne Pivot) repräsentieren (Lemma 3.2), während die starken Module von G (ohne Pivot) durch

die *dead*-label freien Teilbäume der $T'_i \in \mathcal{T}'$ beschrieben sind (Lemma 3.4). Außerdem wird die Reihenfolge der Kinder der *dead*-gelabelten Knoten im Hinblick auf die Pivot-faktorisierenden Permutation angepasst. Als Nächstes wird die berechnete Verfeinerung $\mathcal{T}'$ der Funktion `Factorize` übergeben (Zeile 12). Nun werden die $T'_i \in \mathcal{T}'$ jeweils durch Sequenzen, der in ihnen enthaltenen *dead*-Label freien Teilbäume, ersetzt. Entsprechend enthält die hierdurch definierte Baum-Partition $\mathcal{T}''$, ausschließlich die zu den starken Modulen von G (ohne Pivot) korrespondierenden Teilbäume des zu konstruierenden, modularen Zerlgungsbaumes T_G. Des weiteren sind die Ersetzungen so gestaltet, dass eine Anordnung der Blätter gemäß einer in-order Traversierung der $T''_i \in \mathcal{T}''$ – unter Vernachlässigung des Pivotknoten x – eine faktorisierende Permutation von G definiert. Die zuvor vergebenen Lables (Zeile 6-10) erlauben es, die Blätter der $T''_i \in \mathcal{T}''$ entsprechend den Ko-Komponenten in $G[P_2]$ sowie den Komponenten in den Graphen $G[P_i]$, für $i > 2$ zu gruppieren, hierbei durch je einen Knoten zu repräsentieren und den Pivotknoten x hinter den Knoten aus P_2 zu platzieren. Dies definiert die Pivot-faktorisierende Permutation $\sigma_x = C'_a, \ldots, C'_1, x, C_1, \ldots, C_b$ bzw. das Argument der Funktion `BuildSpine`. Bevor die starken Module von G (mit Pivot) berechnet werden, erfolgt die Bestimmung der hierzu nötigen Werte $\mu(C'_i)$, $\mu(C_i)$ und $\rho(C_i)$ für alle $C'_i, C_i \in \sigma_x$ (Zeilen 14-29). Durch den Aufruf der Funktion `BuildSpine` wird dann ein gewurzelten Baum T berechnet, welcher die starken Module von G (mit Pivot) darstellt. Die Blätter von T sind der Pivotknoten x und die Repräsentanten $C'_i, C_i \in \sigma_x$. Im Anschluss werden letztere, durch die entsprechenden Bäume in $\mathcal{T}''$ ersetzt (Zeile 31).v Konstruktionsbedingt besteht die Möglichkeit, dass der Baum T zu diesem Zeitpunkt degenerierte Knoten mit degenerierten Vätern des gleichen Typs (*seriell/parallel*) enthält. Bezeichnet u einen solchen Knoten und v dessen Vater, so wird u aus T entfernt, wobei die Kinder von u zu den Kindern von v werden (Zeile 37-38). Der Baum T entspricht nun dem modularen Zerlegungsbaum T_G und wird zurückgegeben (Zeile 39).

Algorithmus 8 : `ConquerMDTree`($\mathcal{T}$)

Input : A maximal slice tree partition $\mathcal{T} = T_1, \ldots, T_k$ of some graph G, such that if $L_1, \ldots, L_k$ are the corresponding leaf sets, then T_i is the MD tree for $G[L_i]$. Moreover, each leaf $x \in L_i$ has an associated set $\alpha(x)$ consisting of its neighbours amongst the leaves of the T_j's, $j < i$.

Output : The MD tree for G.

let x be the pivot of $\mathcal{T}$;
if there is no $y \in L_k$ such that $|\alpha(y)| > 0$ **then**
    $k' \leftarrow k - 1$;
else
    $k' \leftarrow k$;
for $i \in [\,1, k'\,]$ **do**
    if $i = 2$ **then**
        label the leaves in L_i by their co-components in $G[L_i]$;
    else
        label the leaves in L_i by their components in $G[L_i]$;
$\mathcal{T}' \leftarrow$ `TreeRefinement`$(T_1, \ldots, T_{k'})$;
$\mathcal{T}'' \leftarrow$ `Factorize`$(\mathcal{T}')$;
let $\sigma_x = C'_a, \ldots, C'_1, x, C_1, \ldots, C_b$ be the pivot factorizing permutation defined by $\mathcal{T}''$;
`// Compute` $\mu(y)$ `for each` $y \in L_i$
for $i \in [\,2, k'\,]$ **do**
    for $y \in L_i$ **do**
        if $i = 2$ **then**
            $\mu(y) \leftarrow \min(\{\, j \mid$ every $z \in C_l$, $l > j$, is non-adjacent to $y \,\})$;
        else
            $\mu(y) \leftarrow \min(\{\, j \mid$ every $z \in C'_l$, $l > j$, is adjacent to $y \,\})$;
`// Compute` $\rho(y)$ `for each` $y \in C_i$
for $C_i \in \sigma_x$ **do**
    for $y \in C_i$ **do**
        $\rho(y) \leftarrow \max(\{\, j \mid$ there exists a $z \in C_l$, $l > j$, to whom y is adjacent to $\,\})$;
        if $\rho(y) = \varnothing$ **then**
            $\rho(y) \leftarrow 0$;
`// Compute` $\mu(C'_i)$, $\mu(C_i)$ `and` $\rho(C_i)$ `for each (co-)component in` σ_x
for $C'_i \in \sigma_x$ **do**
    $\mu(C'_i) \leftarrow \max(\{\, \mu(y) \mid y \in C'_i \,\})$;
for $C_i \in \sigma_x$ **do**
    $\mu(C_i) \leftarrow \max(\{\, \mu(y) \mid y \in C_i \,\})$;
    $\rho(C_i) \leftarrow \max(\{\, \rho(y) \mid y \in C_i \,\})$;
$T \leftarrow$ `BuildSpine`(σ_x);
replace the leaves of T with the corresponding trees in $\mathcal{T}''$;
if $k' = k - 1$ **then**
    if T'_k's root is *parallel* **then**
        update T by making the root of T'_k the parent of T's root;
    else
        update T by adding a new *parallel* node as the parent of T's root, and adding the root of T'_k as a child of this new root;
foreach degenerate node $u \in T$ whose parent has the same type **do**
    replace u by its children;
return T;

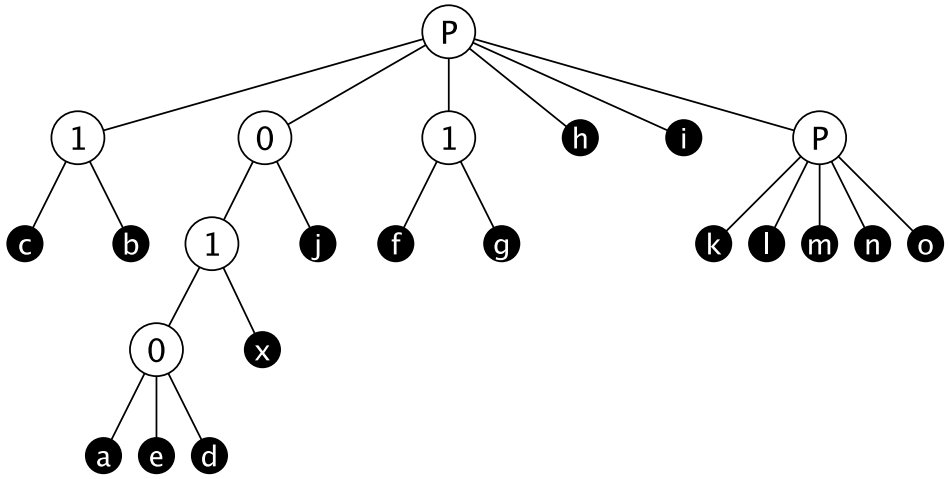


Abbildung 13: Der modulare Zerlegungsbaum T_G von G (Abbildung 3a), in der durch den Algorithmus berechneten Form. Prime Knoten sind hierbei nicht durch Quotientengraphen spezifiziert. Für eine detailliertere Darstellung siehe Abbildung 5

4 Laufzeitanalyse

Ein besonderes Charakteristikum des vorliegenden Algorithmus ist, dass dieser unter Nutzung relativ einfacher Datenstrukturen, eine lineare Laufzeit erreicht. Das folgende Kapitel stellt die nötigen Datenstrukturen vor und begründet die Laufzeit. Hinsichtlich der in Pseudocode angegebenen Algorithmen sei darauf hingewiesen, dass diese primär zur Verdeutlichung der nötigen Berechnungsschritte dienen sollen. Um eine linear Laufzeit zu erreichen, ist es teilweise nötig, bei der Implementierung von der Struktur des Pseudocode abzuweichen.

4.1 Datenstrukturen

Das Argument von `MDTree` (Algorithmus 3) ist ein ungerichteter Graph $G = (V, E)$, in der Darstellung durch eine Adjazenzliste. Hierbei verweist jeder Knoten $x \in V$ auf eine einfach verkettete Liste, welche die benachbarten Knoten $N(x)$ enthält. Eine aktive Liste $\alpha(x) \subseteq N(x)$ eines Knotens $x \in V$ lässt sich auf die gleiche Weise realisieren. Über den Verlauf des Algorithmus werden geordnete Baum-Partitionen verwaltet und die hierin enthaltenen Bäume strukturell verändert. Ein verwalteter Baum T ist stets gewurzelt und geordnet. Zudem hat jeder innerer Knoten mindestens zwei Kinder. Technisch besitzt jeder Knoten einen Zeiger auf seinen Vater sowie einen Zeiger auf sein am weitesten links stehendes

Kind, welches, zusammen mit den restlichen Kindern, in einer doppelt verketteten Liste verwaltet wird. Zusätzlich wird ein Zeiger benötigt, welcher von der Wurzel eines Baums auf ein Blatt verweist. Der Typ eines Knoten (*parallel*/*seriell*/*prim*) lässt sich durch eine an den Knoten vergebene Konstante repräsentieren. Hierzu benötigt jeder Knoten ein entsprechendes Feld. Temporäre Labels, wie etwa die von `TreeRefinement` vegebenen *dead*-Labels, lassen sich hierdurch ebenfalls realisieren. Gemäß dieser Darstellung ist ein Baum durch seine Wurzel bestimmt. Entsprechend kann eine geordnete Baum-Partition als eine doppelt verkettete Liste von Wurzeln implementiert werden, wobei ein Zeiger auf den Anfang der Liste verweist.

4.2 `DivideMDTree`

Über ihre rekursiven Aufrufe verteilt, realisiert die Funktion `DivideMDTree` eine LBFS des Graphen $G = (V, E)$ mittels der Technik der Partitionsverfeinerung (vgl. Abschnitt 3.1). Die Argumente von `DivideMDTree` sind eine Menge $S \subseteq V$, sowie eine Partition $\mathcal{P}$ einer Menge $S' \subseteq V$ mit $S \cap S' = \varnothing$. Hierfür wird die in Abschnitt 1.2 beschriebene Datenstruktur zur Repräsentation einer Partition von Knoten herangezogen. Jeder Knoten $x \in V$ ist genau einmal Pivot eines Aufrufs von `DivideMDTree`, wobei es jeweils zu einer Partitionsverfeinerung der Menge S sowie der Partition $\mathcal{P}$ anhand $N(x)$ kommt. Entsprechend der Argumentation in den Abschnitten 1.2 sowie 1.3 ergeben sich die Kosten hierfür insgesamt zu $\mathcal{O}(n + m)$. Darüber hinaus, werden die aktiven Listen der Knoten durch die Funktion `DivideMDTree` berechnet. Dies lässt sich während den Verfeinerungsschritten, d.h. während dem Traversieren von $N(x)$ erledigen: Zu jeder aktiven Liste $\alpha(y)$ für $y \in N(x) \cap (S \cap S')$ wird der Knoten x hinzugefügt. Hierdurch ändern sich die Kosten nicht, sodass die Funktion `DivideMDTree` insgesamt $\mathcal{O}(n+m)$ zu den Kosten des Algorithmus beiträgt.

4.3 `TreeRefinement`

Eine geordnete Baum-Partition der maximalen Slices $\mathcal{T} = [T_1, \ldots, T_k]$ eines Graphen $G[S]$, $S \subseteq V(G)$ bildet das Argument der Funktion `TreeRefinement`. Die aktiven Listen der Blätter der $T_i \in \mathcal{T}$ bilden die Grundlage der hier ausgeführten Berechnungen (vgl. Abschnitt 3.2). Die Begründung der Laufzeit von `TreeRefinement` erfolgt anhand der Beobachtung, dass jede Kante von G über den Verlauf des Algorithmus genau einmal als aktive Kante interpretiert wird und jedes Blatt $l \in L(T_i)$ für $i \in [1, k]$ mindestens eine aktive Kante hat. Zur Veranschaulichung sei auf Definition 3.1, Definition 3, Lemma 1.3

sowie den Umstand verwiesen, dass ein Aufruf der Funktion `TreeRefinement` innerhalb der Funktion `ConquerMDTree` erfolgt[27].

Während der Ausführung von `TreeRefinement` werden die Bäume $[T_1, \ldots, T_{i-1}] \subseteq \mathcal{T}$ bezüglich eines Blattes $y \in L(T_i)$, $T_i \in \mathcal{T}$ untersucht: Zunächst werden alle Knoten in $[L(T_1) \cap \alpha(y), \ldots, L(T_i) \cap \alpha(y)]$ mit temporären Markierungen ausgestattet. Hierzu wird die aktive Liste $\alpha(y)$ einmalig traversiert und die Knoten temporär markiert. Die Kosten für das Markieren sind konstant, sodass der Aufwand insgesamt $\mathcal{O}(|\alpha(y)|)$ beträgt. Im Anschluss werden alle inneren Knoten in $[T_1, \ldots, T_{i-1}]$ ebenfalls temporär markiert, falls jeweils all ihre Kinder markiert sind. Dies lässt sich bottom-up rekursiv umsetzen, indem ausgehend von einem markierten Blatt, alle Vorfahren markiert werden, falls sie der Vater von ausschließlich markierten Kindern sind. Bei wiederholter Anwendung dieser Regel für alle markierten Blätter, können so alle zu markierenden Knoten markiert werden. Da jeder innere Knoten mindestens zwei Kinder hat, werden auf diese Weise höchstens $|\alpha(y)| - 1$ weitere Knoten markiert. Entsprechend belaufen sich die Kosten hierfür ebenfalls auf $\mathcal{O}(|\alpha(y)|)$.

Im folgenden Schritt, werden unmarkierte Knoten mit mindestens einem markierten Kind durch ein *dead*-Label ausgezeichnet und deren markierte bzw. unmarkierte Kinder ggf. unter neuen Knoten gruppiert. Dieser Vorgang ähnelt dem der Partitionsverfeinerung, wobei hier innere Knoten in $[T_1, \ldots, T_{i-1}]$ verfeinert werden. Anhand der Datenstruktur für Bäume, lässt sich dies entsprechend umsetzen. Als Pivotmenge kann eine temporäre, einfach verkettete Liste $\beta(y)$ herangezogen werden, welche alle temporär markierten Knoten in $[T_1, \ldots, T_{i-1}]$ enthält. Diese kann während dem Markieren der Knoten erzeugt werden. Gemäß der obigen Argumentation gilt $|\beta(y)| \leq |2\alpha(y)| - 1$, sodass sich ein zusätzlicher Aufwand von $\mathcal{O}(|\alpha(y)|)$ ergibt.

Um die unmarkierten Knoten mit mindestens einem markierten Kind zu verfeinern, wird die Liste $\beta(y)$ einmalig traversiert und für jeden zu verfeinernden Knoten u die Reihenfolge der Kinder so verändert, dass die markierten Knoten vor den unmarkierten Knoten erscheinen. Dies ist in $\mathcal{O}(|\alpha(y)|)$ realisierbar. Während $\beta(y)$ ein weiteres Mal traversiert wird, werden die markierten Kinder von u entfernt und unter einem neuen Knoten u_A gruppiert. Hierfür fallen weitere Kosten in $\mathcal{O}(|\alpha(y)|)$ an. Anschließend wird der Knoten u durch einen neu erzeugten, *dead*-gelabelten Knoten u' ersetzt; die Knoten u_A und u werden zu den Kindern von u'. Falls u_A oder u nur ein Kind haben, werden die Knoten durch ihre Kinder ersetzt. Für das Erzeugen/Löschen neuer Knoten ergeben

[27] Zur weiteren Veranschaulichung kann Abbildung 6 herangezogen werden: Jedem Paar $u, v \in V(G)$ lässt sich genau eine Partition maximaler Slices $\mathcal{P}_S$ zuordnen, sodass für zwei verschiedene Klassen $P, P' \in \mathcal{P}_S$ gilt, dass $u \in P$, $v \in P'$ ($uv \in E(G)$ ist eine aktive Kante).

sich für jeden zu verfeinernden Knoten konstante Kosten, wobei höchstens $|\alpha(y)|$ Knoten verfeinert werden. Das Entfernen der temporären Markierung ist durch ein abschließendes Traversieren von $\beta(y)$ zu bewerkstelligen. Auch dies kostet $\mathcal{O}(|\alpha(y)|)$.

Somit belaufen sich die Kosten für das Verfeinern von $[T_1, \ldots, T_{i-1}]$ bezüglich y insgesamt auf $\mathcal{O}(|\alpha(y)|)$. Die bei der Verfeinerung beteiligten Knoten $L(T_1) \cap \alpha(y) \cup \ldots \cup L(T_i) \cap \alpha(y)$, können anschließend aus der Liste $\alpha(y)$ entfernt werden, da sie kein weiteres Mal benutzt werden. Die Kosten sind ebenfalls in $\mathcal{O}(|\alpha(y)|)$. Da jede Kante von G über den Verlauf des gesamten Algorithmus genau einmal aktiv ist und jedes Blatt $l \in T_i$ für alle $T_i \in \mathcal{T}$ mindestens eine aktive Kante besitzt, ergeben sich die gesamten Kosten der Funktion `TreeRefinement` folglich zu $\mathcal{O}(m)$.

4.4 `Factorize`

Eine Verfeinerung einer geordneten Baum-Partition der maximalen Slices $\mathcal{T} = [T_1, \ldots, T_k]$ eines Graphen $G[S]$, $S \subseteq V(G)$ bildet das Argument der Funktion `Factorize`.

Während der Ausführung von `Factorize`, werden alle $T_i \in \mathcal{T}$ nacheinander betrachtet und dabei jeweils die folgenden Schritte ausgeführt: Zunächst werden alle ungelabelten Vorfahren der *dead*-gelabelten Knoten eines T_i mit einem *zombie*-Label ausgestattet. Hierzu genügt es, jeden Baum T_i einmalig gemäß einer in der Wurzel begonnenen Tiefensuche zu traversieren und die Labels zu setzten. Die Kosten hierfür sind linear zur Größe des Baums T_i, d.h. in $\mathcal{O}(|L(T_i)|)$.

Als eine Vorbereitung auf das Ersetzten des Baumes T_i durch die darin enthaltenen, *dead*-Label-freien Teilbäume, folgt die Anpassung der Reihenfolge der Kinder der *zombie*-gelabelten Knoten in T_i. Im Fall der *zombie*-gelabelten, degenerierten Knoten mit mehr als zwei ungelabelten Kindern, werden diese zuvor unter neuen Knoten gruppiert. Dies lässt sich analog zu dem Verfeinern in `TreeRefinement` während einer Traversierung von T_i umsetzen, sodass sich Kosten in Höhe von $\mathcal{O}(|L(T_i)|)$ ergeben. Zur Anpassung der Reihenfolge genügt es, den Baum T_i ein weiteres Mal zu traversieren und jeden Knoten mit einem *zombie*-gelabelten Elternknoten – in Abhängigkeit seines Labels (*dead*/*zombie*/kein Label) sowie des Index i – vor seinen Geschwisterknoten zu platzieren. Die Kosten hierfür sind ebenfalls $\mathcal{O}(|L(T_i)|)$.

Schließlich erfolgt das Ersetzten des Baumes T_i durch die darin enthaltenen, *dead*-Label-freien Teilbäume. Hierzu wird eine temporäre doppelt verkettete Liste $\mathcal{T}'$ herangezogen, welche zu Beginn mit der Wurzel w des Baums T_i initialisiert wird. Es wird geprüft, ob w gelabelt (*dead*/*zombie*) ist: Falls ja, so wird w durch die Liste der Kinder von w (d.h. die darin gewurzelten Teilbäume) ersetzt. Die hierbei anfallenden Kosten sind linear zu

der Länge der Liste der Kinder. Im Folgenden werden äquivalente Ersetzungen für die nun in $\mathcal{T}'$ enthaltenen Wurzeln durchgeführt, sodass sich hierfür Kosten in Höhen von $\mathcal{O}(|L(T_i)|)$ ergeben.

Somit lassen sich die Berechnungen der Funktion `Factorize` durch mehrmaliges (konstant) Traversieren aller Bäume $T_i \in \mathcal{T}$ realisieren, wobei der Aufwand linear zu Größe von $\mathcal{T}$ ist. Da jedes Blatt $l \in L(T_i)$ für alle $i < 1$ mindestens eine aktive Kante besitzt, jede Kante in G über den Verlauf des Algorithmus genau einmal aktiv ist und da jeder innere Knoten in einem $T_i \in \mathcal{T}$ mindestens zwei Kinder hat, ergibt sich der Platz für alle von `Factorize` bearbeiteten Baum-Partitionen zu $\mathcal{O}(m)$. Entsprechend sind die Kosten der Funktion `Factorize` über den Verlauf des gesamten Algorithmus in $\mathcal{O}(m)$.

4.5 `BuildSpine`

Grundlage für die Bestimmung der starken Module mit Pivot ist eine Pivot-faktorisierende Permutation $\sigma_x = C'_a, \ldots, C'_1, x, C_1, \ldots, C_b$. Hierfür kann die Datenstruktur der geordneten Baum-Partitionen verwendet werden. Jede Komponente/Ko-Komponente in σ_x wird dabei durch einen Baum der Höhe 1 implementiert, wobei die μ- und ρ-Werte direkt in den Wurzeln gespeichert werden.

Die Ermittlung der starken Module mit Pivot erfolgt innerhalb einer Schleife, anhand der Auswertung der μ- bzw. ρ-Werte der Komponenten/Ko-Komponenten in σ_x. Hierbei gilt, dass jede Ko-Komponente $C'_i \in \sigma_x$ höchstens zweimal betrachtet wird, da sie entweder zur Bildung eines seriellen oder eines primen Moduls beiträgt. Gleiches gilt für die Komponenten $C_i \in \sigma_x$, welche bei der Bestimmung eines parallelen oder eines primen Moduls ausgewertet werden. Während jeder Betrachtung einer Komponente/Ko-Komponente ergeben sich konstante Kosten, für das Setzen der verwendeten Indexe und die Auswertung der μ- bzw. ρ-Werte. Schließlich wird für jedes starke Modul mit Pivot ein neuer Knoten erzeugt. Diese Knoten bilden die inneren Knoten des berechneten Baums T. Da die Anzahl der Blätter von T durch $a + b + 1$ gegeben ist, werden höchsten $a + b$ neue Knoten erzeugt. Entsprechend belaufen sich die Kosten einer Ausführung von `BuildSpine` insgesamt auf $\mathcal{O}(a + b)$.

Für die Pivot-faktorisierende Permutation σ_x gilt, dass keine Komponente bzw. Ko-Komponenten in σ_x leer ist. Folglich ist $a + b$ linear zur Anzahl der Knoten in den Komponenten/Ko-Komponenten in σ_x. Zudem gilt, dass der Aufruf der Funktion `BuildSpine` innerhalb der Funktion `ConquerMDTree` erfolgt, wobei die Pivot-faktorisierende Permutation σ_x durch die von `Factorize` berechnete Baum-Partition definiert wird. Somit ist sichergestellt, dass jedes Blatt in σ_x mindestens eine aktive Kante besitzt. Da jede Kante

nur einmal aktiv ist, gilt für die gesamte Größe aller Eingaben von `BuildSpine`, dass diese linear zu m ist. Folglich ergeben sich die Kosten von `BuildSpine`, in Bezug auf den gesamten Algorithmus, zu $\mathcal{O}(m)$.

4.6 `ConquerMDTree`

Eine geordnete Baum-Partition der maximalen Slices $\mathcal{T} = [T_1, \ldots, T_k]$ eines Graphen $G[S]$, $S \subseteq V(G)$ bildet das Argument der Funktion `ConquerMDTree`. Direkt zu Beginn wird der Pivotknoten x bestimmt. Da x dem Baums $T_1 = x$ entspricht (Definition 15) sind die Kosten hierfür konstant. Als Nächstes wird geprüft, ob $\mathcal{T}$ eine geordnete Baum-Partition der maximalen Slices eines zusammenhängenden Graphen darstellt. Gemäß der Argumentation in Abschnitt 3.5 genügt es hierzu, die aktive Liste $\alpha(y)$ eines Blattes $y \in L(T_k)$ zu überprüfen. Falls $\alpha(y)$ leer ist, so folgt die Zuweisung $k' \leftarrow k-1$; andernfalls die Zuweisung $k' \leftarrow k$. Um zu dem Knoten y gelangen, ist $\mathcal{T}$ zunächst bis zu dem Baum T_k zu traversieren. O.B.d.A. besitzt der Baum T_k einen Zeiger auf das Blatt y, sodass für die Überprüfung von $\alpha(y)$ insgesamt $\mathcal{O}(k)$ anfallen. Um die Blätter der $T_i \in [T_1, \ldots, T_{k'}]$ entsprechend ihrer Komponenten/Ko-Komponenten in den Graphen $G[L_i]$ zu labeln, sind alle Bäume $T_i \in [T_1, \ldots, T_{k'}]$ jeweils vollständig zu traversieren. Die Kosten hierfür sind linear zur Anzahl der Blätter der $T_i \in [T_1, \ldots, T_{k'}]$. Es folgen die Aufrufe der Funktionen `TreeRefinement` sowie `Factorize`, deren Kosten bereits berücksichtigt sind. Im Anschluss wird aus der durch `Factorize` berechneten Baum-Partition $\mathcal{T}''$, die Pivot-faktorisierende Permutation $\sigma_x = C'_a, \ldots, C'_1, x, C_1, \ldots, C_b$ generiert. Hierfür genügt es, die Bäume $T_i \in \mathcal{T}''$ zu traversieren, die Blätter gemäß ihrer diesbezüglichen Labels unter neuen Knoten zu gruppieren und den Pivotknoten x entsprechend zu platzieren. Die Kosten für das Traversieren und das Auswerten der Labels sind linear zur Anzahl der Blätter der $T_i \in \mathcal{T}''$; die Anzahl der neu erzeugten Knoten ist durch die Anzahl der Blätter beschränkt ist. Um die μ- und ρ-Werte der Komponenten $C_i \in \sigma_x$ bzw. der Ko-Komponenten $C'_i \in \sigma_x$ zu berechnen, können die aktiven Listen der Blätter in σ_x herangezogen werden: Für jedes Blatt y wird eine temporäre Liste $\alpha'(y)$ berechnet, sodass für paarweise verschiedene Blätter y, y' gilt, dass $y' \in \alpha'(y)$ gdw. $y \in \alpha(y')$. Die α'-Listen lassen sich während dem einmaligen Traversieren aller α-Listen erzeugen. Mit Vorliegen der α'-Listen, können die μ- und ρ-Werte der Blätter in σ_x während dem einmaligen Traversieren der α'-Listen ermittelt werden. Die μ- und ρ-Werte der Komponenten/Ko-Komponenten lassen sich abschließen während dem Traversieren der entsprechenden Bäume bestimmen. Es folgt der Aufruf der Funktion `BuildSpine`, wodurch ein gewurzelter Baum T berechnet wird. Die Kosten hierfür sind bereits berücksichtigt. Die für die Repräsentation der Elemente in

σ_x herangezogen Teilbäume in T sind nun durch die Bäume in $\mathcal{T}''$ zu ersetzen. Dies lässt sich mit einem Aufwand linear zur Anzahl der Blätter in $\mathcal{T}''$ erledigen. Falls $k' = k - 1$ erfolgt das Vereinigen des Baums T mit dem Baum $T_k \in \mathcal{T}$. Die Kosten hierfür sind konstant. Das abschließende Entfernen degenerierter Knoten mit degenerierten Kinder des selben Typs sowie das Setzten eines Zeigers von der Wurzel auf ein Blatt lässt sich wiederum während einer Traversierung von T bewerkstelligen.

Zusammenfassend sind alle Berechnungen der Funktion `ConquerMDTree` mit einem zur Anzahl der Blätter in $\mathcal{T}$ linearen Aufwand realisierbar. Da jedes Blatt in $\mathcal{T}$ mindestens eine aktive Kante besitzt und jede Kante von G nur einmal aktiv ist, ergeben sich die gesamten Kosten von `ConquerMDTree` zu $\mathcal{O}(m)$.

Da die Funktionen `TreeRefinement`, `Factorize` und `BuildSpine` ebenfalls Kosten in Höhe von $\mathcal{O}(m)$ aufweisen und der Aufwand der Funktion `DivideMDTree` $\mathcal{O}(n + m)$ beträgt, gilt die folgende Aussage:

Theorem 4.1. *[49] Der modulare Zerlegungsbaum T_G eines ungerichteten Graphen G kann durch* `MDTree` *(Algorithmus 3) in $\mathcal{O}(n + m)$ berechnet werden.*

5 Experimentelle Untersuchung der Laufzeit

Das abschließende Kapitel untersucht das Laufzeitverhalten der angefertigten Python3-Implementierung des Algorithmus von Tedder et al. Entsprechend Theorem 4.1 kann die modulare Zerlegung eines Graphen $G = (V, E)$ durch den vorgestellten Algorithmus in $\mathcal{O}(n + m)$ berechnet werden. Im Folgenden wird versucht anhand möglichst aussagekräftiger Experimente zu zeigen, inwiefern das Ziel einer linearen Implementierung gelungen ist. Darüber hinaus werden Erklärungen für besondere Laufzeitphänomene gegeben.

Grundsätzlich sind hierbei verschiedene Parameter zu berücksichtigen. Offensichtlich sind die Anzahl der Knoten $n = |V|$ sowie die Anzahl der Kanten $m = |E|$ eines Graphen $G = (V, E)$ zwei Parameter, welche die Laufzeit des Algorithmus bestimmen. Des Weiteren besteht die Möglichkeit, dass strukturelle Eigenschaften eines Graphen ebenfalls Einfluss auf die Laufzeit nehmen. Beispielsweise könnten die Anzahl der Komponenten eines Graphen oder spezielle induzierte Subgraphen das Laufzeitverhalten beeinflussen, sodass zwei Graphen G, G' zu verschiedenen Laufzeiten führen selbst wenn $|V(G)| = |V(G')|$ und $|E(G)| = |E(G')|$. Diesbezüglich voll umfänglich zu testen, ist im Rahmen dieser Arbeit unmöglich. Daher werden sich die Tests darauf beschränken, hinsichtlich der Knotenanzahl n, der Kantenanzahl m sowie des strukturellen Parameters der modularen Weite mw zu testen[28]. Hierbei wird versucht die Experimente so zu gestalten, dass die genannten Parameter möglichst unabhängig untersucht werden können.[29]

5.1 Knotenanzahl, Kantenanzahl

Konkret könnte eine obere Schranke für die Laufzeit einer optimalen Implementierung durch eine Gleichung der Form $t(n, m) = C_1 \cdot n + C_2 \cdot m + C_3$ beschrieben sein, wobei $C_1, C_2, C_3 > 0$ konstant sind. Da zunächst keinerlei Information über die Größenordnung der Konstanten vorliegt, ist es sinnvoll jeweils einen der beiden Parameter zu fixieren und den anderen Parameter zu variieren. Dies erlaubt es das Verhalten des Algorithmus in Hinblick auf die Anzahl der Knoten bzw. der Anzahl der Kanten zu untersuchen. Hierfür werden pseudozufällig erzeugte Graphen als Eingabe des Algorithmus gewählt. Diese werden gemäß der in [28] vorgeschlagenen Methode generiert und sind daher durch eine feste Knotenanzahl n sowie eine Kantenwahrscheinlichkeit p charakterisiert[30].

[28] In einem geringen Umfang wird die modulare Tiefe md ebenfalls berücksichtigt. Sie wird im Folgenden zunächst nicht erwähnt, es gilt jedoch $mw = n \Rightarrow md = 1$, falls $n \geq 4$.

[29] Alle Berechnungen wurden mit einem MacBook Pro mit 4GB RAM und einem Intel Core i5 Prozessor mit 2,3 GHz durchgeführt. Die Berechnungszeiten sind stets in Sekunden angegeben.

[30] Entsprechende Graphen können über die Eingabeoberfläche des Programms unter dem Menüpunkt „Graph Generator / Generator A“ erzeugt werden.

Die Kantenanzahl eines pseudozufälligen Graphen $G = (n, p)$ lässt sich hierbei durch $m \approx \binom{n}{2} \cdot p$ gut abschätzen. Der Parameter der modularen Weite mw wird im Folgenden zunächst vernachlässigt. Es kann jedoch angenommen werden, dass fast alle betrachteten Graphen mit $n \geq 1000$ prim sind, d.h. $mw = n$. Das folgende Experiment (Abbildung 14) legt dies nahe.

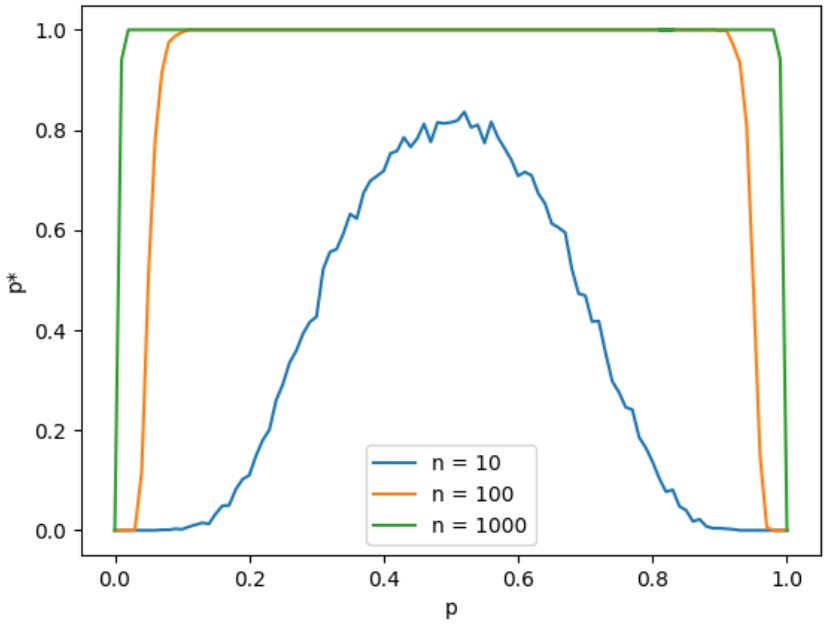


Abbildung 14: Die Wahrscheinlichkeit p^* eines pseudozufälligen Graphen $G = (n, p)$ prim zu sein. Für jedes Paar $(n, p) \in \{10, 100, 1000\} \times \{0.00, 0.01, \ldots, 1.00\}$ wurden 1000 Graphen erzeugt und die Eigenschaft überprüft.

Wie durch Abbildung 15 dargestellt, steigt die Berechnungsdauer der modularen Zerlegung für Graphen mit fixierter Knotenanzahl n bzw. einer bei $0.05 \leq p \leq 0.95$ erwartungsgemäß fixierten modularen Weite $mw = n$, proportional zur Kantenanzahl $m \approx \binom{n}{2} \cdot p$. Es sei darauf hingewiesen, dass die Laufzeitverläufe für verschiedene Knotenanzahlen anhand der Darstellung nur eingeschränkt vergleichbar sind, da eine Veränderung der Kantenwahrscheinlichkeit p für größere n zu einer stärkeren Veränderung der Kantenanzahl m führt. Vergleicht man jedoch exemplarisch die Werte $t(n = 2000, p = 0.05) \approx 2.645s$ und $t(n = 2000, p = 0.95) \approx 10.068s$ mit den Werten $t(n = 4000, p = 0.05) \approx 10.620s$ und $t(n = 4000, p = \frac{733}{2666} \approx 0.275) \approx 16.399s$ – in beiden Fällen ist eine Veränderung der Kantenanzahl um $\Delta m = 1799100$ zu erwarten – so zeigt sich jeweils eine ähnliche Änderung der Berechnungsdauer. Unter Berücksichtigung von technischen Schwankungen zur

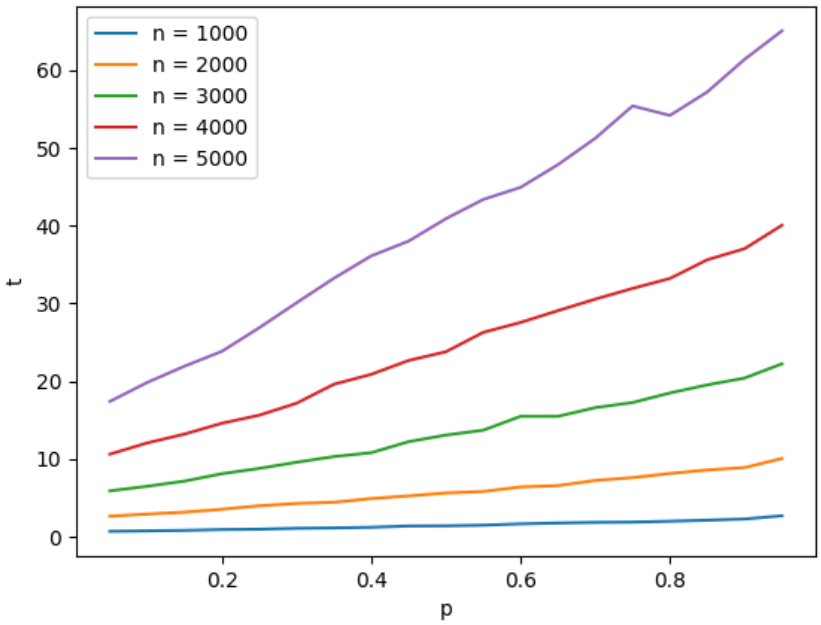


Abbildung 15: Das Laufzeitverhalten in Abhängigkeit der Kantenwahrscheinlichkeit p. Für jedes Paar $(n, p) \in \{1000, 2000, \ldots, 5000\} \times \{0.05, 0.10, \ldots, 0.95\}$ wurden 5 Graphen erzeugt und die durchschnittliche Berechnungsdauer der modularen Zerlegung ermittelt.

Ausführungszeit und einer eventuell zu geringen Testanzahl (5 Läufe je Datenpunkt) kann vermutet werden, dass die Implementierung bezüglich der Kantenanzahl m ein lineares Laufzeitverhalten aufweist. Entsprechend der Argumentation in Kapitel 4 ließe dies weiter vermuten, dass die Umsetzungen der Funktionen `ConquerMDTree`, `TreeRefinement`, `Factorize` und `BuildSpine` – theoretisch jeweils in $\mathcal{O}(m)$ realisierbar – gelungen sind.

Da es sich bei Pseudozufallsgraphen mit $p \approxeq 0$ oder $p \approxeq 1$ meist um Graphen mit $mw < n$ handelt, sind diese in Abbildung 15 nicht erfasst. Tabelle 4 ergänzt Abbildung 15 um numerische Werte für $t(n, p)$ in den Grenzbereichen.

Was bei dem Vergleich von Tabelle 4 mit Abbildung 15 hervorsticht, ist der enorme Anstieg der Berechnungsdauer[31] für $p = 1$, d.h. $t(n, p = 1) - t(n, p = 0.95) \gg t(n, p = 0.95) - t(n, p = 0.90)$ für $n \in \{1000, 2000, \ldots, 5000\}$. Hierbei scheint eine jeweilige Erhöhung der Kantenanzahl um $\Delta m = \binom{n}{2} \cdot 0.05$ nicht der Grund für die erhöhte Berechnungsdauer zu sein, da $t(n, p) - t(n, p - 0.05) \approx t(n, p - 0.05) - t(n, p - 0.10)$ für

[31] Für $G(n = 5000, p = 1)$ wurde nur eine Messung durchgeführt.

	$p = 0$	$p = 0.05$	$p = 0.95$	$p = 1$
$n = 1000$	$0.926s$	$0.699s$	$2.697s$	$23.993s$
$n = 2000$	$2.589s$	$2.645s$	$10.068s$	$161.793s$
$n = 3000$	$5.433s$	$5.900s$	$22.234s$	$549.852s$
$n = 4000$	$9.263s$	$10.619s$	$40.064s$	$1288.758s$
$n = 5000$	$15.328s$	$17.401s$	$65.064s$	$9910.594s$

Tabelle 4: Die durchschnittliche Berechnungsdauer in den Grenzbereichen $p \approxeq 0$ und $p \approxeq 1$. (Ergänzung zu Abbildung 15.)

$(n, p) \in \{1000, 2000, \ldots, 5000\} \times \{0.10, 0.15 \ldots, 0.95\}$.

Grundsätzlich unterscheiden sich die Graphen $G(n, p = 1)$ für $n \in \{1000, 2000, \ldots, 5000\}$ von den Graphen $G(n, p)$ für $(n, p) \in \{1000, 2000, \ldots, 5000\} \times \{0.05, 0.10, \ldots, 0.95\}$ dadurch, dass Erstere als vollständige Graphen spezielle Cographen mit $mw = 2$ sind, während für Letztere angenommen wird, dass diese prim sind, also $mw = n$. Besonders die Eigenschaft der Vollständigkeit wird als ein Grund für die erhöhte Berechnungsdauer vermutet, da vollständige Graphen in Bezug auf den Algorithmus, zu einer maximalen Rekursionstiefe führen. Diese wird durch die Funktion `DivideMDTree` (Zeile 16-19) verursacht, weil die in der While-Schleife verwaltete Partition stets nur die Klasse der Nachbarschaft des gewählten Pivotknotens enthält, welche bei einem rekursiven Aufruf übergeben wird. Entsprechend könnte ein besonders hoher Aufrufstapel Grund für eine technische Verlangsamung des Programms sein.

Graphen ohne Kanten führen auf eine ähnliche Weise zu einer maximalen Rekursionstiefe, wobei Tabelle 4 lediglich für $n = 1000$ dokumentiert[32], dass $t(n, p = 0) > t(n, p = 0.05)$. Trotzdem sind die Differenzen der Berechnungszeiten im Grenzbereich $p \approxeq 1$ deutlich größer, als im Bereich $p \approxeq 0$. Eine mögliche Ursache könnte darin bestehen, dass die Aufrufrahmen im Fall der vollständigen Graphen einen wesentlich höheren Speicherbedarf haben, als die Aufrufrahmen im Fall der Graphen ohne Kanten.

Abbildung 16 dokumentiert das Laufzeitverhalten des Algorithmus in Abhängigkeit der Knotenanzahl n für eine fixierte Kantenanzahl m. Die dünnsten Graphen ($n = 5000, m = 50000$) sind mit einer Kantenwahrscheinlichkeit $p = \frac{20}{4999} \approx 0.004$, die dichtesten Graphen ($n = 1000, m = 450000$) mit einer Kantenwahrscheinlichkeit $p = \frac{100}{111} \approx 0.901$ erzeugt. Stichprobenartige Tests sowie Abbildung 14 lassen annehmen, dass alle in Abbildung 16

[32] Ein feinerer Test (nicht dargestellt) zeigte, dass $t(n, p = 0) > t(n, p = 0.05)$ für $n \leq 1750$ stets erfüllt ist. Erst danach scheinen durch die quadratisch steigende Kantenanzahl ausgelöste Effekte den Effekt der Rekursionstiefe zu überwiegen.

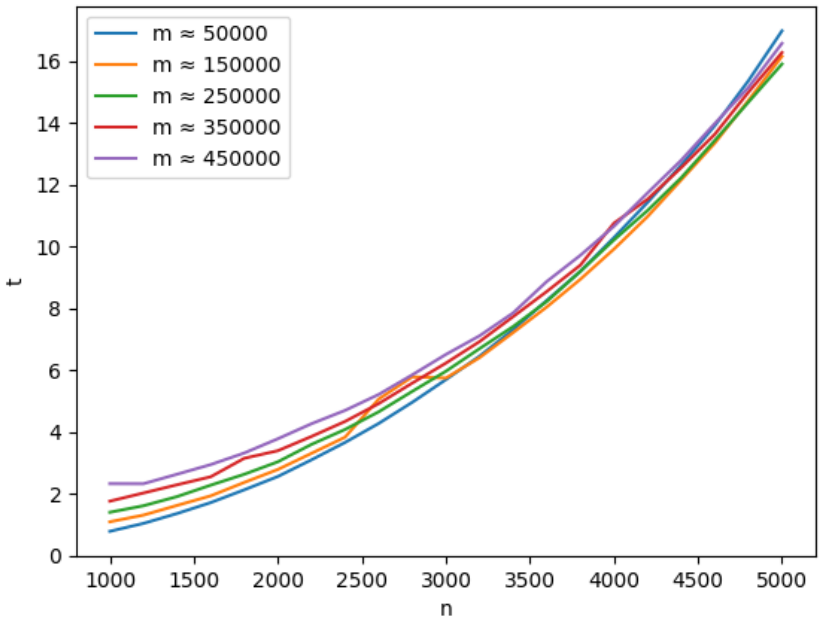


Abbildung 16: Das Laufzeitverhalten in Abhängigkeit der Knotenanzahl n. Für jedes Paar $(n, m) \in \{1000, 1200, \ldots, 5000\} \times \{\approx 50000, \approx 150000, \ldots, \approx 450000\}$ wurden 5 Graphen erzeugt und die durchschnittliche Berechnungsdauer der modularen Zerlegung ermittelt.

untersuchten Graphen prim sind, d.h. $mw = n$.

Wie durch Abbildung 16 gezeigt, besteht kein linearer Zusammenhang zwischen der Knotenanzahl n und der Berechnungszeit $t(n, m)$. Außerdem scheint die Anzahl der Knoten n einen schwerwiegenderen Einfluss auf die Berechnungsdauer $t(n, m)$ zu haben, als die Kantenanzahl m. Da beispielsweise $t(n = 1000, m \approx 450000) \approx t(n = 2000, m \approx 50000)$ wird deutlich, dass eine Erhöhung der Kantenanzahl um $\Delta m \approx 400000$ zu Beginn etwa zu einer ähnlichen Laufzeitveränderung führt, wie eine Erhöhung der Knotenanzahl um $\Delta n = 1000$. Mit steigender Knotenanzahl nimmt der Einfluss der Kanten auf die Berechnungsdauer weiter ab und es treten Fälle auf, bei denen Graphen mit weniger Kanten zu einer größeren Berechnungsdauer führen als Graphen mit einer gleichen Knotenanzahl und einer höheren Kantenanzahl. Eine technische Ursache für die stärkere Gewichtung der Knoten könnte sein, dass die Knotenobjekte einen höheren Speicherbedarf haben als Kanten, welche lediglich durch Zeiger realisiert werden. Eine Erklärung für den nicht-linearen Laufzeitverlauf liefert die – bezüglich der Partitionsverfeinerung – ungünstig implementierte Funktion `DivideMDTree`. Hier wurde bewusst auf eingebaute Python3-

Datentypen und eine hinsichtlich der Komplexität ungünstige Schleife zurückgegriffen, da eine eigens implementierte Methode zur Partitionsverfeinerung zu praktisch schlechteren Berechnungszeiten führte.[33]

5.2 Modulare Weite

Der vorhergehende Abschnitt untersucht das Laufzeitverhalten bezüglich der Knotenanzahl n und der Kantenanzahl m anhand von Pseudozufallsgraphen gemäß [28]. Derartige Graphen bilden eine einfache Möglichkeit die Anzahl der Knoten bzw. Kanten zu regulieren. Allerdings bergen sie den Nachteil, das Laufzeitverhalten lediglich mit der Beschränkung auf prime Graphen mit speziellen strukturellen Eigenschaften abzubilden.

In dem folgenden Experiment wird daher die Struktur der Eingabegraphen bezüglich dem Parameter der modularen Weite mw verändert und der Einfluss auf die Berechnungsdauer analysiert. Es werden ausschließlich Graphen mit einer konstanten Knotenanzahl $n = 1000$ und einer Kantenanzahl $m = 249750 \pm 5\%$ berücksichtigt. Mit der Absicht den Einfluss der modularen Weite möglichst prägnant nachweisen zu können, sind die Graphen so gewählt, dass eine maximale Anzahl der inneren Knoten der modularen Zerlegungsbäume mw-viele Kinder hat. Die entsprechenden modularen Zerlegungsbäume können daher näherungsweise als volle k-äre Bäume mit $k = mw$ beschrieben werden, falls $mw \geq 4$.[34]

Zusätzlich wird der Einfluss des Parameters der modularen Tiefe md – der Tiefe des modularen Zerlegungsbaums eines Graphen – in geringem Umfang mit untersucht. Dazu sind die betrachteten Graphen so konstruiert, dass ihre modularen Zerlegungsbäume, unter der Einschränkung der modularen Weite, entweder eine maximale, eine minimale oder eine weitestgehend zufällige Tiefe haben.[35]

Abbildung 17 deutet an, dass Graphen mit einer kleinen modularen Weite durchschnittlich zu einer höheren Berechnungsdauer führen. Insbesondere sind die Berechnungszeiten im Fall von Cographen um ein Vielfaches höher, als bei Graphen mit $mw \geq 4$. Ein ähnlicher Effekt wurde bereits für den Fall der vollständigen Graphen beobachtet (vgl. Tabelle 4). Hier wurde eine maximale Rekursionstiefe als eine mögliche, technische Ursache für lange Berechnungszeiten vermutet, welche in dem vorliegenden Fall jedoch

[33] Die entsprechenden Teile in `md.py` sind durch `TODO`s gekennzeichnet, sodass ersatzweise die theoretisch günstigere Methode verwendet werden kann.
[34] Eine Ausnahme bilden die modularen Zerlegungsbäume von Cographen.
[35] Entsprechende Graphen können über die Eingabeoberfläche des Programms unter dem Menüpunkt „Graph Generator / Generator B“ erzeugt werden.

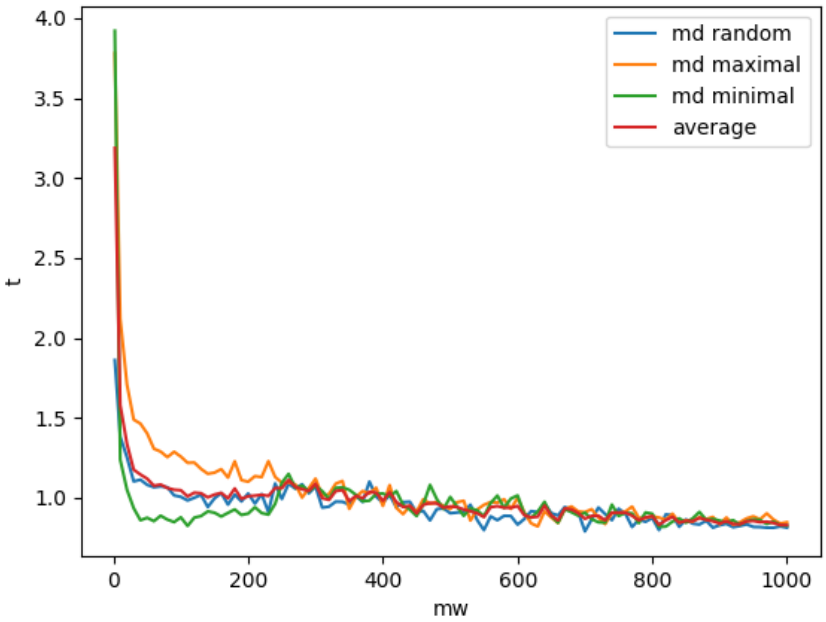


Abbildung 17: Das Laufzeitverhalten in Abhängigkeit der modularen Weite mw und der modularen Tiefe md. Für jedes Paar $(mw, md) \in \{2, 10, 20 \ldots, 1000\} \times \{\text{random}, \text{maximal}, \text{minimal}\}$ wurden 10 Graphen mit $n = 1000$, $m = 249750 \pm 5\%$ erzeugt und die durchschnittliche Berechnungsdauer der modularen Zerlegung ermittelt.

ausgeschlossen werden kann.

Grundsätzlich gilt für die untersuchten Graphen mit $mw \geq 4$, dass deren modulare Zerlegungsbäume bei kleiner modularer Weite mehr innere Knoten haben, als modulare Zerlegungsbäume von Graphen mit einer großen modularen Weite. Hierzu beachte man, dass die Anzahl der inneren Knoten i eines vollen k-ären Baums in Abhängigkeit der Anzahl der Blätter l durch $i = \frac{l-1}{k-1}$ angegeben werden kann[36]. Folglich könnte ein erhöhter Speicherbedarf bei kleiner modularer Weite $mw \geq 4$ eine technische Ursache für größere Berechnungszeiten sein. Außerdem werden geordnete Baum-Partitionen während den Conquer-Phasen des Algorithmus mehrfach, vollständig traversiert. Unter der Annahme, dass die in einer Baum-Partition enthaltenen Bäume dem resultierenden modularen Zerlegungsbaum strukturell ähneln[37], kann vermutet werden, dass der Aufwand für das Traversieren antiproportional zur modularen Weite ist.

[36] Dies folgt aus $n = l + i$ und $n = ki + 1$, wobei n die Anzahl der Knoten eines k-ären Baums ist.
[37] Bezüglich der modularen Weite.

Die auffallend hohen Berechnungszeiten bei Cographen lassen sich hiermit allerdings nur eingeschränkt erklären. Deren modulare Zerlegungsbäume lassen sich nicht näherungsweise als k-äre Bäume mit $k = mw$ auffassen, da degenerierte Knoten des selben Typs (*parallel*/*seriell*) während der Conquer-Phase zusammengefasst werden (`ConquerMDTree`, Zeile 37-38). Somit haben modulare Zerlegungsbäume von Cographen möglicherweise sogar weniger innere Knoten, als die modularen Zerlegungsbäume der betrachteten Graphen mit einer kleinen modularen Weite $mw \geq 4$. Das Traversieren bezüglich dem Resultat ähnlicher Baum-Partitionen, würde dann sogar einen geringeren Aufwand bedeuten. Trotzdem könnte der Aufwand für das Zusammenfassen degenerierter Knoten auch eine Ursache für die zusätzliche Berechnungsdauer sein.

Des Weiteren werden in den Funktionen `TreeRefinement` (Zeile 9-12) und `Factorize` (Zeile 6-7) unter Umständen neue Knoten erzeugt. Die Bedingung hierfür ist, dass ein *dead*- bzw. *zombie*-gelabelter Knoten degeneriert ist. Somit kann vermutet werden, dass dieser Aufwand vermehrt im Fall von Cographen geleistet wird.

Hinsichtlich der modularen Tiefe md, zeigen sich besonders für Graphen mit fester modularer Weite $2 \leq mw \lessapprox 250$ unterschiedliche Berechnungszeiten. Grundsätzlich ist darauf zu verweisen, dass die modulare Tiefe sowie die Differenz zwischen einer maximalen modularen Tiefe und einer minimalen modularen Tiefe, mit steigender modularer Weite, abnimmt. Hierdurch lassen sich ähnlichere Berechnungszeiten im Bereich $mw \gtrapprox 250$ erklären. Entsprechend der Konstruktion der Graphen, sind Aussagen über die modulare Tiefe für $mw = 2$ hinfällig. Da der sich in dem Bereich $10 \lessapprox mw \lessapprox 250$ abzeichnende Trend durch die Werte bei $mw = 2$ ebenfalls nicht bestätigt wird, werden Cographen im Folgenden nicht betrachtet.

Der für Auswertung bezüglich der modularen Tiefe md interessante Bereich, beschränkt sich somit auf Graphen mit $10 \lessapprox mw \lessapprox 250$. Hier scheinen Graphen mit einer großen modularen Tiefe größere Berechnungszeiten zu verursachen, als Graphen mit einer kleinen modularen Tiefe[38]. Wiederum lässt sich dieses Phänomen anhand der These erläutern, dass die Bäume in einer Baum-Partition während der Conquer-Phase, dem resultierenden modularen Zerlegungsbaum strukturell ähneln. Entsprechend handelt es sich um potenziell tiefere bzw. flachere Bäume, falls das Ergebnis ein tiefer bzw. flacher Baum ist. Die Anzahl der inneren Knoten ist bei einer festen modularen Weite potenziell ähnlich, sodass unterschiedlich hohe Kosten für das Traversieren der Baum-Partition unwahrscheinlich sind. Eine mögliche Erklärung liefert die Funktion `Factorize` (Zeile 2-3), d.h. die bottom-

[38] Für die betrachteten Graphen gilt $md_{max} = \lceil \frac{1000-1}{mw-1} \rceil$ und $md_{min} = \lceil \log_{mw} 1000 \rceil$. Gemäß der Konstruktion der Graphen, können sich die Werte um 1 erhöhen.

up Vergabe von *zombie*-Labels in einem vollen mw-ären Baum mit l Blättern und genau einem *dead*-gelabelten Knoten: Falls der Baum eine maximale Tiefe hat, werden durchschnittlich mehr Vorfahren mit einem *zombie*-Label ausgestattet, als in einem Baum mit einer minimalen Tiefe.

5.3 Abschließende Bemerkungen

Die Experimente in den Abschnitten 5.1 und 5.2 dokumentieren akzeptable sowie gut abzuschätzende Berechnungszeiten für viele Graphen. Einzig für Cographen zeigen sich explosionsartige Anstiege in den Berechnungszeiten, deren Ursachen nicht voll ergründet werden konnten. Diesbezüglich wären Experimente hinsichtlich dem Verhältnis von parallelen zu seriellen Knoten oder der Anzahl innerer Knoten über den Verlauf des Algorithmus denkbar aufschlussreich. Ebenso könnte in weiteren Experimenten zu einer festen modularen Weite untersucht werden, wie sich das Verhältnis von kleinen zu großen Quotientengraphen auf das Verhalten des Algorithmus auswirkt. Motiviert durch den Anwendungsfall der modularen Zerlegung im Kontext parametrisierter Algorithmen bleibt die Frage, ob Graphen grundsätzlich zu längeren Berechnungszeiten führen, wenn deren modulare Weite klein ist. Die bisherigen Ergebnisse deuten an, dass bei der Betrachtung schneller parametrisierter Algorithmen – bei kleiner modularer Weite – zu bedenken ist, dass zumindest leicht erhöhte Kosten bei der Berechnung der modularen Zerlegung anfallen.

Prinzipiell könnte das Verhalten der Funktionen (`DivideMDTree`, `TreeRefinement`, `Factorize`, `BuildSpine`, `ConquerMDTree`) dediziert untersucht werden. Dies ließe sich mittels eines Profilers realisieren und könnte zu stichhaltigen Argumenten für oder gegen die aufgestellten Thesen führen. Ebenso ließen sich technisch ausgerichtete Tests durchführen, welche beispielsweise den Speicherbedarf oder die Rekursionstiefe berücksichtigen.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Literatur

[1] Alfred V. Aho, Michael R Garey und Jeffrey D. Ullman. „The transitive reduction of a directed graph“. In: *SIAM Journal on Computing* 1.2 (1972), S. 131–137.

[2] Shikha Anirban, Junhu Wang und Md Saiful Islam. „Modular Decomposition-Based Graph Compression for Fast Reachability Detection“. In: *Data Science and Engineering* 4.3 (2019), S. 193–207.

[3] Andreas Brandstädt, Van Bang Le und Jeremy P Spinrad. *Graph classes: a survey*. SIAM, 1999.

[4] B-M Bui-Xuan u. a. „Algorithmic aspects of a general modular decomposition theory“. In: *Discrete Applied Mathematics* 157.9 (2009), S. 1993–2009.

[5] Binh-Minh Bui-Xuan. „Tree-representation of set families in graph decompositions and efficient algorithms“. Diss. Université Montpellier II-Sciences et Techniques du Languedoc, 2008.

[6] Binh-Minh Bui-Xuan, Michel Habib und Michaël Rao. „Tree-representation of set families and applications to combinatorial decompositions“. In: *European Journal of Combinatorics* 33.5 (2012), S. 688–711.

[7] Christian Capelle. „Decompositions de graphes et permutations factorisantes“. Diss. Montpellier 2, 1997.

[8] Christian Capelle und Michel Habib. „Graph decompositions and factorizing permutations“. In: *Proceedings of the Fifth Israeli Symposium on Theory of Computing and Systems*. IEEE. 1997, S. 132–143.

[9] Maw-Shang Chang. „Algorithms for maximum matching and minimum fill-in on chordal bipartite graphs“. In: *International Symposium on Algorithms and Computation*. Springer. 1996, S. 146–155.

[10] Michel Chein, Michel Habib und Marie-Catherine Maurer. „Partitive hypergraphs“. In: *Discrete mathematics* 37.1 (1981), S. 35–50.

[11] Derek G Corneil. „Lexicographic breadth first search–a survey“. In: *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer. 2004, S. 1–19.

[12] Derek G Corneil und Richard Krueger. „Simple vertex ordering characterizations for graph search:(expanded abstract).“ In: *Electron. Notes Discret. Math.* 22 (2005), S. 445–449.

[13] David Coudert, Guillaume Ducoffe und Alexandru Popa. „Fully polynomial FPT algorithms for some classes of bounded clique-width graphs“. In: *ACM Transactions on Algorithms (TALG)* 15.3 (2019), S. 1–57.

[14] Alain Cournier und Michel Habib. „A new linear algorithm for modular decomposition“. In: *Colloquium on Trees in Algebra and Programming*. Springer. 1994, S. 68–84.

[15] Elias Dahlhaus, Jens Gustedt und Ross M McConnell. „Efficient and practical algorithms for sequential modular decomposition“. In: *Journal of Algorithms* 41.2 (2001), S. 360–387.

[16] Rodney G Downey und Michael Ralph Fellows. *Parameterized complexity*. Springer Science & Business Media, 2012.

[17] Feodor F Dragan, Falk Nicolai und Andreas Brandstädt. „LexBFS-orderings and powers of graphs“. In: *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer. 1996, S. 166–180.

[18] Jack Edmonds und Rick Giles. „A min-max relation for submodular functions on graphs“. In: *Annals of Discrete Mathematics*. Bd. 1. Elsevier, 1977, S. 185–204.

[19] Andrzej Ehrenfeucht und Grzegorz Rozenberg. „Theory of 2-structures, part I: Clans, basic subclasses, and morphisms“. In: *Theoretical Computer Science* 70.3 (1990), S. 277–303.

[20] Andrzej Ehrenfeucht und Grzegorz Rozenberg. „Theory of 2-structures, part II: Representation through labeled tree families“. In: *Theoretical Computer Science* 70.3 (1990), S. 305–342.

[21] Andrzej Ehrenfeucht u. a. „An O (n2) divide-and-conquer algorithm for the prime tree decomposition of two-structures and modular decomposition of graphs“. In: *Journal of Algorithms* 16.2 (1994), S. 283–294.

[22] Michael R Fellows u. a. „What is known about vertex cover kernelization?“ In: *Adventures Between Lower Bounds and Higher Altitudes*. Springer, 2018, S. 330–356.

[23] Jean-Luc Fouquet, Igor Parfenoff und Henri Thuillier. „An O (n) time algorithm for maximum matching in P4-tidy graphs“. In: *Information processing letters* 62.6 (1997), S. 281–287.

[24] Harold N Gabow. „Data structures for weighted matching and extensions to b-matching and f-factors“. In: *ACM Transactions on Algorithms (TALG)* 14.3 (2018), S. 1–80.

[25] Tibor Gallai. „Transitiv orientierbare graphen“. In: *Acta Mathematica Hungarica* 18.1-2 (1967), S. 25–66.

[26] Vassilis Giakoumakis, Florian Roussel und Henri Thuillier. „On P_4-tidy graphs“. In: *Discrete Mathematics and Theoretical Computer Science* 1 (1997), S. 17–41.

[27] Vassilis Giakoumakis und Jean-Marie Vanherpe. „On extended P4-reducible and extended P4-sparse graphs“. In: *Theoretical Computer Science* 180.1-2 (1997), S. 269–286.

[28] Edgar N Gilbert. „Random graphs“. In: *The Annals of Mathematical Statistics* 30.4 (1959), S. 1141–1144.

[29] Michel Habib, Fabien De Montgolfier und Christophe Paul. „A simple linear-time modular decomposition algorithm for graphs, using order extension“. In: *Scandinavian Workshop on Algorithm Theory*. Springer. 2004, S. 187–198.

[30] Michel Habib und Marie-Catherine Maurer. „On the X-join decomposition for undirected graphs“. In: *Discrete Applied Mathematics* 1.3 (1979), S. 201–207.

[31] Michel Habib und Christophe Paul. „A survey of the algorithmic aspects of modular decomposition“. In: *Computer Science Review* 4.1 (2010), S. 41–59.

[32] Michel Habib, Christophe Paul und Laurent Viennot. „Partition refinement techniques: An interesting algorithmic tool kit“. In: *International Journal of Foundations of Computer Science* 10.02 (1999), S. 147–170.

[33] Michel Habib, Christophe Paul und Laurent Viennoti. „A synthesis on partition refinement: a useful routine for strings, graphs, boolean matrices and automata“. In: *Annual Symposium on Theoretical Aspects of Computer Science*. Springer. 1998, S. 25–38.

[34] Michel Habib u. a. „Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing“. In: *Theoretical Computer Science* 234.1-2 (2000), S. 59–84.

[35] John Hopcroft. „An n log n algorithm for minimizing states in a finite automaton“. In: *Theory of machines and computations*. Elsevier, 1971, S. 189–196.

[36] Lee O James, Ralph G Stanton und Donald D Cowan. „Graph decomposition for undirected graphs“. In: *Proceedings of the Third Southeastern Conference on Combinatorics, Graph Theory, and Computing (Florida Atlantic Univ., Boca Raton, Fla., 1972)*. 1972, S. 281–290.

[37] Stefan Kratsch und Florian Nelles. „Efficient and adaptive parameterized algorithms on modular decompositions“. In: *arXiv preprint arXiv:1804.10173* (2018).

[38] Stefan Kratsch und Florian Nelles. „Efficient parameterized algorithms for computing all-pairs shortest paths“. In: *arXiv preprint arXiv:2001.04908* (2020).

[39] Ross M McConnell und Fabien De Montgolfier. „Linear-time modular decomposition of directed graphs“. In: *Discrete Applied Mathematics* 145.2 (2005), S. 198–209.

[40] Ross M McConnell und 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*. 1994, S. 536–545.

[41] Ross M McConnell und Jeremy P Spinrad. „Linear-time transitive orientation“. In: *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*. 1997, S. 19–25.

[42] Ross M McConnell und Jeremy P Spinrad. „Modular decomposition and transitive orientation“. In: *Discrete Mathematics* 201.1-3 (1999), S. 189–241.

[43] Ross M McConnell und Jeremy P Spinrad. „Ordered Vertex Partitioning“. In: *Discrete Mathematics and Theoretical Computer Science* 4.1 (2000).

[44] Silvio Micali und Vijay V Vazirani. „An O (v| v| c| E|) algoithm for finding maximum matching in general graphs“. In: *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*. IEEE. 1980, S. 17–27.

[45] John H Muller und Jeremy Spinrad. „Incremental modular decomposition“. In: *Journal of the ACM (JACM)* 36.1 (1989), S. 1–19.

[46] Charis Papadopoulos und Constantinos Voglis. „Drawing graphs using modular decomposition“. In: *International Symposium on Graph Drawing*. Springer. 2005, S. 343–354.

[47] Donald J Rose, R Endre Tarjan und George S Lueker. „Algorithmic aspects of vertex elimination on graphs“. In: *SIAM Journal on computing* 5.2 (1976), S. 266–283.

[48] David P Sumner. „Graphs indecomposable with respect to the X-join“. In: *Discrete Mathematics* 6.3 (1973), S. 281–298.

[49] Marc Tedder. „Applications of Lexicographic Breadth-First Search to Modular Decomposition, Split Decomposition, and Circle Graphs“. Diss. 2011.

[50] Marc Tedder u. a. „Simple, linear-time modular decomposition“. In: *arXiv preprint arXiv:0710.3901* (2007).

[51] Marc Tedder u. a. „Simpler linear-time modular decomposition via recursive factorizing permutations“. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2008, S. 634–645.

[52] Douglas Brent West u. a. *Introduction to graph theory*. Bd. 2. Prentice hall Upper Saddle River, 2001.

[53] Ming-Shing Yu und Cheng-Hsing Yang. „A linear time algorithm for the maximum matching problem on cographs“. In: *BIT Numerical Mathematics* 33.3 (1993), S. 420–432.