



Bachelorarbeit

Kreuzungsminimierung in Bäumen durch Knotenduplizieren für hierarchische Layouts

Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Algorithmik
Jonas Klötzl, jonas.kloetzl@uni-tuebingen.de, 2023

Bearbeitungszeitraum: 01. November 2022 - 28. Februar 2023

Betreuer/Gutachter: Prof. Dr. Michael Kaufmann, Universität Tübingen

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Jonas Klötzl (Matrikelnummer 5500688), 28. Februar 2023

Kurzfassung

Das Visualisieren von Graphen ist ein wichtiges Problem der Graphentheorie und findet auch im Alltag vielerlei Anwendungen. In Bereichen wie der Bioinformatik oder bei der Analyse von (sozialen) Netzwerken ist das Visualisieren von Daten sowie deren Auswertung essentiell. Dabei kann es bei großen Datenmengen schnell dazu kommen, dass einzelne Kanten aufgrund von zu vielen Kreuzungen nicht mehr nachverfolgt werden können und dementsprechend Informationen verloren gehen. Das Knotenduplizieren versucht dies durch das Minimieren von Kantenkreuzungen zu verhindern, indem Kopien von einzelnen oder mehreren Knoten erstellt und anschließend in den Graphen eingefügt werden. Dabei stellt das Knotenduplizieren, neben dem Kanten- oder Knotenlöschen nur eine von vielen Methoden dar, die die Kantenkreuzungen in einem Graphen reduzieren. In dieser Bachelorarbeit wird die Methode des Knotenduplizierens dazu verwendet, um für hierarchische Layouts Kreuzungen in Bäumen zu minimieren beziehungsweise komplett zu entfernen. Dabei werden zuerst grundlegende graphentheoretische Begriffe wie die der Knotenduplizierung oder des planaren Graphens definiert und anschließend anhand verwandter Arbeiten bereits existierende Ansätze im Bereich der Kreuzungsminimierung durch Knotenduplizieren präsentiert. Die Vorstellung verschiedener Modelle zum Lösen des Problems soll dabei die eigens konstruierten Algorithmen motivieren, welche sowohl auf ihre Korrektheit als auch auf ihre Laufzeit hin untersucht werden.

Inhaltsverzeichnis

1 Einleitung	9
1.1 Grundbegriffe	10
1.2 Anwendungen	10
1.2.1 Anatomische Strukturen und Zelltypen	10
1.2.2 Soziale Netzwerke	11
1.3 Planare Graphen	12
1.3.1 Definitionen	12
1.3.2 Wie macht man einen Graph planar ?	13
2 Verwandte Arbeiten	15
2.1 Knotenduplizieren in Zeichnungen von 2-schichtigen Graphen	15
2.1.1 Einleitung	15
2.1.2 Entfernen von Kreuzungen mit k Duplizierungen - CRS(k)	16
2.1.3 Entfernen von Kreuzungen mit k Duplizierungsknoten - CRSV(k)	18
2.1.4 Minimierung von Kreuzungen mit k Duplizierungen - CMS(k,M)	18
2.2 Über die planare Duplizierungsdicke von Graphen	19
2.3 Planarisierung von Graphen und ihren Zeichnungen mit Hilfe von Knotenduplizierungen	20
2.4 Ein FPT Algorithmus für bipartites Knotenduplizieren	21
3 Modelle	23
3.1 Modell 1	23
3.2 Modell 2	24
3.3 Modell 3	25
3.4 Modell 4	26
3.5 Diskussion	26
4 Algorithmen zur Duplizierungsminimierung	29
4.1 Vorarbeit	30
4.2 Entfernen von Kreuzungen mit k Duplizierungen - CRS(k)	34
4.2.1 Algorithmus Finde_C*	35
4.2.2 Algorithmus Optimiere_II	39
4.2.3 Algorithmus Erstelle_Baum_ω	41
4.2.4 Beispiel	42
4.3 Entfernen von Kreuzungen mit k Duplizierungsknoten - CRSV(k)	44
4.4 Minimierung von Kreuzungen mit k Duplizierungen - CMS(k,M)	46
4.4.1 Algorithmus Berechne_X	50
4.4.2 Algorithmus Berechne_Möglichkeiten	51
4.4.3 Algorithmus Mische	52
5 Zusammenfassung	55

1 Einleitung

Obwohl das Graphenzeichnen bereits zur Jahrtausendwende ein aktives Forschungsgebiet der Informatik darstellte, ist es in einer Welt, in der immer größere Mengen an Daten generiert werden können, heutzutage nicht minder von Bedeutung. Wie in *Fleischer und Hirsch* [FH01] beschrieben, zeichnen sich Methoden zur Graphbasierten Visualisierung dadurch aus, Daten, die aus in Relation stehenden Objekten bestehen, zu visualisieren. Dabei werden die Objekte als Knoten und die Relationen zwischen den Objekten als Kanten bezeichnet. Obwohl Graphen aus dem Teilgebiet der Graphentheorie stammen, findet man die abstrakten mathematischen Konstrukte, unbeachtet vom Betrachter, überall im Alltag vor (*Fleischer und Hirsch* [FH01]).

Ein Beispiel wäre der Liniennetzplan des lokalen ÖPNV von Tübingen (Abbildung 1.1a). Falls man das Smartphone mit der richtigen App gerade nicht zur Hand hat, ist der Liniennetzplan immer noch das Mittel der Wahl, um herauszufinden, mit welcher Linie das Ziel erreicht werden kann. Dabei werden die Haltestellen als Kreise oder Ovale gezeichnet und die Kanten stehen für die Linien, die zwischen zwei Haltestellen verkehren. Zusätzlich sind die Haltestellen mit ihren Namen und die Linien mit der spezifischen Zuglinie beschriftet.

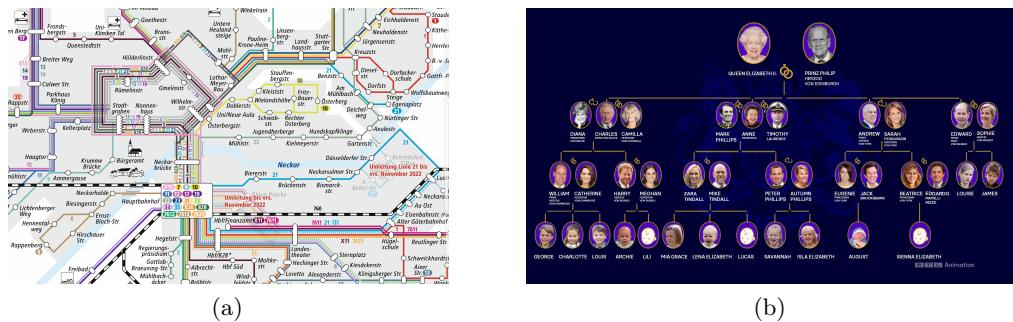


Abbildung 1.1: Verschiedene Arten von Graphen findet man im Alltag beispielsweise
(a) im Busliniennetzplan von Tübingen¹ oder (b) im Stammbaum der britischen Königsfamilie².

¹https://www.swtue.de/fileadmin/user_upload/3Fahrgaeste/TueBus_Liniennetzplan_2022.pdf [abgerufen am 18.02.2023]

²<https://www.n-tv.de/leute/Wurzeln-der-britischen-Royals-Name-im-Krieg-geaendert-wie-deutsch-sind-King-Charles-III-Co--article23578273.html> [abgerufen am 18.02.2023]

Ein weiteres Beispiel sind Stammbäume, wie die der britischen Königsfamilie in Abbildung 1.1b, bei denen die Knoten Personen und die Kanten die Abstammungen zwischen ihnen darstellen.

1.1 Grundbegriffe

Allgemein betrachtet, besteht ein *Graph* $G = (V, E)$ laut *Cormen et al.* [CLRS09] aus einer Menge von Knoten V und einer Menge von Kanten $E \subset V \times V$. Dabei kann eine Kante $e \in E$ zwischen den Knoten $u, v \in V$ sowohl ungerichtet $\{u, v\}$ als auch gerichtet (u, v) sein.

Ein Stammbaum, wie in Abbildung 1.1b betrachtet, wird formal als *Baum* bezeichnet und ist eine Untergruppe eines Graphen, der sich laut *Dasgupta et al.* [DPV06] dadurch auszeichnet, dass er ungerichtet, zusammenhängend und azyklisch ist. Dabei heißt ein ungerichteter Graph *zusammenhängend*, wenn es für alle Knoten $u, v \in V$ einen Pfad von u nach v gibt, und *azyklisch*, wenn kein Pfad von Knoten $v_0 - v_1 - \dots - v_k - v_0$ existiert, sodass man wieder am Startknoten ankommt.

Eine dritte wichtige Klasse von Graphen, deren Anwendung in Abschnitt 1.2.1 beispielhaft betrachtet wird, ist die Klasse der bipartiten Graphen. In *Dasgupta et al.* [DPV06] wird diese wie folgt definiert: Ein *bipartiter Graph* $G(V_1 \cup V_2, E)$ ist ein Graph, dessen Kanten ausschließlich zwischen den zwei Knotenmengen V_1 und V_2 verlaufen, wobei zusätzlich gilt, dass V_1 und V_2 disjunkt sind ($V_1 \cap V_2 = \emptyset$). Somit gibt es keine Kanten zwischen Knoten derselben Knotenmenge und es gilt $(u, v) \notin E$, falls $u, v \in V_1$ oder $u, v \in V_2$.

1.2 Anwendungen

Auch Graphen, denen man im Alltag nicht unbedingt begegnet, spielen im Hintergrund von Systemen in vielerlei Hinsicht eine wichtige Rolle. Wir stellen im Folgenden zwei typische Anwendungen vor.

1.2.1 Anatomische Strukturen und Zelltypen

Wie von *Ahmed et al.* [AAB+22] beschrieben, können Graphen und insbesondere die Klasse der bipartiten Graphen verwendet werden, um Beziehungen zwischen zwei verschiedenen Mengen von Objekten zu visualisieren. So besteht jede Zelle des menschlichen Körpers aus Genen und Proteinen, wobei Zellen nicht aus anderen Zellen und Proteine/Gene nicht aus anderen Proteinen/Genen bestehen (siehe Abbildung 1.2a und 1.2b).

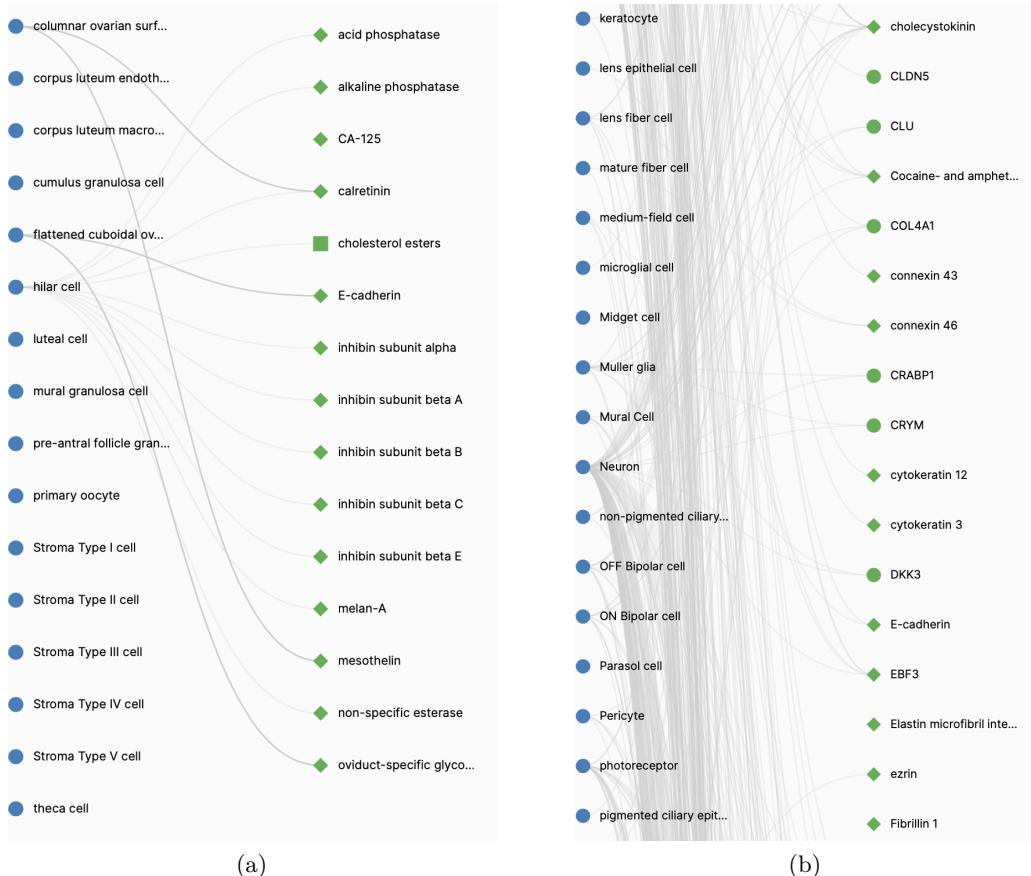


Abbildung 1.2: (a) Alle Zellen mit ihren Proteinen/Genen des Organs Eierstock sowie (b) ein Ausschnitt der Zellen mit ihren Proteinen/Genen für das Organ Auge. Die Abbildung wurde mit Hilfe des Tools von *Paul et al.* [PBH+22] erstellt.

1.2.2 Soziale Netzwerke

Besonders durch das Aufkommen sozialer Netzwerke wie Facebook oder Twitter sowie bei der Kontaktverfolgung in der Corona Pandemie gewann das Visualisieren und Auswerten solcher Netzwerke wieder an Bedeutung. Dabei stehen, wie in *Fleischer* [FH01] beschrieben, zum Beispiel die Knoten für Personen und die Kanten für Relationen wie *befreundet sein mit* oder *hatte physischen Kontakt in den letzten drei Tagen mit*. Cluster-Algorithmen auf Graphen ermöglichen es, zu entscheiden, welche Person am meisten Kontakt zu anderen Personen hatte und somit zuerst geimpft werden sollte, oder welche Person als Freundschaftsvorschlag angezeigt wird. Obwohl es bei den vorangegangenen Beispielen relativ klar war, wie man die Graphen am besten zeichnen sollte (Liniennetzplan: Knoten so nah wie möglich an der geographischen Lage, Stammbaum: Alle Personen der gleichen Generation auf einer

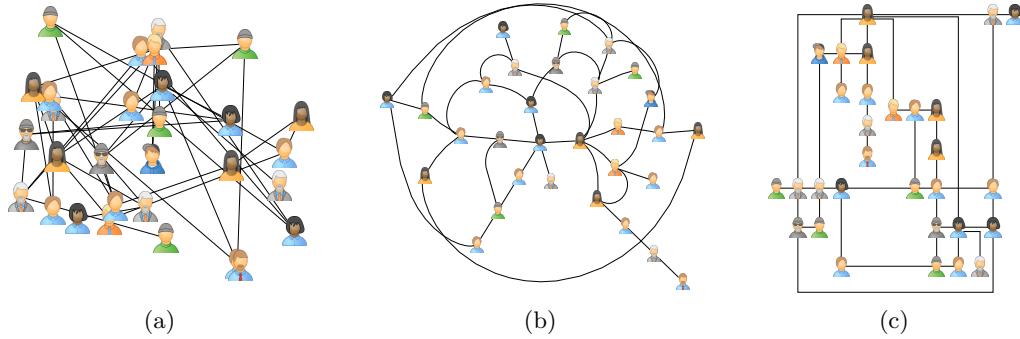


Abbildung 1.3: Verschiedene Zeichnungen desselben, zufällig generierten Ausgangsgraphen (a) und die dazugehörige radiale Zeichnung (b) sowie die optimale Zeichnung laut yEd One-Click-Layout (c). Alle Zeichnungen wurden mit der yEd-Software [yWo] erstellt.

Ebene), ist dies im Fall sozialer Netzwerke und vor allem im Allgemeinen nicht trivial. Wie man am Beispielgraphen in 1.3 sehen kann, gibt es verschiedene Möglichkeiten, die gleichen Informationen wie die Bekanntschaft zwischen Personen zu visualisieren. Werden die Abbildungen in 1.3 betrachtet, so wird festgestellt, dass es einen großen Unterschied macht, wie ein Graph gezeichnet wird. Während man beim zufällig gezeichneten Ausgangsgraphen 1.3a aufgrund der zum Teil sich überdeckenden Knoten und Kanten nur sehr schwer erkennen kann, wer mit wem befreundet ist, funktioniert dies in Abbildung 1.3b besser. Trotzdem gibt es auch hier insbesondere am äußeren Rand der Zeichnung Kreuzungen, die ein Nachverfolgen der Kanten sehr erschweren. Hierbei ist anzumerken, dass radiale Graphen im Allgemeinen für Bäume konzipiert und deshalb für den gegebenen Graphen nicht wirklich geeignet sind. Beim One-Click-Layout hingegen kann man die Freundschaften zwischen den einzelnen Personen sehr leicht nachvollziehen, was auch daran liegt, dass es nur eine einzige Kante gibt, die von einer anderen gekreuzt wird.

1.3 Planare Graphen

1.3.1 Definitionen

In Abbildung 1.3 ist zu sehen, was auch in *Weiskircher* [Wei01] beschrieben wird: Es ist nicht trivial, wie man einen Graphen zeichnen muss, damit er die enthaltenen Informationen am besten darstellt, beziehungsweise zu definieren, was gute und schlechte Zeichnungen von Graphen ausmacht. Wie oben bereits angedeutet, gibt es jedoch eine Eigenschaft von Graphen, die besonders wichtig ist, um Informationen in einer einfach zugänglichen Struktur mit so wenig Kantenkreuzungen wie möglich darstellen zu können: die Planarität. Folgende Definitionen stammen ebenfalls aus *Weiskircher* [Wei01].

Definition 1.1 (Planare Repräsentation). Eine planare Repräsentation D eines Graphen $G = (V, E)$ ist eine Abbildung der Knoten $v \in V$ in verschiedene Punkte in einer Ebene \mathbb{R}^2 und von Kanten $e \in E$ in offene Jordan Kurven mit den folgenden Eigenschaften:

- Für alle Kanten $e \in E$ verbindet die Repräsentation der Kante $e = (v_1, v_2)$ die Repräsentation von v_1 mit der Repräsentation von v_2 .
- Die Repräsentation zweier disjunkter Kanten $e_1 = (v_1, v_2)$ und $e_2 = (v_3, v_4)$ haben keine gemeinsamen Punkte außer bei gemeinsamen Endpunkten.
- Die Repräsentation der Kante $e = (v_1, v_2)$ enthält nicht die Repräsentation von $v_3 \in V$ mit $v_3 \notin \{v_1, v_2\}$.

Die Planare Repräsentation kann nun verwendet werden, um den Begriff Planarer Graph zu definieren:

Definition 1.2 (Planarer Graph). Ein Graph G ist genau dann planar, wenn eine planare Repräsentation von G existiert.

1.3.2 Wie macht man einen Graph planar ?

Wie in Weiskircher [Wei01] beschrieben, gibt es unzählige Herangehensweisen, einen Graphen planar zu machen. Die naheliegendste Methode ist, die Knoten und Kanten des Graphen so zu zeichnen, dass es entweder zu keinen Kreuzungen mehr kommt, oder, falls dies nicht möglich ist, die Kreuzungen zu minimieren.

Aber es gibt auch drastischere Wege, die die im Graphen enthaltene Struktur und Information verändern. Ein Beispiel ist das Knoten- oder Kantenlöschen. Dabei werden so lange Knoten beziehungsweise Kanten gelöscht, bis ein planarer Graph vorliegt.

Eine weitere Methode, einen Graphen planar zu machen, ohne darin enthaltene Informationen zu löschen, ist das Knotenduplizieren. Hierbei werden Kopien von vorhandenen Knoten so eingefügt, dass ein planarer Graph entsteht. Die formale Definition lautet wie folgt (aus Weiskircher [Wei01]):

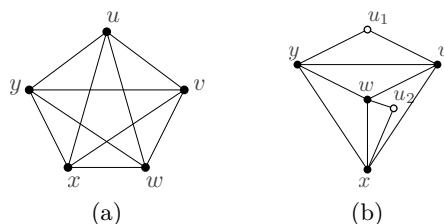


Abbildung 1.4: (a) zeigt den vollständigen Graphen K_5 mit 5 Knoten und 5 Kantenkreuzungen; (b) zeigt eine planare Zeichnung mit Hilfe der Duplizierung des Knotens u .

Definition 1.3 (Knoten duplizieren). *Seien $G = (V, E)$ und $G' = (V', E')$ zwei Graphen. Man erhält G' indem man den Knoten v von G dupliziert, sodass man zwei Knoten v_1 und v_2 erhält, falls die folgenden Bedingungen erfüllt sind:*

$$\begin{aligned} V &= (V' \setminus \{v_1, v_2\}) \cup \{v\}, \\ E &= (E' \setminus \{uv_i \mid u \in V' \text{ und } uv_i \in E' \text{ für } i \in \{1, 2\}\}) \\ &\quad \cup \{uv \mid u \in V \setminus \{v\} \text{ und } (uv_1 \in E' \text{ oder } uv_2 \in E')\}. \end{aligned}$$

2 Verwandte Arbeiten

2.1 Knotenduplizieren in Zeichnungen von 2-schichtigen Graphen

Der folgende Abschnitt basiert auf dem Paper *Splitting Vertices in 2-Layer Graph Drawings* von Ahmed et al. [AAB+22]. Dieses handelt davon, die Kantenkreuzungen in bipartiten Graphen durch Knotenduplizierungen zu minimieren oder zu entscheiden, ob Kreuzungen mit einer bestimmten Anzahl an Duplizierungen beziehungsweise Duplizierungsknoten eliminiert werden können. Anzumerken ist hierbei die Annahme, dass auf einer Seite des bipartiten Graphen weder Knoten dupliziert noch ihre Reihenfolge verändert werden dürfen.

2.1.1 Einleitung

Sei $G = (V_t \cup V_b, E)$ ein bipartiter Graph mit den dazugehörigen Ordnungen π_t und π_b . Im Folgenden bedeutet $u \prec v$, dass der Knoten u unmittelbar links vom Knoten v steht. Ferner sei die Ordnung π_b der unteren Knoten unveränderlich.

Im Paper werden Algorithmen für folgende drei Probleme vorgestellt und deren NP-Vollständigkeit bewiesen. Alle Probleme bekommen sowohl einen bipartiten Graphen $G = (V_t \cup V_b, E)$ mit π_t und π_b als auch einen Duplizierungsparameter k als Eingabe:

- **Entfernen von Kreuzungen mit k Duplizierungen - CRS(k):** Der Algorithmus entscheidet, ob es eine planare Zeichnung des Graphen G mit höchstens k Knotenduplizierungen in V_t gibt.
- **Entfernen von Kreuzungen mit k Duplizierungsknoten - CRSV(k):** Der Algorithmus entscheidet, ob es eine planare Zeichnung des Graphen G mit höchstens k Duplizierungsknoten in V_t gibt.
- **Minimierung von Kreuzungen mit k Duplizierungen - CMS(k, M):** Der Algorithmus entscheidet, ob es eine Zeichnung des Graphen G mit höchstens M Kantenkreuzungen und höchstens k Knotenduplizierungen in V_t gibt.

Der Begriff *Anzahl der Knotenduplizierung* gibt an, wie viele Knoten dupliziert werden. Im Gegensatz dazu gibt die *Anzahl der Duplizierungsknoten* an, wie viele Knoten aus V_t dupliziert werden. Wird beispielsweise der Knoten v i -mal dupliziert, so beträgt die *Anzahl der Duplizierungsknoten* nur 1, die *Anzahl der Duplizierungen* hingegen i . In Abbildung 2.1 wird der Unterschied der beiden Begriffe veranschaulicht.

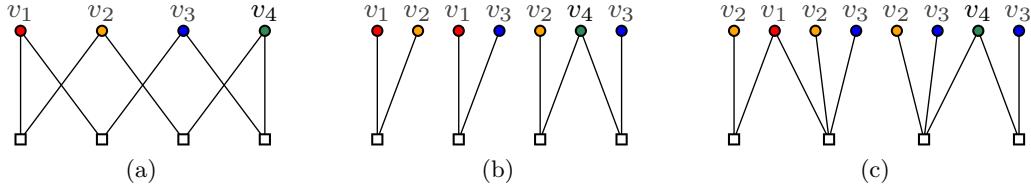


Abbildung 2.1: Ausgangsgraph (a) und dessen planare Zeichnungen mit (b) 3 Knotenduplizierungen (v_1, v_2, v_3) und (c) 2 Duplizierungsknoten (v_2, v_3). Illustration nach Ahmed et al. [AAB+22]

2.1.2 Entfernen von Kreuzungen mit k Duplizierungen - CRS(k)

Der Algorithmus CRS(k) ist rekursiv aufgebaut und betrachtet in jedem Rekursionsschritt die Nachbarn des laut π_b ersten Knotens v_{b_1} in V_b . Sei N^1 eine Menge, die alle Nachbarn des Knotens v_{b_1} enthält, dessen Grad $\deg(v_{b_1}) = 1$ ist und N^+ alle anderen Nachbarn von v_{b_1} . Ob diese dupliziert werden oder nicht, muss im Folgenden nur für die Knoten in N^+ entschieden werden. Dabei wird in drei Fällen unterschieden, wie diese Wahl zu treffen ist. Wenn alle notwendigen Duplizierungen der Nachbarn stattgefunden haben, wird die Ordnung in π_1 gespeichert und zusätzlich v_1 und die dazugehörigen Nachbarn in V_t gelöscht. Somit erhält man einen neuen Graphen $G' = (V'_t \cup V'_b, E')$ und die dazugehörige Ordnung $\pi_b = v_2, \dots, v_{|V_b|}$.

Der Basisfall tritt auf, wenn V_b nur noch einen Knoten enthält. Da die Nachbarn des einen Knotens in V_b nur Grad 1 haben können, müssen keine Knoten dupliziert werden und die Ordnung π_t besteht aus einer beliebigen Anordnung der verbliebenen Knoten in V_t , wobei der erste Knoten festgesetzt sein kann und als u_1^* bezeichnet wird. Wie im vorigen Absatz bereits angedeutet, ist die Wahl der zu duplizierenden Knoten nicht trivial und muss in die nachfolgend näher beschriebenen drei Fälle unterschieden werden.

Fall 1 (v_1 und v_2 haben keinen gemeinsamen Nachbarn): Wie in Abbildung 2.2a zu sehen ist, gibt es keinen Knoten u (wie in Abbildung 2.2b zum Beispiel), der sowohl Nachbar von v_1 als auch von v_2 ist. Somit muss man keine Knoten festsetzen. Gleichzeitig werden alle Knoten in N^+ dupliziert, sodass die jeweilige Kopie nur zum Knoten v_1 inzident ist. Da es trotzdem sein kann, dass bei der vorherigen Iteration ein Knoten u_1^* festgesetzt wurde, muss dies bei der Konstruktion der Ordnung π_t beachtet werden. Man fügt, falls vorhanden, den festgesetzten Knoten u_1^* als ersten Knoten von v_1 und anschließend alle verbleibenden Nachbarn von v_1 in beliebiger Reihenfolge in die Ordnung π_t ein. Somit ergibt sich für die Anzahl der Duplizierungen die Anzahl der Knoten in N^+ plus die Duplizierungen der Instanz G' tieferer Rekursionsstufen: $s = |N^+| + s'$.

2.1. Knotenduplizieren in Zeichnungen von 2-schichtigen Graphen

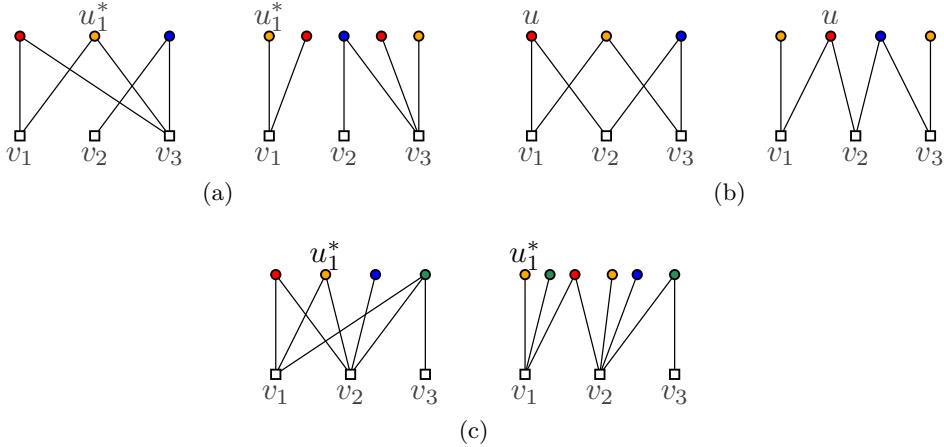


Abbildung 2.2: Beispielgraphen für (a) den Fall 1; (b) den Fall 2; und (c) für den Fall 3. Illustration nach Ahmed et al. [AAB+22]

Fall 2 (v_1 und v_2 haben genau einen gemeinsamen Nachbarn u): Falls im vorherigen Rekursionsschritt der Knoten u nicht festgesetzt wurde, $u_1^* \neq u$, so wird dieser für den nächsten Knoten v_2 festgesetzt. Wie man in Abbildung 2.2b sehen kann, werden die restlichen Knoten in N^+ dupliziert, sodass die jeweilige Kopie nur zum Knoten v_1 inzident ist. Falls u jedoch schon festgesetzt ist und v_1 noch weitere Nachbarn hat, so muss u dupliziert werden, da es sonst unmöglich ist, dass u sowohl letzter als auch erster Nachbar von v_1 ist. Die Konstruktion von π_t verläuft nun identisch zum Fall 1 mit dem Unterschied, dass man für die Anzahl der Duplizierungen bei einem festgesetzten Knoten ungleich u von der Anzahl der Knoten in N^+ noch eins abziehen muss: $s = |N^+| - 1 + s'$.

Fall 3 (v_1 und v_2 haben mehr als einen gemeinsamen Nachbarn): Im Fall, dass v_1 und v_2 nur zwei gemeinsame Knoten u und u' haben und einer davon ein festgesetzter Knoten ist wie zum Beispiel $u_1^* = u$, dann muss der andere Knoten u' der letzte Nachbar von v_1 sein. Behandelt man nun u' wie den einzigen gemeinsamen Nachbarn von v_1 und v_2 , so kann wie in Fall 2 fortgefahren werden. Falls v_1 und v_2 aber mehr als zwei gemeinsame Knoten haben, so ist die Wahl für den festzusetzenden Knoten nicht trivial, was man auch in Abbildung 2.2c sehen kann. Hätte man hier den grünen Knoten für v_2 festgesetzt, so würde eine Duplizierung mehr benötigt und die Duplizierungsanzahl wäre nicht minimal. Dies ist der Grund, weshalb man in diesem Fall den Graph G' ohne festgesetzten Knoten konstruiert und den Knoten auswählt, der nicht von G' als letzter Nachbar von v'_1 vorgesehen wäre. Anschließend werden alle Knoten außer u in N^+ dupliziert und die Ordnung π_t wie im Fall 1 konstruiert. Die Anzahl der Duplizierungen hingegen ist identisch zum Fall 2 mit einer Anzahl an Duplizierungen s von $s = |N^+| - 1 + s'$.

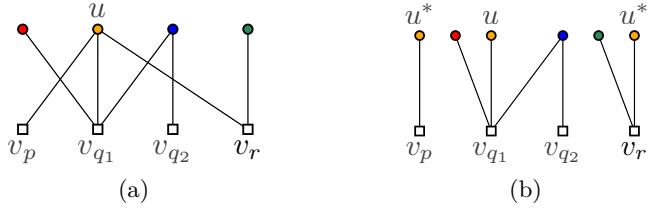


Abbildung 2.3: Ausgangsgraph (a) mit dem gemeinsamen Nachbarn u der Knoten v_p und v_r sowie dessen planare Zeichnung (b) mit den zwei Knoten u^* , die durch zweimaliges Duplizieren des Knoten u entstehen.

2.1.3 Entfernen von Kreuzungen mit k Duplikierungsknoten - CRSV(k)

Diese Methode verläuft ähnlich zu CRS(k) in Abschnitt 2.1.2, wobei hier die Anzahl der Duplikierungsknoten maximal k sein darf. Zudem wird zuerst eine Vorauswahl an Knoten (Preprocessing-Schritt) getroffen wird, die auf jeden Fall dupliziert werden müssen. Dabei wird für zwei nebeneinander liegende Knoten v_p und v_r in V_b mit $v_p \prec \dots \prec v_r$ und gemeinsamem Nachbarn u überprüft, ob dazwischenliegende Knoten v_q mit $v_p \prec v_q \prec v_r$ nur zu u adjazent sind. Ist dies nicht der Fall, so muss der Knoten u so lange dupliziert werden, bis dieser nur noch zu v_p und v_r adjazent ist. Abbildung 2.3 illustriert einen solchen Fall, in dem die obigen Bedingungen gegeben sind, sodass der Knoten u dupliziert werden muss. Falls k^* die Variable ist, in der die Anzahl der benötigten Duplikierungsknoten gespeichert wird, so ist hierbei anzumerken, dass diese nur um eins erhöht wird, da immer nur derselbe Knoten dupliziert wird. Ebenfalls ähnlich zu Abschnitt 2.1.2 werden wieder drei Fälle unterschieden: Falls zwei unmittelbar nebeneinander liegende Knoten v_i und v_{i+1} in V_b mit $v_i \prec v_{i+1}$ keinen gemeinsamen Nachbarn haben, so müssen auch keine Knoten dupliziert werden. Im Fall, dass sie genau einen gemeinsamen Knoten u haben, müssen ebenfalls keine Knoten dupliziert werden, da die zuvor geleistete Vorarbeit im Preprocessing-Schritt dafür sorgt, dass u nur noch v_i und v_{i+1} als Nachbarn haben. Andernfalls, wenn die Knoten v_i und v_{i+1} mehr als einen gemeinsamen Nachbarn haben, so werden alle Knoten bis auf einen einzigen Knoten u dupliziert. Zuletzt wird die Ordnung π_t konstruiert. Dabei werden für alle Knoten $v_i \in \{v_1, v_2, \dots, v_{|V_t|}\}$ die Nachbarn in π_t abgespeichert, wobei auf die richtige Platzierung der Knoten u , die beiden nebeneinanderliegenden Nachbarn v_i und v_{i+1} , geachtet wird.

2.1.4 Minimierung von Kreuzungen mit k Duplikierungen - CMS(k, M)

Anders als bei den vorangegangenen Problemen CRS(k) (2.1.2) und CRSV(k) (2.1.3) bekommt der Algorithmus nicht nur einen Graph $G = (V_t \cup V_b, E)$ und die Duplikationszahl i als Eingabe, sondern zusätzlich eine Zahl M , die die Anzahl der maximalen Kantenkreuzungen angibt. Der vorgestellte Algorithmus von Ahmed et al. [AAB+22] beginnt damit, Knoten aus V_t auszuwählen, die gesplittet werden sollen.

Da die Zahl der Duplizierungen durch den Wert von k beschränkt ist, werden k Knoten aus V_t ausgewählt, die gesplittet werden sollen. Insbesondere ist es demnach auch möglich, dass einzelne Knoten mehrmals ausgewählt werden. All diese Möglichkeiten \mathcal{X} werden abgespeichert, damit unter ihnen später die optimale Lösung gesucht werden kann. Die i Duplizierungen für einen festgelegten Knoten $v \in V_t$ werden nun realisiert, indem dieser durch die Menge der Kopien $D = \{v_1, \dots, v_{i+1}\}$ ersetzt wird. Dabei definiert Ahmed et al. [AAB+22] die Teilmenge von Knoten aus V_b , die gemäß π_b geordnet sind als *Nachbarschaft* $N(v)$ eines Knotens $v \in V_t$. Da bis jetzt die duplizierten Knoten immer noch die gleiche *Nachbarschaft* besitzen wie der Ausgangsknoten v , muss dafür gesorgt werden, dass ein Knoten aus V_b nicht mit zwei Knoten in D adjazent ist. Als erstes werden die Teilmengen von $N(v)$ gemäß \mathcal{X} konstruiert, indem die Menge $N(v)$ in alle gültigen $i+1$ Teilmengen zerlegt und anschließend in \mathcal{Y} abgespeichert wird. Somit enthält die Menge \mathcal{Y} für jede Wahl an duplizierten Knoten in \mathcal{X} die verschiedenen Möglichkeiten, diese mit ihren Nachbarn über eine Kante zu verbinden. Als nächstes kombiniert der Algorithmus die Werte in \mathcal{X} und \mathcal{Y} insoweit, dass für jede Menge in \mathcal{X} alle möglichen Aufteilungen der *Nachbarschaft* in \mathcal{Y} zugeordnet und schließlich in \mathcal{Z} gespeichert werden. Zuletzt wird für jede Zuordnung in \mathcal{Z} jede mögliche Platzierung der duplizierten Knoten zwischen den vorhandenen nicht-duplizierten Knoten in V_t betrachtet und die Anzahl der Kantenkreuzungen M^* berechnet. Der konstruierte Graph mit der kleinsten Anzahl an Kantenkreuzungen wird dazu verwendet, um zu entscheiden, ob der Algorithmus TRUE (für $M^* \leq M$) oder FALSE (für $M^* > M$) ausgibt.

2.2 Über die planare Duplizierungsdicke von Graphen

Eppstein et al. [EKK+17] untersuchen in ihrem Paper *On the Planar Split Thickness of Graphs* unter anderem die k -*Duplizierbarkeit* von vollständigen Graphen und bipartiten Graphen. Dabei gilt ein Graph G als k -*duplizierbar* in G^k , falls es möglich ist, eine planare Repräsentation des Graphen G durch höchstens k -maliges Duplizieren jeden Knotens zu erhalten. Sei \mathcal{G} eine Graph Invariante, dann hat eine k -*Duplizierung* (jeder Knoten wird höchstens k -Mal dupliziert) des Graphen G die Eigenschaft \mathcal{G} genau dann, wenn G k -*duplizierbar* in \mathcal{G} ist. Das kleinste k , das noch die Eigenschaft der k -*Duplizierbarkeit* erfüllt, wird dann als \mathcal{G} *Duplizierungsdicke* eines Graphen G definiert.

Für die vollständigen Graphen formulieren Eppstein et al. [EKK+17] unter anderem den folgenden Satz, wobei $f(G)$ als planare Duplizierungsdicke des Graphen G definiert ist (*Theorem 1*): Falls $n \leq 4$, dann ist $f(K_n) = 1$, und falls $5 \leq n \leq 12$, dann ist $f(K_n) = 2$. Andernfalls gilt $f(K_n) = \lceil n/6 \rceil$.

Es folgen weitere Aussagen zu der k -*Duplizierbarkeit* vollständiger bipartiter Graphen $K_{m,n}(V_1 \cup V_2, E)$ mit $m = |V_1|$ und $n = |V_2|$: Lemma 3: Die Graphen $K_{5,16}$, $K_{6,10}$ und $K_{7,8}$ sind 2-duplizierbar und ihre 2-Duplizierungs-Graphen sind Quadrangulationen. Dies bedeutet, dass die vollständigen bipartiten Graphen $K_{m,n}$, mit $m = 5, 6, 7$ die größtmöglichen Graphen mit einer planaren Duplizierungsdicke von 2 sind. In Ab-

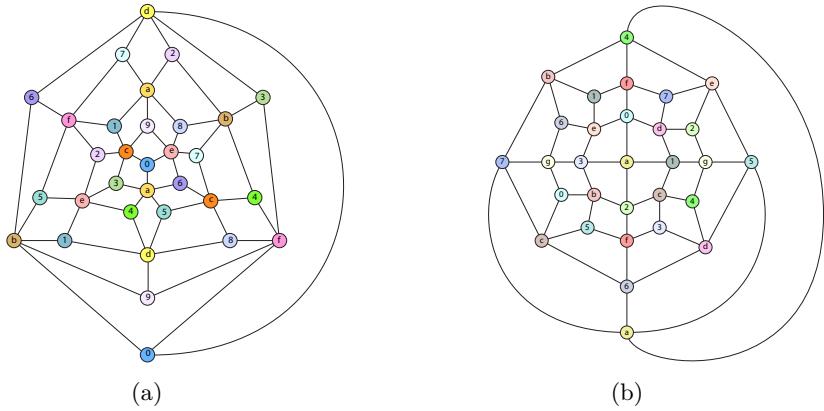


Abbildung 2.4: Planare 2-Duplizierungsgraphen der bipartiten Graphen (a) $K_{6,10}$ und (b) $K_{7,8}$. Illustration aus Eppstein et al. [EKK+17].

bildung 2.4 sind die dazugehörigen planaren Graphen von $K_{6,10}$ und $K_{7,8}$ zu sehen. Außerdem gelte die folgende Aussage (*Theorem 2, Corollary 1*): Jeder vollständige bipartite Graph $K_{m,n}$ ist 2-duplizierbar genau dann, wenn er ein Teilgraph von $K_{4,n}$, $K_{5,16}$, $K_{6,10}$ und $K_{7,8}$ ist oder wenn gilt $nm \leq 4(n+m) - 4$. Anschließend zeigen Eppstein et al. [EKK+17] durch eine Reduktion von einer Variante von 3-SAT (genauer PLANAR CYCLE 3-SAT) auf einen Graphen, dass das Erkennen von 2-duplizierbaren Graphen NP-schwer ist.

Zuletzt beweisen Eppstein et al. [EKK+17], dass die Approximierung eines Werts für die planare Graphendicke mit einem konstanten Faktor möglich ist.

2.3 Planarisierung von Graphen und ihren Zeichnungen mit Hilfe von Knotenduplizierungen

Anders als das Paper *Splitting Vertices in 2-Layer Graph Drawings* von Ahmed et al. [AAB+22] (Abschnitt 2.1) beschränkt sich das Paper *Planarizing Graphs and their Drawings by Vertex Splitting* von Nickel et al. [NNS+23] nicht auf die Klasse der bipartiten Graphen.

Um für allgemeine Graphen $G = (V, E)$ eine planare Zeichnung Γ zu erstellen, betrachten die Autoren dabei im Wesentlichen zwei Probleme: Das erste Problem beschäftigt sich mit der *Kandidaten Auswahl* eines Graphen $G = (V, E)$, dessen Zeichnung Γ gegeben ist. Für einen gegebenen ganzzahligen Wert k soll eine Menge $S \subset V$ an höchstens k Knoten gefunden werden, sodass die Zeichnung Γ des Graphen $G[V \setminus S]$ planar ist. Aus dem Graphen in Abbildung 2.5a werden zum Beispiel die bunten runden Knoten mit ihren dazugehörigen Kanten gelöscht, sodass der Graph in Abbildung 2.5b entsteht.

Als nächstes müssen diese Knoten dupliziert und anschließend wieder in die Zeichnung

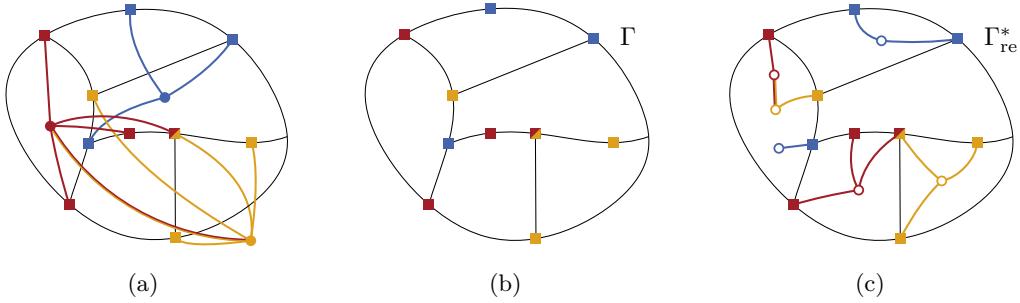


Abbildung 2.5: Ausgangsgraph (a) und dessen planare Zeichnungen Γ ohne die Knoten der *Kandidaten Auswahl* (b) sowie Γ_{re}^* mit den wiedereingebetteten Knoten (c). Abbildung aus Nickel et al. [NNS+23].

Γ eingebettet werden. Dazu lässt sich das zweite Problem formulieren, die *Wiedereinbettung der Duplizierungsmenge*. Seien zusätzlich zum Ausgangsgraphen $G = (V, E)$ die Ergebnisse aus dem Problem *Kandidaten Auswahl* mit der Menge an Kandidaten $S \subset V$, sowie der Zeichnung Γ des resultierenden Subgraphen $G[V \setminus S]$ gegeben. Ist es möglich einen Graphen G^* zu konstruieren, indem höchstens k Knoten aus S mit $k \geq |S|$ mindestens einmal dupliziert werden, sodass dessen Zeichnung Γ_{re}^* im reduzierten Fall $G^*[V \setminus S]$ äquivalent ist zur Zeichnung Γ von $G[V \setminus S]$? Abbildung 2.5c zeigt die Zeichnung der Einbettung der aus dem Ausgangsgraphen gelöschten Knoten S . Außerdem präsentieren Nickel et al. [NNS+23] einen FPT-Algorithmus (*fixed-parameter tractable algorithm*) für das Problem *Wiedereinbettung der Duplizierungsmenge* sowie Beweise für die NP-Vollständigkeit beider Probleme. Dabei benutzen die Autoren für die *Kandidaten Auswahl* eine Reduktion von PLANAR 3-SAT und für das zweite Problem zeigen Nickel et al. [NNS+23], dass *Wiedereinbettung der Duplizierungsmenge* eine Generalisierung des NP-vollständigen FACE COVER Problems ist.

2.4 Ein FPT Algorithmus für bipartites Knotenduplizieren

Das Paper *An FPT Algorithm for Bipartite Vertex Splitting*, von Ahmed et al. [AKK23] kombiniert das Vorgehen von Nickel et al. [NNS+23] mit einer der Problemstellungen des Papers *Splitting Vertices in 2-Layer Graph Drawings* von Ahmed et al. [AAB+22]. Dabei wird für das Problem *Entfernen von Kreuzungen mit k Duplizierungsknoten* CRSV(k) für einen bipartiten Graphen $G = (T \cup B, E)$ entschieden, ob es möglich ist durch das Duplizieren von k Knoten aus der unteren Schicht B eine planare 2-Layer Zeichnung zu erstellen. Hierbei ist zu beachten, dass im Gegensatz zum Paper von Ahmed et al. [AAB+22] die Reihenfolge der Knoten in B nur indirekt durch Wiedereinfügen von duplizierten Knoten, nicht aber direkt durch Verschieben einzelner Knoten verändert werden kann. Genauso wie Nickel et al. [NNS+23] präsentieren Ahmed et al. [AAB+22] einen FPT-Algorithmus, der obiges Problem

Kapitel 2. Verwandte Arbeiten

löst. Dazu beweisen die Autoren folgendes Theorem: *Für einen gegebenen bipartiten Graph $G = (T \cup B, E)$ kann das $\text{CRSV}(k)$ Problem in Zeit $2^{\mathcal{O}(k^6)} \cdot m$ entschieden werden, wobei m die Anzahl der Kanten in G darstellt.* Für den Beweis verwenden Ahmed et al. [AKK23] eine Problemkern-Reduktion. Diese versucht den Graphen G zu reduzieren, sodass der daraus entstehende schwer zu berechnende Teilgraph dazu verwendet werden kann, einen langsamem, dafür aber exakten Algorithmus darauf anzuwenden.

3 Modelle

In diesem Kapitel werden vier verschiedene Modelle präsentiert, die versuchen, das einseitig mit einem Baum erweiterte Problem aus Kapitel 2.1 zu lösen, und die auf der Methode von *Ahmed et al.* [AAB+22] aufbauen. Bei deren Anwendung werden, wie in Abbildung 3.1 beispielhaft dargestellt, zwar Kreuzungen im 2-Layer Graphen beseitigt, gleichzeitig aber entstehen neue Kreuzungen im darüberliegenden Baum. Dabei zeigen die vier Modelle verschiedene Möglichkeiten auf, wie man die Kreuzungen im 2-Layer Graphen beseitigen und gleichzeitig die restlichen Kreuzungen im Baum minimieren und falls möglich beseitigen kann. Der Grund für die einseitige Baumerweiterung wird im Abschnitt 4.1 näher motiviert.

3.1 Modell 1

Das Vorgehen beim ersten Modell ist, den Algorithmus von *Ahmed et al.* [AAB+22] ohne Modifikation auf den Graphen anzuwenden und danach einen neuen Baum oberhalb der Blätter aufzubauen. Dabei geht man so vor, dass für jedes Blatt l_i der tiefste gemeinsame Vorfahre (*Lowest Common Ancestor, LCA*) für den jeweils linken und rechten Nachbarn berechnet wird und anschließend der tiefste davon für

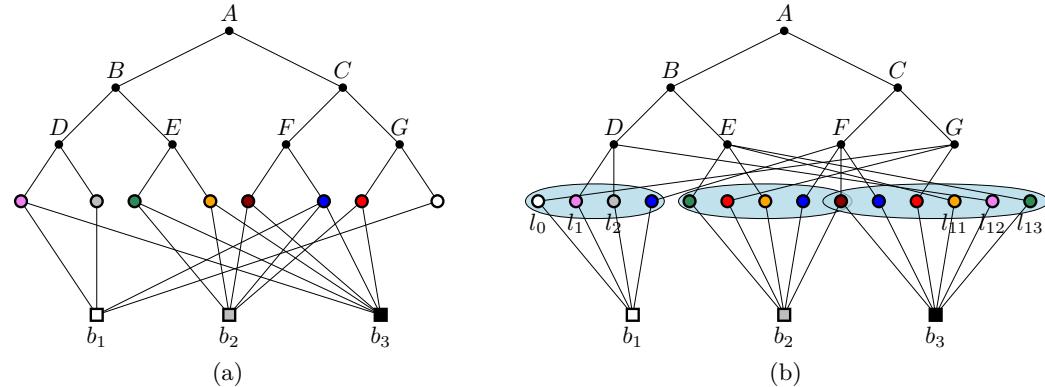


Abbildung 3.1: (a) Beispielinstanz eines Graphen, dessen Kreuzungen beziehungsweise beseitigt werden sollen; (b) Beispielinstanz nach dem Anwenden des Algorithmus von *Ahmed et al.* [AAB+22]. Dupliizierte Knoten werden durch dieselben Farben dargestellt.

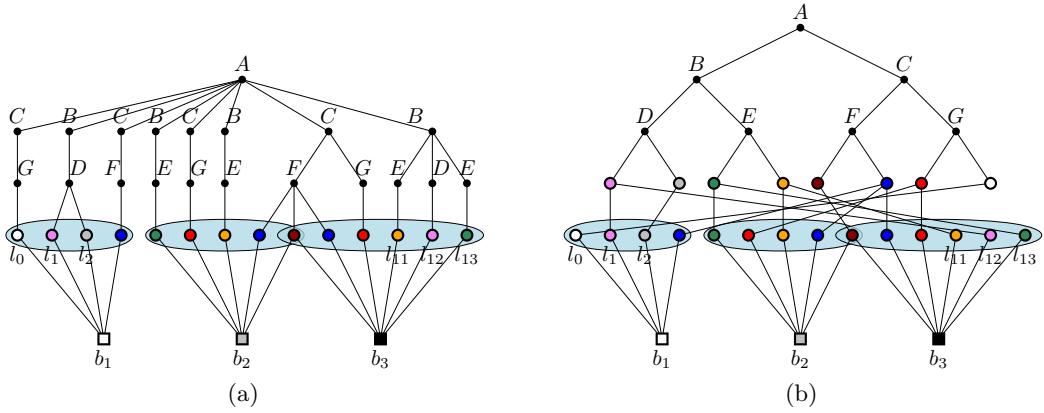


Abbildung 3.2: Beispielhaftes Anwenden der beschriebenen Vorgehensweisen auf den Ausgangsgraphen von Abbildung 3.1a mithilfe des (a) Modells 1 und des (b) Modells 2.

den Knoten l_i in λ_i gespeichert wird. Dies ermöglicht, dass nicht einzelne Pfade für jedes Blatt konstruiert werden, sondern dass nur ein neuer Pfad für die Knoten, die niedriger als λ_i sind, benötigt wird und dieser dann an den Knoten λ_i angehängt werden kann. Vorteile dieses Modells sind, dass zum einen alle Kreuzungen entfernt werden können und zum anderen, dass keine weitere Ebene hinzugefügt wurde und der Baum sehr übersichtlich ist. Gleichzeitig ist festzuhalten, dass sich die Anzahl an Knoten im Baum im Normalfall vervielfachen kann.

Betrachtet man den Knoten l_{11} im Beispiel von Abbildung 3.2a, so wäre der LCA von l_{11} zum linken Knoten l_{10} der Wurzelknoten A und der LCA von l_{11} zum rechten Knoten l_{12} der Knoten B . Es wird folglich ein Knoten gespart, da man l_{11} am Knoten B anhängen kann.

Gleichzeitig könnte man eine Zeichnung mit noch weniger duplizierten Knoten erhalten, falls die Verschiebung von Knoten mit demselben Nachbarn b_i in der unteren Schicht des bipartiten Graphen erlaubt werden würde. Beim Vertauschen der Blätter l_5 und l_6 könnte beispielsweise das Blatt l_6 an den Knoten E angehängt und somit zwei Duplikationen im Baum vermieden werden.

3.2 Modell 2

Beim zweiten Modell ist der Plan, dass man ebenfalls wie auch bei den Modellen 3 und 4 zuvor den Algorithmus von Ahmed et al. [AAB+22] anwendet und anschließend eine komplett neue Ebene aufbaut, die die Blätter des Ausgangsgraphen enthält. Zusätzlich werden alle alten Blätter des Ausgangsgraphen mit den Blättern der neuen Ebene verbunden. Abgesehen davon kann man nun noch versuchen, die Kreuzungen zu verringern, indem man die Ordnung des Baums durch eine *Vertauschen*-Funktion ändert. Würde man beispielsweise auf den Knoten C die *Vertauschen*-Funktion

anwenden, so wäre die Reihenfolge der neuen Schicht nicht mehr *pink* \prec *grau* \prec *grün* \prec *gelb* \prec *braun* \prec *blau* \prec *rot* \prec *weiß*, sondern *pink* \prec *grau* \prec *grün* \prec *gelb* \prec **rot** \prec **weiß** \prec **braun** \prec **blau**. Insbesondere bei großen Bäumen führt dies aber zu einer hohen Laufzeit, da alle Kombinationen von Vertauschungen auf ihre resultierende Kreuzungsanzahl überprüft werden müssen. Für einen Knoten u mit j Kindern gibt es nämlich $j!$ verschiedene Reihenfolgen, wie diese durch die *Vertausche*-Funktion angeordnet werden können.

Anders als beim ersten Modell werden hier nicht alle Kantenkreuzungen entfernt, sondern aus der Baumstruktur verlagert. Dies sorgt zwar dafür, dass der Baum wieder in seinem ursprünglichen Zustand ist, die Anzahl der Kantenkreuzungen werden aber dadurch unter Umständen nicht viel geringer, wie man in Abbildung 3.2a sehen kann. Gleichzeitig ist die die Anzahl der duplizierbaren Knoten in jedem Fall durch die Anzahl der Blätter des Ursprungbaums gegeben.

3.3 Modell 3

Als Erweiterung des zweiten Modells erlaubt das dritte Modell das Hinzufügen einer weiteren Ebene, die nun aus Duplizierungen der unteren Schicht des bipartiten Graphen besteht. Im Beispiel von Abbildung 3.3a also die Knoten b_1, b_2 und b_3 . Dies bedeutet, dass die Vorgehensweise aus Abschnitt 3.2b dahingehend erweitert wird, dass entschieden werden muss, wie oft welche Knoten von (b_1, b_2, b_3) dupliziert werden. Anschließend ist es auch hier erforderlich, zu überprüfen, welche Anordnung der duplizierten Knoten zur kleinsten Anzahl an Kreuzungen führt. Insbesondere wird folglich auch die Verschiebung dieser duplizierten Knoten erlaubt, was dazu

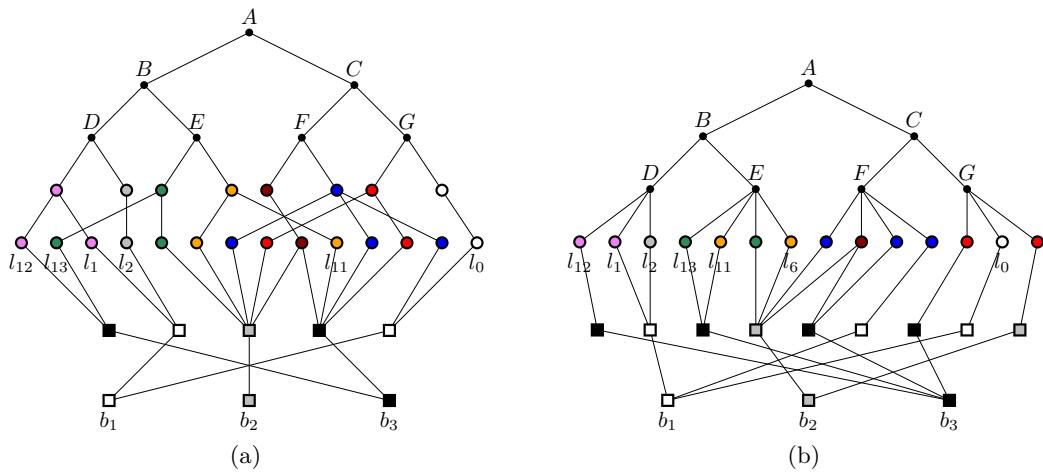


Abbildung 3.3: Beispielhaftes Anwenden der beschriebenen Vorgehensweisen auf den Ausgangsgraphen von Abbildung 3.1a mithilfe des (c) Modells 3 und des (d) Modells 4.

führt, dass die Blätter l_0 bis l_{13} jetzt ebenfalls komplett frei verschoben werden können. Es müsste also untersucht werden, inwiefern dann noch der Algorithmus von *Ahmed et al.* [AAB+22] benötigt wird. Beim Betrachten der Knoten l_{12} und l_1 in Abbildung 3.3a fällt nämlich auf, dass ein Knoten eingespart werden könnte, wenn l_1 direkt mit dem linken Duplikationsknoten von b_3 verbunden würde.

Der Nachteil dieser Methode, dass Kreuzungen in zwei Ebenen entstehen können, ist gleichzeitig auch ein Vorteil, da dadurch die Anzahl der Kreuzungen auf den einzelnen Ebenen jeweils kleiner ist. Betrachtet man die dazugehörige Abbildung 3.3a, so kann man sehen, dass die Verteilung der Kreuzungen auf die zwei Ebenen dazu führt, dass man die einzelnen Kanten im Vergleich zum Modell 2 in Abbildung 3.2b wieder nachverfolgen kann.

3.4 Modell 4

Im letzten Modell ist die extra Ebene mit den duplizierten Blättern aus Modell 2 und 3 nicht mehr vorhanden (siehe Abbildung 3.3b). Dafür gibt es wie in Modell 3 eine zusätzliche Ebene mit Duplikaten der unteren Schicht V_b des bipartiten Graphen, die man frei verschieben kann. Man versucht so viele Knoten aus V_b zu duplizieren, sodass es nur noch zwischen dem neu hinzugefügten Level an Knoten und V_b zu Kreuzungen kommt.

Wie man in Abbildung 3.3b sehen kann, werden also wie auch schon in Modell 3.3a die Knoten b_1 , b_2 und b_3 dupliziert und auf einer neuen Ebene angeordnet. Dabei fällt ebenfalls auf, dass es Blätter gibt, die durch *Ahmed et al.* [AAB+22] nicht dupliziert werden müssen. Betrachtet man die Kinder des Knoten E in Abbildung 3.3b, so wird festgestellt, dass der Knoten l_6 eingespart werden kann, falls der Knoten l_{11} mit dem linken Duplikationsknoten von b_2 verbunden wird. Somit müsste auch hier untersucht werden, inwiefern der Algorithmus von *Ahmed et al.* [AAB+22] benötigt wird.

Obwohl die Kreuzungen nur auf einer Ebene vorkommen, ist es auch hier noch möglich, die einzelnen Kanten nachzuverfolgen, was im Vergleich zu Modell 2 eine deutliche Verbesserung darstellt. Gleichzeitig ist es immer noch so, dass Kreuzungen auf der untersten Ebene, wie zum Beispiel auf Ebene fünf der Abbildung 3.3b, vorhanden sind. Es ist abzusehen, dass insbesondere bei größeren Graphen mit vielen Knoten in V_b die Kanten genauso wie in Modell 2 oder Modell 3 nicht mehr nachverfolgt werden können.

3.5 Diskussion

Vergleicht man die vier vorgestellten Modelle miteinander, so fällt auf, dass das Modell 1 das einzige ist, bei dem alle Kantenkreuzungen immer entfernt werden. Zudem ist es für den Betrachter am einfachsten zu verstehen, da keine neue Ebene hinzugefügt wird. Falls die Knoten im Baum des Graphen höhere Relevanz als die

Knoten des bipartiten Graphen haben, so ist unter Umständen das Modell 4 am besten, da es hier nur auf der *unwichtigeren* Ebene Kantenkreuzungen gibt.

In dieser Arbeit soll jedoch jedes Level und jede Kantenkreuzung gleich gewertet werden, was bedeutet, dass eine Kreuzung im bipartiten Teil des Graphen genauso schlecht ist wie eine, die im Baum vorkommt. Außerdem soll das Resultat der Methode von *Ahmed et al.* [AAB+22] auf jeden Fall dazu verwendet werden, um darauf aufbauend einen Algorithmus zu entwickeln, der die Kantenkreuzungen im gegebenen Graphen minimiert.

Aufgrund der oben genannten Argumente wird deshalb im folgenden Kapitel 4 eine Erweiterung des Modells 1 vorgestellt. Wie im Abschnitt 3.1 bereits erwähnt, geschieht dies durch eine spezielle Anordnung der Blätter, die dafür sorgt, dass die Anzahl der zu duplizierenden Baumknoten minimiert wird, sodass daraus ein planarer Graph gezeichnet werden kann.

4 Algorithmen zur Duplizierungsminimierung

Im folgenden Kapitel wird eine Erweiterung der Methode von *Ahmed et al.* [AAB+22] vorgestellt, die auf dem Modell 1 aus Kapitel 3 aufbaut. Ziel ist es dabei, die zugrunde liegende Hierarchie der Zellen in Form eines Baums miteinzubeziehen. Abbildung 4.1 zeigt zum Beispiel den dazugehörigen Baum des Eierstocks. Es fällt auf, dass der Baum sowie der bipartite Graph selbst für eine kleine Anzahl der Knoten schnell sehr unübersichtlich werden kann. Insbesondere wenn die Anzahl der Knoten und Kanten größer werden, wie zum Beispiel beim Organ Thymus, dessen bipartiter Graph aus 552 Knoten besteht, wird es sehr schwierig, individuelle Kanten nachzuverfolgen.

Dies motiviert zum einen das Anwenden des Algorithmus von *Ahmed et al.* [AAB+22], um die Kreuzungen zwischen den Knoten des bipartiten Graphen (in Abbildung 4.1 zwischen den blauen und grünen Knoten) zu minimieren und im besten Fall zu

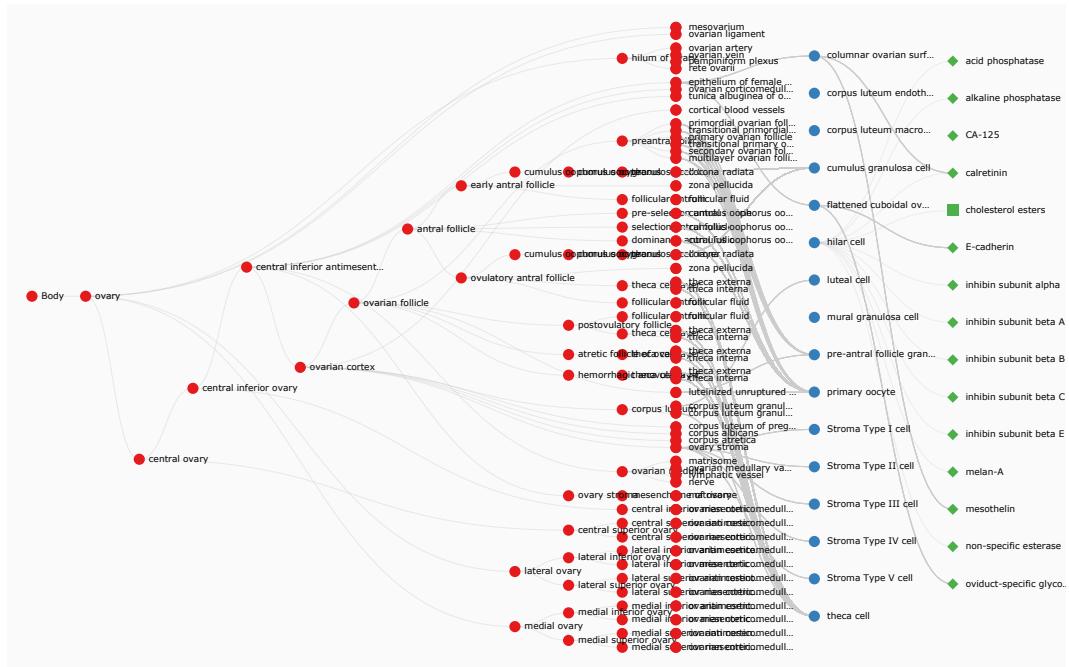


Abbildung 4.1: Baum und bipartiter Graph für das Organ Eierstock. Die Abbildung wurde mit Hilfe des Tools von *Paul et al.* [PBH+22] erstellt.

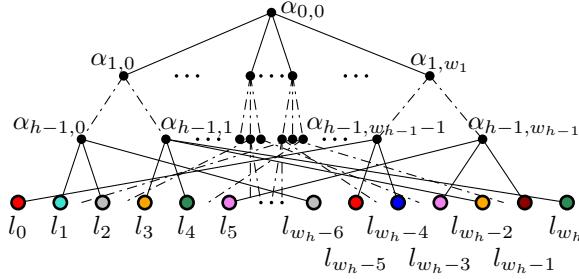


Abbildung 4.2: Instanz eines Graphen T mit den Blattknoten l_j und den Baumknoten $\alpha_{i,j}$. Man beachte, dass der zweite Index unabhängig von der Ordnung π ist, und dass der erste Index das Level angibt, auf welchem sich der Baumknoten α befindet.

beseitigen; zum anderen motiviert dies, ebenso die Kreuzungen zwischen der oberen linken Schicht des bipartiten Graphen und der rechten Schicht des Baums (in Abbildung 4.1 zwischen den blauen und roten Knoten) zu minimieren beziehungsweise im besten Fall zu beseitigen.

Es folgt eine formelle Definition des Problems (Abschnitt 4.1) sowie die Vorstellung der Algorithmen zur Duplizierungsminimierung (Abschnitt 4.2, 4.3 und 4.4).

4.1 Vorarbeit

Im ersten Schritt wird davon ausgegangen, dass der Algorithmus von *Ahmed et al.* [AAB+22] angewendet und anschließend die untere Schicht des bipartiten Graphen gelöscht wird. Das bedeutet, dass ein Baum entsteht, dessen duplizierte Blätter aufgrund der fehlenden unteren Schicht frei bewegen können. Die dazugehörige Definition sieht wie folgt aus:

Definition 4.1 (Baum mit Ordnung). *Sei $T = (V, E, \pi)$ ein Baum mit Blättern $l \subset V$, wobei gilt, dass sowohl innere Knoten als auch Blätter mehrfach vorkommen dürfen. Zusätzlich unterliegen die Blätter einer Ordnung, die in π definiert ist. Hierbei bedeutet $l_i \prec l_j$, dass Blatt l_i links von Blatt l_j steht. Ferner ist l_{w_k} das Blatt auf Level k , das am weitesten rechts steht.*

Definition 4.2 (Level eines Baums). *Wir definieren die Level eines Baums mit Ordnung als den Abstand eines Knotens zur Wurzel. Falls v der Wurzelknoten ist, so beträgt das Level 0, ansonsten ist das Level definiert als das Level des Elternknotens + 1. Geht man von einem vollen Baum aus, bei dem alle Level gefüllt sind, so haben alle Blätter das maximale Level h . Zusätzlich bezeichnet man mit den Kanten des Level i alle Kanten, die die Knoten des Level $i-1$ und i verbinden. Die vom Wurzelknoten ausgehenden Kanten haben somit Kantenlevel 1 und die Kanten, an denen die Blätter befestigt sind, das Level h .*

Algorithmus 1: Optimiere $_{\pi}$

(Algorithmus zum Erstellen einer optimalen Ordnung)

Input : Graph mit Ordnung und Wurzel als Startknoten $(T(V, E, \pi), v)$ **Output :** optimale Ordnung π

```

1  $\pi \leftarrow \emptyset$ 
2  $S \leftarrow v$ 
3 while  $S \neq \emptyset$  do
4    $v \leftarrow \text{dequeue } S$ 
5   foreach Kind  $c$  von  $v$  von links nach rechts do
6     if  $c == \text{Blatt}$  then
7       |  $\pi \leftarrow \pi \prec c$ 
8     end
9      $S \leftarrow \text{enqueue } c$ 
10   end
11 end
12 return  $\pi$ 

```

Wie in Abbildung 4.2 zu erkennen, entstehen nun Kreuzungen zwischen den Leveln h und $h - 1$. Da wie oben bereits angemerkt die Blätter in diesem Fall frei verschoben werden können, kann durch eine in-order Baumtraversierung die Ordnung π der Blätter gefunden werden, sodass keine Kreuzungen mehr entstehen. Algorithmus 1 zeigt eine Implementierung mit Hilfe von Breadth-First-Search.

Satz 4.1. *Algorithmus 1 benötigt für die Erzeugung eines kreuzungsfreien Baums mit Ordnung eine Laufzeit von $\mathcal{O}(n)$.*

Beweis. Da wir als Eingabe einen Baum mit Ordnung ohne duplizierte innere Knoten haben, ist sichergestellt, dass durch die Traversierung diese Kreuzungsfreiheit nicht verändert wird. Eine Charakteristik der Traversierung mittels BFS ist, dass der Baum Ebene für Ebene durchlaufen wird. Ist der Algorithmus nun auf Ebene $h - 1$ angelangt, werden immer alle Kinder eines Knotens, also die Blätter, nacheinander in die Ordnung π eingefügt. Insgesamt sind folglich nach dem Anwenden des Algorithmus 1 alle Blätter eines Knotens nebeneinander in π abgespeichert. Insbesondere stellt der Zusatz *von links nach rechts* in Zeile 5 dabei sicher, dass alle Blätter eines Knotens $\alpha_{h-1,i}$ links neben den Kindern eines Knotens $\alpha_{n-1,j}$ stehen, wenn laut Ordnung π gilt, dass $i \prec j$.

Es ist aus Cormen et al [CLRS09] bekannt, dass BFS eine Laufzeit von $\mathcal{O}(|V| + |E|)$ hat. Da die Kantenzahl in Bäumen immer um eins niedriger ist als die Knotenzahl, wird insgesamt eine Laufzeit von $\mathcal{O}(n)$ erreicht. \square

Die folgenden Definitionen von Optimalität werden später verwendet, um die Korrektheit der Algorithmen zu beweisen.

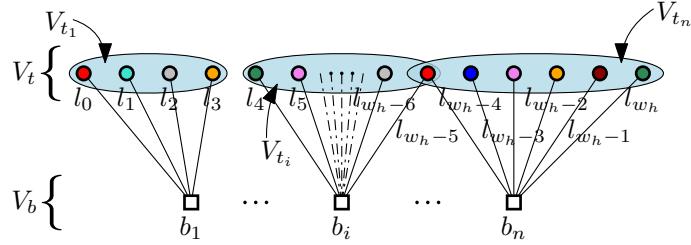


Abbildung 4.3: Instanz eines 2-Layer Graphen mit Ordnung. Man beachte, dass es vorkommen kann, dass ein Knoten wie l_{w_h-5} in zwei Mengen von V_t enthalten ist.

Definition 4.3 (l_i -Optimalität von T). Ein Baum T ist genau dann l_i -optimal, $i \in \mathbb{N}_0$, wenn alle Level $k < i$ optimal sind und wenn es keinen anderen Baum geben kann, dessen Level $j \leq i$ aus weniger Knoten besteht. Anders ausgedrückt, seien V_{l_i} die Mengen der Knoten auf allen Leveln $j \leq i$ und zusätzlich $T = (V, E, \pi)$ sowie $T^* = (V^*, E^*, \pi^*)$ zwei Bäume mit identischem Blattlevel l ($l \subset V \wedge l \subset V^*$), dann gilt:

$$T^* \text{ ist } l_i\text{-optimal} \Leftrightarrow \nexists T: |V_{l_i}| < |V_{l_i}^*|.$$

Definition 4.4 (Optimalität von T). Ein Baum T ist genau dann optimal, wenn das oberste Level h des Baums l_h -optimal ist.

Definition 4.5 (\hat{T}). Sei $\hat{T} = (\hat{V}, \hat{E}, \hat{\pi})$ der Baum ohne mehrfach vorkommende Blätter und innere Knoten des Baums $T = (V, E, \pi)$, sodass gilt: $\hat{T} \subseteq T$.

Korollar 4.1. Ein optimaler Baum T mit t doppelten Blättern hat genau $|E| = |\hat{E}| + t$ Kanten.

Der 2-Layer Graph mit Ordnung wird wie folgt definiert und in Abbildung 4.3 illustriert.

Definition 4.6 (2-Layer Graph mit Ordnung). Sei $L = (V_L, E_L, \Pi_t, \Pi_b)$ ein 2-Layer bipartiter Graph, für dessen Knoten sowohl $V_L = V_t \cup V_b$ als auch $V_t \cap V_b = \emptyset$ gilt. Dabei unterliegen, anders als bei der allgemeinen Definition eines bipartiten Graphen, die Knoten $l_i \subset V_{t_i} \subset V_t$ der oberen Schicht der Ordnung Π_t und die Knoten $b_i \subset V_b$ der unteren Schicht der Ordnung Π_b . Zusätzlich ist in diesem Fall diese Ordnung π_b der unteren Knoten festgeschrieben und darf nicht verändert werden.

Fügt man nun beide Definitionen des *Baums mit Ordnung* und des *2-Layer Graphen mit Ordnung* zusammen, so erhält man die folgende Definition für einen *Two-Layer Graph mit Einseitig Festgeschriebener Ordnung und Baumerweiterung (TLGEFOB)*, welche in Abbildung 4.4 verdeutlicht wird.

Definition 4.7 (TLGEFOB). Sei G ein Graph, der eine Kombination aus einem Baum mit Ordnung T und einem 2-Layer Graph mit Ordnung L ist und für den $G = (V, E, \Pi_t, \Pi_b)$ mit $E = E_T \cup E_L$ und $V = V_T \cup V_L$ gilt. Weiterhin wird die obere

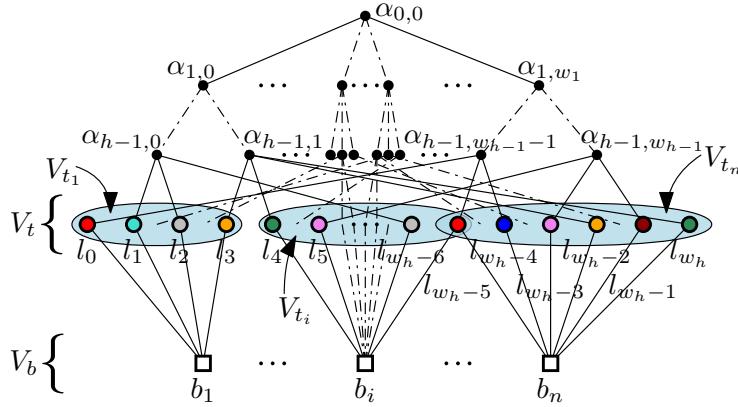


Abbildung 4.4: Instanz eines TLGEFOB. Dabei kann man resultierend aus Definition 4.7 sehen, dass die Blätter aus T gleich den Knoten der oberen Schicht V_t des Graphen L sind.

Schicht des 2-Layer Graphen mit V_t und die untere Schicht mit V_b bezeichnet, wobei die Blätter l des Baums T identisch zu der oberen Schicht V_t des 2-Layer Graphen L sind. Die Ordnung Π_b der unteren Knoten V_b ist unveränderlich, wo hingegen die Ordnung Π_t der oberen Knoten V_t aus den einzelnen Ordnungen $\Pi_t = \pi_1, \dots, \pi_n$ besteht. Dabei beschreibt die i -te Ordnung π_i die Reihenfolge der Knoten der Menge V_{t_i} und somit auch der Nachbarn des unteren Knotens b_i .

Der Knoten b_i beschreibt den i -ten Knoten aus der Menge V_b gemäß der Ordnung π_b . Die Menge V_{t_i} enthält alle Knoten aus V_t , die Nachbarn zum Knoten b_i sind.

In Anlehnung an die drei Probleme im Paper von Ahmed et al. [AAB+22] werden auf den kommenden Seiten die folgenden drei Probleme betrachtet. Dabei erhalten alle als Eingabe sowohl einen $\text{TLGEFOB}(V, E, \Pi_t, \Pi_b)$ nach Anwendung des Algorithmus von Ahmed et al. [AAB+22] als auch einen Duplizierungsparameter k :

- **Entfernen von Kreuzungen mit k Duplizierungen - CRS(k):** Der Algorithmus entscheidet, ob es eine planare Zeichnung des TLGEFOB mit höchstens k Knotenduplizierungen in V_t gibt.
- **Entfernen von Kreuzungen mit k Duplizierungsknoten - CRSV(k):** Der Algorithmus entscheidet, ob es eine planare Zeichnung des TLGEFOB mit höchstens k Duplizierungsknoten in V_t gibt.
- **Minimierung von Kreuzungen mit k Duplizierungen - CMS(k, M):** Der Algorithmus entscheidet, ob es eine Zeichnung des TLGEFOB mit höchstens M Kantenkreuzungen und höchstens k Knotenduplizierungen in V_t gibt.

4.2 Entfernen von Kreuzungen mit k Duplizierungen - CRS(k)

Im folgenden Abschnitt wird ein $\mathcal{O}(n^3)$ Algorithmus präsentiert, der überprüft, ob ein gegebener *TLGEFOB* mit höchstens k Duplizierungen planar gemacht werden kann.

Satz 4.2. *Für einen $TLGEFOB(V, E, \Pi_t, \Pi_b)$ trifft der Algorithmus 2 die korrekte Entscheidung, ob eine Planarisierung des gegebenen $TLGEFOB$ mit k Knotenduplizierungen möglich ist.*

Beweis. In einem ersten Schritt werden über die *Finde_C**-Funktion alle $i \in \{1, \dots, |V_t|\}$ der Knoten höchsten Levels berechnet, die sowohl Vorfahren von Blättern in V_{t_i} als auch von Blättern in $V_{t_{i+1}}$ sind. Ist dies geschehen, wird in den Zeilen 2 bis 5 die Reihenfolge der Blätter mit Hilfe der *Optimiere_Pi*-Funktion berechnet. Zuletzt wird die *Erstelle_Baum_omega*-Funktion verwendet, um die Reihenfolgen der inneren Knoten auf den einzelnen Leveln zu bestimmen. Bei der Berechnung der Anzahl der Duplizierungen wird nun wie folgt vorgegangen: Da ein Knoten bei einmaligem Vorkommen nicht für die Anzahl der Duplizierungen gezählt werden darf, genügt es, von der neuen Anzahl an Knoten im Inneren des Baums $|\omega|$ die Anzahl der ursprünglichen inneren Knoten, die durch $|V \setminus V_L|$ gegeben ist, abzuziehen. Zuletzt wird eine Fallunterscheidung in den Zeilen 8 bis 12 verwendet, um zu überprüfen, ob diese kleiner oder gleich der gegebenen Zahl k ist. \square

Satz 4.3. *Sei $n = |V|$ die Anzahl der Knoten eines $TLGEFOB(V, E, \Pi_t, \Pi_b)$, dann entscheidet der Algorithmus 2 das gegebene Problem in Zeit $\mathcal{O}(n^3)$.*

Algorithmus 2: CRS(k)

Input : $TLGEFOB(V, E, \Pi_t, \Pi_b)$, maximale Anzahl an Duplizierungen k

Output : Boolean

```

1  $C^* \leftarrow \text{Finde\_C}^*(TLGEFOB(V, E, \Pi_t, \Pi_b), p)$ 
2  $\text{opt}\Pi_t \leftarrow \emptyset$ 
3 for  $i$  in  $\{1, \dots, |V_t| - 1\}$  do
4   |  $\text{opt}\Pi_t \leftarrow \text{opt}\Pi_t \leftarrow \text{Optimiere\_Pi}(TLGEFOB(V, E, \Pi_t, \Pi_b), \pi_i, \text{root}, C^*, i)$ 
5 end
6  $\omega \leftarrow \text{Erstelle\_Baum\_omega}(TLGEFOB(V, E, \text{opt}\Pi_t, \Pi_b))$ 
7  $duplics \leftarrow |\omega| - |V \setminus V_L|$ 
8 if  $duplics \leq k$  then
9   | return TRUE
10 else
11   | return FALSE
12 end

```

Beweis. Der Aufruf der *Finde_C**-Funktion benötigt $\mathcal{O}(n^3)$.

Anschließend wird mit einer **for**-Schleife die Funktion *Optimiere_II*-Funktion mit einer Laufzeit von $\mathcal{O}(n^2) \cdot |V_t|$ -mal aufgerufen, was ebenso eine kubische Laufzeit impliziert ($\mathcal{O}(|V_t| \cdot n^2) = \mathcal{O}(n \cdot n^2) = \mathcal{O}(n^3)$).

Die Berechnung von ω kann in $\mathcal{O}(n^2)$ mit Hilfe der Funktion *Erstelle_Baum_ω* durchgeführt werden.

Da die Zeilen 7 bis 12 nur aus einfachen Zuweisungen und **if**-Abfragen bestehen, ist dies in konstanter Zeit $\mathcal{O}(1)$ erledigt.

Somit erhält man insgesamt eine Laufzeit von $\mathcal{O}(n^3 + n^3 + n^2 + 1) = \mathcal{O}(n^3)$. \square

In den folgenden Abschnitten werden die Unterfunktionen *Finde_C**, *Optimiere_II* sowie *Erstelle_Baum_ω* definiert sowie ihre Korrektheit und Laufzeit bewiesen.

4.2.1 Algorithmus Finde_C*

Bevor der Algorithmus *Finde_C** näher erläutert wird, ist es sinnvoll, die darin benötigte Hilfsfunktion *Propagiere_durch* zu betrachten. Hierbei ist anzumerken, dass C_i die i -te Menge in C ist, die alle Knoten enthält, die sowohl Vorfahren von Blättern in V_{t_i} als auch von Blättern in $V_{t_{i+1}}$ sind. Das Element C_i^* wiederum ist das i -te Element der Menge C^* und besteht aus einem Knoten aus der Menge C_i .

Lemma 4.1. Für die Mengen C und C^* berechnet der Algorithmus 3 das korrekte Ergebnis für die Menge C , nachdem die Zuweisung von C^* durchpropagiert wurde.

Beweis. Damit der Algorithmus für ein gegebenes i den Wert in C_i^* durchpropagieren kann, muss in der Menge links und rechts von C_i der Knoten C_i^* gelöscht werden, falls dieser dort vorhanden ist. Zuerst schaut die **while**-Schleife alle Knoten V_{t_j} ($0 \leq j < i$) links von V_{t_i} an und löscht den Knoten C_{j+1}^* aus C_j , falls dieser dort enthalten ist. Damit wird sichergestellt, dass der Knoten $v \in C_j^*$ nicht mehr für C_{j-1}^* gewählt werden kann, da dies sonst zu einer Zweiteilung der Menge der Blätter eines Vorfahren führen würde. Die nachfolgende **if**-Abfrage fängt nun den Fall ab, dass C_j durch das Löschen von C_{j+1}^* einelementig wird. Falls dies eintritt, kann der Knoten in C_j gleich in C_j^* gespeichert und C_j anschließend geleert werden (Zeile 7 und 8). Im nächsten Schritt wird j dekrementiert und eventuelle neue Konflikte durch die Zuweisung in Zeile 7 behoben, indem dieselben Zeilen nochmals für $j - 1$ durchlaufen werden.

Die nachfolgenden Zeilen 13 bis 22 sind äquivalent zu den Zeilen 2 bis 11, nur dass die Mengen C_k ($i < k < |C|$) rechts von C_i betrachtet werden. \square

Lemma 4.2. Für die Mengen C und C^* berechnet der Algorithmus 3 das korrekte Ergebnis für die Menge C , nachdem die Zuweisung von C^* durchpropagiert wurde mit einer Laufzeit von $\mathcal{O}(n)$.

Algorithmus 3: Propagiere_durch(C^*, C, i)

Input : Die Mengen C und C^* sowie der Wert der Laufvariablen i
Output : C und C^*

```

1  $j \leftarrow i - 1$ 
2 while  $j \geq 0$  do
3   if  $C_{j+1}^* \in C_j$  then
4      $C_j \leftarrow C_j \setminus \{C_{j+1}^*\}$ 
5   end
6   if  $|C_j| == 1$  then
7      $C_j^* \leftarrow C_j$ 
8      $C_j \leftarrow \emptyset$ 
9   end
10   $j \leftarrow j - 1$ 
11 end
12  $k \leftarrow i + 1$ 
13 while  $k \leq |C| - 1$  do
14   if  $C_{k-1}^* \in C_k$  then
15      $C_k \leftarrow C_k \setminus \{C_{k-1}^*\}$ 
16   end
17   if  $|C_k| == 1$  then
18      $C_k^* \leftarrow C_k$ 
19      $C_k \leftarrow \emptyset$ 
20   end
21    $k \leftarrow k + 1$ 
22 end
23 return  $C^*, C$ 

```

Beweis. Die **while**-Schleifen in Zeile 2 bis 11 und 13 bis 22 werden insgesamt $|V_t| - 1$ -mal durchlaufen. Die erste Schleife wird nämlich für alle $0 \leq j \leq i - 1 \leq |C| = |V_t|$ durchlaufen, die zweite für alle $i + 1 \leq k \leq |C| = |V_t|$. Innerhalb der Schleifen wiederum finden nur einfache **if**-Abfragen und Zuweisungen in $\mathcal{O}(1)$ statt. Die Laufzeit beträgt folglich insgesamt $\mathcal{O}(|V_t|) = \mathcal{O}(n)$. \square

Lemma 4.3. Für einen TLGEFOB(V, E, Π_t, Π_b) berechnet der Algorithmus 4 mit Hilfe der festgesetzten Knoten aus dem Algorithmus von Ahmed et al. [AAB+22] die korrekten Knoten für C^* . Korrekt bedeutet hierbei, dass für C_i^* ein Knoten mit einem so hohen Level wie möglich berechnet wird, sodass dessen Nachfahren sowohl in V_{t_i} als auch in der Menge $V_{t_{i+1}}$ enthalten sind.

Beweis. Ziel dieses Algorithmus ist es, am Ende $|V_t| - 1$ verschiedene nicht-leere Mengen C_i^* zu erhalten, die anschließend für die Berechnung für π in der Funktion *Adapt_Optimierte_π* benötigt werden.

Algorithmus 4: Finde_C* ($TLGEFOB(V, E, \Pi_t, \Pi_b), p$)

Input : Graph mit Ordnung $TLGEFOB(V, E, \Pi_t, \Pi_b)$ und festgesetzten Blättern p

Output: C^*

```

1 for  $i$  in  $\{1, \dots, |V_t| - 1\}$  do
2    $C_i \leftarrow \emptyset$ 
3    $C_i^* \leftarrow \emptyset$ 
4 end
5  $max\_level \leftarrow$  Level der Blätter
6  $level \leftarrow max\_level - 1$ 
7 while  $\exists C_i^* = \emptyset \wedge level \geq 0$  do
8    $D \leftarrow$  alle Knoten auf Level  $level$ 
9    $\hat{C}^* \leftarrow \emptyset$ 
10  for  $i$  in  $\{1, \dots, |V_t| - 1\}$  do
11    if  $p_i \neq \emptyset$  then
12       $C_i^* \leftarrow p_i$ 
13       $p_i \leftarrow \emptyset$ 
14       $\hat{C}_i^* \leftarrow$  Vorfahr von  $C_i^*$  auf Level  $level$ 
15    else if  $C_i^* == \emptyset$  then
16       $| C_i \leftarrow$  Knoten aus  $D$ , die sowohl Blätter in  $V_{t_i}$  als auch in  $V_{t_{i+1}}$  haben.
17    else
18       $| \hat{C}_i^* \leftarrow$  Vorfahr von  $C_i^*$  auf Level  $level$ 
19    end
20  end
21  for  $i$  in  $\{1, \dots, |V_t| - 1\}$  do
22     $| C \leftarrow \text{Propagiere\_durch}(\hat{C}^*, C, i)$ 
23  end
24  for  $i$  in  $\{1, \dots, |V_t| - 1\}$  do
25    if  $|C_i| == 1$  then
26       $C_i^* \leftarrow C_i$ 
27       $C_i \leftarrow \emptyset$ 
28       $C \leftarrow \text{Propagiere\_durch}(C^*, C, i)$ 
29    end
30  end
31  for  $i$  in  $\{1, \dots, |V_t| - 1\}$  do
32    if  $|C_i| \geq 2$  then
33       $C_i^* \leftarrow$  wähle bel. Element aus  $C_i$ 
34       $C_i \leftarrow \emptyset$ 
35       $C \leftarrow \text{Propagiere\_durch}(C^*, C, i)$ 
36    end
37  end
38 end
39  $level \leftarrow level - 1$ 

```

Da der Algorithmus den Baum von unten nach oben durchläuft, wird in Zeile 6 das Level oberhalb der Blätter als Startlevel definiert. Dies ist darin begründet, dass es generell bevorzugt wird, gemeinsame Knoten auf einem so hohen Level wie möglich zu finden, da dadurch die meisten Baumknoten und -kanten eingespart werden können. Die folgende **while**-Schleife wird solange wiederholt, bis entweder alle C_i^* gefunden wurden, oder der Algorithmus an der Wurzel angelangt ist. Zu Beginn der Schleife (Zeile 10-20) läuft man einmal über alle Knoten V_t , wobei in Zeile 11 überprüft wird, ob V_{t_i} einen von Ahmed et al. [AAB+22] festgesetzten ersten Nachbarn hat. Falls ja, kann man diesen direkt in C_i^* eintragen. Falls nicht, wird die Menge C mit Knoten aus D gefüllt, die sowohl Blätter in V_{t_i} als auch in $V_{t_{i+1}}$ haben. Falls aber in C_i^* ein Wert eingetragen ist, so wird in C_i^* der Vorfahr von C_i^* auf dem betrachteten Level *level* geschrieben. Die folgende **for**-Schleife propagiert mit Hilfe der Menge C_i^* die bereits gewählten Knoten auch auf niedrigeren Levels durch. In der nachfolgenden **for**-Schleife über alle oberen Knoten des bipartiten Teils des Graphen *TLGEFOB* (Zeile 24 - 30) wird deshalb der Basisfall abgearbeitet, bei dem nur ein Knoten in C_i vorhanden ist und somit dieser auch gewählt und durchpropagiert werden muss. Im zweiten Schritt wird nun aus allen verbleibenden Mengen C_i , die mindestens zwei Elemente besitzen, ein beliebiges Element ausgewählt, in C_i^* abgespeichert und durchpropagiert. Hier ist zu beachten, dass dies nicht zu einem Konflikt mit C_{i-1}^* oder C_{i+1}^* führen kann, da bereits zugewiesene Knoten immer über die *Propagiere_durch* Funktion, falls in den Nachbarmengen C_{i-1} und C_{i+1} vorhanden, gelöscht werden. Zuletzt wird die Variable *level* um eins dekrementiert, damit die **while**-Schleife auf der Ebene darüber angewandt und damit sichergestellt werden kann, dass der Algorithmus terminiert.

Es ist ebenfalls garantiert, dass für alle C_i^* ein Element gefunden wird, da die **while**-Schleife auch auf der Wurzel ausgeführt wird. Da die Wurzel selbst natürlich per Konstruktion für alle $i \in \{1, \dots, |V_t| - 1\}$ immer sowohl Blätter in V_{t_i} als auch in $V_{t_{i+1}}$ hat, wird den leeren C_i^* somit auf jeden Fall in Zeile 26 die Wurzel als Knoten zugewiesen. \square

Lemma 4.4. Für einen $TLGEFOB(V, E, \Pi_t, \Pi_b)$ berechnet der Algorithmus 4 die korrekten Knoten für C^* in der Zeit $\mathcal{O}(n^3)$.

Beweis. Die erste **for**-Schleife in Zeile 1 bis 4 wird $|V_t| - 1$ -mal durchlaufen und dient nur zu Initialisierungszwecken. Somit beträgt die Laufzeit $\mathcal{O}(|V_t|) = \mathcal{O}(n)$.

Ebenso können die zwei Zuweisungen in Zeile 5 und 6 in $\mathcal{O}(1)$ erledigt werden.

Im Inneren der **while**-Schleife wird die **for**-Schleife in Zeile 10 bis 20 $|V_t| - 1$ -mal durchlaufen, wobei die **if**-Abfragen und die Zuweisungen in konstanter Zeit geschehen. Um C_i in Zeile 16 mit Werten zu füllen, werden alle Knoten in V_{t_i} und $V_{t_{i+1}}$ durchlaufen, damit alle Knoten aus D bestimmt werden können, die Blätter in beiden Mengen haben. Diese Wahl der Knoten, die in C_i gespeichert werden sollen, kann somit auf jeden Fall in $\mathcal{O}(|V_t|)$ getroffen und zugewiesen werden. Für die Zeilen 10 bis 20 erhält man eine Laufzeit von $\mathcal{O}(|V_t|^2) = \mathcal{O}(n^2)$.

Die **for**-Schleifen in Zeile 21 bis 23, Zeile 24 bis 30 und Zeile 31 bis 37 werden alle

ebenfalls $|V_t| - 1$ -mal durchlaufen. In allen drei Schleifen wird die *Propagiere_durch*-Funktion mit einer Laufzeit von $\mathcal{O}(|V_t|)$ aufgerufen. Zusätzlich finden **if**-Abfragen und einfache Zuweisungen in $\mathcal{O}(1)$ statt. Dies resultiert in einer Laufzeit von $\mathcal{O}((|V_t| \cdot |V_t|) + (|V_t| \cdot |V_t|)) = \mathcal{O}(|V_t|^2) = \mathcal{O}(n^2)$.

Da die **while**-Schleife höchstens so oft durchlaufen wird, wie der Baum des Graphen G Level hat, besitzt diese eine Laufzeit von $\mathcal{O}(|V|)$. Bezieht man die inneren Abläufe auch noch mit ein, so erhält man eine Gesamlaufzeit der **while**-Schleife von $\mathcal{O}(n) \cdot (\mathcal{O}(n^2) + \mathcal{O}(n^2) + \mathcal{O}(n^2)) = \mathcal{O}(n^3)$.

Werden nun die Zeilen 1 bis 6 mit einbezogen, hat der Algorithmus eine Gesamlaufzeit von $\mathcal{O}(n^3) + \mathcal{O}(n^2) = \mathcal{O}(n^3)$. \square

4.2.2 Algorithmus Optimiere_Π

Lemma 4.5. Für einen TLGEFOB(V, E, Π_t, Π_b) sowie der Menge C^* berechnet der Algorithmus 5, aufgerufen auf dem Wurzelknoten v mit Ordnung π , die korrekte Sub-Ordnung π_k für die Knoten in V_{t_k} .

Beweis. Die **for**-Schleife in Zeile 4-15 stellt sicher, dass alle Kinder des Knotens v betrachtet werden. Anschließend überprüft die nächste **for**-Schleife in Zeile 5-9 für alle Blätter $l_i \in \pi_t$, ob diese das betrachtete Kind c aus Zeile 4 als Vorfahr haben und sortiert sie in die entsprechende Menge π_c ein. Da nur jene Blätter Nachfahren eines Knotens auf Level i sein können, die auch Nachfahren dessen Elternknotens auf Level $i - 1$ sind, reicht es hier, nur alle Blätter der Menge π zu überprüfen.

Falls der betrachtete Kindsknoten c Vorfahr von C_k^* oder sogar selbst C_k^* ist, werden dessen Nachfahren in π_c als erste Knoten (*first*) deklariert. Genau dieselbe Überprüfung findet mit C_{k+1}^* statt, sodass man dessen Nachfahren in π_c als letzten Knoten (*last*) deklarieren kann. Die folgende Fallunterscheidung stellt sicher, dass die Mengen π_c in richtiger Reihenfolge zurückgegeben werden.

Die Abbruchbedingung des rekursiven Algorithmus ist erfüllt, wenn er auf den Blättern mit Level h aufgerufen wird. Dann können die Kinder nämlich nicht mehr optimal geordnet werden, da Blätter per Konstruktion keine Kinder haben. In diesem Fall wird das π , mit dem der Aufruf gestartet wurde, wieder zurückgegeben. \square

Lemma 4.6. Für einen TLGEFOB(V, E, Π_t, Π_b) sowie mit Hilfe der Menge C^* berechnet der Algorithmus 5, aufgerufen auf dem Wurzelknoten, die korrekte Sub-Ordnung π für die Knoten in V_{t_k} in Zeit $\mathcal{O}(n^2)$.

Beweis. Die **for**-Schleife in Zeile 4 betrachtet bei initialem Aufruf auf der Wurzel alle Kinder der Wurzel einmal. Anschließend folgt der rekursive Aufruf auf allen Kindern der Wurzel, hierbei werden dann wiederum alle Kinder einmal betrachtet. Beim Aufruf auf den Blättern bricht der Algorithmus bereits in Zeile 2 ab. Die Knoten werden Level für Level abgearbeitet, was dazu führt, dass die **for**-Schleife in Zeile 4 bis 15 insgesamt nur $|V_T| - |l|$ -mal durchlaufen wird. In der **for**-Schleife in Zeile 5 bis 9, die $|l|$ -mal abgearbeitet wird, finden nur **if**-Abfragen und eine Zuweisung in

$\mathcal{O}(1)$ statt. Ebenso können die **if**-Bedingungen in Zeile 10 und 12 mit dazugehörigen Zuweisungen in konstanter Zeit erledigt werden. Somit erhält man eine Laufzeit von $\mathcal{O}((|V_T| - |l|) \cdot |l|) = \mathcal{O}(|V_T|^2) = \mathcal{O}(n^2)$.

Auch die **if**-Bedingungen in den Zeilen 16, 18 und 20 können, genauso wie die Zuweisungen von π , in konstanter Zeit überprüft werden.

Insgesamt beläuft sich dadurch die Laufzeit auf $\mathcal{O}(|V_T|^2) = \mathcal{O}(n^2)$. \square

Algorithmus 5: Optimiere _{Π} ($TLGEFOB(V, E, \Pi_t, \Pi_b), \pi, v, C^*, k$)

Input : $TLGEFOB(V, E, \Pi_t, \Pi_b)$, die Ordnung π , der Aufrufknoten v , die Menge C^* und den Wert der aktuellen Laufvariablen k

Output : Die Ordnung π_k der Nachbarn des Knoten b_k

```

1 if  $v == Blatt$  then
2   | return  $\pi$ 
3 else
4   | foreach Kind  $c$  von  $v$  do
5     |   foreach  $l_i \in \pi$  do
6       |       if  $c$  Vorfahr von  $u$  then
7         |           |  $\pi_c \leftarrow \pi_c \cup l_i$ 
8         |       end
9       |   end
10      |   if  $c$  Vorfahr von  $C_k^* \vee C_k^* == c$  then
11        |       first  $\leftarrow [\pi_c, c]$ 
12      |   else if  $c$  Vorfahr von  $C_{k+1}^* \vee C_{k+1}^* == c$  then
13        |       last  $\leftarrow [\pi_c, c]$ 
14      |   end
15   end
16   if  $C_k^* \neq \emptyset \wedge C_{k+1}^* \neq \emptyset$  then
17     |   return  $\pi \leftarrow$ 
18     |       [Adapt_Optimierte $\pi$ ( $TLGEFOB(V, E, first[1], \Pi_b), first[2], C^*, k$ )  $\prec \dots$ 
19     |        $\prec$  Adapt_Optimierte $\pi$ ( $TLGEFOB(V, E, last[1], \Pi_b), last[2], C^*, k$ )]
20   else if  $C_k^* \neq \emptyset$  then
21     |   return  $\pi \leftarrow$ 
22     |       [Adapt_Optimierte $\pi$ ( $TLGEFOB(V, E, first[1], \Pi_b), first[2], C^*, k$ )  $\prec \dots$ 
23     |        $\prec$  Adapt_Optimierte $\pi$ ( $TLGEFOB(V, E, \pi_{c_n}, \Pi_b), c_n, C^*, k$ )]
24   else if  $C_{k+1}^* \neq \emptyset$  then
25     |   return  $\pi \leftarrow$ 
26     |       [Adapt_Optimierte $\pi$ ( $TLGEFOB(V, E, \pi_{c_1}, \Pi_b), c_1, C^*, k$ )  $\prec \dots \prec$  Adapt_Optimierte $\pi$ ( $TLGEFOB(V, E, last[1], \Pi_b), last[2], C^*, k$ )]
27   else
28     |   return  $\pi \leftarrow$ 
29     |       [Adapt_Optimierte $\pi$ ( $TLGEFOB(V, E, \pi_{c_1}, \Pi_b), c_1, C^*, k$ )  $\prec \dots \prec$  Adapt_Optimierte $\pi$ ( $TLGEFOB(V, E, \pi_{c_n}, \Pi_b), c_n, C^*, k$ )]
30 end
31 end

```

4.2.3 Algorithmus Erstelle_Baum_ω

Lemma 4.7. Für einen $TLGEFOB(V, E, \Pi_t, \Pi_b)$ berechnet der Algorithmus 6 die korrekte Baumstruktur ω .

Beweis. Der Algorithmus 6 erstellt die optimale Baumstruktur mit Hilfe der Ordnung Π_t der oberen Schicht des bipartiten Graphen. Dabei läuft dieser in der **for**-Schleife in Zeile 1 bis 10 einmal über alle Level und berechnet die korrekte Ordnung für das Level i . Innerhalb der Schleife sorgen die zwei **for**-Schleifen in Zeile 3 bis 9 und 4 bis 8 dafür, dass für jedes Blatt $l_k \in \Pi_t$ von links nach rechts jeder Knoten auf dem jeweiligen Level i angeschaut wird. Dabei wird mit Hilfe der **if**-Abfrage in Zeile 5 überprüft, ob der aktuell betrachtete Knoten $\alpha_{i,j}$ Vorfahr von l_k ist und ob der Knoten $\alpha_{i,j}$ bereits beim letzten Mal zu ω_i hinzugefügt worden ist. Diese zweite Bedingung soll sicherstellen, dass der Baum auf dem jeweiligen Level die l_i -Optimalität erfüllt. Ohne diese Abfrage würden zwei identische Knoten $\alpha_{i,j}$ nebeneinander im Baum stehen, obwohl einer ausreichend wäre. Insbesondere wird beim letzten Durchlauf der **for**-Schleife in Zeile 1 bis 10 die Ordnung eines l_{h-1} optimalen Baums T erstellt, wenn h das maximale Level ist. Da die Anzahl der Blätter von der Methode von Ahmed et al. [AAB+22] vorgegeben wurde, ist der gesamte Graph optimal. Zuletzt wird die berechnete Baumordnung ω mit den darin enthaltenen Subordnungen der einzelnen Level ω_i zurückgegeben. □

Lemma 4.8. Für einen $TLGEFOB(V, E, \Pi_t, \Pi_b)$ berechnet der Algorithmus 6 die korrekte Baumstruktur ω in Zeit $\mathcal{O}(n^2)$.

Beweis. Betrachtet man den Algorithmus 6 im Ganzen, so stellt man fest, dass die Zeile 5 bis 7 für jedes Blatt l_k , $|V_T \setminus l|$ -Mal durchlaufen wird. Somit ergibt sich mit der

Algorithmus 6: Erstelle_Baum_ω($TLGEFOB(V, E, \Pi_t, \Pi_b)$)

Input : $TLGEFOB(V, E, \Pi_t, \Pi_b)$

Output : ω

```

1 foreach Level  $i$  ohne Blattlevel do
2    $\omega_i \leftarrow \emptyset$ 
3   foreach  $l_k \in \Pi_t$  do
4     foreach Knoten  $j$  auf Level  $i$  do
5       if  $\alpha_{i,j}$  Vorfahr von  $l_k \wedge$  letztes Element in  $\omega_i \neq \alpha_{i,j}$  then
6          $\omega_i \leftarrow \omega_i \prec \alpha_{i,j}$ 
7       end
8     end
9   end
10 end
11 return  $\omega$ 

```

if-Abfrage in Zeile 5 sowie der Zuweisung in Zeile 6, die beide in konstanter Zeit $\mathcal{O}(1)$ abgearbeitet werden können, eine Gesamlaufzeit von $\mathcal{O}(|l| \cdot |V_T \setminus l|) = \mathcal{O}(n^2)$. \square

4.2.4 Beispiel

Angenommen, wir haben den Graph aus Abbildung 4.5 mit den Ordnungen

$$\Pi_t = (l_0 \prec l_1 \prec l_2 \prec l_3) \prec (l_4 \prec l_5 \prec l_6 \prec l_7 \prec l_8) \prec (l_9 \prec l_{10} \prec l_{11} \prec l_{12} \prec l_{13})$$

und

$$\Pi_b = b_1 \prec b_2 \prec b_3.$$

Zusätzlich wurden die von der Methode von *Ahmed et al.* [AAB+22] festgesetzten Knoten in p gespeichert,

$$p = [\emptyset, l_8].$$

Nun wird überprüft, ob es eine planare Zeichnung des Graphen *TLGEFOB* mit höchstens $k = 9$ Knotenduplizierungen in V_t gibt. Das heißt, *CRS(9)* muss aufgerufen werden.

Zuerst wird die *Finde_C**-Funktion aufgerufen:

Finde_C*

Da p_1 nicht gegeben ist, wird C_1 mit den entsprechenden Werten in Zeile 15 gefüllt,

$$C = [\{\alpha_{2,2}, \alpha_{2,3}\}, \emptyset].$$

Anschließend wird aufgrund der Tatsache, dass p_2 gegeben ist, dieser Wert in Zeile 12 des Algorithmus 4 in C_2^* geschrieben und aus p_2 in Zeile 13 gelöscht. Zusätzlich wird \hat{C}^* berechnet,

$$n = [\emptyset, l_8], \quad p = [\emptyset, \emptyset], \quad \hat{C}^* = [\emptyset, \alpha_{2,2}].$$

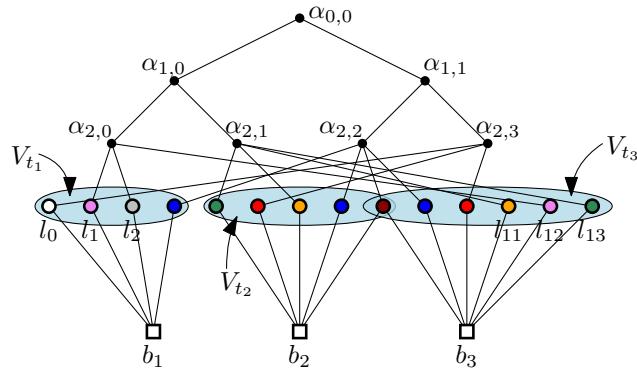


Abbildung 4.5: Beispiel für einen TLGEFOB

4.2. Entfernen von Kreuzungen mit k Duplizierungen - CRS(k)

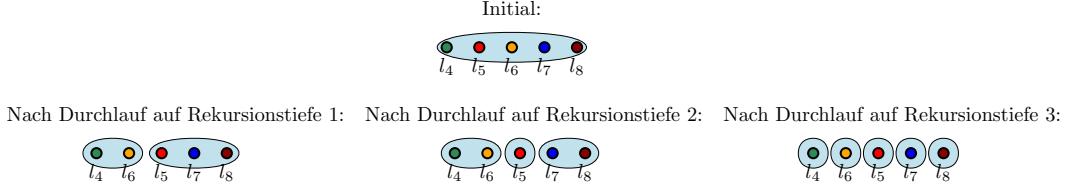


Abbildung 4.6: Werte von π in verschiedenen Rekursionstiefen

Nun wird \hat{C}^* in C durchpropagiert, sodass gilt

$$C = [\emptyset, \emptyset], \quad n = [\alpha_{2,3}, l_8].$$

Jetzt kann die *Optimierte_II*-Funktion auf alle oberen Knoten V_t in V_L aufgerufen werden. Im Folgenden wird beispielhaft der Funktionsaufruf mit $i = 2$ betrachtet, da $i = 1$ und $i = 3$ identisch verlaufen:

Optimierte_II

Während beim Funktionsaufruf die Menge π noch leer ist, wird sie in den Zeilen 5 bis 9 gefüllt:

$$\pi_{\alpha_{1,0}} = l_4 \prec l_6, \quad \pi_{\alpha_{1,1}} = l_5 \prec l_7 \prec l_8.$$

Anschließend werden den Vektoren *first* und *last* die entsprechenden Werte zugewiesen. Da $\alpha_{0,1}$ Vorfahr von $C_2^* = \alpha_{2,3}$ ist, wird dieser Wert als *first* festgeschrieben, Analoges gilt für *last*,

$$first = [\pi_{\alpha_{1,0}}, \alpha_{1,0}], \quad last = [\pi_{\alpha_{1,1}}, \alpha_{1,1}].$$

Somit erhält man nach dem ersten Durchlauf in Rekursionstiefe 1 folgende Ordnung für π :

$$\pi = (l_4 \prec l_6) \prec (l_5 \prec l_7 \prec l_8).$$

Abbildung 4.6 zeigt die Werte von π in den einzelnen Rekursionstiefen.

Zuletzt wird durch Aufrufen der *Erstelle_Baum_* ω -Funktion die Baumstruktur erstellt:

Erstelle_Baum_ ω

Zum einfacheren Nachvollziehen werden die Knoten im Folgenden nur mit ihren Labels gezeichnet:

$$\alpha_{0,0} = A, \quad \alpha_{1,0} = B, \quad \alpha_{1,1} = C, \quad \alpha_{2,0} = D, \quad \alpha_{2,1} = E, \quad \alpha_{2,2} = F \quad \text{und} \quad \alpha_{2,3} = G.$$

Angefangen auf der Wurzelebene (Level 0) berechnet der Algorithmus als erstes ω_0 , was trivialerweise der Wurzelknoten bleibt. Für die Durchläufe höherer Level kann man die Auswirkungen des Algorithmus in Abbildung 4.7 sehen.

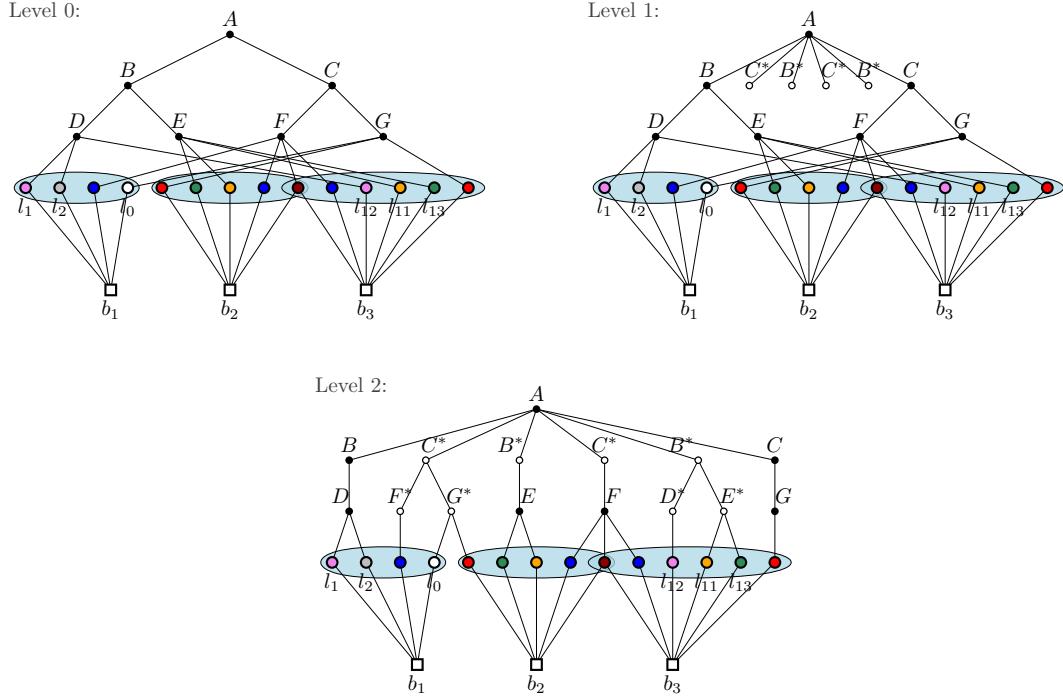


Abbildung 4.7: Werte von ω auf verschiedenen Leveln. Duplizierte Knoten werden als Kreise gezeichnet und deren Labels mit einem Sternchen versehen.

Zum Schluss muss noch die Anzahl der Duplizierungen mit der Formel in Algorithmus 2 berechnet werden:

$$duplicates = |\omega| - |V \setminus V_L| = 15 - 7 = 8.$$

Da 8 kleiner $k = 9$ ist, mit dem die Funktion $CRS(k)$ aufgerufen wurde, gibt der Algorithmus TRUE zurück.

4.3 Entfernen von Kreuzungen mit k Duplizierungsknoten - CRSV(k)

Für das Problem $CRSV(k)$ wird in diesem Abschnitt ein Algorithmus vorgestellt, der mit einer kubischen Laufzeit entscheidet, ob ein $TLGEFOB$, bei dem höchstens k Duplizierungsknoten dupliziert werden, planar gezeichnet werden kann.

Satz 4.4. *Für einen gegebenen $TLGEFOB(V, E, \Pi_t, \Pi_b)$ trifft der Algorithmus 7 die korrekte Entscheidung, ob eine Planarisierung des gegebenen $TLGEFOB$ mit höchstens k Duplizierungsknoten möglich ist.*

Beweis. Da die Zeilen 1 bis 6 identisch zum Algorithmus CRS in Abschnitt 4.2 sind, wird für dessen Korrektheit auf Satz 4.2 verwiesen.

4.3. Entfernen von Kreuzungen mit k Duplizierungsknoten - CRSV(k)

Algorithmus 7: CRSV(k)

Input : TLGEOF(B(V, E, Π_t, Π_b)), maximale Anzahl an Duplizierungsknoten k

Output : Boolean

```

1  $C^* \leftarrow \text{Finde\_}C^*(\text{TLGEOF}(V, E, \Pi_t, \Pi_b), p)$ 
2  $\text{opt}\Pi_t \leftarrow \emptyset$ 
3 for  $i$  in  $\{0, \dots, |V_t| - 1\}$  do
4   |  $\text{opt}\Pi_t \leftarrow \text{opt}\Pi_t \cup \text{Optimierte\_}\Pi(\text{TLGEOF}(V, E, \Pi_t, \Pi_b), \pi_i, \text{root}, C^*, i)$ 
5 end
6  $\omega \leftarrow \text{Erstelle\_Baum\_}\omega(\text{TLGEOF}(V, E, \text{opt}\Pi_t, \Pi_b))$ 
7  $\text{dupvertices} \leftarrow 0$ 
8 foreach level  $i$  ohne Blattlevel do
9   | sortiere  $\omega_i$ 
10  |  $j \leftarrow 1$ 
11  | while  $j < |\omega_i|$  do
12    |   | if  $\omega_{j-1} == \omega_j$  then
13    |   |   |  $\text{dupvertices} \leftarrow \text{dupvertices} + 1$ 
14    |   |   |  $j \leftarrow j + 1$ 
15    |   |   | while  $\omega_{j-1} == \omega_j \wedge j < |\omega_i|$  do
16    |   |   |   |  $j \leftarrow j + 1$ 
17    |   |   | end
18    |   | else
19    |   |   |  $j \leftarrow j + 1$ 
20    |   | end
21  | end
22 end
23 if  $\text{dupvertices} \leq k$  then
24   | return TRUE
25 else
26   | return FALSE
27 end

```

Bei der Berechnung der Anzahl der Duplizierungsknoten wird nun wie folgt vorgegangen: Formal gilt ein Knoten als Duplizierungsknoten, wenn dieser mindestens zweimal auf einem Level vorkommt. Deshalb wird in der nachfolgenden **for**-Schleife nach der Initialisierung der Variable dupvertices für jedes Level i die entsprechende Menge ω_i sortiert, sodass alle identischen Knoten nebeneinander stehen. Mit Hilfe der **while**-Schleife in Zeile 11 bis 21 wird für alle Knoten überprüft, ob der vorherige Knoten ω_{j-1} identisch zum aktuell betrachteten Knoten ω_j ist. Da ω_{j-1} in Zeile 13 bereits als Duplizierungsknoten vermerkt wird, müssen alle weiteren zu ω_{j-1} identischen Knoten mit der nachfolgenden **while**-Schleife übersprungen werden. Sind zwei aufeinanderfolgende Knoten nicht identisch, ist dies für die Berechnung der Duplizierungsknoten irrelevant und wird vom **else**-Fall abgefangen, indem j um eins

inkrementiert wird.

Zuletzt wird eine Fallunterscheidung in den Zeilen 23 bis 27 verwendet, um zu überprüfen, ob die berechnete Variable *dupvertices* kleiner oder gleich der gegebenen Zahl k ist. \square

Satz 4.5. Für einen $TLGEFOB(V, E, \Pi_t, \Pi_b)$ entscheidet der Algorithmus 7 das gegebene Problem in Zeit $\mathcal{O}(n^3)$.

Beweis. Der Aufruf der *Finde_C**-Funktion benötigt $\mathcal{O}(n^3)$.

Anschließend wird mit einer **for**-Schleife die Funktion *Optimiere_Π* $|V_t|$ -mal mit einer Laufzeit von $\mathcal{O}(n^2)$ aufgerufen, was in Zeit $\mathcal{O}(|V_t| \cdot n^2) = \mathcal{O}(n \cdot n^2) = \mathcal{O}(n^3)$ geschieht.

Die Berechnung von ω geschieht anschließend mit Hilfe der *Erstelle_Baum_ω*-Funktion in Zeit $\mathcal{O}(n^2)$.

In den Zeilen 8 bis 21 werden alle Knoten im Baum V_T ohne das Blattlevel zweimal betrachtet. Hinzu kommt das Sortieren von ω_i auf jedem Level i , was zum Beispiel mit dem *Insertion Sort* Algorithmus im Worst-Case in $\mathcal{O}(n^2)$ geschieht. Die Zeilen 8 bis 22 haben inklusive der darin enthaltenen Zuweisungen und **if**-Abfragen, die Zeit $\mathcal{O}(1)$ benötigen, eine Laufzeit von $\mathcal{O}(n \cdot n^2) = \mathcal{O}(n^3)$.

Zuletzt benötigt die Fallunterscheidung in den Zeilen 23 bis 27 höchstens die Zeit $\mathcal{O}(1)$.

Insgesamt ergibt sich damit eine Laufzeit von $\mathcal{O}(n^3 + n^3 + n^2 + n^3 + 1) = \mathcal{O}(n^3)$. \square

4.4 Minimierung von Kreuzungen mit k Duplizierungen - CMS(k, M)

Im folgenden Abschnitt wird ein $2^{\mathcal{O}((k+1) \cdot n^{n!})}$ Algorithmus präsentiert, der überprüft, ob für einen gegebenen TLGEFOB höchstens k Duplizierungen ausreichen, damit dieser höchstens M Kantenkreuzungen besitzt.

Satz 4.6. Für einen gegebenen $TLGEFOB(V, E, \Pi_t, \Pi_b)$ trifft der Algorithmus 8 die korrekte Entscheidung, ob das Erstellen einer Zeichnung mit höchstens M Kantenkreuzungen durch höchstens k Knotenduplizierungen möglich ist.

Beweis. Nach der Initialisierung aller benötigten Mengen in Zeile 1 werden alle Möglichkeiten, k Knoten im Baum von TLGEFOB zu duplizieren, in der Menge \mathcal{X}^* abgespeichert. Anschließend erhält man durch die disjunkte Vereinigung mit der ursprünglichen Knotenmenge V alle möglichen Bäume, die in \mathcal{X} geschrieben werden. Die nachfolgende **foreach**-Schleife berechnet nun alle Möglichkeiten, wie die Knoten des Levels $h - 1$ angeordnet sein können, und speichert diese in der Menge \mathcal{Y}^* ab. Hier ist zu beachten, dass es dazu kommen kann, dass die erstellten Ordnungen in \mathcal{Y}^* nicht mit den zur Verfügung stehenden Baumknoten realisiert werden können. Deshalb folgt in den Zeilen 9 bis 14 für jede Ordnung $y \in \mathcal{Y}$ eine Überprüfung, ob

4.4. Minimierung von Kreuzungen mit k Duplizierungen - CMS(k,M)

Algorithmus 8: CMS(k, M)

Input : TLGEFOB(V, E, Π_t, Π_b), maximale Anzahl an Duplizierungen k , maximale Anzahl an Kantenkreuzungen M

Output: Boolean

```

1  $CSET \leftarrow \mathcal{W} \leftarrow \mathcal{N} \leftarrow \mathcal{X} \leftarrow \mathcal{Y} \leftarrow \mathcal{Z} \leftarrow \{\}$ 
2  $\mathcal{X}^* \leftarrow \text{Berechne\_}\mathcal{X}(\text{TLGEFOB}, k, CSET, \mathcal{X}^*)$ 
3 foreach  $x \in \mathcal{X}^*$  do
4    $\mathcal{X} \leftarrow \{\mathcal{X}, x \sqcup V\}$ 
5 end
6 foreach  $x \in \mathcal{X}$  do
7    $A \leftarrow \alpha_{h-1,j} \in x$ 
8    $\mathcal{Y}^* \leftarrow \text{Berechne\_Möglichkeiten}(A, CSET, \mathcal{Y}^*)$ 
9   foreach  $y \in \mathcal{Y}^*$  do
10     $\omega \leftarrow \text{Erstelle\_Baum\_}\omega(y)$ 
11    if  $\omega \neq x$  then
12       $\mid \text{delete}(y, \mathcal{Y}^*)$ 
13    end
14  end
15   $\mathcal{Y} \leftarrow \{\mathcal{Y}, \mathcal{Y}^*\}$ 
16 end
17 for  $i \in \{1, \dots, V_{t_i}\}$  do
18    $\mathcal{Z}_i \leftarrow \{\}$ 
19    $\mathcal{Z}_i \leftarrow \text{Berechne\_Möglichkeiten}(V_{t_i}, CSET, \mathcal{Z}_i)$ 
20   foreach  $J \in \mathcal{Z}_i$  do
21     if  $\text{last}(V_{t_i}) \neq p(i) \wedge p(i) \neq \emptyset$  then
22        $\mid \text{delete}(J, \mathcal{Z}_i)$ 
23     end
24   end
25    $\mathcal{Z}^* \leftarrow \{\mathcal{Z}^*, \mathcal{Z}_i\}$ 
26 end
27  $\mathcal{Z} \leftarrow \text{Mische}(\mathcal{Z}^*, CSET, \mathcal{Z})$ 
28  $\mathcal{W} \leftarrow \text{Mische}(\{\mathcal{Y}, \mathcal{Z}\}, CSET, \mathcal{W})$ 
29 foreach  $w \in \mathcal{W}$  do
30   foreach  $v \in V_t \in w$  do
31      $\mid \mathcal{N}_w(v) \leftarrow v$  und alle Duplizierungen in  $w$  davon
32   end
33    $TLG \leftarrow \{TLG, \text{Mische}(\mathcal{N}_w, CSET, TLG)\}$ 
34 end
35  $M^* \leftarrow \text{calcCrossings}(TLG)$ 
36  $\min M^* \leftarrow \min(M^*)$ 
37 if  $\min M^* \leq M$  then
38    $\mid \text{return } \text{TRUE}$ 
39 else
40    $\mid \text{return } \text{FALSE}$ 
41 end

```

die Knoten identisch zum Baum ω sind. Dabei wird ω von der leicht modifizierten Funktion *Erstelle_Baum_* ω berechnet, indem diese nicht die Blätter, sondern die Knoten auf dem Level $h - 1$, die in y gespeichert sind, dazu verwendet. Sind die Knotenmengen nicht identisch (Zeile 11), so wird die Ordnung y wieder aus der Menge \mathcal{Y}^* gelöscht.

Als nächstes müssen auch alle verschiedenen Anordnungen der Blätter für jede Menge V_{t_i} erstellt werden (Zeile 19). Da der Algorithmus von Ahmed et al. [AAB+22] den ersten und letzten Knoten einer Menge V_{t_i} festsetzen kann, muss in Zeile 21 überprüft werden, ob dies für die aktuelle Menge V_{t_i} der Fall ist. Falls ja, so werden alle berechneten Mengen $J \in \mathcal{Z}_i$, deren letzter Knoten nicht $p(i)$ ist, aus \mathcal{Z}_i gelöscht. Da es nun für jede der i Knotenmengen V_{t_i} viele verschiedene berechnete Anordnungen gibt, müssen alle Kombinationen über die *Mische*-Funktion zusammengeführt werden (Zeile 27). Dasselbe geschieht in Zeile 28, in der alle potenziellen Anordnungen der Baumknoten auf Level $h - 1$ mit allen möglichen Anordnungen der Blätter auf Level h kombiniert und in \mathcal{W} abgespeichert werden.

Da jetzt alle verschiedenen Möglichkeiten an konstruierbaren Knotenmengen von Bäumen, die durch k Knotenduplizierungen erstellt werden können, berechnet worden sind, müssen zum Schluss noch die möglichen Kanten zwischen dem Blattlevel h und dem Level $h - 1$ aufgezählt werden. Dies geschieht, indem zuerst alle Nachbarn eines Blattknotens für eine Möglichkeit $w \in \mathcal{W}$ in $\mathcal{N}_w(v)$ abgespeichert werden (Zeile 31); anschließend wird durch die *Mische*-Funktion für jeden Knoten dieser Möglichkeit w jede mögliche Kombination an Nachbarn berechnet und in TLG abgespeichert. Dieses Ergebnis TLG entspricht somit dem gewünschten 2-Layer Graphen, wobei die untere Schicht die Blätter und die obere Schicht die Knoten des Levels $h - 1$ enthält. Für jeden dieser Graphen in TLG wird nun in Zeile 35 die Anzahl der Kantenkreuzungen berechnet und in Zeile 38 das Minimum davon bestimmt. Die *if*-Abfrage von Zeile 37 bis 41 entscheidet nun, ob der berechnete Wert an Kreuzungen M^* höchstens so groß ist wie der übergebene Wert M . \square

Satz 4.7. *Für einen TLGEFOB(V, E, Π_t, Π_b) entscheidet der Algorithmus 8, ob mit Hilfe von k Knotenduplizierungen ein Graph mit höchstens M Kantenkreuzungen konstruiert werden kann in Zeit $2^{\mathcal{O}((k+1) \cdot n^{n!})}$.*

Beweis. Wie später im Lemma 4.10 bewiesen werden wird, beträgt die Laufzeit für den Aufruf der *Berechne_X*-Funktion $\mathcal{O}((n+k-1)!/((n-1)! \cdot k!))$, falls n als Anzahl der Knoten im TLGEFOB mit $n = |V|$ definiert ist und k der Parameter der Funktion ist.

Die nachfolgende **foreach**-Schleife fügt in jede Menge $x \in \mathcal{X}^*$ höchstens n Knoten hinzu, was bedeutet, dass man die Zeit $\mathcal{O}(n \cdot ((n+k-1)!/((n-1)! \cdot k!)))$ benötigt. Die nächste **foreach**-Schleife wird aufgrund der Mächtigkeit der Menge \mathcal{X} $(n+k-1)!/((n-1)! \cdot k!)$ -mal durchlaufen. Darin werden jedes Mal großzügig nach oben abgeschätzt n Knoten herausgenommen und darauf die *Berechne_Möglichkeiten*-Funktion mit einer Laufzeit (laut Lemma 4.12) von $\mathcal{O}(|A|!)$, also $\mathcal{O}(n!)$ aufgerufen. Dies bedeutet, dass für jedes der $(n+k-1)!/((n-1)! \cdot k!)$ Elemente von \mathcal{X}

4.4. Minimierung von Kreuzungen mit k Duplizierungen - CMS(k,M)

höchstens $n!$ weitere Elemente erstellt werden. Nun wird wiederum für jedes dieser $n! \cdot ((n+k-1)!/((n-1)! \cdot k!))$ Elemente die *Erstelle_Baum_ω*-Funktion mit einer Laufzeit von $\mathcal{O}(n^2)$ aufgerufen. Der darin enthaltene Vergleich in der **if**-Abfrage kann dabei vernachlässigt werden. Somit enthält die Menge \mathcal{Y} höchstens $n! \cdot ((n+k-1)!/((n-1)! \cdot k!)) = (n \cdot (n+k-1)!)/k!$ Elemente und die Zeilen 6 bis 16 besitzen eine Laufzeit von $\mathcal{O}(n^2 n! \cdot ((n+k-1)!/((n-1)! \cdot k!))) = \mathcal{O}((n^3 \cdot (n+k-1)!)/k!)$.

Die **for**-Schleife in Zeile 17 bis 26 wird $|V_{t_i}|$ -mal und somit n -mal durchlaufen, wobei darin die *Berechne_Möglichkeiten*-Funktion mit einer Laufzeit von $\mathcal{O}(|A|!)$, also $\mathcal{O}(n!)$ aufgerufen wird. Somit hat jede Menge \mathcal{Z}_i höchstens $n!$ Elemente. Da in der **foreach**-Schleife von Zeile 20 bis 24 nur **if**-Abfragen und Vergleiche mit konstanter Zeit enthalten sind, beträgt die Laufzeit der gesamten **for**-Schleife in Zeile 17 bis 26 $\mathcal{O}(n \cdot n!)$.

Die Laufzeit der *Mische*-Funktion aus Zeile 27 beträgt laut Lemma 4.14 $\mathcal{O}(k^j)$, falls k die Anzahl der Mengen in \mathcal{Z}^* und j die Anzahl der Elemente der größten Menge in \mathcal{Z}^* ist. Da die Menge \mathcal{Z}^* für jeden unteren Knoten des bipartiten Graphen $|V_b|$ die $|\mathcal{Z}_i| = n!$ verschiedenen Anordnungen der Blätter enthält, beträgt die Laufzeit für den Aufruf der *Mische*-Funktion $\mathcal{O}(n^{n!})$.

Wie man beim Funktionsaufruf in Zeile 28 sehen kann, beträgt die Anzahl der Elemente, auf die die Funktion aufgerufen wird ($\{\mathcal{Y}, \mathcal{Z}\}$), nur $k = 2$. Für den Wert von j muss laut Lemma 4.14 die Mächtigkeit der größten Menge von $\{\mathcal{Y}, \mathcal{Z}\}$ gewählt werden. Somit erhält man als Laufzeit für den Aufruf der *Mische*-Funktion $\mathcal{O}(2^{\max\{n^{n!}, (n \cdot (n+k-1)!)/k!\}}) = \mathcal{O}(2^{(n^{n!})})$. Ebenfalls beträgt die Mächtigkeit der Menge \mathcal{W} höchstens $\mathcal{O}(2^{(n^{n!})})$.

Die **foreach**-Schleife über alle Elemente von \mathcal{W} wird demnach $\mathcal{O}(2^{(n^{n!})})$ -mal durchlaufen. Darin enthalten wird für jeden Knoten v im oberen Teil des bipartiten Graphen der Menge $w \in \mathcal{W}$ die **foreach**-Schleife abgearbeitet und die Anzahl der Duplizierungen von v abgespeichert. Aufgrund des Eingabeparameters k , der die Anzahl der Duplizierungen angibt, ist somit auch die Anzahl der Duplizierungen eines einzelnen Knotens durch k beschränkt. Daraus folgt, dass die *Mische*-Funktion in Zeile 33 höchstens $(2^{(n^{n!})})^k$ Elemente erstellt und diese in *TLG* abspeichert. Die Laufzeit beträgt $\mathcal{O}((2^{(n^{n!})}) \cdot (2^{(n^{n!})})^k) = \mathcal{O}(2^{(n^{n!})})^{k+1} = \mathcal{O}(2^{((k+1) \cdot n^{n!})})$.

Nach der Berechnung der Kantenkreuzungen kann das Minimum M^* linear in der Größe der Eingabe ermittelt werden.

Zuletzt werden die **if**-Abfragen und Vergleiche in konstanter Zeit abgearbeitet.

Für den gesamten Algorithmus 8 erhält man somit eine Laufzeit von:

$$\begin{aligned} & \mathcal{O}\left(\frac{(n+k-1)!}{(n-1)! \cdot k!} + \frac{n \cdot (n+k-1)!}{(n-1)! \cdot k!} + \frac{n^3 \cdot (n+k-1)!}{k!} + n \cdot n! + n^{n!} + 2^{(n^{n!})} + 2^{((k+1) \cdot n^{n!})}\right) \\ &= \mathcal{O}\left(2^{((k+1) \cdot n^{n!})}\right) = 2^{\mathcal{O}((k+1) \cdot n^{n!})}. \end{aligned}$$

□

4.4.1 Algorithmus Berechne _{\mathcal{X}}

Lemma 4.9. Für einen TLGEFOB(V, E, Π_t, Π_b), einen Parameter k sowie die Mengen CNODES und \mathcal{X} berechnet der Algorithmus 9 alle möglichen Knotenduplizierungen.

Beweis. Ziel dieses Algorithmus ist es, dass in der Menge \mathcal{X} verschiedene Knotenmengen der Mächtigkeit k enthalten sind, die dupliziert werden sollen. Dies geschieht rekursiv, indem so lange Knoten in die Menge CNODES hinzugefügt werden, bis diese genau k Elemente enthält. Dabei entstehen durch die Wahl der Knoten Abhängigkeiten, die berücksichtigt werden müssen.

Wird in der **foreach**-Schleife von Zeile 1 ein Baumknoten betrachtet, dessen Level gleich $h - 1$ ist (Zeile 2), so ist es praktisch, wenn zusätzlich einige Vorgänger hinzugefügt werden (Zeile 4). Da es möglich sein kann, dass mehrere Male dieselben Knoten hinzugefügt werden (mehrmaliges Duplizieren eines Knotens), so muss die disjunkte Vereinigung verwendet werden. Zeile 5 stellt dabei die Abbruchbedingung

Algorithmus 9: Berechne _{\mathcal{X}} (TLGEFOB, k , CNODES, \mathcal{X})

Input : TLGEFOB(V, E, Π_t, Π_b), k , CNODES, \mathcal{X}

Output : \mathcal{X}

```

1 foreach  $v \in V_T \setminus l$  do
2   if  $level(v) == h - 1$  then
3     foreach  $j \in \{h - 1, \dots, 1\}$  do
4        $NCNODES \leftarrow \{v, \{\text{Knoten der } j \text{ Vorgänger}\}\} \sqcup NCNODES$ 
5       if  $|NCNODES| == k$  then
6          $\mathcal{X} \leftarrow \{NCNODES, \mathcal{X}\}$ 
7       else if  $|NCNODES| < k$  then
8          $\mathcal{X} \leftarrow \text{Berechne}_{\mathcal{X}}(\text{TLGEFOB}, k, CNODES, \mathcal{X})$ 
9       end
10      end
11    end
12  else
13     $DFSNODES \leftarrow modDFS(v)$ 
14    foreach  $U \in DFSNODES$  do
15       $NCNODES \leftarrow \{v, U\} \sqcup NCNODES$ 
16      if  $|NCNODES| == k$  then
17         $\mathcal{X} \leftarrow \{NCNODES, \mathcal{X}\}$ 
18      else if  $|NCNODES| < k$  then
19         $\mathcal{X} \leftarrow \text{Berechne}_{\mathcal{X}}(\text{TLGEFOB}, k, CNODES, \mathcal{X})$ 
20      end
21    end
22  end

```

4.4. Minimierung von Kreuzungen mit k Duplizierungen - CMS(k,M)

dar, wenn die Menge der Knoten in $NCNODES$ gleich der gewünschten Anzahl an Duplizierungen k ist. Andernfalls wird der Algorithmus rekursiv aufgerufen, um weitere Knoten in die Menge $NCNODES$ hinzuzufügen.

Wird in der `foreach`-Schleife von Zeile 1 hingegen ein Baumknoten betrachtet, dessen Level kleiner als $h - 1$ ist (Zeile 11), so werden zusätzlich die Pfade bis zu den Knoten auf Level $h - 1$ hinzugefügt (Zeile 14), da einzelne Baumknoten die Kreuzungen nicht reduzieren können. Dies lässt sich durch eine modifizierte Tiefensuche (DFS) in Zeile 12 realisieren, die beim Erreichen eines Knotens auf Level $h - 1$ den zugehörigen Pfad von v aus als Menge von Knoten in $DFSNODES$ abspeichert. Zeile 15 stellt dabei wieder die Abbruchbedingung dar, wenn die Menge der Knoten in $NCNODES$ gleich der gewünschten Anzahl an Duplizierungen k ist. Andernfalls wird der Algorithmus rekursiv aufgerufen, um weitere Knoten in die Menge $NCNODES$ hinzuzufügen.

Wird auf jeder Rekursionstiefe mindestens ein Knoten hinzugefügt, ist insbesondere durch die Abbruchbedingungen in Zeile 5 und 15 sichergestellt, dass der Algorithmus terminiert. \square

Lemma 4.10. *Für einen TLGEFOB(V, E, Π_t, Π_b), einen Parameter k sowie die Mengen $CNODES$ und \mathcal{X} berechnet der Algorithmus 9 alle möglichen Knotenduplizierungen in Zeit $\mathcal{O}((n+k-1)!/((n-1)! \cdot k!))$.*

Beweis. Der Algorithmus 10 berechnet alle Möglichkeiten, k verschiedene Knoten aus dem Baum-Teil des TLGEFOB-Graphen mit $n = |V|$ Knoten auszuwählen, wobei die Reihenfolge keine Rolle spielt und es insbesondere auch erlaubt ist, mehrere Male denselben Knoten auszuwählen. Aus der Stochastik ist bekannt, dass man dabei $(n+k-1)!/((n-1)! \cdot k!)$ Möglichkeiten erhält.

Bei jedem Rekursionsschritt wird die Menge $CNODES$ um mindestens eins größer. Daraus folgt, dass die Anzahl der Rekursionsstufen kleiner gleich k ist. Somit beträgt die Gesamtaufzeit des Algorithmus 9 $\mathcal{O}((n+k-1)!/((n-1)! \cdot k!))$. \square

4.4.2 Algorithmus Berechne_Möglichkeiten

Lemma 4.11. *Für die Mengen Q , $CSET$ und \mathcal{A} berechnet der Algorithmus 10 alle möglichen Anordnungen der Elemente in Q .*

Beweis. Der Sinn dieses Algorithmus ist es, alle möglichen Anordnungen der Elemente in Q zu berechnen und in die Menge \mathcal{A} abzuspeichern. Dies ist vergleichbar mit dem Ziehen ohne Zurücklegen von n Kugeln aus einer Urne mit n Kugeln.

Da der Algorithmus rekursiv aufgebaut ist, beginnt dieser mit der Abbruchbedingung in Zeile 1, die überprüft, ob die Menge Q bereits leer ist. Falls ja, dann werden die Knoten, die in der Menge $CSET$ angeordnet sind, zur Menge \mathcal{A} hinzugefügt und zurückgegeben.

Falls dies nicht der Fall ist, so wird das jeweils betrachtete Element x der Menge Q an die Menge $CSET$ angehängt und in $NCSET$ gespeichert. Es folgt der rekursive Aufruf der Funktion (Zeile 6), wobei das Element x zuvor aus der Menge Q gelöscht

Algorithmus 10: Berechne_Möglichkeiten($Q, CSET, \mathcal{A}$)

```

Input :  $Q, CSET, \mathcal{A}$ 
Output :  $\mathcal{A}$ 
1 if  $Q == \emptyset$  then
2    $\mathcal{A} \leftarrow \{\mathcal{A}, CSET\}$ 
3 else
4   foreach  $x \in Q$  do
5      $NCSET \leftarrow CSET \setminus x$ 
6      $\mathcal{A} \leftarrow \text{Berechne\_Möglichkeiten}(Q \setminus \{x\}, NCSET, \mathcal{A})$ 
7   end
8 end

```

wird. Das Löschen dieses Elements x sowie die Abbruchbedingung in Zeile 1 stellen sicher, dass der Algorithmus terminiert. \square

Lemma 4.12. *Sei k die Anzahl der Elemente der Menge Q beim initialen Aufruf, dann berechnet der Algorithmus 10 alle möglichen Anordnungen der Elemente in Q in Zeit $\mathcal{O}(k!)$.*

Beweis. Der Algorithmus 10 berechnet alle Möglichkeiten, n Elemente aus der Menge Q mit k Elementen ohne Zurücklegen zu wählen, d.h. die Reihenfolge spielt eine Rolle. Aus der Stochastik ist bekannt, dass man dabei $k!$ Möglichkeiten erhält. Bei jedem Rekursionsschritt wird beim Funktionsaufruf die Menge Q um eins kleiner. Daraus folgt, dass die **foreach**-Schleife in Zeile 4 bis 7 nach jedem Rekursionsschritt einmal weniger durchlaufen wird. Sei Q^* die Menge, mit der die Funktion initial aufgerufen wird, so wird die **foreach**-Schleife beim initialen Aufruf $|Q^*|$ -mal, in Rekursionstiefe 1 $|Q^*| - 1$ -mal und auf Rekursionstiefe k dann $|Q^*| - k$ -mal durchlaufen.

Falls k die Mächtigkeit der Menge Q beim initialen Aufruf ist, so beträgt die Gesamtaufzeit des Algorithmus 10 $\mathcal{O}(k \cdot (k-1) \cdot \dots \cdot 2 \cdot 1) = \mathcal{O}(k!)$. \square

4.4.3 Algorithmus Mische

Lemma 4.13. *Für die Mengen $Q, CSET$ und \mathcal{A} berechnet der Algorithmus 11 alle möglichen Kombinationen von Elementen der Menge Q .*

Beweis. Da der Algorithmus 11 rekursiv aufgebaut ist, beginnt dieser mit der Abbruchbedingung in Zeile 1, die überprüft, ob die Menge Q bereits leer ist. Falls ja, dann werden die Knoten, die in der Menge $CSET$ angeordnet sind, zur Menge \mathcal{A} hinzugefügt und zurückgegeben.

Falls dies nicht der Fall ist, so wird das erste Element aus der Menge Q entnommen und in *first* abgespeichert. Anschließend wird das jeweils betrachtete Element x der Menge *first* an die Menge *CSET* angehängt und in *NCSET* gespeichert. Es folgt

Algorithmus 11: Mische($Q, CSET, \mathcal{A}$)

```

Input :  $Q, CSET, \mathcal{A}$ 
Output :  $\mathcal{A}$ 

1 if  $Q == \emptyset$  then
2    $\mathcal{A} \leftarrow \{\mathcal{A}, CSET\}$ 
3 else
4    $first \leftarrow first(Q)$ 
5   foreach  $x \in first$  do
6      $NCSET \leftarrow CSET \setminus x$ 
7      $\mathcal{A} \leftarrow Mische(Q \setminus first, NCSET, \mathcal{A})$ 
8   end
9 end

```

der rekursive Aufruf der Funktion (Zeile 6), wobei die Menge $first$ zuvor aus der Menge Q gelöscht wird. Das Löschen dieser Menge sowie die Abbruchbedingung in Zeile 1 stellen somit sicher, dass der Algorithmus terminiert. \square

Lemma 4.14. *Sei k die Anzahl der Mengen in Q und j die Anzahl der Elemente der größten Menge in Q , dann berechnet der Algorithmus 11 alle möglichen Kombinationen der Mengen in Q in Zeit $\mathcal{O}(k^j)$.*

Beweis. Der Algorithmus 11 berechnet alle Möglichkeiten, jeweils ein Element aus den $|Q|$ Mengen aus Q auszuwählen, sodass jede Menge genau $|Q|$ Elemente enthält. Obwohl bei jedem Rekursionsschritt beim Funktionsaufruf die Anzahl der Mengen in Q um eins kleiner wird, wird die **foreach**-Schleife in Zeile 5 bis 8 bei jedem Funktionsaufruf so oft durchlaufen, wie es Elemente in $first$ gibt. Im schlechtesten Fall bedeutet dies, dass es k^j verschiedene Möglichkeiten gibt, wenn alle Mengen in Q gleich viele Elemente j besitzen. Hierbei sei k die Anzahl der Mengen in Q und j die Anzahl der Elemente der größten Menge in Q . Somit beträgt die Gesamlaufzeit des Algorithmus 11 $\mathcal{O}(k^j)$. \square

5 Zusammenfassung

In dieser Bachelorarbeit wurde die Methode des Knotenduplizierens zur Minimierung von Kantenkreuzungen für hierarchische Layouts von Bäumen verwendet. Dazu wurden zuerst grundlegende graphentheoretische Begriffe definiert und anschließend bereits existierende Ansätze im Bereich der Kreuzungsminimierung durch Knotenduplizieren vorgestellt. Besonders hervorzuheben ist hierbei die Arbeit von *Ahmed et al.* [AAB+22], die verschiedene Methoden zur Kreuzungsminimierung in bipartiten Graphen mit einseitig festgeschriebener Ordnung, nicht aber für Bäume präsentierte. Anschließend wurden die vier konstruierten Modelle in Kapitel 3 bewertet und es wurde diskutiert, weshalb sich das Modell 1 am besten dafür eignet, die Kreuzungen eines *Two-Layer Graph mit einseitig festgeschriebener Ordnung und Baumerweiterung (TLGEFOB)* zu minimieren. Die Hauptkontribution dieser Arbeit besteht in der Erweiterung der auf dem Paper von *Ahmed et al.* [AAB+22] aufbauenden Methoden, indem für die Instanz eines (*TLGEFOB*) verschiedene Algorithmen vorgestellt wurden. Dabei zeigt die Analyse der Laufzeit, dass insbesondere die Algorithmen *CRS(k)* und *CRSV(k)* vielversprechende Methoden zur Kreuzungsminimierung in hierarchischen Layouts darstellen. Gleichzeitig wird für die Methode *CMS(k, M)* deutlich, dass diese aufgrund der hohen Ausführungszeit keine Anwendung in der Praxis erlaubt.

Ausblick

In zukünftigen Arbeiten könnte vor allem für den Algorithmus *CMS(k, M)* versucht werden, effizientere Methoden zu konstruieren, um die Praxistauglichkeit weiter zu verbessern. Darüber hinaus könnte auch die Implementierung der vorgestellten Methoden in bestehende Visualisierungstools wie dem von *Paul et al.* [PBH+22] und damit einhergehend die Leistung im Kontext einer praktischen Anwendung evaluiert werden. Schließlich könnten für die in Kapitel 3 vorgestellten Modelle 2, 3 und 4 Algorithmen entwickelt und deren Effizienz analysiert werden. Auch die Qualität in Bezug auf die Übersichtlichkeit der resultierenden gezeichneten Graphen könnte mit den hier präsentierten Methoden verglichen werden.

Literaturverzeichnis

- [AAB+22] R. Ahmed, P. Angelini, M. A. Bekos, G. Di Battista, M. Kaufmann, P. Kindermann, M. Nöllenburg, A. Symvonis, A. Villedieu, M. Wallinger. “Splitting Vertices in 2-Layer Graph Drawings”. In: *Unpublished Manuscript* (Jan. 2022) (zitiert auf S. 10, 15–21, 23, 24, 26, 27, 29, 30, 33, 36, 38, 41, 42, 48, 55).
- [AKK23] R. Ahmed, S. Kobourov, M. Kryven. “An FPT Algorithm for Bipartite Vertex Splitting”. In: *Graph Drawing and Network Visualization*. Hrsg. von P. Angelini, R. von Hanxleden. Cham: Springer International Publishing, 2023, S. 261–268. DOI: 10.1007/978-3-031-22203-0_19 (zitiert auf S. 21, 22).
- [CLRS09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009 (zitiert auf S. 10, 31).
- [DPV06] S. Dasgupta, C. H. Papadimitriou, U. Vazirani. *Algorithms*. 1. Aufl. USA: McGraw-Hill, Inc., 2006 (zitiert auf S. 10).
- [EKK+17] D. Eppstein, P. Kindermann, S. Kobourov, G. Liotta, A. Lubiw, A. Maignan, D. Mondal, H. Vosoughpour, S. Whitesides, S. Wismath. “On the Planar Split Thickness of Graphs”. In: *Algorithmica* 80.3 (Juni 2017), S. 977–994. DOI: 10.1007/s00453-017-0328-y (zitiert auf S. 19, 20).
- [FH01] R. Fleischer, C. Hirsch. “Graph Drawing and Its Applications”. In: *Drawing Graphs: Methods and Models*. Hrsg. von M. Kaufmann, D. Wagner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, S. 1–22. DOI: 10.1007/3-540-44969-8_1 (zitiert auf S. 9, 11).
- [NNS+23] S. Nickel, M. Nöllenburg, M. Sorge, A. Villedieu, H.-Y. Wu, J. Wulms. “Planarizing Graphs and their Drawings by Vertex Splitting”. In: *Graph Drawing and Network Visualization*. Cham: Springer International Publishing, 2023, S. 232–246. DOI: 10.1007/978-3-031-22203-0_17 (zitiert auf S. 20, 21).
- [PBH+22] H. Paul, K. Börner, B. W. Herr II, E. M. Quardokus, S. A. Vutukuri, M. Vogelsang, N. Mahadevaswamy. *ASCT+B REPORTER*. 2022. URL: <https://hubmapconsortium.github.io/ccf-asct-reporter/> (besucht am 28.01.2023) (zitiert auf S. 11, 29, 55).
- [Wei01] R. Weiskircher. “Drawing Planar Graphs”. In: *Drawing Graphs: Methods and Models*. Hrsg. von M. Kaufmann, D. Wagner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, S. 23–45. DOI: 10.1007/3-540-44969-8_2 (zitiert auf S. 12, 13).

[yWo] yWorks GmbH. URL: <https://www.yworks.com/products/yed> (besucht am 27.02.2023) (zitiert auf S. 12).