Heuristiken zur Optimierung des Backtrackingverfahrens für 1-Planarität

Bachelorarbeit
von

Manuel Binder

An der Fakultät für Informatik und Mathematik
Lehrstuhl für Informatik mit Schwerpunkt Theoretische Informatik

Gutachter:	Prof. Dr. Ignaz Rutter
Betreuender Mitarbeiter:	Simon Fink, M. Sc.

Bearbeitungszeit: 8. Juni 2020 – 7. September 2020

Selbstständigkeitserklärung

Hiermit versichere ich, dass diese Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt wurden. Textpassagen, die wörtlich oder dem Sinn nach auf Publikationen oder Vorträgen anderer Autoren beruhen, sind als solche kenntlich gemacht. Die Bachelorarbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Passau, 7. September 2020

Kurzzusammenfassung

Ein Graph ist 1-planar, wenn er in der Ebene mit maximal einer Kreuzung pro Kante gezeichnet werden kann. Dies erweitert das Konzept eines planaren Graphen, welcher in der Ebene ohne Kantenkreuzungen gezeichnet werden kann. Die Bestimmung, ob ein Graph 1-planar ist, ist NP-vollständig. Binucci, Didimo und Montecchiani entwickelten einen Algorithmus, der mit exponentiell vielen Schritten überprüfen kann, ob ein Graph 1-planar ist. Dazu wählt der Algorithmus zwei beliebige Kanten aus und kreuzt diese. Dies wird nach und nach mit allen anderen Kanten wiederholt, bis der Graph planar ist oder alle Möglichkeiten durchprobiert wurden. Die Auswahl der Kanten, welche man als nächstes kreuzen will, spielt dabei eine entscheidende Rolle, wie schnell ein planarer Graph entsteht. In dieser Arbeit werden verschiedene Heuristiken vorgestellt, welche bei der Bestimmung des nächsten Kantenpaares verwendet werden können. Die Kombination der einzelnen Heuristiken bringt neue Algorithmen hervor, welche abhängig von der Dichte eines Graphen bevorzugte Anwendungsgebiete aufweisen.

Inhaltsverzeichnis

1. Einleitung **1**

2. Grundlagen **3**
- 2.1. Graphen, Subgraphen und Zusammenhang 3
- 2.2. Planarität und 1-Planarität 4
- 2.3. Spezielle Graphen und Graphzeichnungen 4

3. Backtracking-Algorithmus **7**
- 3.1. Die Backtrackingroutine 7
- 3.2. Kreuzungen und Drachenkanten 9
- 3.3. Die `VerifyNode` Routine 10
- 3.4. Reihenfolge der Abarbeitung des Suchbaumes 11
- 3.5. Beispiel 12

4. Optimierungstechniken zur Priorisierung von Kanten **15**
- 4.1. Skew-Kanten Optimierung 15
- 4.2. K_4 Optimierung 16
- 4.3. Erweiterte Drachenkanten Optimierung 17
- 4.4. Kanteneliminierung an Grad-2-Knoten 18
- 4.5. Kuratowski-Optimierung 19

5. Kombination der Optimierungstechniken **21**
- 5.1. Lokale Kanteneliminierung an Grad-2-Knoten 22
- 5.2. Globale Kanteneliminierung an Grad-2-Knoten 23
- 5.3. Iterative Erhöhung der zulässigen Kreuzungen 23

6. Experimentelle Auswertung **25**
- 6.1. Auswertung einzelner Optimierungstechniken 26
- 6.2. Auswertung kombinierter Optimierungstechniken 31

7. Fazit **39**

Literaturverzeichnis **41**

Anhang **43**
- A. Pseudocode von NextBestCrossing 43
- B. Detaillierte Ergebnisse der Algorithmen 44

1. Einleitung

Einige Graphen können in der Ebene ohne jede Kantenkreuzung gezeichnet werden. Diese nennt man *planare* Graphen. Wie ein Graph gezeichnet wird, spielt eine entscheidende Rolle hinsichtlich der Lesbarkeit, Verständlichkeit und Interpretation. Als Kriterien für eine gute Graphdarstellung haben sich eine minimale Anzahl an Kreuzungen, Symmetrie, möglichst geradlinige Kanten und rechtwinklige Kantenkreuzungen herauskristallisiert (siehe [11, 17, 16, 18]). Somit gelten planare Graphen als gut verständlich.

In der realen Welt gibt es viele Anwendungsfälle in der eine möglichst verständliche Graphvisualisierung wichtig ist. Beispielsweise gibt es das Gebiet der Rechnernetze, bei der jede Komponente mit vielen anderen Komponenten kommunizieren muss. Die Komponenten bilden in diesem Beispiel die Knoten des Graphen und die Kanten sind die Kommunikationsverbindungen zwischen den Netzwerkkomponenten. Dabei existieren zumeist mehrere Pfade zwischen verschiedenen Komponenten, was ab einer gewissen Komplexität zwangsläufig zu Kantenkreuzungen führt. Ein weiteres Beispiel sind soziale Netzwerke. Die Personen des Netzwerkes entsprechen den Knoten im Graphen. Die Kanten zwischen den Personen sagen aus, wer wen kennt. Betrachtet wird nun ein Freundeskreis bestehend aus zehn Personen. Jede der zehn Personen kennt die neun anderen Personen. Das heißt für den Graphen, dass jeder Knoten mit jedem anderen Knoten verbunden ist. Versucht man diesen Graphen zu zeichnen, gelingt dies nicht ohne Kanten zu kreuzen, da dieser Graph nicht planar ist.

Ist ein Graph nicht planar, so kann man als nächstes versuchen, den Graphen mit maximal einer Kreuzung pro Kante zu zeichnen. Dies führt zum Konzept der 1-Planarität. Ein Graph ist *1-planar*, wenn er in der Ebene mit nur einer Kreuzung pro Kante gezeichnet werden kann. Wir beschäftigen uns mit der Frage: Ist ein Graph 1-planar und welche Kanten müssen gegebenenfalls dazu gekreuzt werden? Es wurde bewiesen, dass die Überprüfung eines Graphen auf 1-Planarität NP-vollständig ist [13, Theorem 5]. Im Gegensatz dazu ist es in Linearzeit möglich, einen Graphen auf Planarität zu prüfen.

Um nun solche nicht planaren Graphen auf 1-Planarität zu testen, haben Binucci, Didimo und Montecchiani [3] einen Algorithmus erstellt, der ermittelt, ob ein Graph 1-planar ist. Der Algorithmus benötigt exponentiell viele Schritte. Dieser Tatsache bedient sich auch der Algorithmus, indem er systematisch alle möglichen Paare von Kanten miteinander kreuzt und so versucht einen planaren Graphen aus dem zuvor nicht planaren Graphen zu erstellen. Beim Einfügen einer Kreuzung wird ein Hilfsknoten genau am Schnittpunkt der beiden zu kreuzenden Kanten ergänzt, so dass die gekreuzte Stelle im Graphen planar wird.

Binucci, Didimo und Montecchiani lösen damit auf Basis der bekannten Graph-Datenbanken NORTH und ROME [2] einen Großteil der Instanzen mit bis zu 30 Knoten. Lässt sich ein Graph mit mehr als 30 Knoten in mehrere separat abzuarbeitende Teilgraphen aufteilen, so können auch größere Graphen gelöst werden. Da es exponentiell viele Kombinationen von Kantenkreuzungen zum Durchprobieren gibt und die Rechenzeit in der praktischen Umsetzung begrenzt ist, liefert der Algorithmus für große nicht 1-planare Graphen keine Lösung. Lediglich Graphen, welche zu viele Kanten besitzen, nämlich mehr als $4 \cdot \#\text{Knoten} - 8$ und somit ohnehin nicht mehr 1-planar sein können [15, Theorem 1], wurden als nicht 1-planar klassifiziert. Neben der Bestimmung von 1-planaren Graphen legten Binucci, Didimo und Montecchiani als weiteres Kriterium die Anzahl an eingefügten Kreuzungen zugrunde. Diese Anzahl verglichen sie mit der Anzahl an Kreuzungen eines moderner Planarisierer [10], welcher eine Kante mehr als nur einmal kreuzen darf. Ihr Algorithmus benötigte dabei im Durchschnitt nicht mehr als 1,8 mal der Anzahl an Kreuzungen eines modernen Planarisierers.

Wir wollen auf Grundlage des Algorithmus von Binucci, Didimo und Montecchiani mögliche Heuristiken zur Optimierung der Auswahl des als nächsten zu kreuzenden Kantenpaares vorstellen. Es werden anstelle einer zufälligen Reihenfolge gezielt Kanten gewählt, um schneller einen planaren Graphen zu erhalten. Der Aufbau des Algorithmus erlaubt es außerdem, an einigen Stellen bereits früher abbrechen zu können und somit effektiv nicht alle Kantenpaare durchprobieren zu müssen. Im Rahmen dieser Arbeit wird ein zusätzliches Abbruchkriterium erläutert, um mehr Graphen als nicht 1-planar klassifizieren zu können.

Kombiniert man die verschiedenen Heuristiken erhält man je nach Kombinationsreihenfolge neue, unterschiedliche Algorithmen, welche unterschiedliche Ergebnisse in Abhängigkeit der Dichte des Graphen aufweisen. Somit wird je nach Dichte eines Graphen eine Empfehlung gegeben, welcher Algorithmus die besten Ergebnisse liefert. Wählt man den besten Algorithmus für die NORTH Graphen und betrachtet die Zunahme an Graphen, welche zusätzlich bezüglich des Algorithmus von Binucci, Didimo und Montecchiani klassifiziert werden können, so kommt es zu einer Steigerung um 37,0%. Bei den ROME Graphen beträgt die prozentuale Zunahme 122,4%.

In Kapitel 2 werden die grundlegenden Definitionen und Konzepte erklärt. Anschließend wird in Abschnitt 3 der generelle Ablauf des Algorithmus von Binucci, Didimo und Montecchiani erläutert. In Kapitel 4 werden Möglichkeiten zur gezielten Auswahl von Kantenpaaren vorgestellt, welche in Abschnitt 5 kombiniert werden. Danach werden in Kapitel 6 die vorgenommenen Optimierungen evaluiert und verglichen, um eine effiziente Kombinationsreihenfolge der einzelnen Optimierungen zu finden. Abschließend folgt in Abschnitt 7 das Fazit mit weiterführenden Themen.

2. Grundlagen

In diesem Kapitel werden die grundlegenden Begriffe und Konzepte, auf die diese Arbeit aufbaut, dargelegt. Die Definitionen für die 2-Zusammenhangskomponente und die 1-planare Einbettung sowie die dafür notwendigen Konzepte stammen aus der Bibliographie für 1-planare Graphen [12].

2.1. Graphen, Subgraphen und Zusammenhang

Ein (ungerichteter) *Graph* $G = (V,\ E)$ ist ein geordnetes Paar von einer endlichen, nicht leeren Menge V, den *Knoten*, und einer endlichen Menge E, den *Kanten*. Eine Kante $e \in E$ ist ein ungeordnetes Paar $(u,\ v)$ mit $u,\ v \in V$, wobei u und v als *Endpunkte* bezeichnet werden. Die *Dichte* eines Graphen ist definiert als $\frac{|E|}{|V|}$. Zwei Kanten sind *adjazent*, wenn sie sich einen Knoten als Endpunkt teilen. Der *Grad* eines Knotens v, geschrieben $\deg(v)$, ist die Anzahl seiner adjazenten Kanten. Beim *Kreuzen* zweier Kanten $e_1 = (u_1, v_1), e_2 = (u_2, v_2) \in E$ eines Graphen G entsteht ein neuer Graph $G' = (V \cup \{x\}, (E \setminus \{e_1, e_2\}) \cup \{(u_1, x), (x, v_1), (u_2, x), (x, v_2)\})$ wobei x als Hilfsknoten in den Graphen eingefügt wird. Kanten der Form (v, v) werden als *Schlinge* bezeichnet. Liegen zwischen zwei Knoten v, w mehrere Kanten, so heißen diese *Mehrfachkanten*. Diese Arbeit beschäftigt sich nur mit *einfachen* Graphen, bei denen keine Schlingen oder Mehrfachkanten vorkommen. Liegt zwischen jedem Knotenpaar eine Kante, so ist der Graph *vollständig*.

Häufig lässt sich ein Problem auf Teilen des Graphen einzeln lösen. Dazu benötigen wir folgende Konzepte. Ein *Subgraph* G' eines Graphen G ist ein Graph $G' = (V',\ E')$, so dass $V' \subseteq V$ und $E' \subseteq E$ gilt. Der Subgraph G', der von V' *induziert* wird, ist der Graph $G' = (V',\ E')$ mit $E' \subseteq E$, wobei E' alle Kanten $(u,\ v) \in E$ enthält mit $u \in V'$ und $v \in V'$. Ein *Pfad* ist ein Graph mit Knoten $V = \{v_1, ..., v_n\}$ und Kanten $E = \{e_1, ..., e_{n-1}\}$ mit $e_i = (v_i, v_{i+1})$ für $i \in \{1, ..., n-1\}$. Eine *Zusammenhangskomponente* $G' = (V',\ E')$ eines Graphen G ist ein maximaler Subgraph von G, so dass für jedes Knotenpaar $u, v \in V'$ ein Pfad von u nach v in G' existiert. Ein Graph mit exakt einer Zusammenhangskomponente heißt *zusammenhängend*. Ein Graph G ist 2-*zusammenhängend*, wenn ein beliebiger Knoten $v \in V$ entfernt werden kann, ohne die Anzahl an Zusammenhangskomponenten von G zu erhöhen. Eine 2-*Zusammenhangskomponente* ist ein maximal 2-zusammenhängender Subgraph G' eines Graphen G. Jeder Graph lässt sich in seine 2-Zusammenhangskomponenten zerlegen, was später im Algorithmus in Kapitel 3 ein wesentliches Mittel zur Verringerung der Größe eines Graphen ist. Abbildung 2.1 zeigt eine Zerlegung eines Graphen in seine 2-Zusammenhangskomponenten.

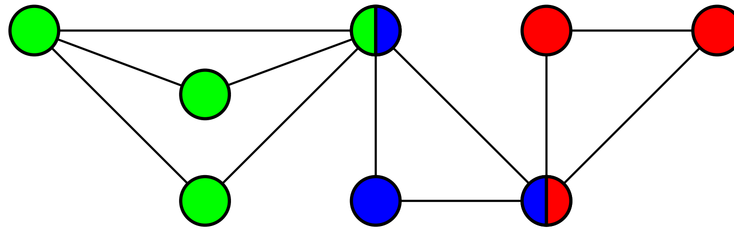


Abbildung 2.1.: Ein Graph, welcher sich in drei farblich markierte 2-Zusammenhangskomponenten aufteilt.

2.2. Planarität und 1-Planarität

Eine *Zeichnung* eines Graphen $G = (V, E)$ bildet jeden Knoten $v \in V$ auf einen eindeutigen Punkt p_v in der Ebene ab. Jede Kante $(u, v) \in E$ wird dabei als Jordan-Kurve von p_u zu p_v dargestellt. Eine Zeichnung ist *planar*, wenn sich keine Kanten kreuzen. Wir nennen einen Graphen *planar*, wenn er eine planare Zeichnung besitzt. In einer Zeichnung bilden die von Kanten eingeschlossenen Flächen die *Facetten*. Ebenfalls zählt dazu die *äußere Facette*, welche den Graphen umgibt. Eine *Einbettung* eines Graphen G ist eine Äquivalenzklasse von Zeichnungen von G mit der gleichen Facettenmenge und äußeren Facette. Eine *planare Einbettung* ist eine Einbettung bezüglich einer Äquivalenzklasse von planaren Zeichnungen. Die *Planarisierung* eines nicht planaren Graphen G ist ein planarer Graph G', welcher durch die Ersetzung aller Kreuzungspunkte mit Hilfsknoten ermittelt wird. Eine 1-*planare* Zeichnung eines Graphen ist eine Zeichnung, bei der jede Kante maximal einmal gekreuzt ist. Ein Graph ist 1-*planar*, wenn er eine 1-planare Zeichnung besitzt. Eine 1-*planare Einbettung* ist eine Einbettung bezüglich einer Äquivalenzklasse von 1-planaren Zeichnungen.

Zusätzlich weiß man über 1-planare Graphen, dass jeder Graph mit weniger als sieben Knoten 1-planar ist, weil der vollständige Graph von diesem Graphen 1-planar ist (siehe Abbildung 2.2). Wir bezeichnen einen Graphen als *trivial* 1-*planar*, wenn der Graph planar ist oder weniger als sieben Knoten besitzt. Anzumerken ist dabei, dass die Überprüfung auf Planarität in Linearzeit erfolgt [5]. Analog dazu ist bekannt, dass ein 1-planarer Graph nicht zu viele Kanten im Verhältnis zu Knoten besitzen darf. Deshalb nennen wir einen Graphen *trivial nicht* 1-*planar*, wenn $|E| > 4 \cdot |V| - 8$ gilt [15, Theorem 1]. In Abbildung 2.1 erkennt man, dass jede der drei Komponenten trivial 1-planar ist, da jede sowohl planar ist als auch weniger als sieben Knoten besitzen. Hat ein Graph genau $4 \cdot |V| - 8$ Kanten, so heißt er *optimal*. In 2016 zeigte Brandenburg [6], dass ein optimaler Graph in Linearzeit auf 1-Planarität getestet und eine 1-planare Einbettung berechnet werden kann.

2.3. Spezielle Graphen und Graphzeichnungen

Zunächst benötigen wir zwei besondere Graphen, den K_5 und $K_{3,3}$ (siehe Abbildung 2.3), welche in jedem nicht planaren Graphen als Substruktur enthalten sind [14]. Ein K_n ist ein Graph bestehend aus n Knoten, wobei jeder Knoten mit jedem anderen Knoten verbunden ist. Ein $K_{3,3}$ ist ein Graph mit sechs Knoten, wobei sich diese disjunkt in zwei dreielementige Knotenmengen aufteilen. Jeder Knoten der ersten Knotenmenge ist mit jedem Knoten der zweiten Knotenmenge verbunden. Ein *Drachenviereck* ist eine Zeichnung eines K_4 (siehe Abbildung 2.4). Dabei wird die äußere Facette durch vier kreuzungsfreie Kanten gebildet. Diese nennen wir *Drachenkanten*. Die verbleibenden zwei Kanten kreuzen sich im Inneren der Drachenkanten. Diese Struktur benötigen wir beim Einfügen einer Kreuzung, da man zusätzlich Drachenkanten in den Graphen einfügen kann, falls sie noch nicht vorhanden sind, ohne seine 1-Planarität zu beeinflussen.

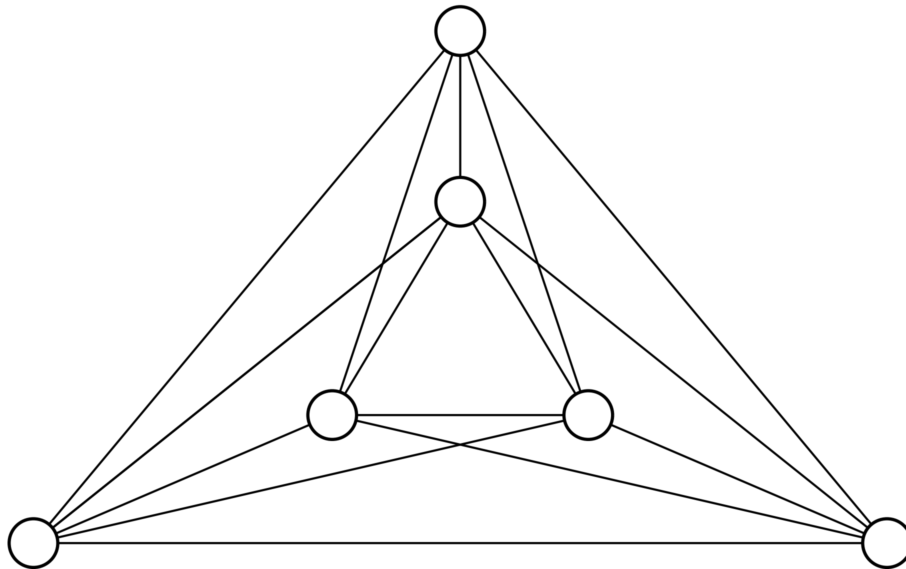


Abbildung 2.2.: Eine 1-planare Zeichnung des vollständigen Graphen K_6. [9, Figure 1]

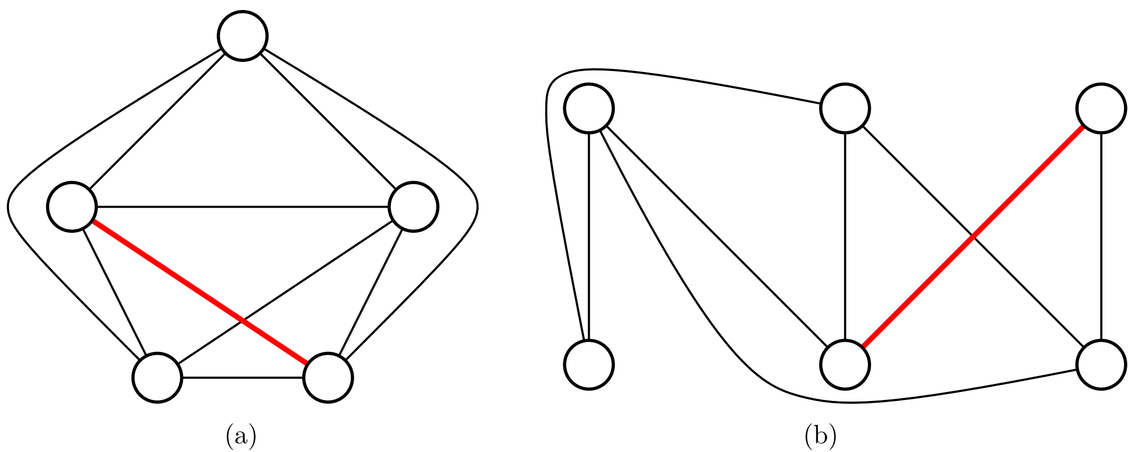


Abbildung 2.3.: Zeichnungen eines (a) K_5 und (b) $K_{3,3}$ mit roter Kante, die gekreuzt wird.

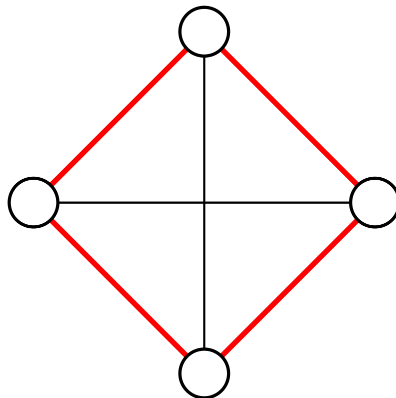


Abbildung 2.4.: Ein Drachenviereck mit rot markierten Drachenkanten.

3. Backtracking-Algorithmus

In diesem Kapitel wird der entwickelte Algorithmus von Binucci, Didimo und Montecchiani [3] erklärt. Da dieser auf einer bekannten Problemlösungsmethode basiert, dem Backtracking, wird dieses zuerst erläutert. Genauer wird dabei nach dem „Versuch und Irrtum“-Prinzip (trial and error) vorgegangen. Es wird ein möglicher Schritt zur Lösung des Problems gewählt. Dabei erhält man eine Teillösung. Auf Grundlage dieser Teillösung wird wieder der nächste Schritt gewählt bis eine Lösung des Problems gefunden ist oder bis eine Teillösung nicht mehr zu einer Lösung führen kann. Kann eine Teillösung nicht mehr zu einer Lösung führen, so wird zurück zur letzten Teillösung gegangen, welche noch nicht überprüfte Auswahlmöglichkeiten bietet und ein anderer Schritt probiert.

Binucci, Didimo und Montecchiani haben ein Backtrackingverfahren entwickelt, welches mit exponentiell vielen Schritten ermittelt, ob ein Graph 1-planar ist. Der Algorithmus heißt `1PlanarTester`. Ist der Graph 1-planar, so wird eine 1-planare Einbettung zurückgegeben. Zunächst wird der Graph in seine 2-Zusammenhangskomponenten aufgeteilt (siehe Abbildung 2.1), welche unabhängig voneinander betrachtet werden können. Der Grund hierfür ist, dass ein Graph genau dann 1-planar ist, wenn jede seiner 2-Zusammenhangskomponenten 1-planar ist [3, Property 1]. Daraus resultieren folgende Bedingungen, um auf die 1-Planarität eines Graphen G schließen zu können:

- Existiert eine 2-Zusammenhangskomponente, welche nicht 1-planar ist, so ist G nicht 1-planar.
- Sind alle 2-Zusammenhangskomponenten 1-planar, so ist G 1-planar.

Da es somit ausreichend ist, die 2-Zusammenhangskomponenten eines Graphen zu betrachten, werden diese im Folgenden nur noch als Komponenten bezeichnet.

Als nächstes betrachten wir, wie eine Komponente $G_i = (V_i, E_i)$ abgearbeitet wird. Der `1PlanarTester` überprüft zunächst die Komponente, ob sie trivial 1-planar ist. Ist sie trivial 1-planar, so wird eine 1-planare Einbettung zurückgegeben. Ansonsten wird geprüft, ob die Komponente trivial nicht 1-planar ist. Ist sie trivial nicht 1-planar, so wird ein negatives Ergebnis zurückgegeben. Trifft keiner der beiden Fälle zu, so startet die eigentliche Backtrackingroutine.

3.1. Die Backtrackingroutine

Das Vorgehen der Backtrackingroutine lässt sich in zwei Schritte einteilen:

Schritt 1: Wähle zwei Kanten aus und kreuze diese.

Schritt 2: Überprüfe die entstandene Teillösung. Dieser Algorithmus heißt `VerifyNode`. Dabei können drei Fälle eintreten:

a) Die Teillösung ist eine planare Zeichnung von der Planarisierung der Komponente.
Die 1-planare Einbettung der Komponente wird zurückgegeben.
b) Über die Teillösung kann noch keine Entscheidung bezüglich 1-Planarität getroffen werden.
Es wird mit Schritt 1 weitergemacht.
c) Die Teillösung kann zu keiner 1-planaren Einbettung führen.
Es wird die letzte Kreuzung rückgängig gemacht und mit Schritt 1 fortgefahren, wobei eine andere Kreuzung gewählt werden muss.

Betrachten wir zuerst Schritt 1. Sei dazu $\overline{E}$ die Menge aller möglichen Paare von Kanten der Komponente mit der Einschränkung, dass die Kanten eines Paares nicht zueinander adjazent sein dürfen. Würde man eine Kreuzung mit zwei adjazenten Kanten einfügen, so kann man diese Kreuzung immer entfernen, ohne Einfluss auf die 1-Planarität der Komponente zu nehmen. Um eine Ordnung auf den Elementen von $\overline{E}$ zu definieren, benötigen wir eine beliebige Permutation σ von $\overline{E}$ (siehe Abbildung 3.1). Diese gibt an, in welcher Reihenfolge die Paare nacheinander gekreuzt werden, beginnend mit $\sigma(0)$.

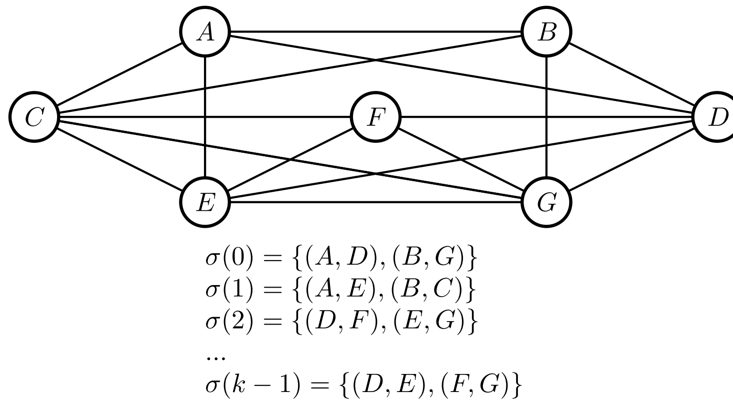


Abbildung 3.1.: Eine 2-Zusammenhangskomponente mit einer Permutation σ von $\overline{E}$.

Kreuzt man $\sigma(0)$, so erhält man eine Teillösung auf deren Basis erneut Kreuzungen stattfinden können. Dabei gilt zu beachten, dass in jeder darauf aufbauenden Teillösung $\sigma(0)$ gekreuzt ist. Kommt man zu der Entscheidung, dass aufbauend auf der Kreuzung von $\sigma(0)$ keine Lösung erreicht werden kann, so ergibt sich eine neue Teillösung, in der $\sigma(0)$ nicht mehr gekreuzt werden darf. Dadurch bildet sich ein binärer Suchbaum, welcher in Abbildung 3.2 dargestellt ist. Jede der Ebenen steht genau für ein Paar an Kanten $\sigma(i)$ mit $0 \leq i < |\overline{E}|$. Jeder Knoten $v_{c_0 \dots c_j}$ mit $c_i \in \{0, 1\}$ enthält in seiner Kodierung, ob $\sigma(i)$ gekreuzt ist für $0 \leq i < j$, wobei j der aktuellen Ebene im Suchbaum entspricht. Die Ebene der Wurzel ist 0. Zum Beispiel bedeutet Knoten v_{011}, dass $\sigma(0)$ nicht gekreuzt ist und $\sigma(1)$, $\sigma(2)$ gekreuzt sind. Jedem Knoten $v_{c_0 \dots c_j}$ des Suchbaumes kann ein Graph $G(v_{c_0 \dots c_j})$ zugeordnet werden, welcher die kodierten Kreuzungen enthält. Die Funktion $C_{c_0 \dots c_j}$ gibt für einen Knoten $v_{c_0 \dots c_j}$ an, ob ein Kantenpaar $\sigma(i)$ in $G(v_{c_0 \dots c_j})$ gekreuzt ist. Für Paare von Kanten, welche erst in einer tieferen Ebene überprüft werden, wird `undefiniert` zurückgegeben.

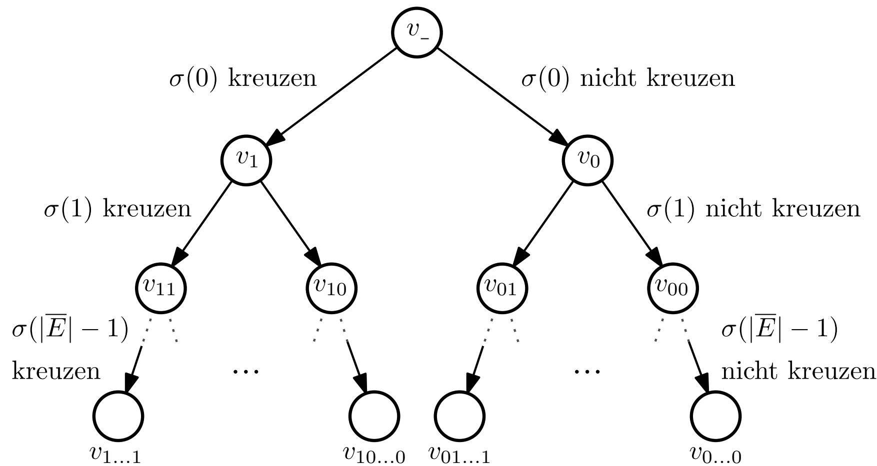


Abbildung 3.2.: Der binäre Suchbaum des Algorithmus.

3.2. Kreuzungen und Drachenkanten

In Abschnitt 2.3 wurde das Konzept des Drachenvierecks eingeführt. Dies findet nun Anwendung beim Einfügen einer Kreuzung in eine Komponente G. Eine Kreuzung besteht immer aus einem Kantenpaar $(e_1, e_2) = \sigma(i) \in \overline{E}$. Betrachtet man nur den Graphen bestehend aus den beiden Kanten, so entspricht das Kreuzen der Planarisierung dieses Graphen.

Da die Kanten e_1 und e_2 somit gekreuzt sind, dürfen sie nicht nochmal gekreuzt werden, damit die ursprüngliche Komponente G 1-planar sein kann. Fügt man nun zwischen den Endpunkten von e_1 und e_2 Kanten ein, falls sie noch nicht im Graphen existieren, so entsteht ein Drachenviereck. Die Drachenkanten kann man beliebig nahe an e_1 und e_2 anlegen (siehe Abbildung 3.3). Da nun e_1 und e_2 nicht mehr gekreuzt werden dürfen und die Drachenkanten beliebig nahe an sie angelegt werden können, brauchen diese auch nicht mehr gekreuzt werden, da ansonsten auch e_1 oder e_2 gekreuzt werden würde. Somit hat das Einfügen dieser zusätzlichen Drachenkanten keine Auswirkungen auf die 1-Planarität der Komponente G. Damit wird die Kantenzahl erhöht, was die Chance verbessert, dass bereits früh im Suchbaum bei einer Teillösung ein Abbruchkriterium angewendet werden kann.

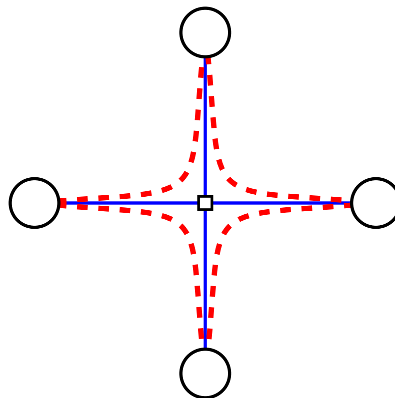


Abbildung 3.3.: In blau die eingefügte Kreuzung mit quadratischem Hilfsknoten und rot gestrichelten Drachenkanten.

In Abbildung 3.4 ist das Einfügen einer Kreuzung am Beispiel der Kanten (A, D) und (B, G) dargestellt. Basierend auf Abbildung 3.1 enthält die Komponente bereits drei der vier Drachenkanten. Die zusätzlich eingefügte Drachenkante ist rot gestrichelt und die

bereits vorhandenen drei Kanten sind grün markiert. Dies schließt die Beschreibung von Schritt 1 ab.

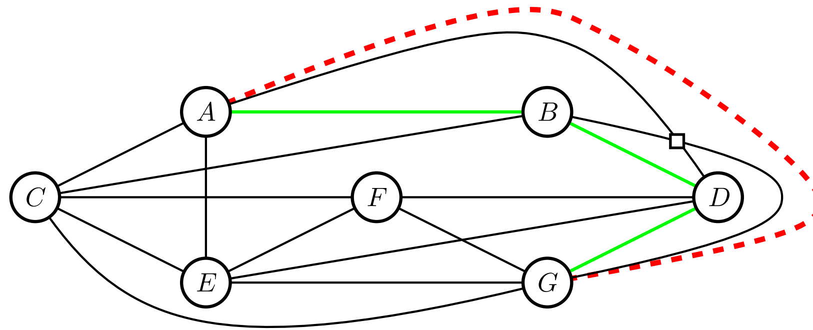


Abbildung 3.4.: Beispiel für das Einfügen einer Kreuzung in eine Komponente basierend auf Abbildung 3.1. Der Hilfsknoten ist als weißes Quadrat dargestellt. Die neue Drachenkante ist rot gestrichelt. Die bereits vorhandenen Drachenkanten sind in grün hervorgehoben.

3.3. Die `VerifyNode` Routine

In Schritt 2 des Algorithmus wird basierend auf dem Graphen $G(v_{c_0 \ldots c_j})$ des aktuellen Knotens $v_{c_0 \ldots c_j}$ im Suchbaum eine der drei aufgezählten Entscheidungen getroffen. Der Algorithmus, der dies entscheidet, wird `VerifyNode` genannt.

Die `VerifyNode` Routine entscheidet darüber, wie an einem Knoten $v_{c_0 \ldots c_j}$ im Suchbaum weiter verfahren wird. Dazu existieren verschiedene Rückgabewerte (SOL, CNT, CUT), um auszudrücken, ob weiter im Suchbaum abgestiegen werden muss. Mit SOL wird ausgedrückt, dass eine 1-planare Einbettung für die Komponente gefunden wurde (entspricht Schritt 2a). Wird hingegen CNT zurückgegeben, so kann noch keine Entscheidung getroffen werden und es muss weiter im Suchbaum abgestiegen werden (entspricht Schritt 2b). Die letzte Option CUT beschreibt, dass unterhalb des Knotens $v_{c_0 \ldots c_j}$ keine Einbettung für die Komponente sein kann (entspricht Schritt 2c) und es muss im Suchbaum wieder zurückgegangen werden.

Um diese Entscheidung treffen zu können, wird zunächst überprüft, ob Kanten mehrfach oder Drachenkanten gekreuzt sind. Falls dies vorliegt, wird CUT zurückgegeben. Um sich diesen CUT zu sparen, kann auch bereits bei der Wahl der Kanten darauf geachtet werden, dass diese Kombination keine der eben aufgezählten Kriterien verletzt. Ansonsten betrachtet man die Kanten im Graphen $G(v_{c_0 \ldots c_j})$, welche bereits gekreuzt wurden oder in keinem Teilbaum ausgehend von $v_{c_0 \ldots c_j}$ noch gekreuzt werden dürfen. Das sind alle Kanten, die unterhalb von $v_{c_0 \ldots c_j}$ nicht mehr gekreuzt werden dürfen, weil sie schon vollständig abgearbeitet wurden. Dies entspricht dem Konzept der gesättigten Kanten, welches im Folgenden definiert wird.

Definition 3.1. *Eine Kante $e \in E$ eines Graphen $G(v_{c_0 \ldots c_j}) = (V, E)$ ist* gesättigt, *wenn mindestens eines der nachfolgenden Kriterien zutrifft.*

- *Die Kante e ist gekreuzt in $G(v_{c_0 \ldots c_j})$. Dies bedeutet es existiert ein Kantenpaar $(e_1, e_2) \in \overline{E}$, welches e enthält, mit $C_{c_0 \ldots c_j}(e_1, e_2) = wahr$.*
- *Die Kante e ist eine Drachenkante.*
- *Es existiert kein Kreuzungspartner mehr für die Kante e, weil alle möglichen Partner bereits ausprobiert wurden. Formal bedeutet dies, dass für jedes Kantenpaar $\sigma(i) \in \overline{E}$, welches e enthält, gilt $C_{c_0 \ldots c_j}(\sigma(i)) = falsch$.*

- *Es existiert kein Kreuzungspartner mehr für die Kante e, weil alle möglichen Partner gekreuzt sind. Formal bedeutet dies, dass für jedes Kantenpaar $(e_1, e_2) \in \overline{E}$, welches e enthält, gilt, dass die andere Kante Teil eines Kantenpaares $\sigma(i)$ ist mit $C_{c_0 \dots c_j}(\sigma(i)) =$ wahr.*

Die tatsächliche Verwendung der gesättigten Kanten zur Bestimmung, ob im Suchbaum weiter gesucht werden muss, findet sich in Algorithmus 3.1. Dieser wendet zunächst alle Kreuzungen des Knotens $v_{c_0 \dots c_j}$ auf den Ursprungsgraphen G an und man erhält den neuen Graphen $G(v_{c_0 \dots c_j})$. Anschließend wird der Teilgraph G_s, welcher nur aus den gesättigten Kanten besteht, extrahiert. Nach Binucci, Didimo und Montecchiani [3, Lemma 2] enthält der Graph G_s nur Kanten, welche schon vollständig abgearbeitet sind. Dies führt dazu, dass CUT zurückgegeben werden kann, wenn G_s nicht planar ist, da dieser Graph auch in allen Teilbäumen unterhalb $v_{c_0 \dots c_j}$ nicht planar werden kann [3, Lemma 3]. Auch wenn $G(v_{c_0 \dots c_j})$ zu viele Kanten hat, um noch 1-planar sein zu können, wird CUT zurückgeliefert. Ist jedoch der Graph $G(v_{c_0 \dots c_j})$ planar, haben wir die richtigen Kreuzungen eingefügt und der Ursprungsgraph G ist 1-planar. In diesem Fall wird SOL und eine 1-planare Einbettung zurückgegeben. Tritt keiner dieser Fälle ein, muss weiter im Suchbaum abgestiegen werden und CNT wird zurückgegeben.

Anzumerken ist hier eine Abweichung vom Vorgehen von Binucci, Didimo und Montecchiani, welche aber nicht weiter ins Gewicht fällt. Anstatt die Teillösung auf Planarität zu prüfen, wie es im Algorithmus 3.1 gemacht wird, wird die Teillösung vervollständigen. Das bedeutet, ausgehend vom Knoten $v_{c_0 \dots c_j}$ wird ein beliebiger Pfad zu einem Blatt gewählt. Die dabei neu gewählten Kreuzungen werden eingefügt. Anschließend wird dieser Graph auf Planarität getestet. Der Algorithmus 3.1 vervollständigt die Lösung in dem Sinne, dass keine neuen Kreuzungen eingefügt werden. Also immer ausgehend von $v_{c_0 \dots c_j}$ der rechte Weg bis zum Blatt $v_{c_0 \dots c_j 0 \dots 0}$ gewählt wird. Dies hat den Vorteil, dass SOL zurückgegeben werden kann, wenn der Graph $G(v_{c_0 \dots c_j})$ planar ist. Dennoch wirkt sich dies nicht auf das Gesamtergebnis aus, da die zufällig gewählten Kreuzungen ohnehin in tieferen Ebenen überprüft werden. In welcher Reihenfolge tiefere Ebenen besucht werden, wird im Folgenden erläutert.

Algorithmus 3.1 : Pseudocode von der VERIFYNODE Routine

Input : Graph $G = (V, E)$, Knoten $v_{c_0 \dots c_j}$
Output : SOL, CUT, CNT

```
   // Wende Kreuzungen auf G an
1  G(v_{c0...cj}) ← applyCrossings(G, v_{c0...cj})
2  G_s ← saturatedGraph(G(v_{c0...cj}), v_{c0...cj})
3  if !isPlanar(G_s) or isTrivialNot1Planar(G(v_{c0...cj})) then
4      return CUT
5  else if isPlanar(G(v_{c0...cj})) then
6      return SOL
7  else
8      return CNT
```

3.4. Reihenfolge der Abarbeitung des Suchbaumes

Nach dem Aufbau des Algorithmus ist nun die konkrete Reihenfolge, wie die einzelnen Knoten im Suchbaum besucht werden, relevant. Zunächst ist anzumerken, dass es $k \in \mathcal{O}(|V|^2)$ viele Kreuzungskombinationen und somit Ebenen im Suchbaum gibt, weil $|E| \leq 4 \cdot |V| - 8$ gilt. Der Suchbaum wird deshalb mittels Tiefensuche durchlaufen, damit die

Platzkomplexität des Algorithmus in $\mathcal{O}\left(|V|^2\right)$ liegt. Binucci, Didimo und Montecchiani gehen dabei so vor, dass im Suchbaum solange wie möglich nach links unten gegangen wird. Es werden also die ersten Kreuzungen basierend auf σ bis zum ersten CUT eingefügt. Gibt $VerifyNode$ beim Knoten $v_{c_0 \ldots c_j}$ CUT zurück, so wird zum Elternknoten von $v_{c_0 \ldots c_j}$ gegangen also $v_{c_0 \ldots c_{j-1}}$ und überprüft, ob noch ein unbesuchter Pfad existiert. Wann ja, wird dieser als nächstes gewählt. Ansonsten wird zum nächsten Elternknoten zurückgesprungen und es findet wieder die gleiche Überprüfung statt. Dieser Vorgang wiederholt sich bei jedem CUT. Hat man den gesamten Suchbaum ohne Erfolg durchsucht, so ist der Graph nicht 1-planar.

Neben diesem Vorgehen kann auch jede andere beliebige Durchlaufreihenfolge gewählt werden. Wichtig ist dabei nur, dass alle Knoten des Suchbaumes besucht werden. Eine Ausnahme hiervon bilden Knoten, welche durch einen CUT abgeschnitten werden.

3.5. Beispiel

Betrachten wir zur Veranschaulichung des Ablaufs die beiden Abbildungen 3.5 und 3.6. Sei dazu G die 2-Zusammenhangskomponente und σ die Permutation wie in Abbildung 3.1. Es wird zuerst das Kantenpaar $\sigma(0)$ gekreuzt. Also wird Kante (A, D) mit (B, G) gekreuzt und es ergibt sich der Graph $G(v_1) = (V', E')$, welcher in Abbildung 3.4 zu sehen ist. Nimmt man nun die gesättigten Kanten, so ergibt sich der Graph G_s aus Abbildung 3.5. Da G_s planar und $G(v_1)$ nicht planar ist, liefert $VerifyNode$ als Ergebnis CNT zurück. Außerdem ist zu beachten, dass das Verhältnis der Kantenzahl $|E'| = 19$ zur Knotenzahl $|V'| = 8$ nicht das Limit für trivial nicht 1-planar übersteigt. Analog wird mit $\sigma(1)$ verfahren. Nach $\sigma(2)$ wird SOL von $VerifyNode$ zurückgegeben, weil der in Abbildung 3.6 dargestellte Graph planar ist. Somit ist eine Planarisierung von G gefunden und G schließlich 1-planar.

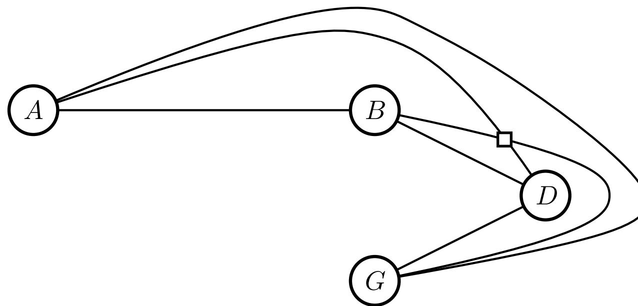


Abbildung 3.5.: Graph G_s bestehend aus nur den gesättigten Kanten.

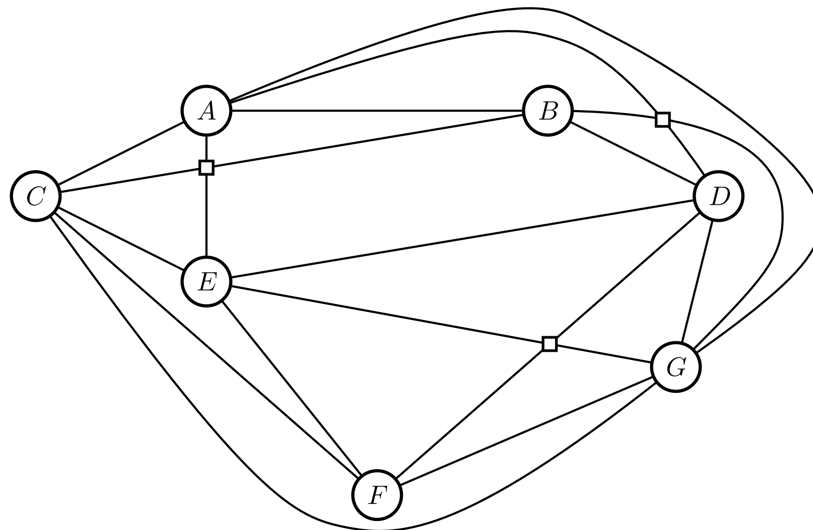


Abbildung 3.6.: Eine mögliche planare Zeichnung mit eingezeichneten Hilfsknoten und Drachenkanten basierend auf der Planarisierung des Graphen von Abbildung 3.1.

4. Optimierungstechniken zur Priorisierung von Kanten

Dieses Kapitel befasst sich mit den Optimierungsmöglichkeiten, um mehr und größere Komponenten lösen zu können. Der generelle Ablauf des Backtrackings, welcher in Abschnitt 3 vorgestellt wurde, ändert sich dabei nicht. Was sich jedoch ändert, ist die Wahl der Permutation σ und die Durchlaufreihenfolge. Das Problem bei der willkürlichen Wahl der Permutation σ ist, dass der Suchbaum bei größeren Komponenten nicht mehr in einer annehmbaren Zeit komplett durchsucht werden kann. Die Zielsetzung ist somit das Suchen von passenden Kantenpaaren, um schneller eine 1-planare Einbettung zu finden oder das Ausschließen von Kanten, um den Suchbaum zu verkleinern. Die Kriterien für die Bestimmung der Kantenpaare findet in diesem Kapitel statisch statt. Das heißt, es wird vor dem Backtracking der Graph analysiert und während des Backtrackings auf Basis dieser Analyse Kanten ausgewählt.

4.1. Skew-Kanten Optimierung

Diese Optimierung stammt von Binucci, Didimo und Montecchiani [3] und wurde auch von Ihnen in Ihrem Algorithmus umgesetzt. Eine *Skew-Kante* eines Graphen $G = (V, E)$ ist jede Kante $e \in E$, welche bei Entfernung dazu führt, dass der resultierende Graph planar ist.

Zur Ermittlung wird immer eine Kante aus G entfernt und überprüft, ob dieser nun planar ist. Ist G planar, so hat man eine Skew-Kante gefunden und kann mit den verbleibenden Kanten analog verfahren. Für Graphen, welche nicht durch die Entfernung einer Kante planar werden, können somit keine Skew-Kanten ermittelt werden.

Hat man nun die Menge S der Skew-Kanten ermittelt, so wird die Menge der möglichen Kreuzungen $\overline{E}$ beschränkt, so dass jedes Kantenpaar mindestens eine Kante aus S beinhaltet. Das führt dazu, dass die ursprüngliche Länge von $\overline{E}$ von $\mathcal{O}\left(|V|^2\right)$ auf $\mathcal{O}\left(|V| \cdot |S|\right)$ reduziert wird. Der Suchbaum schrumpft dabei auf eine Größe von zuvor $\mathcal{O}\left(2^{|V|^2}\right)$ auf $\mathcal{O}\left(2^{|V| \cdot |S|}\right)$. Binucci, Didimo und Montecchiani schränken dabei die Größe von S auf genau eine Skew-Kante ein. Gilt $|S| > 1$ so wird genau eine Kante aus S ausgewählt und die Kreuzungsmöglichkeiten auf diese eine Kante beschränkt. Dies stellt sicher, dass es nicht zu viele Skew-Kanten sind.

Wird beim Durchsuchen des neuen Suchbaumes eine Lösung gefunden, so ist der Graph 1-planar. Wird man jedoch nicht fündig, so kann der Graph dennoch 1-planar sein. Es kann nämlich vorkommen, dass die Entfernung von zwei oder mehr Kanten aus $E \setminus S$ dazu führt, dass der Graphen 1-planar wird. Um vergleichbar mit anderen Algorithmen zu sein, wird anschließend die normale Backtrackingroutine gestartet. Dadurch bleibt der Überblick über die so lösbaren Instanzen erhalten. Abschließend bleibt anzumerken, dass selbst für diese fast planaren Graphen, es NP-schwer ist, die Anzahl an notwendigen Kreuzungen zu berechnen oder den Graphen auf 1-Planarität zu testen, wie Cabello und Mohar [7] beweisen.

4.2. K_4 Optimierung

Bei dieser Optimierung wird die Methodik von Abschnitt 3.2 ausgenutzt. Beim Einfügen einer Kreuzung werden zusätzlich die Drachenkanten ergänzt, welche nicht mehr gekreuzt werden dürfen. Sind bereits Kanten im ursprünglichen Graph vorhanden, dürfen diese auch nicht mehr gekreuzt werden. Die Idee ist nun, dass K_4 Kanten bevorzugt zum Kreuzen verwendet werden, um möglichst viele Kanten vom ursprünglichen Graph zu treffen. Diese Kanten werden dann zu Drachenkanten und dürfen nicht mehr gekreuzt werden. Ist nun eine dieser Kanten Teil eines Kantenpaares, so fällt diese Option im Suchbaum weg.

Ein *Grad-2-Pfad* $(v_0, v_1, ..., v_i)$ mit $i \geq 1$ ist ein Pfad mit $\deg(v_l) = 2$ für $0 < l < i$. Um nun auch K_4 Strukturen, wie in Abbildung 4.1b dargestellt, zu finden, benötigen wir den *erweiterten* K_3 (siehe Abbildung 4.1a). Dieser besteht aus drei Knoten u, v, w und k Kanten mit $k \geq 3, k \in \mathbb{N}$, wobei ein Grad-2-Pfad zwischen den Knoten u und w sowie v und w existieren muss. Der Knoten u muss adjazent zu v sein. Die Grundkante (u, v) wird dabei als *kreuzbar* bezeichnet. Der *erweiterte* K_4 (siehe Abbildung 4.1b) besteht aus zwei erweiterten K_3 mit Knoten u, v, w_1 und u, v, w_2, wobei w_1 zu w_2 adjazent ist. Er besitzt zwei kreuzbare Kanten, nämlich (u, v) und (w_1, w_2).

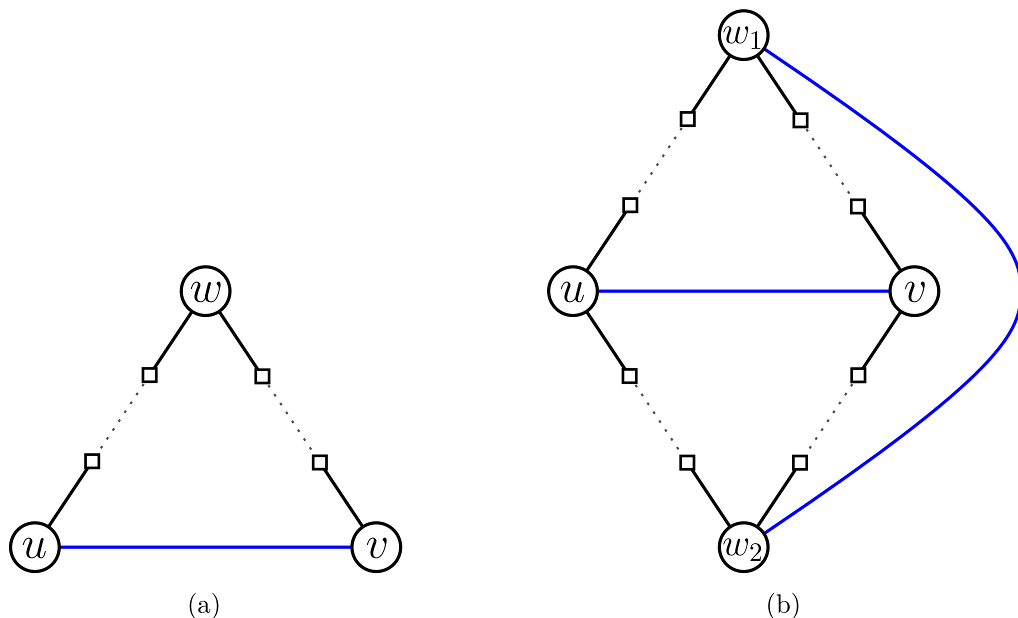


Abbildung 4.1.: (a) Erweiterter K_3 mit Grad-2-Pfaden. Die Quadrate stellen die Grad-2-Knoten dar. Die blauen Linien sind die kreuzbaren Kanten.
(b) Erweiterter K_4 mit den zwei blau markierten, kreuzbaren Kanten.

Zum Finden von erweiterten K_4 Strukturen dient Algorithmus 4.1. Zunächst werden alle erweiterten K_3 gesucht. Diese findet man, indem über alle Kanten iteriert wird und dann

Knoten gesucht werden, welche zu beiden Endknoten der Kante einen Grad-2-Pfad besitzen. Hat man diese gefunden, so sucht man anschließend Paare von erweiterten K_3, welche eine gemeinsame, kreuzbare Kante haben. Sind nun die Knoten adjazent, welche nicht an der gemeinsamen, kreuzbaren Kante beteiligt sind, so hat man einen erweiterten K_4 gefunden. Beachte, dass die Menge K_4 Duplikate automatisch vermeidet.

Algorithmus 4.1 : Pseudocode von FindK4

```
    Input : Graph G = (V, E)
    Output : Menge von K4
    // Finde zuerst erweiterte K3
 1  mapping<edge , K3> ← ∅
 2  forall {u, v} ∈ E do
 3      mapping[{u, v}] ← ∅
 4      forall w ∈ V : w ∉ {u, v} and DEG(w)> 2 do
 5          if EXISTSDEG2PATH(w, v) and EXISTSDEG2PATH(w, u) then
 6              mapping[{u, v}] ← mapping[{u, v}] ∪ {u, v, w}

    // Überprüfe, ob diese erweiterten K3 sogar erweiterte K4 bilden
 7  K4 ← ∅
 8  forall {u, v} ∈ E do
 9      forall {u1, v1, w1} ∈ mapping[{u, v}] : u, v ∈ {u1, v1, w1} do
10          forall {u2, v2, w2} ∈ mapping[{u, v}] : u, v ∈ {u2, v2, w2} and
             {u1, v1, w1} ≠ {u2, v2, w2} do
11              a ∈ {u1, v1, w1} : a ∉ {u, v}
12              b ∈ {u2, v2, w2} : b ∉ {u, v}
13              if ISADJACENT(a, b) then
14                  K4 ← K4 ∪ {u1, v1, w1, b}

15  return K4
```

Nachdem man alle erweiterten K_4 Strukturen gefunden hat, kann wie im normalen Backtracking fortgefahren werden. Lediglich bei der Wahl des nächsten Kreuzungspaares $\sigma(l)$ werden die rot markierten, kreuzbaren Kanten aus Abbildung 4.1b bevorzugt gekreuzt. Wir nennen eine Kreuzung $\sigma(i)$ bezüglich eines Graphen $G(v_{c_0 \dots c_j})$ *erlaubt*, wenn beide Kanten nicht gesättigt und $C_{c_0 \dots c_j}(\sigma(i)) =$ `undefiniert` ist. Somit werden zwei kreuzbare Kanten des selben erweiterten K_4 am Knoten $v_{c_0 \dots c_j}$ immer dann gekreuzt, wenn diese Kreuzung erlaubt ist. Beim Kreuzen zweier kreuzbarer erweiterter K_4 Kanten entsteht ein *erweitertes Drachenviereck*, wobei alle nicht gekreuzten Kanten als *erweiterte Drachenkanten* bezeichnet werden (siehe Abbildung 4.2a). Dadurch erweitert sich Definition 3.1 der gesättigten Kanten um folgende Option: Die Kante e ist eine erweiterte Drachenkante. Die Rechtfertigung hierfür findet sich in Abbildung 4.2b. Die erweiterten Drachenkanten können beliebig nahe an die gekreuzten Kanten gelegt werden und müssen somit nicht mehr gekreuzt werden, weil die rot markierten, gekreuzten Kanten nicht mehr gekreuzt werden dürfen.

4.3. Erweiterte Drachenkanten Optimierung

Das in Abschnitt 4.2 vorgestellte Konzept der erweiterten K_4 Strukturen lässt sich auch auf das Einfügen von beliebigen Kreuzungen ausweiten. Die Kreuzung wird nach wie vor analog zu Kapitel 3.2 eingefügt. Jedoch wird noch zusätzlich zu jeder eingefügten oder bereits existierenden Drachenkante (u, v) ein Grad-2-Pfad $(u, v_1, ..., v_k, v)$ mit $k \geq 1$ gesucht. Alle Kanten dieses Pfades sind erweiterte Drachenkanten. Eine Kreuzung sieht dann wie

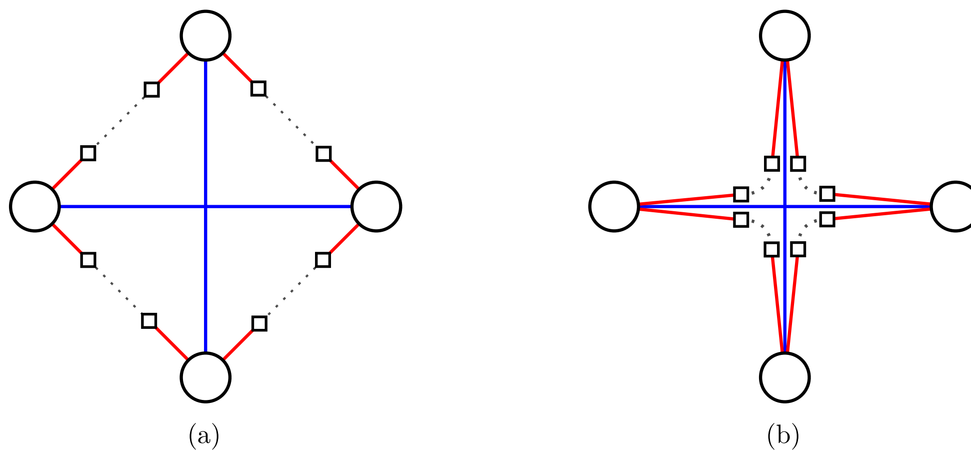


Abbildung 4.2.: (a) Erweitertes Drachenviereck mit rot markierten erweiterten Drachenkanten und der blauen Kreuzung.
(b) Illustration warum erweiterte Drachenkanten nicht gekreuzt werden müssen.

in Abbildung 4.3 aus. Dies schränkt den Suchbaum weiter ein, da mehrere potentielle Kreuzungspaare wegfallen.

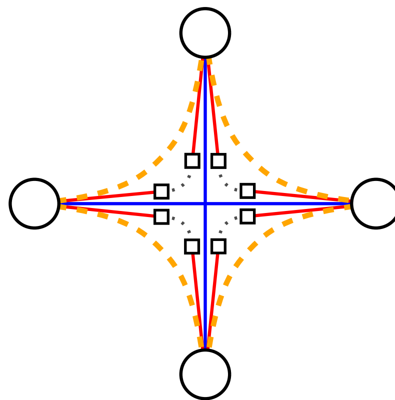


Abbildung 4.3.: In blau die eingefügte Kreuzung mit roten erweiterten Drachenkanten und orange gestrichelten Drachenkanten.

4.4. Kanteneliminierung an Grad-2-Knoten

Überwiegend dünne Graphen enthalten viele Grad-2-Kanten und Pfade. Die Idee hinter dieser Optimierung besteht darin, dass es zu meist nicht zielführend ist, wenn mehrere Kanten eines Grad-2-Pfades gekreuzt werden. Deshalb betrachtet man einen Grad-2-Pfad als einzelne Kante und lässt vorerst nur eine Kreuzung mit dieser Kante zu.

Algorithmisch sucht man sich alle Grad-2-Pfade und wählt aus jedem Pfad eine beliebige Kante aus, welche gekreuzt werden darf. Alle anderen Kanten will man vorerst mit keiner anderen Kante kreuzen. Es wird also zuerst weit nach rechts unten im Suchbaum bis zu einem Knoten $v_{0...0}$ gelaufen, um all diese Kombinationen auszuschließen. Dabei ist zu beachten, dass nach wie vor ein Graph als nicht 1-planar klassifiziert werden kann, da der Algorithmus nach Abarbeitung des aktuellen Teilbaumes beginnt, nach und nach die zuvor blockierten Kanten wieder freizugeben. Somit wird der gesamte Suchbaum abgesucht. Lediglich alle Grad-2-Pfade werden ausgehend vom Knoten $v_{0...0}$ als einzelne Kante wahrgenommen, weil es nur noch einer Kante pro Pfad erlaubt ist, gekreuzt zu werden.

4.5. Kuratowski-Optimierung

Seien H, H' Graphen. Eine *Unterteilung* H' von H ist jeder Graph, der durch das Ersetzen von Kanten aus H durch Grad-2-Pfade entsteht (siehe Abbildung 4.4). Ein Graph H ist ein *topologischer Minor* von G, falls G eine Unterteilung von H als Subgraphen enthält.

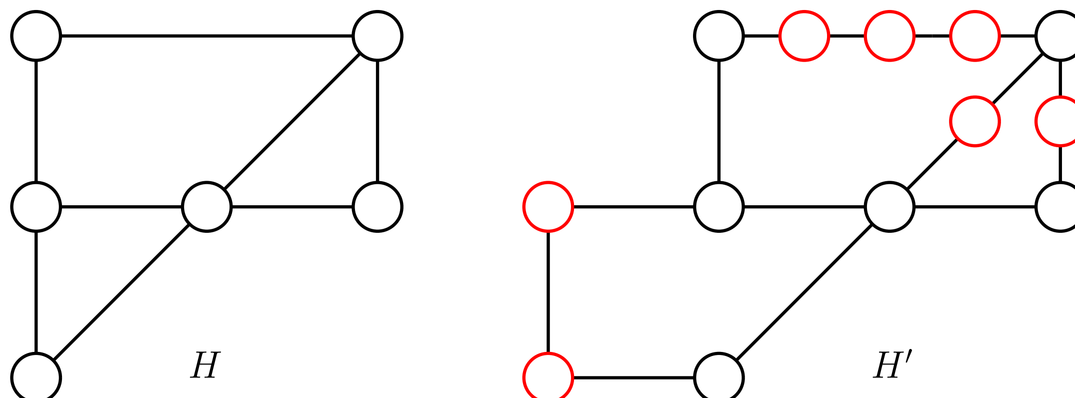


Abbildung 4.4.: Der Graph H' ist eine Unterteilung von H. In rot sind die eingefügten Grad-2-Knoten hervorgehebt.

Satz 4.1 (Satz von Kuratowski [14, Théorème C]). *Ein Graph G ist genau dann planar, wenn er weder $K_{3,3}$ noch K_5 als topologischen Minor enthält.*

Der Satz von Kuratowski 4.1 liefert uns genau die Strukturen, welche dafür sorgen, dass der Graph nicht planar ist. Wir bezeichnen die topologischen $K_{3,3}$ und K_5 Minoren eines Graphen als *Kuratowski-Minoren.* Die Idee hinter diesem Ansatz ist, dass man alle Kuratowski-Minoren des Graphen berechnet und nur in diesen Substrukturen Kreuzungen zulässt. Schafft man es nun, dass alle Kuratowski-Minoren durch das Einfügen von Kreuzungen planar werden und keine weiteren Kuratowski-Minoren dabei entstehen, so liefert uns der Satz von Kuratowski, dass der resultierende Graph planar ist.

Wir schränken die Anzahl an erlaubten Kreuzungen pro Kuratowski-Minor auf eins ein. Dies führt dazu, dass die Anzahl an Kreuzungskombinationen abnimmt, da maximal so viele Kreuzungen zulässig sind, wie anfänglich Kuratowski-Minoren vorhanden sind. Die Extraktion der Kuratowski-Minoren erfolgt mit dem Algorithmus von Boyer und Myrvold [4]. Dies findet bereits vor dem eigentlichen Backtracking statt. Da ein nicht planarer Graph meistens mehrere Kuratowski-Minoren besitzt, sei l die Anzahl an Kuratowski-Minoren. Sei $H_i = (V', E')$ ein Kuratowski-Minor von G mit $V' \subset V$, $E' \subset E$ und $0 \leq i < l$. Algorithmus 4.2 implementiert dabei die Auswahl der nächsten Kante innerhalb eines Kuratowski-Minors H_i. Der Algorithmus überprüft für jedes Kantenpaar $(e_1, e_2) \in E' \times E'$, ob sie eine erlaubte Kreuzung bilden. Bilden sie eine erlaubte Kreuzung, so werden die beiden Kanten als Tupel zurückgegeben. Andernfalls wird $NULL$ zurückgegeben, weil keine Kantenkombination gefunden wurde, welche aktuell zulässig ist. Zu beachten ist, dass auf der Kantenmenge E' eine totale Ordnung definiert ist, so dass keine Kantenpaare doppelt überprüft werden. Ist keine Kreuzung innerhalb von H_i aktuell zulässig, so wird der Algorithmus für den nächsten Kuratowski-Minor H_{i+1} ausgeführt. Hat man alle l Kuratowski-Minoren ohne neue erfolgreiche Kreuzung durchprobiert, so wird ein Cut gemacht und die Backtrackingroutine entfernt die letzte eingefügte Kreuzung.

Mit diesem Vorgehen wird nicht der gesamte Suchbaum abgesucht und es kann somit kein Graph als nicht 1-planar klassifiziert werden. Um dennoch vergleichbar mit den anderen, vorgestellten Algorithmen zu sein, wird nach Abarbeitung die normale Backtrackingroutine gestartet.

Algorithmus 4.2 : Pseudocode von NEXTKURATOWSKICROSSING

```
  Input : Graph H_i = (V', E')
  Output : Kreuzung (e_1, e_2), NULL
1 if isCrossed(H_i) then
2 |  return NULL
3 forall e_1 ∈ E' : !isSaturated(e_1) do
4 |  forall e_2 ∈ E' : e_2 > e_1 and !isSaturated(e_2) do
5 |  |  if C_{c_0...c_j}(e_1, e_2) = undefined then
6 |  |  |  return (e_1, e_2)
7 return NULL
```

5. Kombination der Optimierungstechniken

Nach den zuvor vorgestellten Optimierungstechniken gilt es nun, diese zu vereinen, um ein möglichst gutes Gesamtresultat zu erzielen. Die folgenden Algorithmen beruhen auf den Ergebnissen, welche am Ende von Abschnitt 6.1 erläutert werden. Zunächst wird an jedem Knoten $v_{c0 \dots c_j}$ des Suchbaumes versucht, die beste Kreuzungsmöglichkeit mit den zuvor vorgestellten Möglichkeiten zu finden. Das heißt, es wird von einer statischen Analyse vor dem Backtracking auf eine dynamische Analyse an jedem Knoten des Suchbaumes während des Backtrackings umgestellt. Zum Beispiel bietet sich eine erneute Evaluierung anhand des aktuellen Graphen mit eingefügten Kreuzungen bei der Kuratowski-Optimierung (Abschnitt 4.5) an, da die Anzahl der Kuratowski-Minoren weniger oder auch mehr werden können pro eingefügte Kreuzung. Des Weiteren lässt sich auf Basis der Kuratowski-Minoren folgendes neue Cut-Kriterium formulieren.

Lemma 5.1. *Sei $H = (V', E')$ ein Kuratowski-Minor von $G(v_{c0 \dots c_j})$. Sei S die Menge an gesättigten Kanten von H. Gilt $|E' \setminus S| \leq 1$, so ist $G(v_{c0 \dots c_j c_{j+1} \dots c_k})$ nicht planar mit $k > j$ und beliebigen $c_i \in \{0, 1\}$ für $j < i \leq k$.*

Beweis. Sei $L = E' \setminus S$. Betrachte den Fall $|L| = 0$. Dann sind alle Kanten in E' gesättigt und dürfen nicht mehr gekreuzt werden, weil sie entweder bereits gekreuzt sind oder es nicht mehr erlaubt ist, sie in einen Teilbaum unterhalb von $v_{c0 \dots c_j}$ zu kreuzen [3, Lemma 2]. Somit besitzen alle Graphen $G(v_{c0 \dots c_j c_{j+1} \dots c_k})$ mit $k > j$ den Kuratowski-Minor H als Substruktur. Mit dem Satz von Kuratowski 4.1 folgt, dass $G(v_{c0 \dots c_j c_{j+1} \dots c_k})$ nicht planar ist.

Betrachte nun den Fall $|L| = 1$. Somit existiert eine Kante $e \in L$, welche noch gekreuzt werden darf. Wird e nicht mehr gekreuzt, so folgt analog zum ersten Fall, dass $G(v_{c0 \dots c_j c_{j+1} \dots c_k})$ nicht planar ist. Wird $e = (u, v)$ jedoch gekreuzt, so existiert anschließend ein Knoten w und gesättigte Kanten $e_1 = (u, w)$ und $e_2 = (w, v)$. Dadurch ergibt sich ein Kuratowski-Minor $H' = (V' \cup \{w\}, (E' \setminus \{e\}) \cup \{e_1, e_2\})$. Analog zu oben folgt, dass $G(v_{c0 \dots c_j c_{j+1} \dots c_k})$ nicht planar ist. □

Sei H die Menge an Kuratowski-Minoren des aktuellen Graphen G mit eingefügten Kreuzungen. Sei $l = |H|$ und bezeichne mit H_i das i-te Elemente von H mit $0 \leq i < l$. Wir bezeichnen einen Kuratowski-Minor H_i als *minimal*, wenn H_i die wenigsten noch nicht

gesättigten Kanten unter allen topologischen Minoren aus H besitzt. Die Idee des minimalen Kuratowski-Minors beruht auf Lemma 5.1. Benutzt man für die nächste Kreuzung immer den Kuratowski-Minor mit den wenigsten noch nicht gesättigten Kanten, so werden diese im nächsten Schritt nochmal weniger. Somit gelangt man nach wenigen Schritten an den Punkt, an dem Lemma 5.1 anwendbar ist und ein Cut im Suchbaum eingefügt werden kann.

Algorithmus 5.1 sucht den minimalen Kuratowski-Minor H_i aus H. Sei L_i die Menge der nicht gesättigten Kanten von H_i. Falls $|L_i| < 2$ für mindestens ein i gilt, so kann Lemma 5.1 angewendet und ein CUT eingefügt werden. Ansonsten suche das Minimum über alle $|L_i|$ und gib den zugehörigen topologischen Minor zurück.

Algorithmus 5.1 : Pseudocode von FindMinimalKuratowskiSubgraph

```
    Input : Graph G = (V, E)
    Output : Kuratowski-Minor H_i = (V', E'), CUT
 1  kuratowskis ← findKuratowskis(G)
 2  bestKuratowski ← NULL
 3  minEdges ← INTMAX
 4  forall (V', E') ∈ kuratowskis do
 5  |   edgesToCross ← 0
 6  |   forall e ∈ E' : !isSaturated(e) do
 7  |   |   edgesToCross ← edgesToCross + 1
    |
    |   // Anwendung von Lemma 5.1
 8  |   if edgesToCross < 2 then
 9  |   |   return CUT
 10 |   if edgesToCross < minEdges then
 11 |   |   bestKuratowski ← (V', E')
 12 |   |   minEdges ← edgesToCross
 13 return bestKuratowski
```

5.1. Lokale Kanteneliminierung an Grad-2-Knoten

Nachdem der minimale Kuratowski-Minor gefunden wurde, gilt es die bisher genannten Optimierungsmöglichkeiten innerhalb des Kuratowski-Minors (lokal) zu bündeln. Zunächst wird versucht unter allen erweiterten K_4 diejenigen zu finden, welche auch Skew-Kanten als kreuzbare Kanten beinhalten. Ist ein solcher K_4 vorhanden, so werden dessen kreuzbare Kanten als nächstes gekreuzt. Falls nicht, wird versucht, zwei Skew-Kanten zu finden, welche sich nicht in dem gleichen Grad-2-Pfad befinden. Allgemein werden aus der Menge der Kanten, welche für die nächste Kreuzung zur Verfügung steht, alle bis auf eine Kante eines Grad-2-Pfades entfernt. Existiert in dieser Menge eine Kreuzungsmöglichkeit, so wird diese gewählt. Ansonsten suche ein K_4, welches kreuzbare Kanten innerhalb des minimalen Kuratowski-Minors besitzt und kreuze diese. Befindet sich unter all diesen Kreuzungsmöglichkeiten keine erlaubte nächste Kreuzung, so wird wie in Algorithmus 4.2 verfahren. Der Pseudocode für diese Routine befindet sich im Anhang A. Außerdem werden beim Einfügen einer Kreuzung die erweiterten Drachenkanten ergänzt, wie in Kapitel 4.3 beschrieben.

Hat man nun den minimalen Kuratowski-Minor und die zugehörige nächste Kreuzung ausgewählt, so wird diese zuerst getestet. Dies heißt, dass die Kreuzung in den Graphen $G(v_{c_0 \ldots c_j})$ eingefügt wird und es entsteht dabei G', wobei $v_{c_0 \ldots c_j}$ dem aktuellen Knoten

im Suchbaum entspricht. Sei l (bzw. l') die Anzahl an Kuratowski-Minoren von $G(v_{c_0 \dots c_j})$ (bzw. G'). Gilt anschließend $l' > l$, so wird die Kreuzung nicht gemacht und Knoten $v_{c_0 \dots c_j 0}$ als nächstes besucht, damit diese Kreuzung erst später überprüft wird. Bevorzugt sollen diejenigen Kreuzungen werden, welche nicht mehr Kuratowski-Minoren erzeugen. Rufe also anschließend wieder diesen Algorithmus aus Anhang A auf. Dabei gilt es zu beachten, dass die gleiche Kreuzung nun nicht mehr zulässig ist und eine andere Kreuzung ausprobiert wird.

5.2. Globale Kanteneliminierung an Grad-2-Knoten

Dieser Algorithmus ist sehr ähnlich zur lokalen Kanteneliminierung an Grad-2-Knoten. Lediglich die Reihenfolge der Grad-2-Kanteneliminerung wird angepasst, so dass zuerst die Grad-2-Kanten, wie in Abschnitt 4.4, eliminiert werden. Anschließend wird wie bei der lokalen Kanteneliminierung an Grad-2-Knoten verfahren, wobei innerhalb des minimalen Kuratowski-Minors keine Grad-2-Kanten betrachtet werden müssen. Die Änderung beschränkt sich also auf eine Umstellung von einer lokalen Kanteneliminierung an Grad-2-Knoten (innerhalb eines Kuratowski-Minors) zu einer globalen Kanteneliminierung an Grad-2-Knoten (auf der ganzen Komponente).

5.3. Iterative Erhöhung der zulässigen Kreuzungen

Als Grundlage für diesen Ansatz wird der Algorithmus mit lokaler Kanteneliminierung an Grad-2-Knoten aus Kapitel 5.1 verwendet, weil dieser schneller zu einem Cut führt als die globale Variante (siehe Abbildung 6.5c). Die Idee ist, dass der Algorithmus mehrfach aufgerufen wird und dabei die Anzahl an erlaubten Kreuzungen nach und nach erhöht wird. Fängt man mit einer zulässigen Kreuzung an und erhöht pro Iteration nur um eins, so erhält man nicht nur eine mögliche 1-planare Einbettung, sondern diejenige mit den wenigsten notwendigen Kreuzungen. Ebenfalls wird durch diese Einschränkung der Suchbaum vor allem bei den ersten Iterationen deutlich verkleinert.

Die notwendigen Anpassungen am Algorithmus beschränken sich auf einen zusätzliches Cut. Dieser wird am Ende der VerifyNode Routine 3.1 eingefügt, falls die Zahl der vorgenommenen Kreuzungen beim aktuellen Knoten $v_{c_0 \dots c_j}$ gleich der Anzahl an zugelassenen Kreuzungen ist. Für die Anzahl an möglichen Kreuzungen gibt es zwei Kriterien:

(1) Jede Kante kann maximal einmal gekreuzt werden. Bei ungerader Kantenzahl kann abgerundet werden, da die verbleibende Kante keinen Kreuzungspartner hat. Also ergeben sich maximal $\left\lfloor \frac{|E|}{2} \right\rfloor$ viele Kreuzungen.

(2) Es ist bekannt, dass jeder 1-planare Graph $G = (V, E)$ maximal $|V| - 2$ Kreuzungen hat. Dies ergibt sich aus der maximalen Kantenzahl eines planaren Graphen $G' = (V', E')$ mit $|E'| = 3 \cdot |V'| - 6$ und der maximalen Kantenzahl eines 1-planaren Graphen $G^* = (V^*, E^*)$ mit $|E^*| = 4 \cdot |V^*| - 8$ [15, Theorem 1].

Nimmt man nun das Minimum dieser beiden Grenzen, so erhält man die maximale Anzahl an möglichen Kreuzungen des Graphen. Anschließend wird der um die Kreuzungsbegrenzung modifizierte Algorithmus von Abschnitt 5.1 verwendet, um eine 1-planare Einbettung zu berechnen. Schlägt dies fehl, so wird die Anzahl an erlaubten Kreuzungen um 1 erhöht. Ist nach allen Durchläufen immer noch keine 1-planare Einbettung gefunden, so ist der Graph nicht 1-planar.

Algorithmus 5.2 : Pseudocode von IteratedAlgorithm

```
    Input : Graph G = (V, E)
    Output : 1-planare Einbettung von G mit minimaler Anzahl an Kreuzungen,
             NULL
1   forall i = 1 to min |V| − 2, ⌊|E|/2⌋ do
        // Benutze Algorithmus von Abschnitt 5.1
2       embedding ← RepeatedEvalLocalDeg2Alg(G, i)
3       if embedding.isEmbedded() then
4           return embedding
```

6. Experimentelle Auswertung

Die experimentelle Auswertung gibt einen Überblick über die vorgenommenen Optimierungen und welche zu bevorzugen sind. Ziel der Evaluation ist die Beantwortung folgender Frage:

Was ist die bestmögliche Kombination der einzelnen Optimierungstechniken, um am meisten 2-Zusammenhangskomponenten innerhalb der vorgegebenen Zeit zu lösen.

Lösen bedeutet, dass entschieden wird, ob eine 2-Zusammenhangskomponente (Komponente) 1-planar oder nicht 1-planar ist. Zunächst wird die Umsetzung sowie die Laufzeitumgebung beschrieben. Im Anschluss werden die einzelnen Optimierungen gegenüber gestellt und mit dem Standardalgorithmus aus Kapitel 3 verglichen. Danach werden die verschiedenen Methoden kombiniert und ausgewertet. Dabei liegt der Fokus auf dem Vergleich mit der Umsetzung von Binucci, Didimo und Montecchiani.

Die Umsetzung fand in der Programmiersprache C++ statt. Als Graphframework wurde das Open Graph Drawing Framework (OGDF)[1] [8] mit der Version Catalpa (Februar 2020) verwendet. OGDF bietet viele Algorithmen, wie zum Beispiel die Extraktion der Kuratowski-Minoren oder den Planaritätstest, welche bei der Umsetzung der verschiedenen Algorithmen zum Einsatz kamen. Kompiliert wurde sowohl OGDF als auch die Umsetzung dieser Arbeit mit GCC Version 8.3.0 im Release Mode mit Optimierungsflags O3 und DNDEBUG.

Als Laufzeitumgebung dient ein Cluster mit 17 Knoten mit je einem 10-Kern Intel Xeon E5-2690 v2 Prozessor mit 3,0 GHz und 64 GB RAM. Der L2-Cache beträgt 256 KB. Das Betriebssystem der Knoten ist Debian GNU/Linux 10 (buster). Jede Komponente erhält genau einen Kern, wodurch bis zu 170 Komponenten gleichzeitig getestet werden können.

In den folgenden Auswertungen wird pro Komponente eine Berechnungszeit von drei Stunden zugrunde gelegt. Dies deckt sich mit der gewählten Zeit von Binucci, Didimo und Montecchiani [3, Kapitel 4]. Die Zerlegung der Graphen in Komponenten erfolgt bereits vor der Ausführung der einzelnen Backtrackingalgorithmen. Dabei werden alle trivial 1-planaren und trivial nicht 1-planaren aussortiert, so dass nur noch Instanzen übrig bleiben, welche die Backtrackingroutine benötigen. Binucci, Didimo und Montecchiani führten Ihren Algorithmus auch auf trivial nicht 1-planaren Graphen aus. Außerdem beruht Ihre Auswertung nicht auf den Komponenten, sondern auf Graphen, welche aus vielen Komponenten bestehen. Somit lassen sich Ihre Ergebnisse nicht mit den Ergebnissen dieser

[1] http://www.ogdf.net

Arbeit vergleichen. Um dennoch einen Vergleich ziehen zu können, wurde der Algorithmus von Binucci, Didimo und Montecchiani im Rahmen dieser Arbeit in C++ implementiert. Der Algorithmus entspricht dem Skew-Kanten Algorithmus aus Kapitel 4.1. Dadurch kann der Algorithmus auf den Komponenten und nicht auf den Graphen, welche aus mehreren Komponenten bestehen können, ausgeführt werden.

Die in Abschnitt 4 und 5 vorgestellten Algorithmen wurden auf Basis der NORTH Graphen entwickelt und getestet. Um auszuschließen, dass diese Algorithmen speziell für diese Komponenten konzipiert wurden, werden alle Algorithmen auf den Komponenten der NORTH und ROME Graphen [2] ausgeführt. Da die ROME Graphen mit 8253 deutlich mehr als die NORTH Graphen mit 400 Komponenten umfassen, werden alle Algorithmen nur auf einen Teil der ROME Graphen angewendet. Diesen Teil nennen wir *RESTRICTED-ROME*. Die RESTRICTED-ROME Graphen entsprechen den Graphen, welche Binucci, Didimo und Montecchiani [1] verwendeten. Dabei handelt es sich nur um 1-planare Graphen, weil aus der Auflistung nicht bekannt wird, welche Instanzen Ihr Algorithmus nicht gelöst hat. Als nicht 1-planar klassifizierte der Algorithmus keine Instanzen. Somit umfassen die RESTRICTED-ROME Graphen 412 Komponenten, welche alle 1-planar sind. Dies stellt kein Problem dar, denn bei der Auswertung der einzelnen Optimierungstechniken (siehe Abschnitt 6.1) wird lediglich ein grober Vergleich der einzelnen Ansätze untereinander benötigt. Die kombinierten Optimierungen sowie der Algorithmus von Binucci, Didimo und Montecchiani werden dann aber auf allen ROME Komponenten ausgeführt, wobei die Berechnungszeit auf eine Stunde reduziert wird. Die Reduzierung der Berechnungszeit kann deshalb vorgenommen werden, weil ein Großteil der Komponenten innerhalb der ersten Minuten gelöst wird und somit nur ein geringer Anteil auf das Zeitintervall ab einer Stunde fällt. Außerdem ist die verwendete Hardware deutlich schneller als die Hardware von Binucci, Didimo und Montecchiani.

6.1. Auswertung einzelner Optimierungstechniken

Die detaillierten Ergebnisse jeder einzelnen Optimierungsmethode befindet sich im Anhang in Tabelle B.1 und B.2. Diese zeigen, dass der Algorithmus ohne Optimierung Komponenten bis zu 20 Knoten bereits zu 76,3% (NORTH Komponenten) und sogar 98,3% (RESTRICTED-ROME Komponenten) gelöst werden können. Für Komponenten mit mehr als 20 Knoten liegt die Lösbarkeit hingegeben bei nur noch 11,7% (NORTH Komponenten) und 39,8% (RESTRICTED-ROME Komponenten). Dies zeigt, dass der Standardalgorithmus nur für kleine Instanzen bis 20 Knoten sinnvoll eingesetzt werden kann und eine Optimierung für größere Komponenten notwendig ist.

Anzumerken ist auch, dass trotz der Tatsache, dass alle RESTRICTED-ROME Komponenten 1-planar sind, nicht alle von den verschiedenen Implementierungen gelöst werden. Selbst der Skew-Kanten Ansatz, welcher dem Algorithmus von Binucci, Didimo und Montecchiani entspricht, löst nur 95,9% der RESTRICTED-ROME Komponenten. Der Grund hierfür liegt bei der zufälligen Wahl der Permutation σ. Hat man eine Permutation gewählt, welche genau die richtigen Kreuzungen bereits am Anfang einfügt, so können selbst große Instanzen als 1-planar klassifiziert werden. Hat man dieses Glück jedoch nicht und wählt zuerst Kreuzungen, welche nicht zu einer 1-planaren Einbettung führen können, so benötigt der Algorithmus länger und das Limit der Berechnungszeit kann erreicht werden.

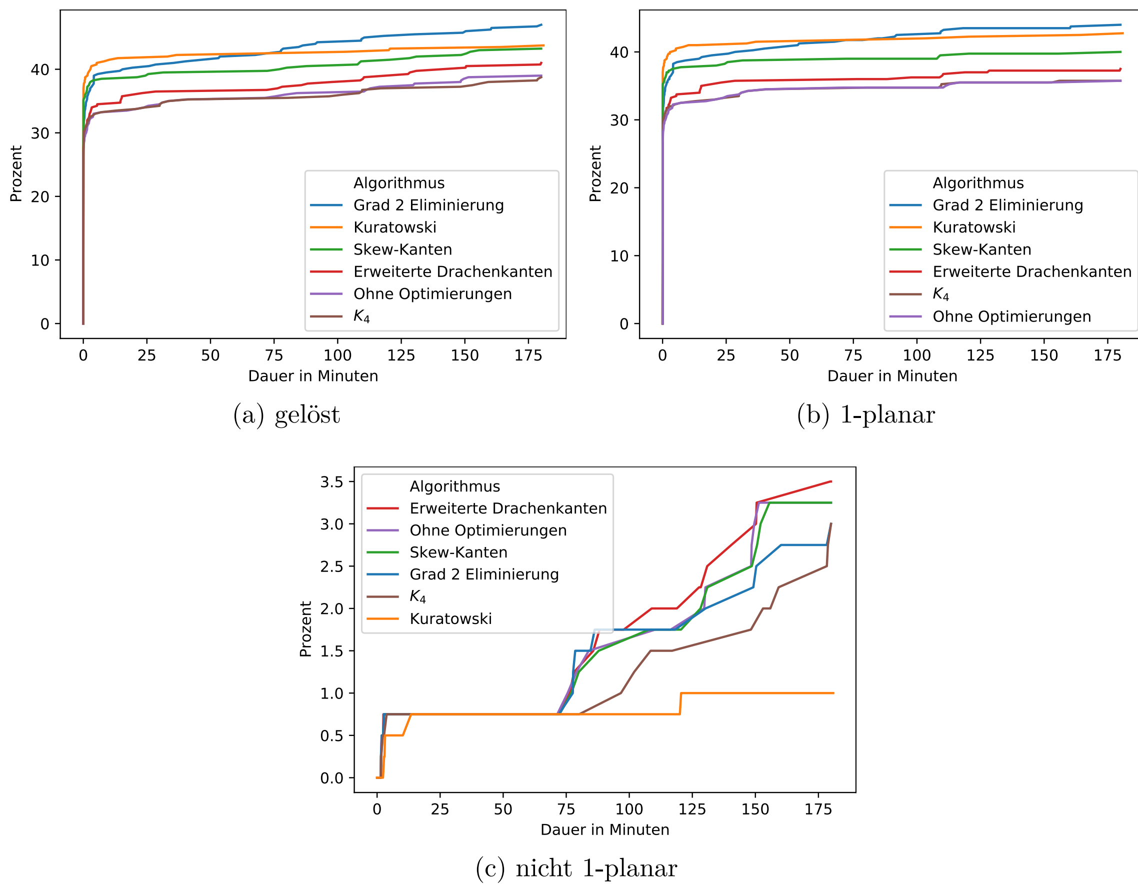
In Abbildung 6.1 ist der prozentuale Anteil an gelösten, 1-planaren und nicht 1-planaren Instanzen bezüglich der Berechnungszeit dargestellt. Betrachtet werden zunächst nur die NORTH Komponenten. Dabei fällt auf, dass der Anteil an gelösten nicht 1-planaren Komponenten maximal 3,5% beträgt und somit nur einen minimalen Einfluss auf die gelösten Instanzen in Abbildung 6.1a hat. Der dominierende Anteil ist für alle Algorithmen die Menge an gelösten 1-planaren Komponenten. Dies liegt daran, dass für die Klassifizierung

einer Komponente als nicht 1-planar der gesamte Suchbaum abgesucht werden muss. Da der Suchbaum exponentiell groß im Bezug zur Knotenanzahl ist, wird der Suchbaum nur bis zu einer Größe von 20 Knoten vollständig abgesucht. Vergleicht man die einzelnen Algorithmen miteinander, so liefert die Kanteneliminierung an Grad-2-Knoten das beste Ergebnis mit 47,0% an gelösten Instanzen gefolgt vom Kuratowski Algorithmus mit 43,8%. Im Vergleich dazu stellt man beim K_4 Algorithmus keine Verbesserung gegenüber dem Standardalgorithmus fest, welcher 39,0% gelöste Instanzen umfasst. Hinsichtlich des Kurvenverlaufs ist erkennbar, dass ein Großteil der 1-planaren Instanzen innerhalb der ersten Minute gelöst werden. Dabei ist der Kuratowski Ansatz am besten. Erst nach einer Stunde schließt der Ansatz der Kanteneliminierung an Grad-2-Knoten auf und überholt die stagnierende Kuratowski Methode. Auch die Steigung nach den ersten Minuten ist bei der Grad 2 Eliminierung am größten, was zeigt, dass durch die Betrachtung der Grad-2-Pfade als eine einzelne Kante die möglichen Kombinationen deutlich abnehmen. Das ist auch der Grund, weshalb kontinuierlich 1-planare Einbettungen durch diese Methode gefunden werden.

Betrachtet man die nicht 1-planaren Instanzen (Abbildung 6.1c), so wird die Berechnungszeit benötigt, um bis zu 14 nicht 1-planaren Komponenten zu lösen. Die erweiterten Drachenkanten bewirken hierbei, dass weniger Kreuzungsmöglichkeiten zur Verfügung stehen, was schneller zu einem Cut im Suchbaum führt. Die Kuratowski-Optimierung kann aufgrund des Aufbaus des Algorithmus, wie in Abschnitt 4.5 erläutert, nur vier nicht 1-planare Komponenten lösen. Dieses Problem hat der Skew-Kanten Algorithmus nicht, obwohl dieser auch zuerst einen optimierten Algorithmus ausführt, welcher nicht den gesamten Suchbaum absucht und anschließend das Backtracking ohne Optimierung startet. Denn das Durchprobieren von nur maximal $|E| - 1$ Kreuzungen benötigt nur einen Bruchteil einer Sekunde. Deshalb sind die Kurven von der Skew-Kanten Optimierung und dem Standardalgorithmus beinahe identisch.

Die RESTRICTED-ROME Komponenten umfassen nur 1-planare Instanzen. Somit ist in Abbildung 6.2 nur der Graph für 1-planare Komponenten abgebildet. Abweichend zu der Auswertung der NORTH Komponenten bemerkt man, dass die Skew-Kanten Optimierung am meisten Instanzen löst. Der Grund hierfür ist, dass Binucci, Didimo und Montecchiani [3, Further Optimizations] ebenfalls die Skew-Kanten Optimierung verwendeten und die RESTRICTED-ROME Komponenten genau die Komponenten sind, welche sie mit diesem Ansatz gelöst haben. Somit sind es überwiegend Komponenten, welche durch das Entfernen einer Kante planar werden und sehr gut vom Skew-Kanten Algorithmus gelöst werden können. Dies relativiert das Resultat der Skew-Kanten Optimierung, schmälert aber nicht das Ergebnis der anderen Methoden. Zusätzlich weisen die RESTRICTED-ROME Komponenten nur eine durchschnittliche Dichte von 1,41 auf, wohingegen die NORTH Komponenten eine durchschnittliche Dichte von 1,95 haben. Hinsichtlich der durchschnittlichen Anzahl an Grad-2-Pfade enthalten beide Graph-Datenbanken ungefähr gleich viele, nämlich 7,1 (NORTH) und 7,2 (RESTRICTED-ROME). Dies führt dazu, dass die Kanteneliminierung an Grad-2-Knoten auf beiden Graph-Datenbanken die meisten Instanzen, abgesehen vom Skew-Kanten Ansatz, lösen kann. Außerdem erkennt man, dass der Kuratowski-Ansatz die richtigen Substrukturen innerhalb einer Komponente identifiziert, in welche Kreuzungen eingefügt werden müssen. Im Durchschnitt enthalten die NORTH Komponenten 110 Kuratowski-Minoren und die RESTRICTED-ROME Komponenten 3,4. Da ungefähr doppelt so viele RESTRICTED-ROME Komponenten wie NORTH Komponenten gelöst wurden, bietet es sich an, dass man eine große Anzahl an Kuratowski-Minoren als Richtwert für tendenziell schwerer lösbare Komponenten verwendet und versucht die Anzahl pro eingefügte Kreuzung zu minimieren.

Betrachtet man die Dichteverteilung der 1-planaren und nicht 1-planaren Komponenten der NORTH Komponenten aus Abbildung 6.3, so fällt auf, dass alle gefundenen 1-planaren



(a) gelöst

(b) 1-planar

(c) nicht 1-planar

Abbildung 6.1.: Darstellung des prozentualen Anteils der gelösten/1-planaren/nicht 1-planaren Komponenten der NORTH Komponenten bezüglich der Berechnungszeit.

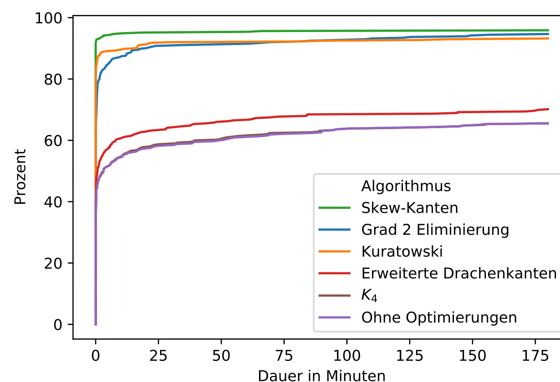


Abbildung 6.2.: Darstellung des prozentualen Anteils der 1-planaren Komponenten der RESTRICTED-ROME Komponenten bezüglich der Berechnungszeit.

Komponenten eine Dichte von 2,5 oder weniger aufweisen. Die nicht 1-planaren Komponenten haben eine Dichte, die genau 2 oder größer ist. Im Dichtebereich von 1 bis 1,5 erkennt man den Einfluss der Optimierungen, wobei die Grad 2 Eliminierung 61 der 80 Instanzen löst. Der Standardalgorithmus löst hingegen nur 37 Instanzen. Somit löst die Grad 2 Eliminierung 30% mehr Instanzen als der Standardalgorithmus in diesem Dichtebereich. Für die restlichen Bereiche gibt es bis auf die bereits erklärte Schwäche des Kuratowski-Ansatzes bei nicht 1-planaren Komponenten keine großen Unterschiede.

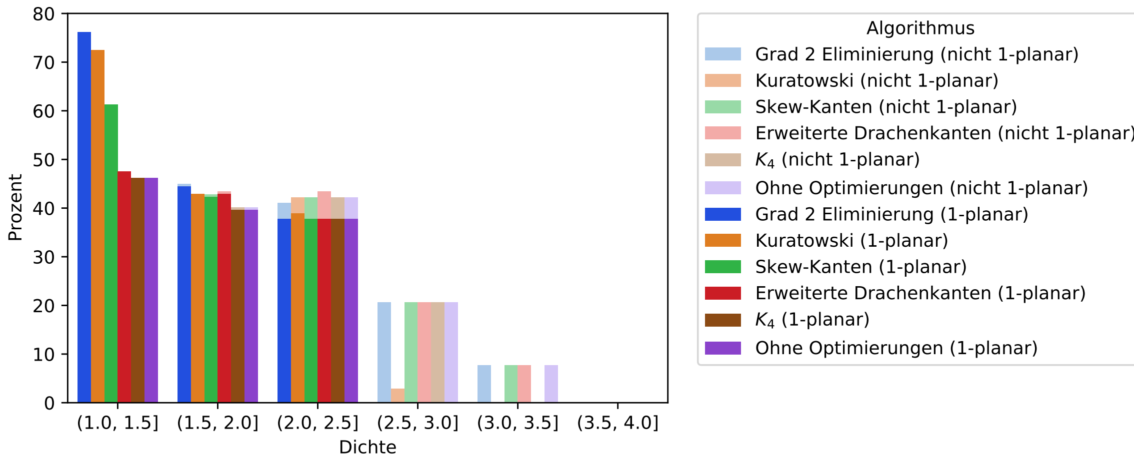


Abbildung 6.3.: Darstellung der 1-planaren und nicht 1-planaren, prozentualen Anteile aufgeteilt nach Algorithmen und Dichte der NORTH Komponenten.

Abbildung 6.4 zeigt die prozentualen Anteile der 1-planaren RESTRICTED-ROME Komponenten pro Dichtebereich. Der Skew-Kanten Algorithmus sticht, wie bereits zuvor erläutert, hervor, welcher fast alle Instanzen innerhalb der ersten Minute löst. Dies spiegelt sich auch in der durchschnittlichen Anzahl an Skew-Kanten pro Komponente wider, welche für die RESTRICTED-ROME Komponenten bei 9,7 und für die NORTH Komponenten bei 2,4 liegt. Vergleicht man die Prozentwerte der K_4 Methode mit dem Standardalgorithmus, so sind diese fast identisch. Dies liegt daran, dass nur im Durchschnitt 0,1 erweiterte K_4 in den Komponenten enthalten sind. Bei den NORTH Komponenten sind es hingegen 6,0. Nichtsdestotrotz ist aus allen Vergleichen erkennbar, dass der K_4 Ansatz nur bedingt sinnvoll einsetzbar ist. Der Kuratowski Algorithmus und die Kanteneliminierung an Grad 2 Knoten liefern analog zu den NORTH Komponenten die besten Ergebnisse neben der Skew-Kanten Methode ab. Im Dichtebereich 2 bis 2,5 befinden sich nur zwei Komponenten, welche von allen Algorithmen gelöst werden konnten.

In Tabelle 6.1 sind die durchschnittlichen Berechnungszeiten nach gelösten, 1-planaren und nicht 1-planaren Komponenten dargestellt. Die Zeit für die nicht 1-planaren Komponenten scheint auf den ersten Blick sehr gut beim Kuratowski Algorithmus. Jedoch hat dieser nur vier Instanzen gelöst und ist dementsprechend nicht besser als alle anderen Methoden, welche zwischen 12 und 14 nicht 1-planare Komponenten klassifiziert haben und dafür deutlich mehr Zeit benötigten. Bei den 1-planaren Komponenten stechen der Skew-Kanten und Kuratowski Algorithmus hervor, welche im Durchschnitt schneller sind als alle anderen.

Zusammenfassend lässt sich bezüglich der am Anfang von Kapitel 6 gestellten Frage sagen, dass der K_4 Ansatz am schlechtesten hinsichtlich der erzielten Lösbarkeit verglichen zu den anderen Methoden abschneidet und keine erkennbare Verbesserung gegenüber dem Standardalgorithmus, welcher 426 Komponenten von den 812 Komponenten beider Graph-Datenbanken löst, mit sich bringt. Es folgt die Optimierung durch erweiterte Drachenkanten, welche 453 Komponenten löst. Das ist eine Verbesserung um 3,3%. Die Skew-Kanten Optimierung löst 568 Instanzen und liefert nochmals eine deutlich erkennbare

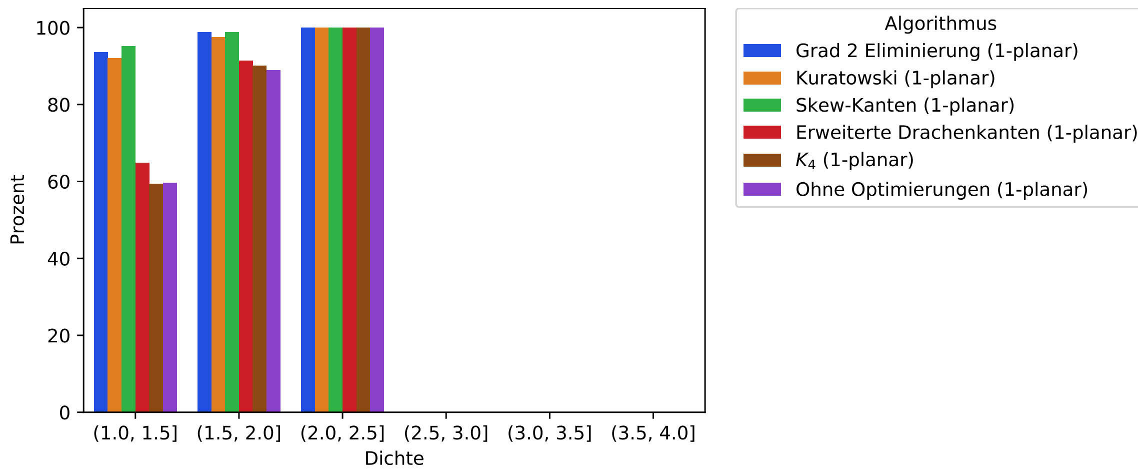


Abbildung 6.4.: Darstellung der 1-planaren, prozentualen Anteile aufgeteilt nach Algorithmen und Dichte der RESTRICTED-ROME Komponenten.

Tabelle 6.1.: Durchschnittliche Dauer der einzelnen Algorithmen.

Algorithmus	NORTH Ø Dauer (min) pro Komp.			RESTRICTED-ROME Ø Dauer (min) pro Komp.		
	gelöst	1-planar	¬1-planar	gelöst	1-planar	¬1-planar
Ohne Optimierungen	12.06	5.20	87.55	10.13	10.13	-
Skew-Kanten	10.17	3.78	88.74	0.46	0.46	-
K_4	12.60	5.11	101.85	9.63	9.63	-
Erweiterte Drachenk.	13.07	6.06	88.21	8.29	8.29	-
Grad 2 Eliminierung	13.35	8.33	86.97	4.72	4.72	-
Kuratowski	4.06	3.36	33.89	1.75	1.75	-

Steigerung um 17,5% gegenüber dem Standardalgorithmus, was aber durch die Wahl der RESTRICTED-ROME Komponenten relativiert werden muss. Bei den verbleibenden zwei besten Algorithmen liegt der Ansatz der Kanteneliminierung an Grad-2-Knoten leicht vor der Kuratowski Methode mit einer Verbesserung um 18,7% und 16,4% gegenüber dem Algorithmus ohne Optimierung. Dies gibt eine grobe Ordnung der Optimierungstechniken vor.

6.2. Auswertung kombinierter Optimierungstechniken

Die Kombination der einzelnen Optimierungen liefert drei Algorithmen, wobei zwei sich lediglich in der Reihenfolge der verschiedenen Optimierungsmöglichkeiten unterscheiden. Die lokale Kanteneliminierung an Grad-2-Knoten wendet die Kanteneliminierung innerhalb eines Kuratowski-Minors an, wohingegen die globale Kanteneliminierung an Grad-2-Knoten dies vor der Bestimmung der Minoren vornimmt. Aufbauend auf dem Ansatz der lokalen Kanteneliminierung an Grad-2-Knoten schränkt der dritte Algorithmus die Anzahl an zulässigen Kreuzungen ein und erhöht dieses Limit nach und nach. Die detaillierten Ergebnisse für jeden Algorithmus aufgeteilt nach Knotenanzahl finden sich im Anhang in Tabelle B.3 und B.4.

Zunächst wird ein Vergleich zwischen den kombinierten Ansätzen und dem Skew-Kanten Algorithmus, welcher der Umsetzung von Binucci, Didimo und Montecchiani entspricht, gezogen (siehe Abbildung 6.5). Anschließend wird die Auswirkung von Lemma 5.1 evaluiert. Bei den 400 NORTH Komponenten liefert die globale Grad 2 Eliminierung das beste Ergebnis mit 237 (59,3%) gelösten Instanzen. Dies beruht vor allem auf dem Ergebnis der 1-planaren Komponenten. Betrachtet man die nicht 1-planaren Komponenten, so liefert die lokale Grad 2 Eliminierung mit 30 (7,3%) nicht 1-planaren Instanzen den höchsten Wert. Dies liegt daran, dass bei der globalen Kanteneliminierung zunächst sehr viele Kantenkombinationen verboten werden und diese gemäß dem Backtracking nach und nach wieder freigegeben werden müssen. Dadurch kommt es zu einer ineffizienteren Abarbeitung des gesamten Suchbaumes verglichen mit der lokalen Grad 2 Eliminierung. Da alle kombinierten Algorithmen das Cut-Kriterium aus Lemma 5.1 verwenden und mehr nicht 1-planare Komponenten wie der Skew-Kanten Algorithmus finden, bestätigt sich dessen praktischer Nutzen. Zieht man den Vergleich zum Algorithmus von Binucci, Didimo und Montecchiani, welcher 173 (43,3%) Instanzen löst, werden durch die globale Grad 2 Eliminierung 64 Komponenten mehr gelöst. Dies entspricht einer prozentualen Zunahmen um 37,0% gegenüber Ihrem Algorithmus.

Von den 8253 ROME Komponenten in Abbildung 6.6 konnten nur 1-planare Komponenten gelöst werden. Dies liegt zum einen an der verringerten Berechnungszeit auf eine Stunde und zum anderen an der geringen Dichte der ROME Komponenten, welche durchschnittlich 1,45 beträgt. Da dünne Komponenten in der Regel weniger Kreuzungen benötigen, liefert die iterative Erhöhung der zugelassenen Kreuzungen das beste Ergebnis mit 31,1% an gelösten Instanzen. Dabei konnten 2564 Komponenten als 1-planar klassifiziert werden. Betrachtet man die durchschnittliche Kreuzungsanzahl dieses Algorithmus, so werden im Durchschnitt 1,86 Kreuzungen benötigt. Zum Vergleich liegt die Anzahl an eingefügten Kreuzungen bei der lokalen Kanteneliminierung an Grad-2-Knoten bei durchschnittlich 3,13, bei der globalen Eliminierung bei 3,73 und bei der Skew-Kanten Methode bei 1,84. Der Skew-Kanten Algorithmus weist dabei den geringsten Wert auf, weil genau eine Kreuzung aufgrund der Skew-Kante eingefügt wird und ansonsten das Standardbacktracking zumeist keine Lösung liefert. Dies ist der Grund für den beinahe rechtwinkligen Knick beim Skew-Kanten Algorithmus. An der Kurve der lokalen Kanteneliminierung an Grad-2-Knoten lässt sich erkennen, dass sich diese Methode, verglichen mit den anderen kombinierten Optimierungen, nicht für dünne Komponenten eignet. Im Gegensatz dazu stellt man

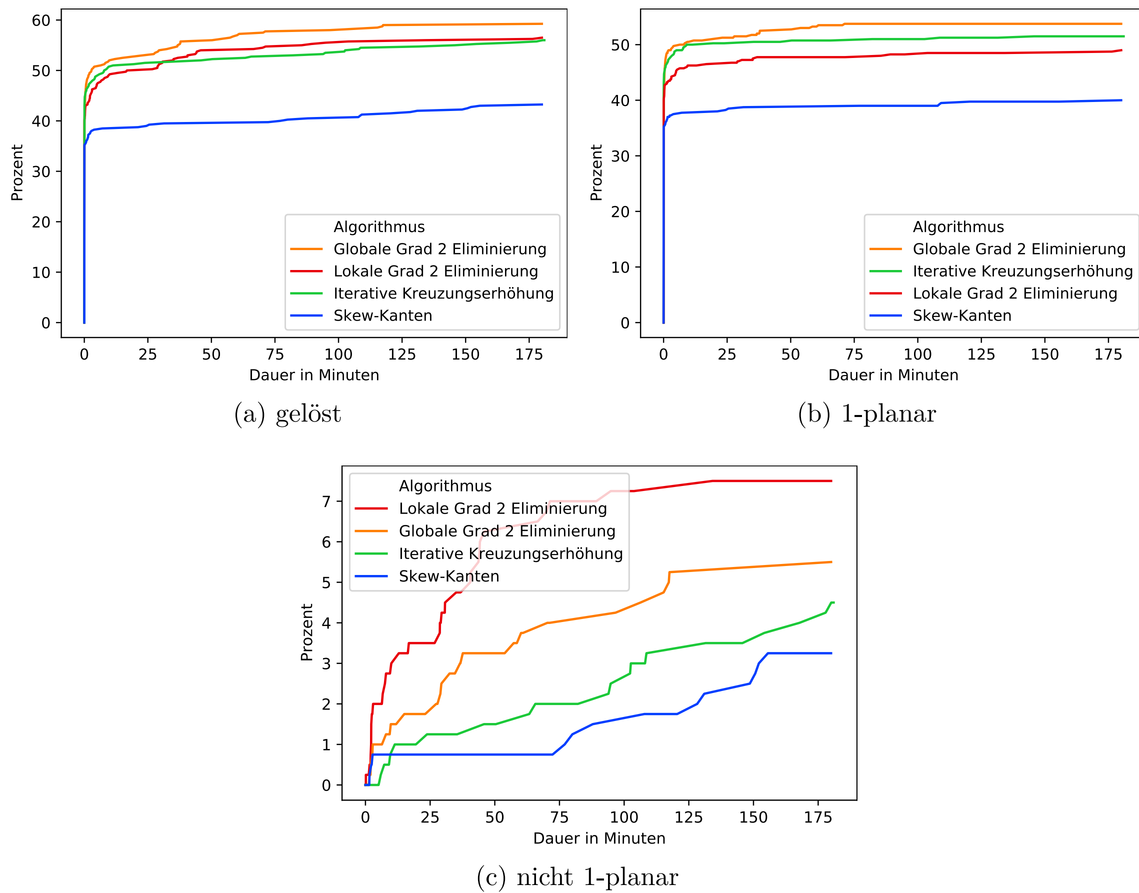


Abbildung 6.5.: Darstellung des prozentualen Anteils der gelösten/1-planaren/nicht 1-planaren NORTH Komponenten bezüglich der Berechnungszeit.

bei der globalen Kanteneliminierung an Grad-2-Knoten fest, dass dieser Ansatz gut für dünne Komponenten funktioniert. Alles in allem löst die iterative Kreuzungserhöhung 1411 Instanzen mehr als die Umsetzung von Binucci, Didimo und Montecchiani, welche 1153 Komponenten löst. Das entspricht einer prozentualen Zunahme um 122,4%.

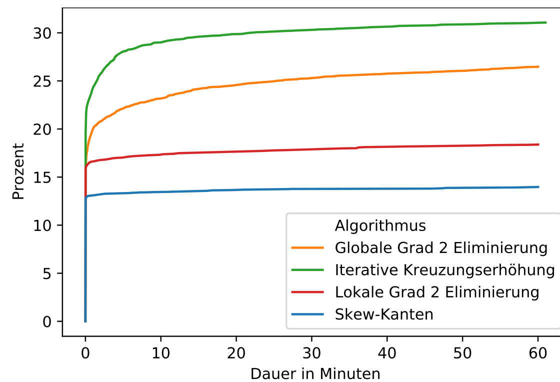


Abbildung 6.6.: Darstellung des prozentualen Anteils der 1-planaren ROME Komponenten bezüglich der Berechnungszeit.

Einen Überblick über die Dichteverteilung der 1-planaren und nicht 1-planaren NORTH Komponenten, welche die verschiedenen Algorithmen lösen können, bietet Abbildung 6.7. Für Komponenten mit einer Dichte von maximal 2 liefert die globale Kanteneliminierung an Grad-2-Knoten die meisten gelösten Instanzen dicht gefolgt von der iterativen Erhöhung der zulässigen Kreuzungen. Letztere bietet den Vorteil, dass man zusätzlich eine 1-planare Einbettung erhält, welche eine optimale Anzahl an Kreuzungen hat. Deshalb ist für manche Anwendungsfälle diese Methode dem Algorithmus der globalen Kanteneliminierung vorzuziehen. Für dichtere Komponenten bringt die lokale Kanteneliminierung an Grad-2-Knoten überzeugende Ergebnisse vor, da diese am meisten nicht 1-planare Komponenten findet, welche in diesen Bereichen vermehrt vorkommen.

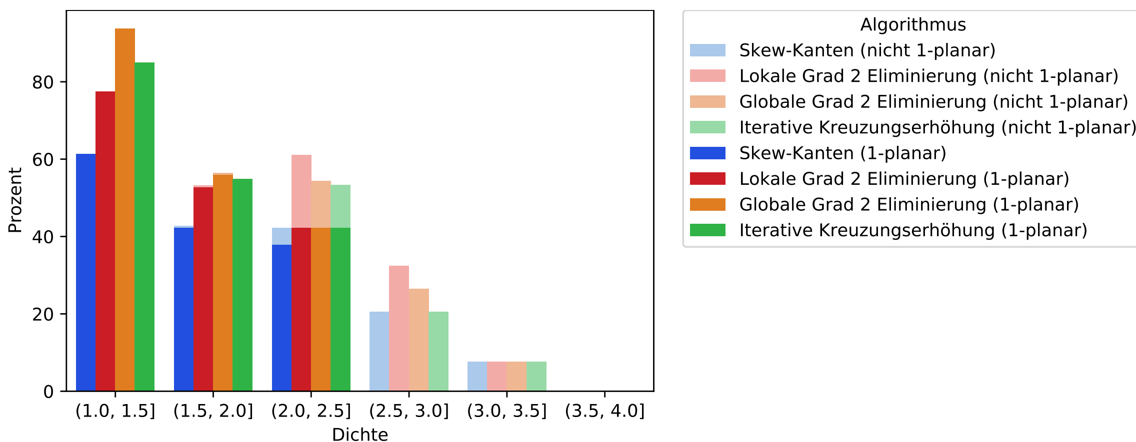


Abbildung 6.7.: Darstellung der 1-planaren und nicht 1-planaren, prozentualen Anteile aufgeteilt nach Algorithmen und Dichte der NORTH Komponenten.

Abbildung 6.8 zeigt den prozentualen Anteil an gelösten, 1-planaren ROME Komponenten pro Dichtebereich. Die 50% an gelösten Instanzen im Bereich von 2 bis 2,5 fällt dabei zuerst ins Auge. Da jedoch nur zwei Komponenten in diesen Bereich fallen und eine davon gelöst wurde, ist dieser Bereich zu vernachlässigen. Bei den dünneren Bereichen sind es 6025 (1 bis 1,5) bzw. 2226 (1,5 bis 2) getestete Komponenten. Abweichend zu den NORTH Komponenten löst die iterative Kreuzungserhöhung die meisten Instanzen bis zu einer Dichte von 2. Der Grund hierfür ist die Größe der Instanzen. Im Dichtebereich von 1 bis

2 haben die ROME Komponenten eine durchschnittliche Knotenanzahl von 47,8, wobei die NORTH Komponenten 32,0 Knoten aufweisen. Somit sind die ROME Komponenten deutlich größer als die NORTH Komponenten, weshalb auch weniger Instanzen gelöst werden konnten. Die iterative Kreuzungserhöhung kommt also mit großen Instanzen mit geringer Dichte am besten zurecht. Folglich spielt neben der Dichte auch die Größe der Komponenten eine entscheidende Rolle, wie auch aus Abbildung 6.9 deutlich wird.

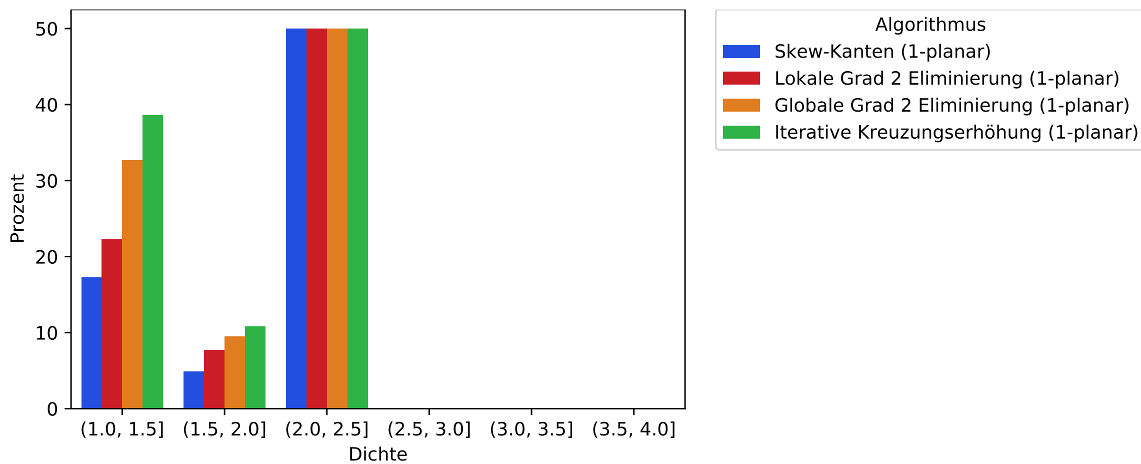


Abbildung 6.8.: Darstellung des 1-planaren, prozentualen Anteils aufgeteilt nach Algorithmen und Dichte aller ROME Komponenten.

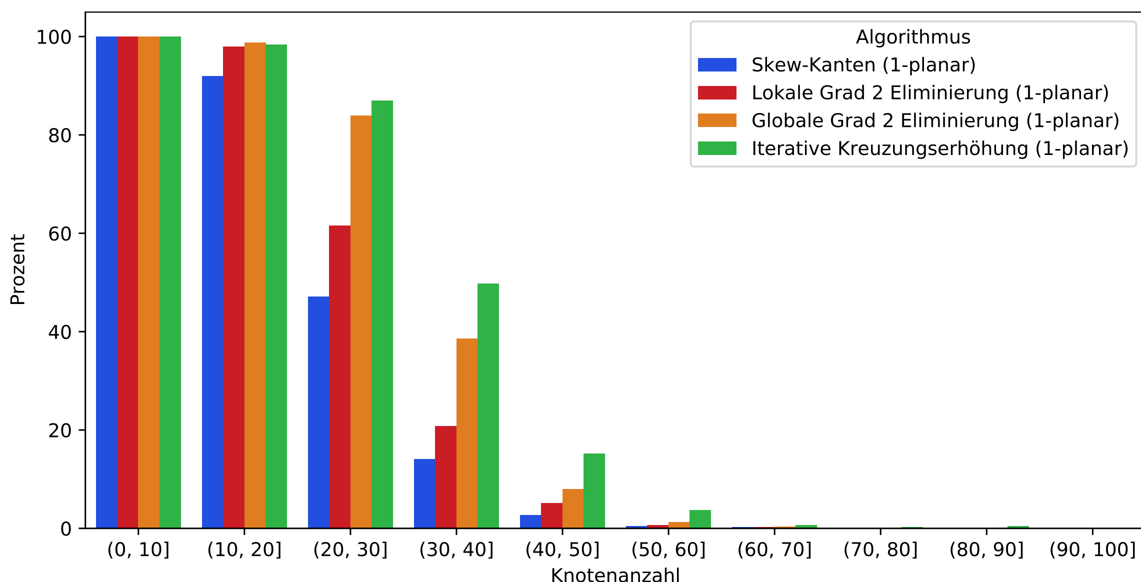
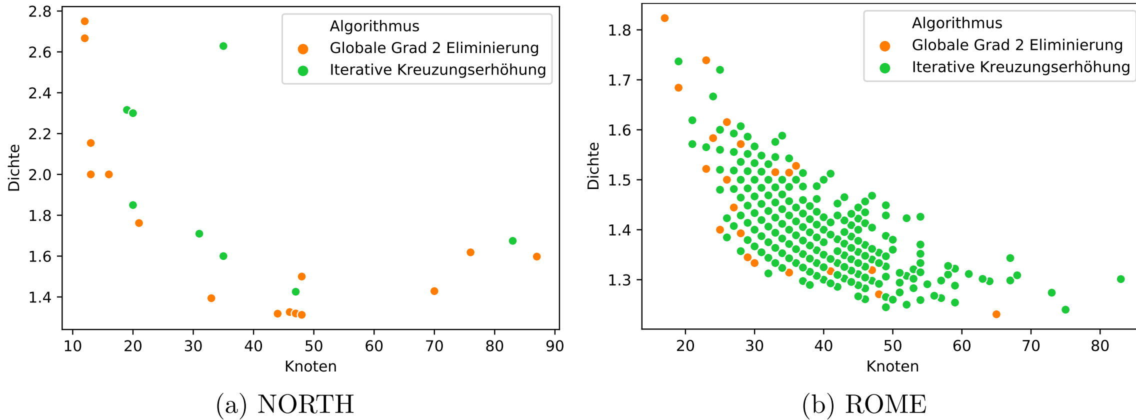


Abbildung 6.9.: Darstellung des 1-planaren, prozentualen Anteils aufgeteilt nach Algorithmen und Knotenanzahl aller ROME Komponenten.

Um den Unterschied zwischen der globalen Kanteneliminierung an Grad-2-Knoten und der iterativen Erhöhung der zulässigen Kreuzungen zu verdeutlichen, bietet Abbildung 6.10 einen Überblick. Es wird dargestellt, welche Komponenten zusätzlich gelöst werden konnten. Das heißt, Komponenten, welche beide Algorithmen gelöst haben, sind nicht in der Abbildung enthalten. Auf den NORTH Komponenten sind es nur wenige unterschiedliche Instanzen, weshalb der Unterschied der beiden Methoden auf dieser Graph-Datenbank gering ist. Auf den ROME Komponenten hingegen erkennt man, dass die Anzahl an zusätzlich gelösten Instanzen mit der iterativen Kreuzungserhöhung deutlich höher ist. Insbesondere Komponenten mit mindestens 30 Knoten und einer Dichte, welche ausgehend

von 1,6 abnimmt je mehr Knoten es werden, löst dieser Algorithmus mehr. Somit ist für Komponente mit einer Dichte bis 1,6 die iterative Kreuzungserhöhung allen anderen Algorithmen vorzuziehen.



(a) NORTH (b) ROME

Abbildung 6.10.: Darstellung der zusätzlich gelösten Komponenten zwischen den beiden Algorithmen aufgeteilt nach Knoten und Dichte. Abbildung (a) zeigt die NORTH Komponenten und (b) die ROME Komponenten.

Betrachtet man die durchschnittliche Berechnungsdauer der einzelnen Algorithmen (siehe Tabelle 6.2) so fällt auf, dass die globale Grad 2 Eliminierung auf den NORTH Komponenten am schnellsten und auf den ROME Komponenten am langsamsten ist. Dies liegt an den größeren ROME Komponenten und dem nicht 1-planaren Anteil bei den NORTH Komponenten. Angemerkt sei nochmal, dass die Berechnungsdauer der NORTH Komponenten drei Stunden und der ROME Komponenten eine Stunde beträgt. Der Skew-Kanten Algorithmus benötigt auf den NORTH Komponenten knapp vier Minuten länger und löst weniger Instanzen wie die globale Grad 2 Eliminierung. Die lokale Grad 2 Eliminierung löst am meisten nicht 1-planare Komponenten und ist zudem am schnellsten in dieser Kategorie. Somit empfiehlt sich diese Methode auch aus zeitlicher Sicht für dichtere Komponenten. Für 1-planare Komponenten ist die iterative Erhöhung der zulässigen Kreuzungen der schnellste Ansatz, was sich vor allem auch auf den ROME Komponenten bestätigt, da dieser Ansatz am meisten Instanzen löst.

Tabelle 6.2.: Durchschnittliche Dauer der Algorithmen über alle ROME und NORTH Komponenten.

Algorithmus	NORTH Ø Dauer (min) pro Komp.			ROME Ø Dauer (min) pro Komp.		
	gelöst	1-planar	¬1-planar	gelöst	1-planar	¬1-planar
Skew-Kanten	10.17	3.78	88.74	1.21	1.21	-
Lokale Grad 2 E.	7.18	3.98	28.08	1.86	1.86	-
Globale Grad 2 E.	6.54	2.50	46.07	3.91	3.91	-
It. Kreuzungserhöhung	8.38	2.15	79.65	2.35	2.35	-

Abschließend wird die Wirksamkeit des Cut-Kriteriums aus Lemma 5.1 nochmals genauer betrachtet. Dazu gibt Tabelle 6.3 die Verteilung der verschiedenen Abbruchkriterien auf den NORTH Komponenten wieder. Bei der Auswertung muss jedoch beachtet werden, dass die Reihenfolge der Anwendung der Kriterien eine entscheidende Rolle spielt, da auch mehrere Abbruchbedingungen gleichzeitig zutreffen können. Für diese Auswertung wurde folgende Reihenfolge verwendet:

(1) Die Kantenanzahl der Komponente ist zu hoch, um noch 1-planar sein zu können. Die Komponente ist folglich trivial nicht 1-planar.

(2) Der von den gesättigten Kanten induzierte Graph ist nicht planar.

(3) Der Kuratowski-Minor hat weniger als zwei nicht gesättigte Kanten.

Treffen mehrere Bedingungen zu, so wird nur immer das erste zutreffende Kriterium gezählt. Zwischen den Algorithmen kann man keine großen Unterschiede im Verhältnis der Cuts zueinander feststellen. Auffallend ist jedoch, dass Kriterium (2) den größten Anteil einnimmt, weshalb der Standardalgorithmus bereits 13 Komponenten als nicht 1-planar klassifiziert. Dennoch erkennt man die verbleibenden knapp 10% von Kriterium (3), die bei der lokalen Grad 2 Eliminierung dazu führt, dass 30 Komponenten als nicht 1-planar eingeordnet werden. Kriterium (1) fällt wegen der vielen Komponenten mit einer Dichte kleiner als 3 nicht auf, wie Abbildung 6.11 erkennen lässt. Erst ab einer Dichte von 3 tritt es auf, dass eine Komponente durch das Einfügen von Kreuzungen und Drachenkanten trivial nicht 1-planar wird. Die Verteilung der Cuts auf den ROME Komponenten ist analog, wobei dort Kriterium (1) gar nicht verwendet wird, da nur Graphdichten bis zu 2,5 vorkommen.

Tabelle 6.3.: Prozentuale Verteilung der Cuts auf den NORTH Komponenten.

Algorithmus	Kriterium 1	Kriterium 2	Kriterium 3
Lokale Grad 2 Eliminierung	0,4%	92,2%	7,4%
Globale Grad 2 Eliminierung	0,3%	89,9%	9,8%

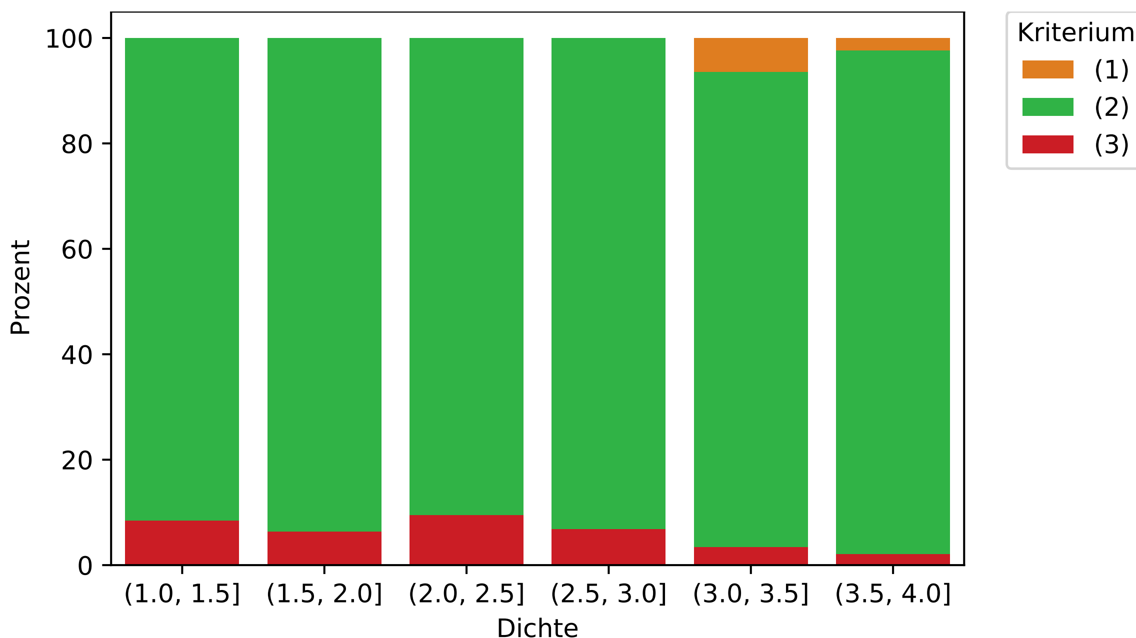


Abbildung 6.11.: Darstellung der prozentualen Verteilung der Abbruchkriterien pro Dichtebereich der NORTH Komponenten.

Aus all den zuvor gesehen Diagrammen lässt sich hinsichtlich der Frage nach der besten Kombinationsmöglichkeit der Einzeloptimierungen für jeden kombinierten Algorithmus ein bevorzugtes Anwendungsgebiet finden. Hat man überwiegend dünne Komponenten mit einer Dichte bis maximal 1,6, so liefert die iterative Erhöhung der zugelassenen Kreuzungen die besten Ergebnisse. Zusätzlich erhält man eine 1-planare Einbettung mit einer optimalen Anzahl an Kreuzungen. Für den Bereich bis zu einer Dichte von 2,0 kann sowohl die globale Kanteneliminierung an Grad-2-Knoten als auch die iterative Kreuzungserhöhung verwendet

werden. Für dichtere Komponenten bringt die lokale Kanteneliminierung an Grad-2-Knoten die besten Resultate hervor. Des Weiteren werden knapp 10% mehr Cuts durch Lemma 5.1 erreicht. Wählt man jeweils den besten Algorithmus auf den jeweiligen Graph-Datenbanken, so liefert die globale Kanteneliminierung an Grad-2-Knoten eine prozentuale Zunahme um 37,0% an mehr gelösten Instanzen gegenüber dem Algorithmus von Binucci, Didimo und Montecchiani. Mit der iterativen Kreuzungserhöhung beträgt die prozentuale Zunahme auf den ROME Komponenten 122,4%.

7. Fazit

Wir verbesserten das von Binucci, Didimo und Montecchiani [3] entwickelte Backtrackingverfahren durch eine gezielte Wahl von Kantenpaaren, welche man kreuzt oder, wie bei der Kanteneliminierung an Grad-2-Knoten, vorerst nicht kreuzen will. Die dabei in C++ implementierten Ideen wurden auf Basis der bekannten NORTH und ROME Graph-Datenbanken getestet und miteinander verglichen. Es stellte sich heraus, dass die Kombination der verschiedenen Einzeloptimierungen zu zwei ähnlichen Algorithmen führt, welche sich lediglich in der Reihenfolge bei der Kanteneliminierung an Grad-2-Knoten und der Minimierung der Anzahl an Kuratowski-Minoren unterscheiden. Ersterer ist die globale Kanteneliminierung an Grad-2-Knoten Methode, welche auf den NORTH Graphen das beste Gesamtergebnis an gelösten 2-Zusammenhangskomponenten (Komponenten) liefert. Der zweite Ansatz ist die lokale Kanteneliminierung an Grad-2-Knoten, welcher sich besonders gut für nicht 1-planare Komponenten eignet. Da nicht 1-planare Komponenten vermehrt bei Komponenten ab einer Dichte von 2 auftreten, empfiehlt sich die lokale Methode für solche Instanzen. Für Komponenten mit einer Dichte bis zu 1,6 bietet sich der Algorithmus mit der iterativen Erhöhung der zulässigen Kreuzungen an, welcher zusätzlich eine 1-planare Einbettung mit einer optimalen Anzahl an Kreuzungen liefert. Im Dichtebereich von 1,6 bis 2 kann sowohl die iterative Kreuzungserhöhung als auch die globale Kanteneliminierung verwendet werden. Basierend auf den Kuratowski-Minoren haben wir ein neues Abbruchkriterium vorgestellt, welches den Suchbaum weiter einschränkt und so für eine bessere Erkennung von nicht 1-planaren Instanzen sorgt.

Weitere reizvolle Themen in diesem Forschungsbereich wäre zum einen weitere Abbruchkriterien für diese Algorithmen zu finden. Vor allem beim Algorithmus mit der iterativen Erhöhung der zulässigen Kreuzungen könnte man durch diese zusätzliche Einschränkung noch weitere Kriterien erforschen. Zum anderen wäre die Erweiterung dieses praktisch anwendbaren Algorithmus, um k-planare Graphen zu identifizieren, interessant. Dies hätte zur Folge, dass pro Kante nicht mehr nur eine Kreuzung sondern k Kreuzungen zulässig wären. Insbesondere müssten sämtliche Optimierungen neu überdacht werden. Zum Beispiel wäre das Einfügen von Drachenkanten nicht ohne weiteres möglich, selbst wenn man das Konzept der gesättigten Kanten allgemeiner fasst und auf k-gesättigte Kanten ausweitet.

Literaturverzeichnis

[1] Ergebnisse. http://mozart.diei.unipg.it/montecchiani/1planarity/labels.xlsx. Zugriff: 08.06.2020.

[2] North und rome graphen. http://www.graphdrawing.org/data.html. Zugriff: 08.06.2020.

[3] C. Binucci, W. Didimo, and F. Montecchiani. An experimental study of a 1-planarity testing and embedding algorithm. *Springer*, 2019.

[4] John M. Boyer and Wendy J. Myrvold. On the cutting edge: Simplified o(n) planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004.

[5] John M. Boyer and Wendy J. Myrvold. On the cutting edge: simplified planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004.

[6] Franz J. Brandenburg. Recognizing optimal 1-planar graphs in linear time. *Algorithmica*, 80(1):1–28, October 2016.

[7] Sergio Cabello and Bojan Mohar. Adding one edge to planar graphs makes crossing number and 1-planarity hard. *SIAM Journal on Computing*, 42(5):1803–1829, January 2013.

[8] Markus Chimani, Carsten Gutwenger, Michael Junger, Gunnar W Klau, Karsten Klein, and Petra Mutzel. *The Open Graph Drawing Framework*, pages 543 – 570. CRC Press, Australia, 2013.

[9] Július Czap and Dávid Hudák. 1-planarity of complete multipartite graphs. *Discrete Applied Mathematics*, 160(4):505 – 512, 2012.

[10] Carsten Gutwenger and Petra Mutzel. An experimental study of crossing minimization heuristics. In Giuseppe Liotta, editor, *Graph Drawing*, pages 13–24, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[11] Weidong Huang, Peter Eades, and Seok-Hee Hong. Larger crossing angles make graphs easier to read. *J. Vis. Lang. Comput.*, 25(4):452–465, August 2014.

[12] Stephen G. Kobourov, Giuseppe Liotta, and Fabrizio Montecchiani. An annotated bibliography on 1-planarity. *Computer Science Review*, 25:49 – 67, 2017.

[13] Vladimir P. Korzhik and Bojan Mohar. Minimal obstructions for 1-immersions and hardness of 1-planarity testing. *Journal of Graph Theory*, 72(1):30–71, 2013.

[14] Casimir Kuratowski. Sur le problème des courbes gauches en topologie. *Fundamenta Mathematicae*, 15(1):271–283, 1930.

[15] János Pach and Géza Tóth. Graphs drawn with few crossings per edge. *Combinatorica*, 17(3):427–439, September 1997.

[16] H.C Purchase. Effective information visualisation: a study of graph drawing aesthetics and algorithms. *Interacting with Computers*, 13(2):147–162, 12 2000.

[17] Helen Purchase. Which aesthetic has the greatest effect on human understanding? In Giuseppe DiBattista, editor, *Graph Drawing*, pages 248–261, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[18] Helen C. Purchase, David Carrington, and Jo-Anne Allder. Empirical evaluation of aesthetics-based graph layout. *Empirical Softw. Engg.*, 7(3):233–255, September 2002.

Anhang

A. Pseudocode von NextBestCrossing

```
Input : Graph mit eingefügten Kreuzungen G = (V, E),
Kuratowski-Minor H_i = (V', E')
Output : Kreuzung (e_1, e_2), NULL
1  skewEdges ← findSkewEdges(H_i)
   // Lösche Skew Kanten, welche bereits gesättigt sind
2  skewEdges ← delSaturatedEdges(skewEdges)
   // Finde erweiterte K_4s wie in Algorithmus 4.1
3  k_4s ← FindK4(G)
   // Finde K_4 mit mindestens einer Skew Kante
4  k_4 ← getK4WithMostSkewEdges(k_4s, skewEdges)
5  if k_4 ≠ NULL and existsCrossing(k_4.edges) then
6  └ return getCrossing(k_4)
   // Finde alle Grad 2 Pfade
7  deg2Paths ← findDeg2Paths(H_i)
   // Nehme alle bis auf eine Kante pro Grad 2 Pfad
8  deg2Edges ← findDeg2Paths(deg2Paths)
9  bestSkewEdges ← skewEdges \ deg2Edges
10 if existsCrossing(bestSkewEdges) then
11 └ return getCrossing(bestSkewEdges)
   // Mache K_4 Kreuzung ohne Skew Kanten innerhalb H_i
12 k_4s ← getK4InH(k_4s, H_i)
13 if k_4 ≠ NULL and existsCrossing(k_4.edges) then
14 └ return getCrossing(k_4)
   // Mache beliebige, noch mögliche Kreuzung innerhalb H_i, wie in
      Algorithmus 4.2
15 return NextKuratowskiCrossing(H_i)
```

B. Detaillierte Ergebnisse der Algorithmen

Tabelle B.1.: Ergebnis des Algorithmus ohne Optimierungen, mit Skew-Kanten Optimierung und mit K_4-Optimierung.

Graphen	#	Ohne Optimierung			Skew-Kanten Optimierung			K_4 Optimierung		
		#gelöst	#1-planar	#¬1-planar	#gelöst	#1-planar	#¬1-planar	#gelöst	#1-planar	#¬1-planar
NORTH 1-10	55	55 (100%)	52 (95%)	3 (5%)	55 (100%)	52 (95%)	3 (5%)	55 (100%)	52 (95%)	3 (5%)
NORTH 11-20	114	74 (65%)	64 (56%)	10 (9%)	76 (67%)	66 (58%)	10 (9%)	74 (65%)	65 (57%)	9 (8%)
NORTH 21-30	84	16 (19%)	16 (19%)	0 (0%)	18 (21%)	18 (21%)	0 (0%)	15 (18%)	15 (18%)	0 (0%)
NORTH 31-40	46	6 (13%)	6 (13%)	0 (0%)	14 (30%)	14 (30%)	0 (0%)	6 (13%)	6 (13%)	0 (0%)
NORTH 41-50	29	5 (17%)	5 (17%)	0 (0%)	7 (24%)	7 (24%)	0 (0%)	5 (17%)	5 (17%)	0 (0%)
NORTH 51-60	43	0 (0%)	0 (0%)	0 (0%)	2 (5%)	2 (5%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
NORTH 61-70	21	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
NORTH 71-80	2	0 (0%)	0 (0%)	0 (0%)	1 (50%)	1 (50%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
NORTH 81-90	4	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
NORTH 91-100	2	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Gesamt NORTH	400	156 (39%)	143 (36%)	13 (3%)	173 (43%)	160 (40%)	13 (3%)	155 (39%)	143 (36%)	12 (3%)
R.-ROME 1-10	9	9 (100%)	9 (100%)	0 (0%)	9 (100%)	9 (100%)	0 (0%)	9 (100%)	9 (100%)	0 (0%)
R.-ROME 11-20	172	169 (98%)	169 (98%)	0 (0%)	171 (99%)	171 (99%)	0 (0%)	169 (98%)	169 (98%)	0 (0%)
R.-ROME 21-30	184	89 (48%)	89 (48%)	0 (0%)	173 (94%)	173 (94%)	0 (0%)	89 (48%)	89 (48%)	0 (0%)
R.-ROME 31-40	47	3 (6%)	3 (6%)	0 (0%)	42 (89%)	42 (89%)	0 (0%)	3 (6%)	3 (6%)	0 (0%)
Gesamt R.-ROME	412	270 (66%)	270 (66%)	0 (0%)	395 (96%)	395 (96%)	0 (0%)	270 (66%)	270 (66%)	0 (0%)
Gesamt	812	426 (52%)	413 (51%)	13 (2%)	568 (70%)	555 (68%)	13 (2%)	425 (52%)	413 (51%)	12 (1%)

Tabelle B.2.: Ergebnis des Algorithmus mit erweiterten Drachenkanten, mit Grad 2 Eliminierung und mit Kuratowski-Optimierung.

		Erweiterte Drachenkanten			Grad 2 Eliminierung			Kuratowski-Optimierung		
Graphen	#	#gelöst	#1-planar	#¬1-planar	#gelöst	#1-planar	#¬1-planar	#gelöst	#1-planar	#¬1-planar
NORTH 1-10	55	55 (100%)	52 (95%)	3 (5%)	55 (100%)	52 (95%)	3 (5%)	55 (100%)	52 (95%)	3 (5%)
NORTH 11-20	114	76 (67%)	65 (57%)	11 (10%)	82 (72%)	73 (64%)	9 (8%)	65 (57%)	64 (56%)	1 (1%)
NORTH 21-30	84	22 (26%)	22 (26%)	0 (0%)	30 (36%)	30 (36%)	0 (0%)	28 (33%)	28 (33%)	0 (0%)
NORTH 31-40	46	6 (13%)	6 (13%)	0 (0%)	12 (26%)	12 (26%)	0 (0%)	14 (30%)	14 (30%)	0 (0%)
NORTH 41-50	29	5 (17%)	5 (17%)	0 (0%)	9 (31%)	9 (31%)	0 (0%)	9 (31%)	9 (31%)	0 (0%)
NORTH 51-60	43	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	3 (7%)	3 (7%)	0 (0%)
NORTH 61-70	21	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
NORTH 71-80	2	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (50%)	1 (50%)	0 (0%)
NORTH 81-90	4	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
NORTH 91-100	2	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Gesamt NORTH	400	164 (41%)	150 (38%)	14 (4%)	188 (47%)	176 (44%)	12 (3%)	175 (44%)	171 (43%)	4 (1%)
R.-ROME 1-10	1.56	9 (100%)	9 (100%)	0 (0%)	9 (100%)	9 (100%)	0 (0%)	9 (100%)	9 (100%)	0 (0%)
R.-ROME 11-20	1.48	170 (99%)	170 (99%)	0 (0%)	172 (100%)	172 (100%)	0 (0%)	169 (98%)	169 (98%)	0 (0%)
R.-ROME 21-30	1.36	107 (58%)	107 (58%)	0 (0%)	181 (98%)	181 (98%)	0 (0%)	168 (91%)	168 (91%)	0 (0%)
R.-ROME 31-40	1.35	3 (6%)	3 (6%)	0 (0%)	28 (60%)	28 (60%)	0 (0%)	38 (81%)	38 (81%)	0 (0%)
Gesamt R.-ROME	412	289 (70%)	289 (70%)	0 (0%)	390 (95%)	390 (95%)	0 (0%)	384 (93%)	384 (93%)	0 (0%)
Gesamt	812	453 (56%)	439 (54%)	14 (2%)	578 (71%)	566 (70%)	12 (1%)	559 (69%)	555 (68%)	4 (0%)

Tabelle B.3.: Ergebnisse der kombinierten Algorithmen.

Graphen	#	Lokale Grad 2 Eliminierung			Globale Grad 2 Eliminierung			Iterative Kreuzungserhöhung		
		#gelöst	#1-planar	#¬1-planar	#gelöst	#1-planar	#¬1-planar	#gelöst	#1-planar	#¬1-planar
NORTH 1-10	55	55 (100%)	52 (95%)	3 (5%)	55 (100%)	52 (95%)	3 (5%)	55 (100%)	52 (95%)	3 (5%)
NORTH 11-20	114	99 (87%)	74 (65%)	25 (22%)	93 (82%)	74 (65%)	19 (17%)	86 (75%)	72 (63%)	14 (12%)
NORTH 21-30	84	40 (48%)	40 (48%)	0 (0%)	45 (54%)	45 (54%)	0 (0%)	44 (52%)	44 (52%)	0 (0%)
NORTH 31-40	46	18 (39%)	16 (35%)	2 (4%)	19 (41%)	19 (41%)	0 (0%)	21 (46%)	20 (43%)	1 (2%)
NORTH 41-50	29	8 (28%)	8 (28%)	0 (0%)	18 (62%)	18 (62%)	0 (0%)	13 (45%)	13 (45%)	0 (0%)
NORTH 51-60	43	3 (7%)	3 (7%)	0 (0%)	3 (7%)	3 (7%)	0 (0%)	3 (7%)	3 (7%)	0 (0%)
NORTH 61-70	21	0 (0%)	0 (0%)	0 (0%)	1 (5%)	1 (5%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
NORTH 71-80	2	2 (100%)	2 (100%)	0 (0%)	2 (100%)	2 (100%)	0 (0%)	1 (50%)	1 (50%)	0 (0%)
NORTH 81-90	4	1 (25%)	1 (25%)	0 (0%)	1 (25%)	1 (25%)	0 (0%)	1 (25%)	1 (25%)	0 (0%)
NORTH 91-100	2	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Gesamt NORTH	400	226 (57%)	196 (49%)	30 (8%)	237 (59%)	215 (54%)	22 (6%)	224 (56%)	206 (52%)	18 (5%)
ROME 1-10	9	9 (100%)	9 (100%)	0 (0%)	9 (100%)	9 (100%)	0 (0%)	9 (100%)	9 (100%)	0 (0%)
ROME 11-20	250	245 (98%)	245 (98%)	0 (0%)	247 (99%)	247 (99%)	0 (0%)	246 (98%)	246 (98%)	0 (0%)
ROME 21-30	1296	798 (62%)	798 (62%)	0 (0%)	1087 (84%)	1087 (84%)	0 (0%)	1128 (87%)	1128 (87%)	0 (0%)
ROME 31-40	1843	383 (21%)	383 (21%)	0 (0%)	712 (39%)	712 (39%)	0 (0%)	917 (50%)	917 (50%)	0 (0%)
ROME 41-50	1358	69 (5%)	69 (5%)	0 (0%)	109 (8%)	109 (8%)	0 (0%)	206 (15%)	206 (15%)	0 (0%)
ROME 51-60	1227	9 (1%)	9 (1%)	0 (0%)	16 (1%)	16 (1%)	0 (0%)	46 (4%)	46 (4%)	0 (0%)
ROME 61-70	1126	3 (0%)	3 (0%)	0 (0%)	4 (0%)	4 (0%)	0 (0%)	8 (1%)	8 (1%)	0 (0%)
ROME 71-80	929	1 (0%)	1 (0%)	0 (0%)	1 (0%)	1 (0%)	0 (0%)	3 (0%)	3 (0%)	0 (0%)
ROME 81-90	214	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (0%)	1 (0%)	0 (0%)
ROME 91-100	1	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Gesamt ROME	8253	1517 (18%)	1517 (18%)	0 (0%)	2185 (26%)	2185 (26%)	0 (0%)	2564 (31%)	2564 (31%)	0 (0%)
Gesamt	8653	1743 (20%)	1713 (20%)	30 (0%)	2422 (28%)	2400 (28%)	22 (0%)	2788 (32%)	2770 (32%)	18 (0%)

Tabelle B.4.: Ergebnis des Skew-Kanten Algorithmus auf den ROME Graphen.

Graphen	Anzahl	Ø Dichte	#gelöst	#1-planar	#nicht 1-planar
NORTH 1-10	55	1.98	55 (100%)	52 (95%)	3 (5%)
NORTH 11-20	114	2.03	76 (67%)	66 (58%)	10 (9%)
NORTH 21-30	84	2.00	18 (21%)	18 (21%)	0 (0%)
NORTH 31-40	46	1.78	14 (30%)	14 (30%)	0 (0%)
NORTH 41-50	29	1.64	7 (24%)	7 (24%)	0 (0%)
NORTH 51-60	43	2.05	2 (5%)	2 (5%)	0 (0%)
NORTH 61-70	21	1.93	0 (0%)	0 (0%)	0 (0%)
NORTH 71-80	2	1.58	1 (50%)	1 (50%)	0 (0%)
NORTH 81-90	4	1.67	0 (0%)	0 (0%)	0 (0%)
NORTH 91-100	2	1.52	0 (0%)	0 (0%)	0 (0%)
Gesamt NORTH	400	1.95	173 (43%)	160 (40%)	13 (3%)
ROME 1-10	9	1.56	9 (100%)	9 (100%)	0 (0%)
ROME 11-20	250	1.50	230 (92%)	230 (92%)	0 (0%)
ROME 21-30	1296	1.41	610 (47%)	610 (47%)	0 (0%)
ROME 31-40	1843	1.44	259 (14%)	259 (14%)	0 (0%)
ROME 41-50	1358	1.45	36 (3%)	36 (3%)	0 (0%)
ROME 51-60	1227	1.45	6 (0%)	6 (0%)	0 (0%)
ROME 61-70	1126	1.45	3 (0%)	3 (0%)	0 (0%)
ROME 71-80	929	1.47	0 (0%)	0 (0%)	0 (0%)
ROME 81-90	214	1.46	0 (0%)	0 (0%)	0 (0%)
ROME 91-100	1	1.42	0 (0%)	0 (0%)	0 (0%)
Gesamt ROME	8253	1.45	1153 (14%)	1153 (14%)	0 (0%)
Gesamt	8653	1.47	1326 (15%)	1313 (15%)	13 (0%)