Karlsruhe Institute of Technology

Visualisierung planarer Graphen mit kleiner äußerer Facette

Evaluation des Kantenlöschen-Zeichnen-Kanteneinfügen-Frameworks

Bachelorarbeit
von

Adrian Herrmann

An der Fakultät für Informatik
Institut für Theoretische Informatik

Erstgutachter:	Prof. Dr. Dorothea Wagner
Zweitgutachter:	Prof. Dr. Peter Sanders
Betreuende Mitarbeiter:	Dipl.-Inform. Thomas Bläsius
	Dr. Tamara Mchedlidze

Bearbeitungszeit: 16. Februar 2014 – 16. Juni 2014

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst habe und dass alle wörtlich oder sinngemäß übernommenen Stellen in dieser Arbeit als solche gekennzeichnet sind.

Karlsruhe, den 16. Juni 2014

Zusammenfassung

In dieser Arbeit wird ein Framework vorgestellt, welches einen Graphen zeichnet. Zuerst werden die verwendeten Begriffe und Algorithmen erklärt. Dann wird das eigentliche Framework vorgestellt, welches aus 3 Schritten besteht. Zuerst werden verschiedene Möglichkeiten aufgezeigt, Kanten auszuwählen um sie temporär zu löschen. Danach zeichnet man den Graphen mit einem Zeichenalgorithmus. Schließlich werden die zuvor gelöschten Kanten kreuzungsfrei in die Zeichnung wieder eingefügt. Zuletzt wird das Framework evaluiert, das heißt sein Ergebnis wird vorgestellt und anhand bestimmter Kriterien bewertet.

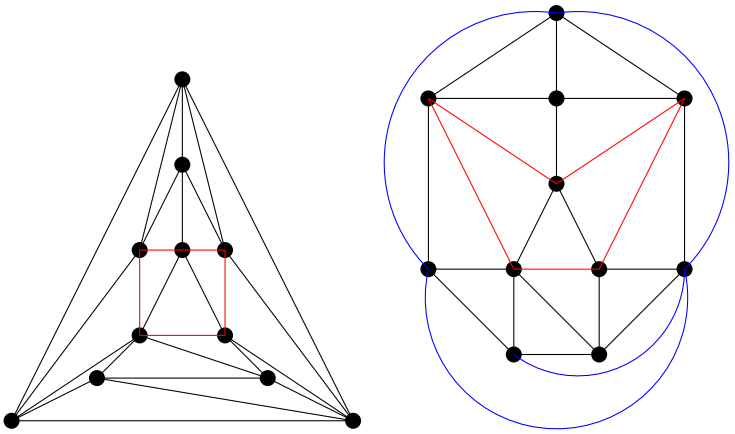
Inhaltsverzeichnis

1. Einleitung

Viele Informationen sind relationaler Natur und lassen sich daher mit Graphen darstellen: Karten jeder Art, z.B. Landkarten oder Straßenkarten, Liniennetze, auch abstrakte Dinge, wie Beziehungen in sozialen Netzen. Oft gibt es davon eine natürliche Zeichnung, z.B. geographische Position bei Karten. Diese Zeichnung ist jedoch nicht immer die beste. Manchmal ist die natürliche Art den Graphen darzustellen aufgrund bestimmter Faktoren, wie z.B. Örtlicher Gegebenheiten, nur schwer lesbar. Dabei gibt es häufig eine bessere Methode den Graphen zu zeichnen. Bei anderen Graphen gibt es keine natürliche Zeichnung, man will sie trotzdem gut lesbar zeichnen. Sehr störend für die Lesbarkeit sind vor allem Kantenkreuzungen (nach [Pur97]). Viele Graphen sind planar, allerdings ist deren natürliche Zeichnung nicht immer planar.

Ein Graph heißt *planar*, wenn er in der Ebene ohne Kreuzungen gezeichnet werden kann.

Es ist daher besser den Graph auch planar zu zeichnen. Doch nicht alle Graphen sind auch planar, man kann sie aber planarisieren. Bei einer *Planarisierung* werden alle Kreuzungen durch Dummy Knoten ersetzt. Diese haben Grad 4, wobei der *Grad* eines Knotens die Anzahl der eingehenden und ausgehenden Kanten angibt. Dass sich mehrere Kanten in einem Punkt kreuzen, schließt man aus. Sollte dies trotzdem vorkommen, kann man eine Kante verschieben. Man kann vorher die Anzahl der Kreuzungen minimieren, dies ist aber nicht notwendig. Dieses Verfahren der Planarisierung wurde von Gutwenger et al. [GMW01] untersucht. Inhalt seiner Arbeit war das Einfügen von Kanten in einen planaren Graphen und dabei möglichst wenig Kanten zu kreuzen, wobei die Kreuzungen durch Dummy Knoten ersetzt wurden. Verwendet man dies zur Planarisierung fängt man mit einem planaren Teilgraph an und fügt die Kanten nacheinander ein. Ein anderes Merkmal für die Lesbarkeit von Graphen ist der Platzverbrauch. Wenn ein Graph mehrere DIN-A 4 Seiten groß ist, ist er schwer lesbar und vor allem kaum zu überblicken. Ihn kleiner zu zeichnen, erhöht nicht immer die Lesbarkeit, da der Abstand zwischen zwei Knoten zu klein werden kann und man sie nicht mehr unterscheiden kann. Bei Graphen mit kleiner äußerer Facette ist dies häufig der Fall (siehe 1(a)). Außerdem führt eine kleine äußere Facette, bei einer orthogonalen Zeichnung, zu vielen Knicken [Bis13]. Man kann eine andere Facette als äußere wählen, falls möglich. Ein anderer Ansatz ist die äußere Facette temporär zu vergrößern, indem man Kanten dieser löscht. Eine Möglichkeit diese Kanten zu finden ist einen induzierten Baum im Dualgraph zu suchen und die entsprechenden Kanten im Graphen zu löschen. Der *Dualgraph* $G^\star$ zu einem planaren Graph G mit einer festen planaren Einbettung enthält für jede Facette in G einen Knoten und zwei Knoten in $G^\star$ sind mit einer Kante verbunden, wenn die dazugehörigen Facette in G eine gemeinsame Kante haben. Haben



(a) Ein Graph mit kleiner äußerer Facette

(b) Der selbe Graph neu gezeichnet.

Abbildung 1.1: Anwendung des Ansatzes mit kleinem Baum

zwei Facetten in G mehrere gemeinsame Kanten, so entstehen in $G^\star$ Mehrfachkanten. Ein Graph (V_1, E_1) heißt *Teilgraph* zu einem Graphen (V_2, E_2), wenn $V_1 \subseteq V_2$ und $E_1 \subseteq E_2$. Ein Teilgraph ist *induziert*, wenn gilt: $\forall u, v \in V1 : (u, v) \in E2 \Leftrightarrow (u, v) \in E1$. Man will dabei die äußere Facette möglichst stark vergrößern, ohne dabei zu viel Struktur zu verlieren. Wie viele Kanten man löscht ist eine schwierige Frage. Man will möglichst viel Struktur behalten und dabei die äußere Facette maximal vergrößern. Löscht man keine Kanten, so hat man die gesamte Struktur behalten, aber die äußere Facette nicht vergrößert. Löscht man alle Kanten bis auf einen Spannbaum, so hat man die äußere Facette maximal vergrößert, aber es ist kaum noch Struktur übrig. Für allgemeine Graphen wurde von Erdős et al. [ESS86] die NP-Vollständigkeit für das Finden von maximalen induzierten Bäumen bewiesen. Bischof [Bis13] hat darüber hinaus gezeigt, dass dies auch in planaren Graphen NP-Vollständig ist. Dieser Ansatz wird in diesem Framework verwendet. Hat man die Facette vergrößert, hofft man, den Graph nun besser zeichnen zu können. Besser heißt hierbei mit weniger Platzverbrauch und mit weniger Knicke pro Kante, wodurch sich die Lesbarkeit erhöht [Pur97]. Der Graph wird dann mit einem bereits bekannten Zeichenalgorithmus neu gezeichnet. Dabei erhofft man sich, dass man aufgrund der größeren äußeren Facette der Graph mit weniger Platzverbrauch gezeichnet werden kann und dadurch besser lesbar wird. Dabei kann man auch mehrere Zeichenalgorithmen ausprobieren und je nach Situation oder Ergebnis einen anderen wählen. Zuletzt fügt man die zuvor entfernten Kanten ein. Dabei werden die Kanten über die äußere Facette geroutet, falls man sie nicht durch eine innere Facette legen kann. Die äußere Facette ist am Ende nicht größer als vorher, da es sich um den selben Graphen handelt und man keine andere Facette als äußere wählt. Trotzdem versucht man damit die oben genannten Ziele zu erreichen. In Bild 1(b) sieht man, wie dieser Ansatz auf den in Bild 1(a) gezeigten Graphen angewendet wurde. Hierbei wurden die drei Kanten der äußeren Facette gelöscht und eine Kante weiter innen. Man sieht leicht, dass dabei ein induzierter Baum verwendet wurde, der nicht maximal ist. Man kann noch mindestens ein Kante aus dem inneren Kreis (in rot) löschen, wobei noch eine weitere Kante gelöscht werden muss, damit ein Baum entsteht.

1.1 Verwandte Arbeiten

Mit der Zeichnung von Graphen haben sich bereits sehr viele Personen beschäftigt. Das Wichtigste hierbei ist zu Wissen woran man eine gute Zeichnung erkennt. Purchase [Pur97] hat dies herausgefunden. Nach ihrer Arbeit, ist das wichtigste Kriterium die Anzahl der

Kreuzungen, d.h. je weniger Kreuzungen desto besser lesbar. Daraus folgt, dass planare Graphen besser lesbar sind, als nicht-planare. Die Kreuzungsminimierung ist nach Garray und Johnson [GJ82] NP-vollständig. Deshalb ist es einfacher eine Planarisierung durchzuführen. Andere Kriterien sind Platzverbrauch, Knickminimierung und Winkelauflösung. Die Zeichnung eines planaren Graphen durch Wahl einer anderen äußeren Facette zu verbessern und dabei die Facettentiefe zu minimieren ist ein Ansatz, der von Gutwenger et al. [GM04] verfolgt wurde. In ihrer Arbeit werden erst beide Kriterien getrennt behandelt und aus allen möglichen Zeichnungen mit minimaler Facettentiefe diejenige mit der größten äußeren Facette gewählt.

1.2 Struktur dieser Arbeit

In dieser Arbeit wird das erstellte Framework vorgestellt. Sie ist folgendermaßen strukturiert: In Kapitel 2 werden diejenigen verwendeten Algorithmen erklärt, die allgemein bekannt sind und in Kapitel 3 als bekannt vorausgesetzt werden. Außerdem werden dort die verwendeten Begriffe definiert. In Kapitel 3 wird das eigentliche Framework vorgestellt. Dieses ist in drei Schritte unterteilt. Der erste Teil (3.1) beschäftigt sich mit dem Löschen der Kanten. Im zweiten Teil (3.2) werden die verwendeten und möglichen Zeichenalgorithmen aufgezeigt. Im dritten Teil (3.3) wird das Einfügen der Kanten erklärt. Schließlich wird in Kapitel 4 das Framework evaluiert, d.h. das Ergebnis wird bewertet und seine Laufzeit wird gemessen.

2. Grundlagen

In diesem Kapitel werden die verwendeten Algorithmen und die wichtigen Begriffe erklärt.

2.1 Begriffe und Darstellung

Ein *Graph* G mit Knoten V und Kanten E wird als G(V,E) geschrieben. Eine *Kante* $e \in E$ die vom Knoten $u \in V$ zum Knoten $v \in V$ geht, wird mit $(u, v) \in E$ beschrieben. Die *Zeichnung* eines Graphen G (V,E) ordnet jedem Knoten $v \in V$ eine Position zu, sodass keine zwei Knoten die selbe Position haben. Jede Kante $e = (u, v)$ wird durch eine Kurve dargestellt, die von u nach v geht. Zwei Kanten schneiden sich nur in endlich vielen Punkten. Eine Zeichnung ist *planar*, wenn sich zwei Kanten nur in den gemeinsamen Eckpunkten schneiden. Ein Graph heißt *planar*, wenn er planar gezeichnet werden kann. Die *Einbettung* eines Graphen ist die Reihenfolge der Kanten um dem Graphen. Eine *Facette* eines Graphen bei fester Einbettung und dazu passender Zeichnung ist eine von Kanten begrenzte Fläche. Die unendliche Fläche außerhalb des Graphen ist die *äußere* Facette. Die *Größe* einer Facette beschreibt die Anzahl der Kanten, an die sie angrenzt. Ein Graph ist *zusammenhängend*, wenn es für je zwei Knoten $u, v \in E$ einen Weg von u nach v gibt, wenn man die Kanten als ungerichtet betrachtet. Eine *Brücke* ist eine Kante e, sodass $G \setminus \{e\}$ unzusammenhängend ist. Ein *Separator-Knoten* ist ein Knoten v, sodass $G \setminus \{v\}$ unzusammenhängend ist.

Ein Graph (V_1, E_1) heißt *Teilgraph* zu einem Graphen (V_2, E_2), wenn $V_1 \subseteq V_2$ und $E_1 \subseteq E_2$. Ein Teilgraph ist *induziert*, wenn gilt: $\forall u, v \in V1 : (u, v) \in E2 \Leftrightarrow (u, v) \in E1$. Der *Dualgraph* $G^\star$ zu einem planaren Graph G mit einer festen planaren Zeichnung enthält für jede Facette in G einen Knoten und für jede Kante e in G enthält $G^\star$ eine Kante $e^\star$, die die rechte Facette von e mit der linken Facette von e verbindet. Beachte dass dabei Schleifen, wenn G ein Brücke enthält, und Mehrfachkanten, wenn zwei Kanten die selben Facetten begrenzen, entstehen können. Ein *Kreis* ist ein Graph, wobei jeder Knoten Grad 2 hat. Ein *Baum* ist ein zusammenhängender Graph, der keinen Kreis als Teilgraph enthält. Ein *Inklusion-maximaler induzierten Baum* B in G ist ein Teilgraph von G, bei dem man keinen Knoten aus G mehr hinzufügen kann, sodass es ein induzierter Baum bleibt. Entweder der Teilgraph ist dann kein Baum mehr oder er ist nicht mehr induziert. Ein induzierter Baum ist *maximal*, wenn von allen möglichen Inklusion-maximalen induzierten Bäumen die meisten Knoten hat.

$P \to Q$ ist die Strecke zwischen P und Q, wobei P und Q Punkte sind.

2.2 Graham Scan

Der Graham Scan (nach Ronald Graham 1972) ist ein effizienter Algorithmus zur Berechnung der konvexen Hülle einer endlichen Menge von Punkten in der Ebene. Ein *Polygon* ist ein gezeichneter Kreis. Ein Polygon ist *konvex*, wenn für je zwei Punkte P und Q im Polygon $P \to Q$ ganz im Polygon liegt.

Definition 2.1 (Konvexe Hülle)**.** *Die Konvexe Hülle von n Punkten in der Ebene ist das kleinste konvexe Polygon, dass alle Punkte enthält.*

Die Laufzeit des Graham Scan liegt in $\mathcal{O}(n \cdot \log n)$, wobei n die Anzahl der Punkte ist. Gegeben ist eine Menge von Punkten, im Anwendungsfall sind dies die Knoten des Graphen und die Bend Punkte der Kanten. Zuerst wird der Startpunkt gesucht, dies ist der Punkt mit der kleinsten y-Koordinate. Falls es mehrere solche Punkte gibt, wähle den, mit der kleinsten x-Koordinate. Dieser Punkt sei P_0. Man kann P_0 in $\mathcal{O}(n)$ Zeit finden. Dann sortiert man die restlichen Punkte P aufsteigend nach dem Winkel zwischen $P_0 \to P$ und der x-Achse. Bei gleichem Winkel wird nach dem Abstand zu P_0 sortiert, wobei beim kleinsten Winkel aufsteigend und beim Rest absteigend sortiert wird. Dies ist bei einem Graphen wichtig, da man keine Kante durch einen Knoten ziehen will, da die schlecht lesbar ist. Man nimmt daher beim kleinsten Winkel und bei größten alle Punkte zu konvexen Hülle dazu. Damit diese dazugenommen werden, muss man die Punkte so sortieren. Dabei ist die Sortierung bei den anderen Winkeln egal, da dort nur der entfernteste Punkt zu konvexen Hülle gehört.

Sei nun S die sortierte Punktemenge. Man verwendet einen *Stack*, auf dem sich die Eckpunkte der konvexen Hülle für alle bereits abgearbeiteten Punkte befinden.

Definition 2.2 (Stack)**.** *Ein Stack (deutsch: Stapel- oder Keller- Speicher) ist eine Datenstruktur, die nur zwei Operationen hat. Push legt ein Element oben auf den Stack und Pop entfernt das oberste Element vom Stack. Es funktioniert somit nach dem LIFO (Last-In-First-Out) Schema.*

Zu Beginn liegen P_0 und P_1 auf dem Stack (P_i ist der i-te Punkt in S). Im k-ten Schritt betrachtet man P_k. Es sei P_t der oberste und P_{t-1} der zweit-oberste Punkt auf dem Stack. Nun wird geprüft ob P_k links von der Gerade durch $P_{t-1} \to P_t$ liegt. Falls ja, wird P_k auf den Stack gepusht. Sonst entferne P_t vom Stack und teste erneut. Sobald P_k auf den Stack gelegt wurde betrachte den nächsten Punkt. Der Algorithmus endet, wenn der letzte Punkt auf den Stack gelegt wurde. Alle Punkte die am Ende auf dem Stack liegen sind die Eckpunkte der konvexen Hülle. Die Laufzeit beträgt $\mathcal{O}(n)$, da jeder Punkt nur einmal auf den Stack gelegt wird und falls nötig entfernt wird. Er wird nicht mehrfach betrachtet. Außerdem sind Stack Operationen in konstanter Zeit möglich. Durch die Sortierung kommt man insgesamt auf $\mathcal{O}(n \cdot \log n)$.

2.3 Ear Clipping Triangulierung

Die *Triangulierung* eines Polygons zerlegt das Polygon in eine Menge von Dreiecken. Der Ear Clipping Algorithmus ist ein einfacher Algorithmus zur Triangulierung von Polygonen. Seine Laufzeit ist in $\mathcal{O}(n^2)$. Es gibt auch schnellere Algorithmen, z.B. Delaunay-Triangulation, die ein Laufzeit von $\mathcal{O}(n \cdot \log n)$ haben. Diese sind deutlich komplizierter. Der Ear Clipping Algorithmus funktioniert nur bei einfachen Polygonen. Ein Polygon heißt *einfach*, wenn es keine Überschneidungen oder Löcher enthält. Die Eckpunkte des Polygons sind im Uhrzeigersinn angeordnet.

Eine *Diagonale* in einem Polygon ist eine Strecke zwischen zwei nicht benachbarten Punkten des Polygons, sodass die Strecke komplett im Polygon liegt und das Polygon in zwei Teile

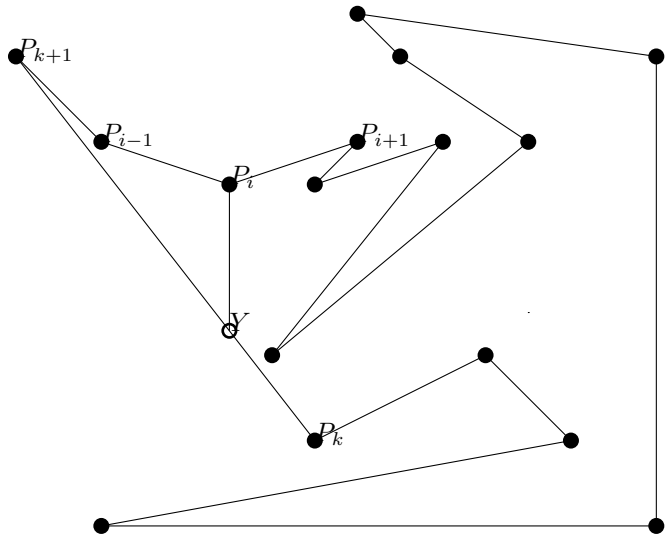


Abbildung 2.1: Konstruieren einer Diagonale

teilt. Ein *Ohr* in einem Polygon entsteht, wenn man eine Diagonale durch das Polygon legt und ein Teil des Polygons ein Dreieck ist. Dieser Teil ist ein Ohr. Der Algorithmus sucht nach einem Ohr und trennt dieses ab. Dies wird wiederholt, solange das Restpolygon kein Dreieck ist (n-3 Mal). Um ein Ohr zu finden, startet man mit einem beliebigen Punkt P_i des Polygons. Zuerst prüft man, ob P_i ein Ohr ist, indem man für die Punkte P_{i-1},P_i und P_{i+1} testet, ob sie eine konvexe Ecke bilden (d.h. der Winkel ist kleiner als 180 °). Ist dies der Fall, so wird noch für jeden anderen Punkt überprüft, ob er in dem Dreieck $P_{i-1} \to P_i \to P_{i+1}$ ist. Findet sich kein solche Punkt, so ist P_i ein Ohr und man ist fertig. Dies geht in $\mathcal{O}(n)$. Falls P_i kein Ohr ist, sucht man nach einem Punkt P_j, sodass $P_i \to P_j$ eine Diagonale bilden. Um diesen Punkt zu finden, legt man eine Winkel-Halbierende in den Winkel bei P_i. Der erste Schnittpunkt mit einer Kante des Polygons sei Y. Falls Y ein Knoten ist, setze $P_j := Y$. Sonst betrachte die Endpunkte dieser Kante P_k und P_{k+1}. Starte mit P_{k+1} und teste ob $P_{k+1} = P_{i-1}$.Falls nein, betrachte das Dreieck $P_{k+1} \to P_{i-1} \to P_i$. Prüfe ob ein Punkt in diesem Dreieck liegt. Falls ja, wähle denjenigen Punkt P_r als P_j, der den Winkel YP_iP_r minimiert. Falls es mehrere davon gibt, wähle denjenigen, der am nächsten an P_i liegt. Falls kein solcher Punkt existiert, setzte $P_j := P_{k+1}$. Falls $P_{k+1} = P_{i-1}$ mache das selbe mit P_k und P_{i+1}. ElGindy et al. haben bewiesen [EET93], dass man auf diese Weise immer eine Diagonale findet, d.h. es kann nicht vorkommen, dass $P_{k+1} = P_{i-1}$ und $P_k = P_{i+1}$ gilt, wenn P_i kein Ohr ist. Hat man die Diagonale gefunden, führe die Suche rekursiv auf der Hälfte des Polygons aus, dass man erhält, wenn $P_0 := P_i$ und $P_{k-1} := P_j$ und man die Punkte im Uhrzeigersinn anordnet. Wähle dabei als P_i den Punkt $P_{\lfloor \frac{k}{2} \rfloor}$. Dabei ist k die Anzahl der Eckpunkte der Polygon-Hälfte.

Nach dem Master-Theorem läuft die Ohr Suche in $\mathcal{O}(n)$. Das *Master-Theorem* ist ein Theorem, dass zeigt, wie man Laufzeiten von rekursiven Algorithmen abschätzt. Dieses Verfahren ein Ohr zu finden, wurde von ElGindy et al. [EET93] gefunden und dessen Laufzeit bewiesen. Damit kommt man insgesamt auf $\mathcal{O}(n^2)$.

2.4 Breitensuche

Die Breitensuche (BFS) ist ein Verfahren um alle Knoten eines Graphen abzulaufen. Es arbeitet alle Kanten eines Knotens ab, bevor es zum nächsten Knoten geht. Man benutzt dabei eine Warteschlange.

Definition 2.3 (Queue)**.** *Eine Queue (deutsch: Warteschlange) ist eine Datenstruktur, die nur zwei Operationen hat. Push hängt ein Element hinten an die Queue an und Pop entfernt das vorderste Element der Queue. Es funktioniert somit nach dem FIFO (First-In-First-Out) Schema.*

Zuerst wählt man einen Startknoten v und legt ihn in die Queue.
Das Verfahren nimmt ein Element aus der Queue und prüft für jede Kante (v,u) in dem Graphen, ob der Knoten u schon gefunden wurde. Falls nicht, markiere u als gefunden und füge ihn in die Queue ein. Wiederhole dies, solange die Queue noch nicht leer ist.

2.5 Tiefensuche

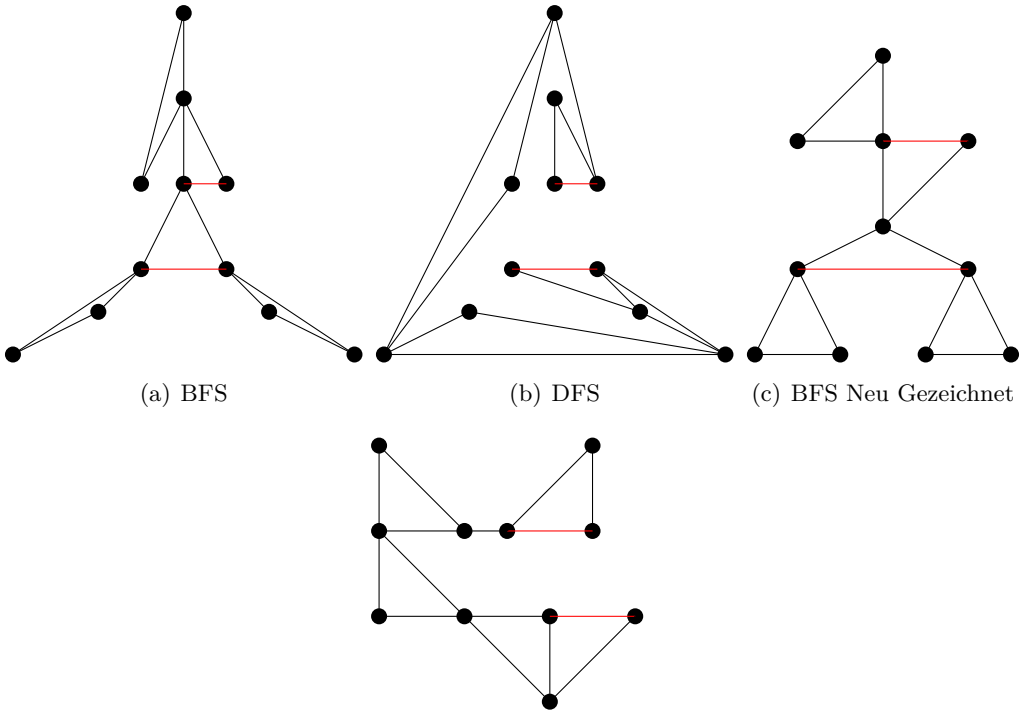
Die Tiefensuche (DFS) ist ein Verfahren um alle Knoten eines Graphen abzulaufen. Es folgt einer Kante, bis es nicht mehr weiter geht und macht dann mit der nächsten Kante weiter. Es wird ein Stack 2.2 verwendet. Zuerst wählt man einen Startknoten v und legt ihn auf den Stack.
Das Verfahren wählt für den obersten Knoten des Stacks eine Kante (v,u) und prüft, ob u schon gefunden wurde. Falls nicht, lege u auf den Stack und markiere ihn als gefunden. Sonst wähle die nächste Kante. Gibt es keine Kante oder sind alle u schon gefunden wurden entferne v vom Stack. Wiederhole dies bis der Stack leer ist.

3. Der Algorithmus

Im folgenden betrachten wir ausschließlich planare zusammenhängende Graphen. Nichtplanare Graphen kann man vorher Planarisieren. Wir beschreiben eine Algorithmus, der in drei Schritte unterteilt ist. Zuerst werden Kanten gelöscht um die äußere Facette zu vergrößern. Dabei erhofft man sich, dass der Graph durch die größere äußere Facette besser gezeichnet werden kann. Man will die äußere Facette möglichst stark vergrößern, ohne zu viel Struktur zu verlieren. Dann wird der Graph gezeichnet. Dabei werden verschiedene Zeichenalgorithmen verwendet und verglichen. Man erhofft sich, dass es durch den ersten Schritt möglich ist, den Graphen schön zu zeichnen. Anschließend fügt man die Kanten wieder ein. Dabei erhofft man sich, dass man die Kanten gut routen kann, sodass der Graph am Ende lesbarer ist als zuvor.

3.1 Schritt 1 - Kanten löschen

Eine Möglichkeit, die Kanten zu bestimmen, die man löschen will, ist im Dualgraph, ausgehend von der äußeren Facette, einen induzierten Baum zu suchen. Man verwendet einen Baum um zu garantiert, dass die äußere Facette auch tatsächlich vergrößert wird und man keine Kanten löscht, die die äußere Facette nicht vergrößern, da ein Baum zusammenhängend ist und die äußere Facette enthält. Außerdem bleibt der Graph dadurch zusammenhängend, da ein Baum keinen Kreis enthält. Der Baum ist induziert, damit keine neuen Brücken entstehen, da dies die Struktur zu stark zerstören würde. Um diesen Baum zu finden gibt es mehrere Möglichkeiten. Man kann einen Breitensuchbaum (BFS Tree) verwenden, das heißt man macht ausgehend von dem Knoten der äußeren Facette eine Breitensuche (siehe 2.4). Man könnte z.B. auch einen Tiefensuchbaum (DFS Tree) verwenden, d.h. man macht ausgehend von dem Knoten der äußeren Facette eine Tiefensuche (siehe 2.5) statt einer Breitensuche. Dabei muss man beachten, dass wenn die Suche einen Knoten findet man überprüfen muss, ob durch das Hinzufügen des Knotens der Baum induziert bleibt. Falls dies nicht der Fall ist, fügt man den Knoten nicht in den Stack bzw. die Queue ein, sodass er am Ende nicht im Baum ist. Beide Verfahren garantieren nicht, dass der Baum maximal wird. Es ist nach [Bis13] NP-Schwer einen maximalen Baum zu finden. Deshalb wird aus Laufzeitgründen darauf verzichtet. Man will einen möglichst großen Baum, weil dies zu einer starken Vergrößerung der äußeren Facette führt. In Bild 3.1 sieht man die Anwendung von BFS und DFS auf den in Bild 1 (a) gezeigten Graphen. Beide Verfahren löschen Kanten aus dem roten Kreis und produzieren einen Inklusion-maximalen induzierten Baum.



(a) BFS (b) DFS (c) BFS Neu Gezeichnet

(d) DFS Neu Gezeichnet

Abbildung 3.1: Anwendung der Baumsuchen

Eine andere Möglichkeit ist durch Löschen von k Kanten die äußere Facette maximal zu vergrößern. Dabei heißt Struktur erhalten nur k Kanten löschen. Dafür kann man auch einen Greedy-Algorithmus verwenden. Ein *Greedy-Algorithmus* ist ein Algorithmus, der in jedem Zustand den Folgezustand auswählt, der in diesem Moment das beste Ergebnis verspricht. In diesem Fall heißt das, dass die Kante gelöscht wird, die die äußere Facette maximal vergrößert. Dies führt man k-mal durch. Greedy-Algorithmen sind nicht immer optimal. Auch in diesem Fall nicht (siehe 3.1). Das Bild(b) zeigt die Anwendung eines Greedy-Algorithmus auf (a) mit $k := 4$. Nach Anwendung des Algorithmus hat die äußere Facette Größe 20. Löscht man in (a) stattdessen die grünen Kanten und die dazu angrenzende rote Kante erhält man eine Größe von 22. Der Grund dafür ist, dass die große rote Facette von kleinen Dreiecken eingeschlossen ist und die Facetten weiter außen größer als 3 sind. Deshalb wird der Greedy-Algorithmus die rote Facette erst erreichen, wenn von außen betrachtet nur noch Dreicke da sind. Dabei kann der Greedy-Algorithmus beliebig schlecht werden, da man die rote Facette beliebig vergrößern kann. Man muss dabei natürlich noch mehr Dreiecke einfügen.

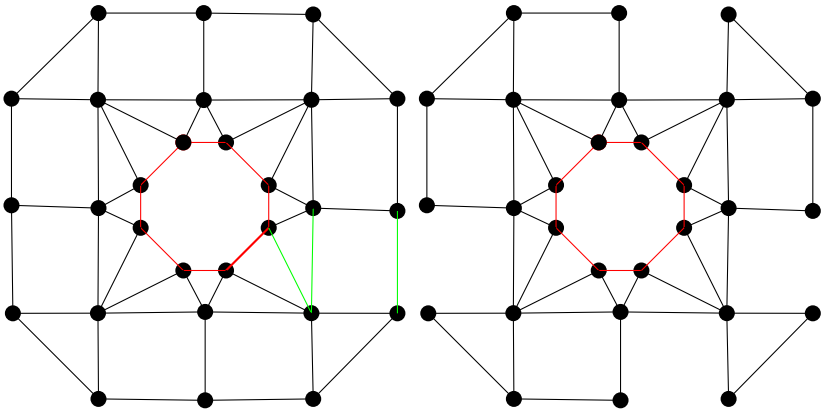
Man könnte auch die Struktur erhalten, indem man keine neuen Separator-Knoten einfügt und dabei möglichst viele Kanten löscht. Zu erlauben, dass der Graph unzusammenhängend werden darf, ist nicht sinnvoll, da das Löschen einer Brücke die äußere Facette sogar verkleinert.

Hat man schließlich diese Kanten mit einem Verfahren gefunden, werden sie gelöscht. Man muss dabei natürlich die Kanten speichern, damit man sie in Schritt 3 einfügen kann.

3.2 Schritt 2 - Graph zeichnen

In diesem Schritt wird ein Algorithmus von OGDF[1] verwendet. Ein Algorithmus ist „PlanarStraightLayout“. Da werden Kanten durch Strecken repräsentiert. Eine andere

[1] http://ogdf.net



(a) Vorher (b) Nachher

Abbildung 3.2: Anwendung eines Greedy-Algorithmus

Möglichkeit ist ein orthogonales Layout. Dabei will man möglichst wenig Knicke haben. In Bild 3.2 sieht man die Anwendung der Layout-Algorithmen. In (a) ist ein Graph gegeben ähnlich dem in Bild 3.1(a), die gelöschten Kanten wurden auch mit einem BFS Tree bestimmt. In (b) wurde „PlanarStraightLayout“auf diesen Graphen angewendet. Dieser Algorithmus trianguliert den Graphen, zeichnet ihn und löscht dann die hinzugefügten Kanten. In (c) wurde in orthogonaler Layouter verwendet.

Man kann auch ganz andere Zeichenalgorithmen verwenden, auf die ich aber nicht näher eingehen werde. Unabhängig vom gewählten Algorithmus will man am Ende möglichst wenig Platz verbrauchen.

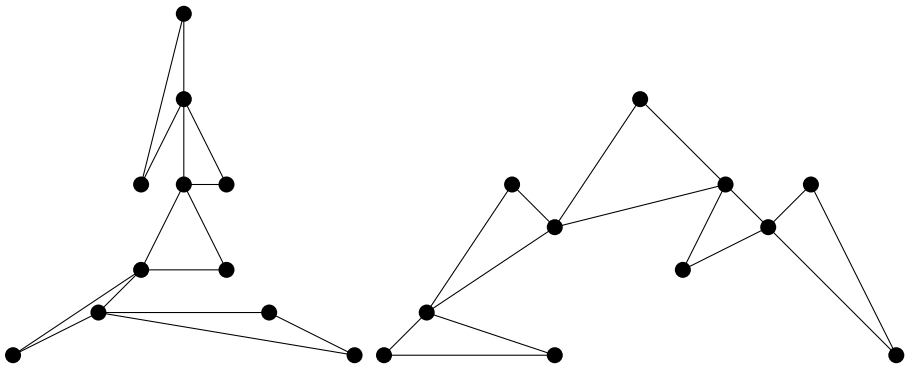
3.3 Schritt 3 - Kanten einfügen

Das Einfügen der Kanten läuft in zwei Schritten ab. Zuerst wird der Graph trianguliert, wobei man zuvor die äußere Facette kopiert, danach wird für jede Kante ein Pfad im Dualgraph gesucht. Die Triangulierung sorgt dafür, dass der Graph nach Einfügen der Kanten planar bleibt.

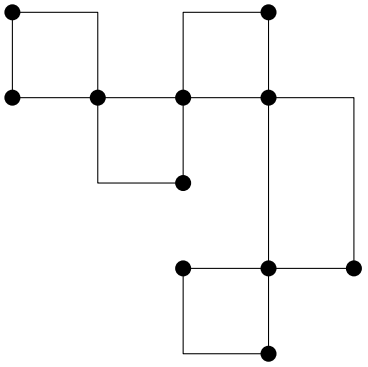
3.3.1 Die Triangulierung

Die Triangulierung läuft in drei Schritten ab. Sie hilft dabei, die Kanten besser einzufügen, da man sonst im Dualgraph keinen Pfad finden kann, der über die äußere Facette geht. Zuerst bildet man die Konvexe Hülle der äußeren Facette. Hierbei wird ein Grahamscan (siehe 2.2) verwendet. Man sieht dies in 3.3.1 (a). Dann wird die konvexe Hülle kopiert, indem man sie an ihrem Mittelpunkt streckt. Außerdem wird jeder Knoten auf der konvexen Hülle mit seiner Kopie durch ein Kante verbunden. Dies sorgt dafür, dass man einen Pfad im Dualgraph auf der äußeren Facette findet. Es wird die konvexe Hülle kopiert und nicht die ganze äußere Facette, da diese zum einen normalerweise sehr groß ist, wenn man dies in Schritt 1 geschafft hat oder sie bereits groß war, und kann zum anderen Dinge haben, die man nicht kopieren will, wie z.B. Grad 1 Knoten.

Der nun erhaltene Graph wird trianguliert, wobei die neue äußere Facette ignoriert wird. Wichtig dabei ist, dass man geometrisch trianguliert, nicht nur den Graphen allgemein. Dabei können alle Facetten als Polygone aufgefasst werden, außer die neue äußere Facette, deshalb wird diese ignoriert. Um die Facetten bzw. Polygone zu triangulieren wird der Ear-Clipping Algorithmus (siehe 2.3) verwendet. Dieser Algorithmus wird verwendet, weil er einfach zu implementieren und nachzuvollziehen ist. Er funktioniert, da die verwendeten



(a) Vorher (b) PlanarStraightLayout



(c) Orthogonales Layout

Abbildung 3.3: Anwendung von Layoutern

Facetten einfach Polygone sind. Das Polygon enthält keine Überschneidungen, weil der Graph planar ist und Überschneidungen im Polygon Kantenkreuzungen repräsentieren würden. Es enthält keine Löcher, weil ein Loch eine Facette repräsentiert, die in einer anderen Facette ist und es keinen Weg von einem Knoten der an diese Facette angrenzt zu einem Knoten, der nicht an diese Facette angrenzt gibt. D. h. der Graph ist unzusammenhängend. In Bild 3.3.1 sieht man die Anwendung dieser Schritte auf den Graphen aus Bild 3.2(b). Die eingefügten Kanten sind nicht immer schön bzw. gut lesbar, da Winkel von fast 180° zu schlecht lesbaren Kanten in der Konvexen Hülle und der Triangulierung führen.

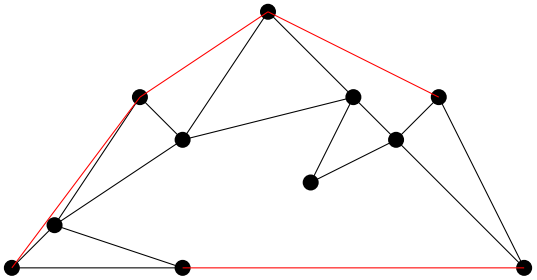
3.3.2 Das Einfügen

Hat man den Graphen trianguliert, so kann man die Kanten einfügen, indem man im Dualgraph einen Pfad sucht, wobei man nur Kanten der Triangulierung und Verbindungen $P \to P'$ kreuzt, wobei P' der kopierte Punkt P der konvexen Hülle ist. Die Kreuzungen werden hierbei durch Dummy Knoten repräsentiert. Dafür gibt es von ogdf eine vorgegebene Funktion [GMW01]. Dieser Funktion gibt man Kosten für die Kantenkreuzungen. Kanten, die man nicht kreuzen darf, erhalten m als Kosten (m = Anzahl Kanten). Alle anderen Kanten, die man kreuzen darf, erhalten Kosten 1. Die Dummy Knoten werde auf der gekreuzten Kante positioniert, z.B. äquidistant.

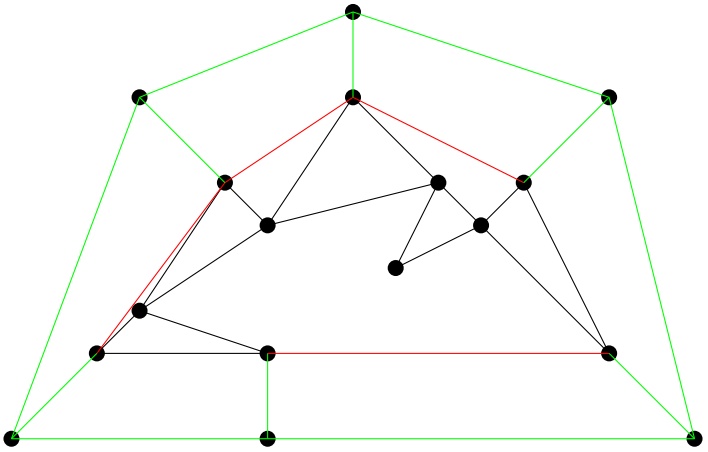
Man kann auch einen Kräfte basierten Ansatz verwenden um den Winkel an den Dummy Knoten zu verkleinern. Dabei wird jedem Dummy Knoten eine Kraft zugeordnet, die ihn auf der gekreuzten Kante in eine Richtung zieht. Diese Kraft hängt vom Winkel der kreuzenden Kante an diesem Knoten ab. Man wählt dann den Knoten mit der größten Kraft und verschiebt ihn. Dann berechnet man seine Kraft und die der angrenzenden Knoten neu und iteriert beliebig oft. Z.B. bis die höchste Kraft klein genug ist oder man legt einen

Parameter k fest und macht es k-Mal oder k · Anzahl Dummy Knoten. Dabei muss man beachten, dass zwei Dummy Knoten nicht ihre Reihenfolge auf der Kante tauschen. Dies erreicht man indem man für jeden Dummy Knoten ein Intervall festlegt, in dem er sich bewegen darf.

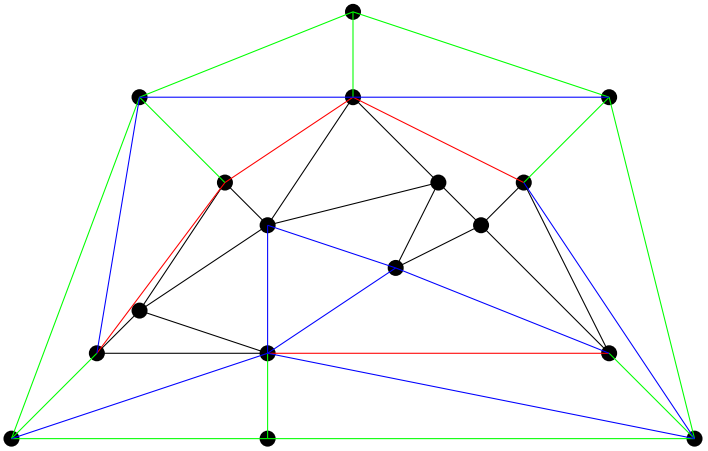
In Bild 3.3.1(d) sieht man, wie eine Kante eingefügt wurde. Man erhofft sich die Kanten mit möglichst wenig Dummy Knoten einfügen zu können, wobei die Dummy Knoten Knicke darstellen. Geht man von dem triangulierten Graph aus, so ist die Anzahl Dummy Knoten minimal, da der Algorithmus, der die Kanten routet auf einem kürzeste Wege Algorithmus beruht. Wie viele Knicke man bekommt hängt als von Schritt 1 und 2 und vom Graphen an sich ab. Zuletzt löscht man alle unnötigen Kanten und Knoten.



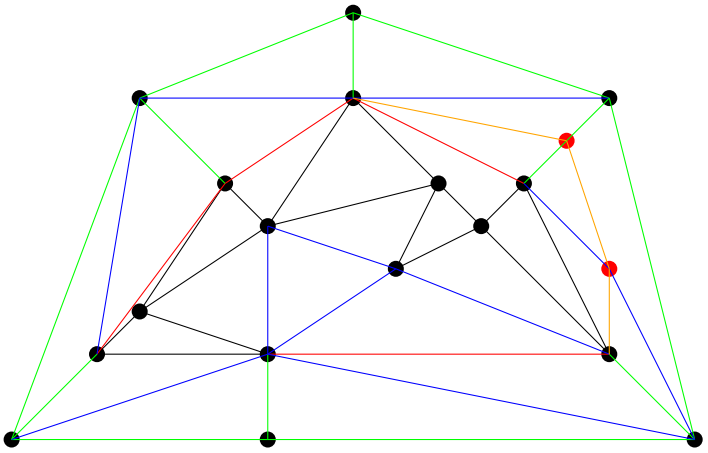
(a) Konvexe Hülle



(b) Die Konvexe Hülle wurde kopiert



(c) Der Graph ist nun trianguliert



(d) Eine Kante wird einfügt

Abbildung 3.4: Die Schritte beim Kanten einfügen

4. Evaluation des Frameworks

Das Framework wurde mit C++ und OGDF implementiert. OGDF stellt eine Repräsentation eines Graphen mit C++ Klassen dar und bietet viele Funktionen und Algorithmen auf Graphen an. Als IDE wurde Visual Studio 2012 Ultimate verwendet. Die Daten des Rechners:

- Windows 8.1 64 bit
- 12GB Arbeitsspeicher
- Prozessor: Intel(R) Core(TM) i5-3350P CPU @ 3.10GHz (4 CPUs), je 3.1GHz
- Microsoft Visual C++ Compiler

Es wurden für Schritt 2 (siehe 3.2) beide Algorithmen verwendet und sie werden im folgenden einzeln betrachtet. Für die Evaluierung des Framework werden die planaren Blöcke ohne Separator-Knoten der „rome-graphs“verwendet. Wobei i_n der i-te Block ist. Nicht planare Graphen wurden nicht betrachtet. Zuerst wird für die planare Graphen der ersten 24 Graphen wichtige Daten in Tabellen in 4.1 und 4.2 dargestellt. Dabei wurden bei 16 Graphen keine Kanten gelöscht. Diese werden im folgenden nicht weiter betrachtet. Es wurden keine Kanten gelöscht, weil der induzierte Baum nur einen Knoten enthält. Es konnte kein Weiterer zum Baum hinzugefügt werden, da, wegen Schleifen und Mehrfachkanten, bereits dadurch schon ein Kreis entsteht. Bei den Graphen wird am Anfang die größte Facette als äußere bestimmt.

n = Anzahl Knoten; m = Anzahl Kanten; x = Anzahl gelöschter Kanten; k:= Anzahl Knicke

Eine Laufzeit von 0.001 s ist kleiner gleich 0.001 s. Mit vorher ist der Graph gemeint, den man erhält, wenn man ihn nur zeichnet, das heißt keine Kanten temporär löscht. Die Fläche wird über die Koordinaten in YEd berechnet, wobei angenommen wird, das eine Koordinatendifferenz von eins einem Millimeter entspricht.

Danach werden 97 Graphen getestet und die Ergebnisse gemittelt. Dabei werden die Flächenänderung, die Laufzeit und die Anzahl Knicke betrachtet und die beiden Zeichenalgorithmen werden anhand dieser Kriterien verglichen.

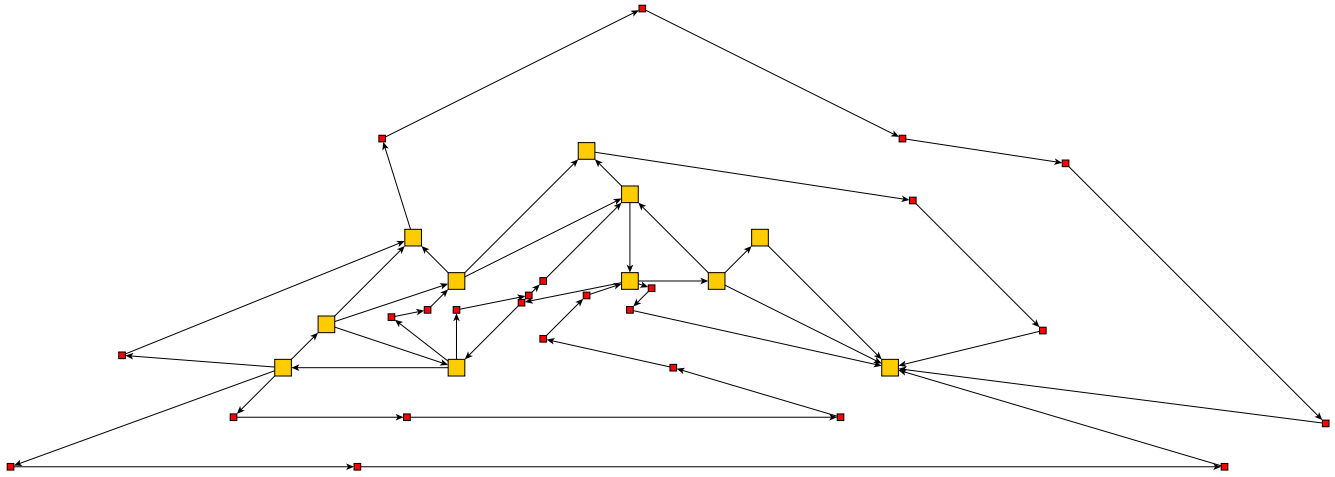


Abbildung 4.1: Ausgabe des Frameworks für den Einführungsgraphen mit PlanarStraightLayout

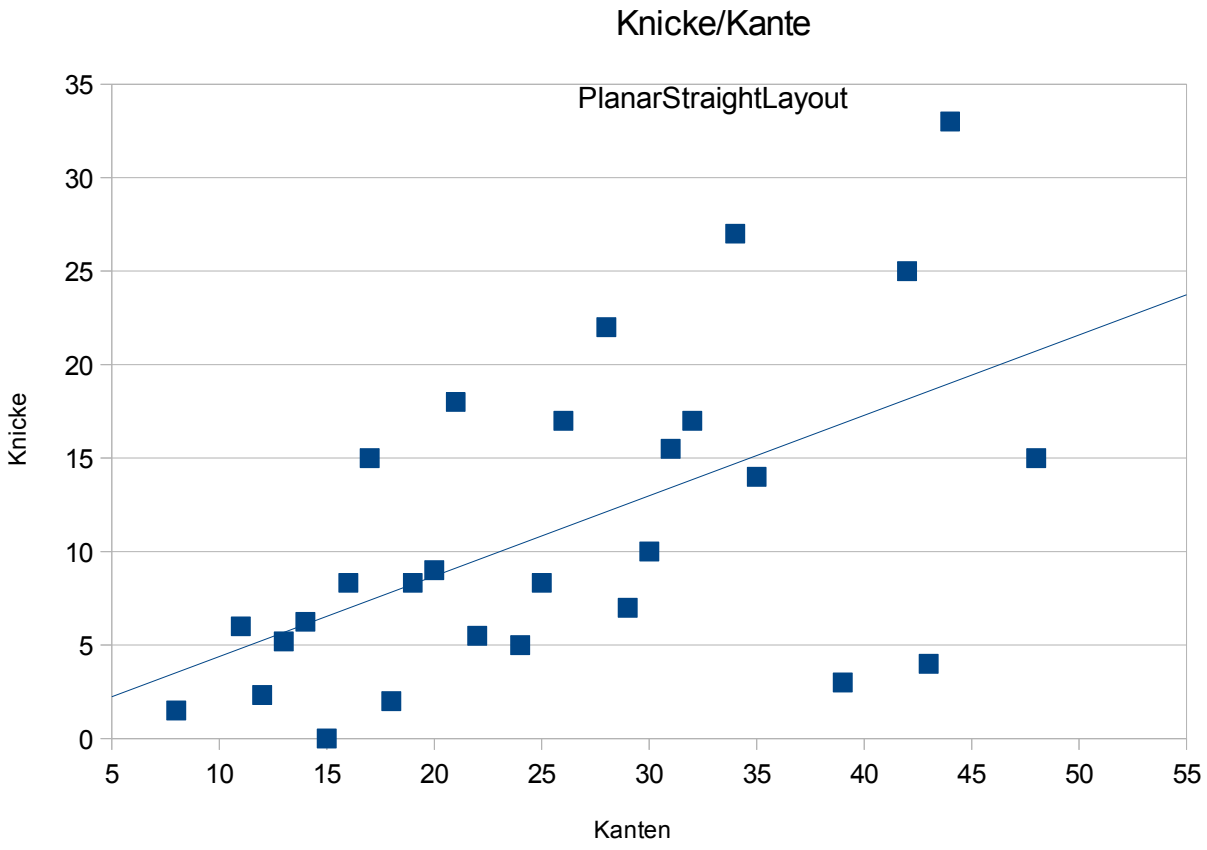
4.1 PlanarStraightLayout

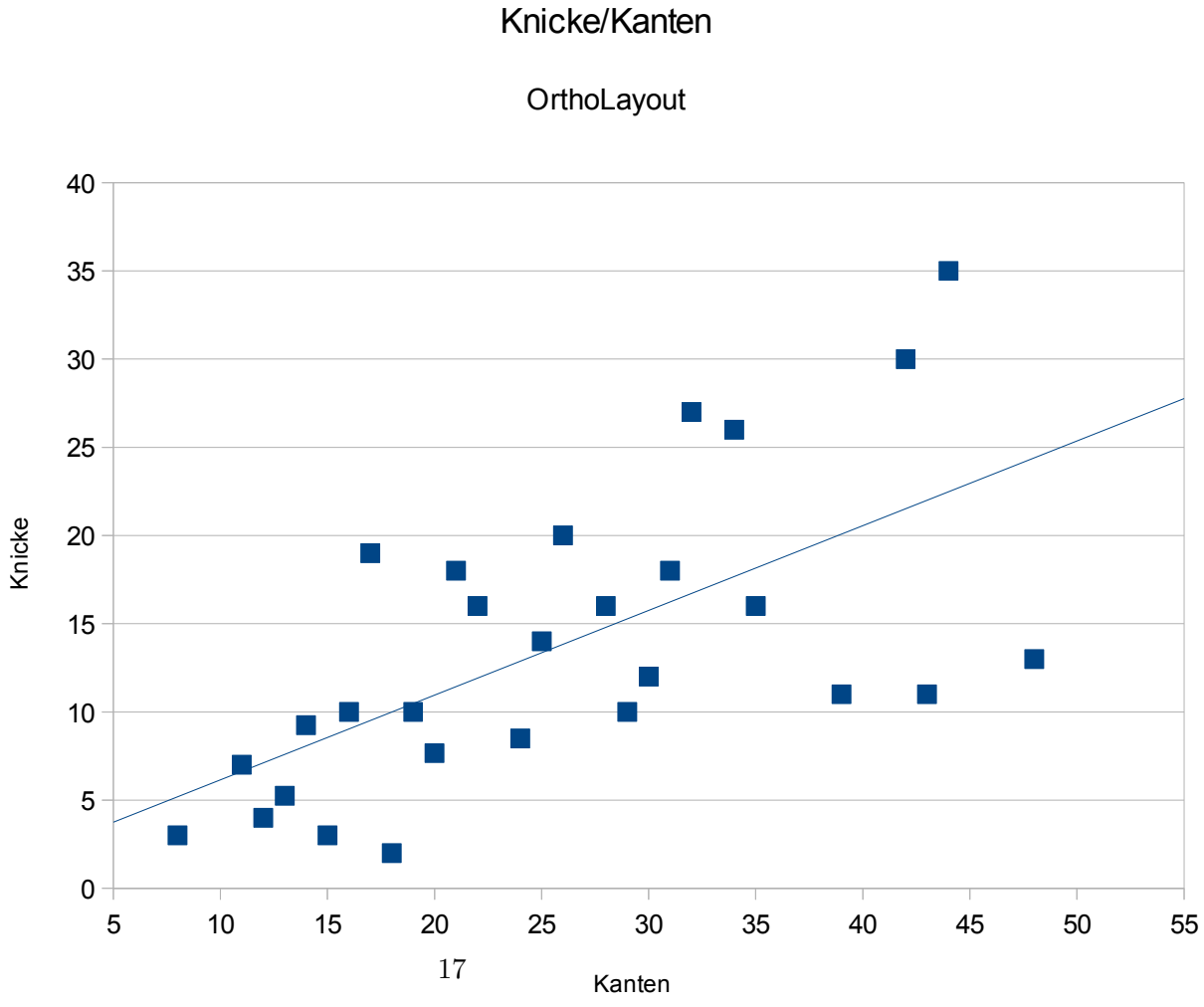
Name	n	m	k	Fläche vor	Fläche nach	Del. Zeit	Ins. Zeit	x	ext. Facette
g einführung	11	27	25	$540800mm^2$	$295750mm^2$	$0.001s$	$0.054s$	12	$3 \rightarrow 15$
grafo116.28i_1	20	29	5	$1622400mm^2$	$768950mm^2$	$0.001s$	$0.047s$	1	$10 \rightarrow 13$
grafo119.24i_0	18	27	7	$1521000mm^2$	$1394250mm^2$	$0.001s$	$0.047s$	3	$11 \rightarrow 16$
grafo123.21i_0	13	18	11	$540800mm^2$	$304200mm^2$	$0.001s$	$0.031s$	3	$7 \rightarrow 13$
grafo125.32i_1	16	20	3	$456300mm^2$	$338000mm^2$	$0.001s$	$0.063s$	2	$13 \rightarrow 17$
grafo130.31i_0	20	25	1	$608400mm^2$	$464750mm^2$	$0.001s$	$0.047s$	1	$13 \rightarrow 14$
grafo132.21i_0	14	20	0	$338000mm^2$	$23600mm^2$	$0.001s$	$0.046s$	2	$13 \rightarrow 15$
grafo135.30i_0	22	32	23	$2028000mm^2$	$1098500mm^2$	$0.001s$	$0.078s$	5	$10 \rightarrow 21$
grafo136.21i_0	15	20	1	$912600mm^2$	$650650mm^2$	$0.001s$	$0.047s$	2	$12 \rightarrow 15$

In Bild 4.1 sieht man die Ausgabe des Frameworks für den Einführungsgraphen mit PlanarStraightLayout. Dass die Dummy Knoten so weit entfernt sind liegt an dem Faktor der Streckung beim Kopieren der äußeren Facette und an der äquidistanten Platzierung der Knoten.

Von den 102 Graphen wurden bei 45 keine Kanten gelöscht. Für die restlichen 57 Graphen gilt:

- Es wurden durchschnittlich 2.16 Kanten gelöscht.
- Die durchschnittliche Flächendifferenz beträgt: $832547mm^2 - 840849mm^2 = -8302mm^2$, das heißt die Graphen wurden im Durchschnitt minimal größer.
- Pro Kante werden durchschnittlich 0.43 Knicke und eine Fläche von $30820mm^2$ benötigt.
- Die durchschnittliche Laufzeit beträgt $0.033s$.


Knicke/Kante
PlanarStraightLayout
Knicke
Kanten
0
5
10
15
20
25
30
35
5
10
15
20
25
30
35
40
45
50
55


Knicke/Kanten
OrthoLayout
Knicke
Kanten
0
5
10
15
20
25
30
35
40
5
10
15
20
25
30
35
40
45
50
55

4.2 Orthogonales Layout

Name	n	m	k (vorh.)	Fläche vor	Fläche nach	Del. Zeit	Ins. Zeit	x	ext. Facette
g einführung	11	27	57 (33)	$156310mm^2$	$32400mm^2$	$0.001s$	$0.152s$	12	$3 \rightarrow 15$
grafo116.28i_1	20	29	15 (12)	$103180mm^2$	$80608mm^2$	$0.001s$	$0.062s$	1	$10 \rightarrow 13$
grafo119.24i_0	18	27	13(12)	$89088mm^2$	$72275mm^2$	$0.001s$	$0.156s$	3	$11 \rightarrow 16$
grafo123.21i_0	13	18	12 (4)	$38415mm^2$	$39852mm^2$	$0.001s$	$0.016s$	3	$7 \rightarrow 13$
grafo125.32i_1	16	20	12 (1)	$48330mm^2$	$69750mm^2$	$0.001s$	$0.031s$	2	$13 \rightarrow 17$
grafo130.31i_0	20	25	8 (4)	$69165mm^2$	$66196mm^2$	$0.001s$	$0.047s$	1	$13 \rightarrow 14$
grafo132.21i_0	14	20	11 (7)	$51338mm^2$	$37074mm^2$	$0.001s$	$0.047s$	2	$13 \rightarrow 15$
grafo135.30i_0	22	32	19 (9)	$107168mm^2$	$78548mm^2$	$0.001s$	$0.063s$	5	$10 \rightarrow 21$
grafo136.21i_0	15	20	5 (4)	$36135mm^2$	$44370mm^2$	$0.001s$	$0.031s$	2	$12 \rightarrow 15$

In Bild 4.2 sieht man die Ausgabe des Frameworks für den Einführungsgraphen mit Orthogonalem Layout. Man stellt fest, dass die Dummy Knoten eng nebeneinander sind, da viele Kanten gelöscht wurden. Dies liegt daran, dass kurze Triangulierungskanten häufig gekreuzt werden und manche davon auch eng nebeneinander liegen.

Von den 102 Graphen wurden bei 45 keine Kanten gelöscht. Für die restlichen 57 Graphen gilt:

- Es wurden durchschnittlich 2.16 Kanten gelöscht.
- Die durchschnittliche Flächendifferenz beträgt: $69643mm^2 - 64850mm^2 = 4793mm^2$, das heißt die Graphen wurden im Durchschnitt kleiner.
- Pro Kante werden durchschnittlich 0.54 Knicke und eine Fläche von $2433mm^2$ benötigt.
- Die durchschnittliche Laufzeit beträgt $0.041s$.

4.3 Zusammenfassung

Die getesteten Graphen waren nicht auf das Problem zugeschnitten, das heißt die äußere Facette war meist nicht besonders klein. Deshalb ist die Fläche selten stark verkleinert und manchmal sogar vergrößert worden. Man sieht in dem von mir gewählten Graphen, dass die Fläche auch stark verkleinert werden kann. Würde man die Dummy-Knoten bei der Berechnung der Fläche berücksichtigen, wäre diese noch größer. Dies kann man verhindern, indem man sie besser positioniert (siehe Vorschlag in 3.3). Dies ist aber nicht so wichtig, da der Platzverbrauch von Kanten einfach reduziert werden kann und für die Lesbarkeit nicht so wichtig ist. Es hängt auch viel vom verwendeten Zeichenalgorithmus ab, weshalb ich diese im folgenden vergleichen werde.

4.3.1 Vergleich der Zeichenalgorithmen

Bei PlanarStraightLayout (PSL) wurde die Fläche nicht kleiner, sogar um ca. $\frac{1}{100}$ größer. Wohingegen die Fläche bei dem orthogonale Layout $\frac{1}{10}$ kleiner wurde. Dafür hat das Orthogonale Layout die größere Laufzeit und benötigt mehr Knicke. PSL hat somit weniger Knicke und kürzere Laufzeit, was auf Kosten der Fläche geht. Welches man wählt hängt somit davon ab, welches Kriterium wichtiger ist und wie groß der Graph ist.

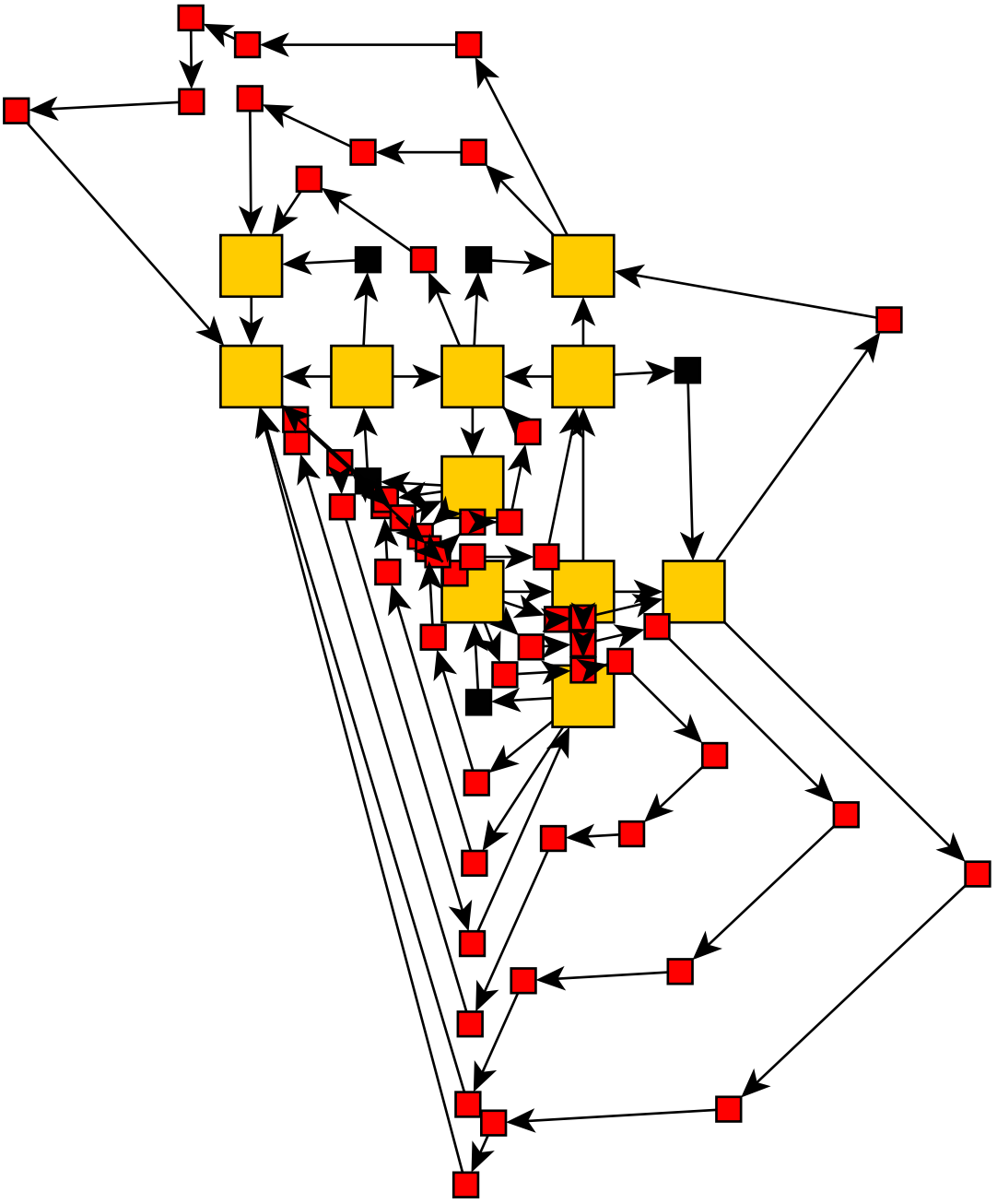


Abbildung 4.2: Ausgabe des Frameworks für den Einführungsgraphen mit Orthogonalem Layout

Literaturverzeichnis

[Bis13] Simon Bischof. Induzierte Bäume in planaren Graphen. Master's thesis, KIT, 2013. Bachelorarbeit.

[EET93] Hossam ElGindy, Hazel Everett, and Godfried Toussaint. Slicing an ear using prune-and-search. *Pattern Recognition Letters*, 14(9):719 – 722, 1993. Computational Geometry.

[ESS86] Paul Erdös, Michael Saks, and Vera T Sós. Maximum induced trees in graphs. *Journal of Combinatorial Theory, Series B*, 41(1):61 – 79, 1986.

[GJ82] M. R. Garey and D. S. Johnson. Crossing number is np-complete. *SIAM Journal on Algebraic Discrete Methods*, 4(3), 1982.

[GM04] Carsten Gutwenger and Petra Mutzel. Graph embedding with minimum depth and maximum external face. In Giuseppe Liotta, editor, *Graph Drawing*, volume 2912 of *Lecture Notes in Computer Science*, pages 259–272. Springer Berlin Heidelberg, 2004.

[GMW01] Carsten Gutwenger, Petra Mutzel, and René Weiskircher. Inserting an Edge into a Planar Graph. In *Proceedings of the 12th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'01)*, pages 246–255, 2001.

[Pur97] Helen Purchase. Which aesthetic has the greatest effect on human understanding? In Giuseppe DiBattista, editor, *Graph Drawing*, volume 1353 of *Lecture Notes in Computer Science*, pages 248–261. Springer Berlin Heidelberg, 1997.