Christian-Albrechts-Universität zu Kiel

Bachelorarbeit

Planarisierung von Hypergraphen

Christian Kutschmar

29. September 2010

Institut für Informatik
Arbeitsgruppe Echtzeitsysteme und Eingebettete Systeme

Prof. Dr. Reinhard von Hanxleden

betreut durch:
Dipl.-Inf. Miro Spönemann

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

Zusammenfassung

Planarität eines Graphen ist die Grundlage für viele Layoutverfahren. In dieser Arbeit wird ein Verfahren vorgestellt, welches es ermöglicht, solch eine planare Form eines Graphen zu erstellen. Dafür werden erst Voraussetzungen wie Planaritätstests und anschließend das eigentliche Verfahren der Planarisierung vorgestellt. Nachdem das grundlegende Verfahren für normale Graphen erläutert wurde, wird dies so erweitert, dass es auch auf Hypergraphen anwendbar ist. Es wird weiterhin aufgezeigt, dass die Planarisierung nicht nur Selbstzweck ist, sondern vielmehr Grundlage anderer Verfahren. Danach werden offene Fragestellungen erörtert und anschließend ein Einblick in die Implementierung gegeben.

Inhaltsverzeichnis

1 Einführung **1**

2 Graphentheoretische Grundlagen **5**
2.1 Graphen . . . 5
2.2 Hypergraphen . . . 6
2.3 Einbettung . . . 7
2.4 Faces . . . 8
2.5 Dualer Graph . . . 10
2.6 Planarität . . . 10
2.6.1 Definition der Planarität . . . 10
2.6.2 Übersicht Planaritätstests . . . 12

3 Verwendete Technolgien und verwandte Arbeiten **15**
3.1 Verwendete Algorithmen . . . 15
3.1.1 Die Breitensuche (BFS) . . . 15
3.1.2 Der Dijkstra-Algorithmus . . . 16
3.2 Verwendete Technologien . . . 17
3.2.1 Eclipse . . . 17
3.2.2 KIELER . . . 17
3.3 Verwandte Arbeiten . . . 17

4 Vorstellung der Verfahren **21**
4.1 Einfügen von Kanten in planare Graphen . . . 21
4.1.1 Ermitteln von Faces und dem dualen Graphen . . . 21
4.1.2 Suche mit Breitensuche . . . 24
4.1.3 Suche mit dem Dijkstra-Algorithmus . . . 24
4.1.4 Das Einfügen . . . 25
4.2 Einfügen von Kanten in planare Hypergraphen . . . 27
4.2.1 Die Point-based Form herstellen . . . 28
4.2.2 Den dualen Graphen erweitern . . . 28
4.2.3 Das Einfügen . . . 29
4.3 Änderung der Graphstruktur . . . 31

5 Implementierung **33**
5.1 Datenstruktur . . . 33
5.1.1 Einbettung . . . 35
5.1.2 Wahl des Suchalgorithmus . . . 35

5.1.3 Testen der Implementierung 35

6 Zusammenfassung und Ausblick **37**

Literaturverzeichnis **39**

Abbildungsverzeichnis

1.1 Ergebnis einer Datenbankabfrage 2
1.2 Die gleiche Abfrage neu gelayoutet 2
1.3 Ein Graph mit einer Kantenkreuzung und dessen planarisierte Form 3

2.1 Ein Hypergraph H in verschiedenen Darstellungsformen 7
2.2 Grafische Veranschaulichung einer Einbettung 7
2.3 Zwei verschiedenen Einbettungen eines Graphen 8
2.4 Ein Graph G mit seinen Faces a, b und c 8
2.5 Unterschiedliche Faces bei unterschiedlichen Einbettungen eines Graphen G 9
2.6 Ein Graph G und sein dualer Graph G' 10
2.7 Diverse Graphen 11
2.8 Das Gas-Wasser-Strom Problem 11
2.9 Die kleinsten nicht planaren Graphen 12

4.1 Finden eines linken und eines rechten Face 21
4.2 Ermitteln der allgemeinen und relevanten Faces 23
4.3 Pfad im dualen Graphen 26
4.4 Einfügen des neuen Knotens und der neuen Kante 27
4.5 Faces a bis f, welche am Hyperknoten von Hypergraph H liegen, werden im dualen Graphen zu einem vollständigen Teilgraphen . . . 29
4.6 Einfügen einer neuen Kante 30
4.7 Änderung einer Hyperkante 32

5.1 Die Interfaces der PGraph Struktur 33
5.2 Die Datenstruktur 34

Verzeichnis der Auflistungen

3.1 Breitensuche 15
3.2 Dijkstra-Algorithmus 16

4.1 Finden des linken Face 22
4.2 Erstellen des dualen Graphen 23
4.3 Ermitteln des optimalen Pfades mit BFS 24
4.4 Ermitteln des optimalen Pfades mit dem Dikstra-Algorithmus 24
4.5 Einfügen eines Knotens auf einer Kante 26
4.6 Einbettung aktualisieren 26
4.7 Einfügen der neuen Kanten mit Hilfe neuer Knoten 27
4.8 Verschmelzen der Hyperknoten 28
4.9 Erweitern des dualen Graphen 29
4.10 Aufspalten eines Hyperknotens 30

1 Einführung

Bei vielen Anwendungstypen werden Zusammenhänge in natürlicher Weise als Graphen modelliert. In mathematischer Sichtweise sind Objekte als Knoten und deren Verbindungen als Kanten eines Graphen zu sehen. So lassen sich sowohl Analysen durchführen, als auch die Grundlage für die bildliche Darstellung legen. Beispiele solcher Darstellungen sind Schaltpläne, Zustandsdiagramme oder die Darstellung von Abhängigkeiten in Datenbanken für Anwendungsentwickler.

Diese Diagramme sind in der Regel zweidimensional und können sehr schnell sehr groß werden, so dass ein Layout von Hand zu aufwendig ist. In solchen Fälle ist ein Softwaretool hilfreich, welches das Layouten übernimmt. Dabei führen Kantenkreuzungen oft zu einem schlecht überschaubaren Ergebnis und die Frage ist nun, ob sich so ein Diagramm auch ohne Kantenkreuzungen zeichnen lässt.
In dieser Arbeit wird ein Verfahren vorgestellt, das es ermöglicht, eben solche Kantenkreuzungen aus Graphen und Hypergraphen zu entfernen. Dadurch entsteht eine gut lesbares bzw. verständliche Version der Zeichnung, wobei gut lesbar natürlich immer ein subjektives Kriterium ist.

Ein Beispiel wird aufzeigen, wie unterschiedlich ein Diagramm mit verschiedenen Layouts aussehen kann und wie dies das Verständnis beeinflusst. Die folgenden Abbildungen 1.1 und 1.2 wurden aus der Arbeit *Zeichnen von Diagrammen - Theorie und Praxis* von Petra Mutzel entnommen [19]. Abbildung 1.1 zeigt das grafisch aufbereitete Ergebnis einer Datenbankabfrage. Dieses ist sehr unübersichtlich und erschwert das Verständnis enorm. In Abbildung 1.2 wurde diese Grafik nach einem kräfte-basierten Verfahren gelayoutet. Das Ergebnis unterscheidet sich schon auf den ersten Blick deutlich von der Vorgabe und ermöglicht es Zusammenhänge schnell nachzuvollziehen.

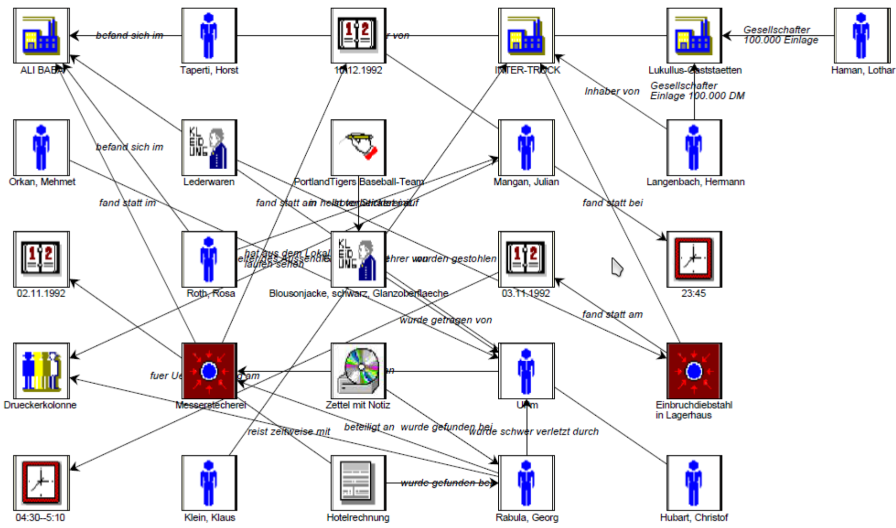


Abbildung 1.1: Ergebnis einer Datenbankabfrage

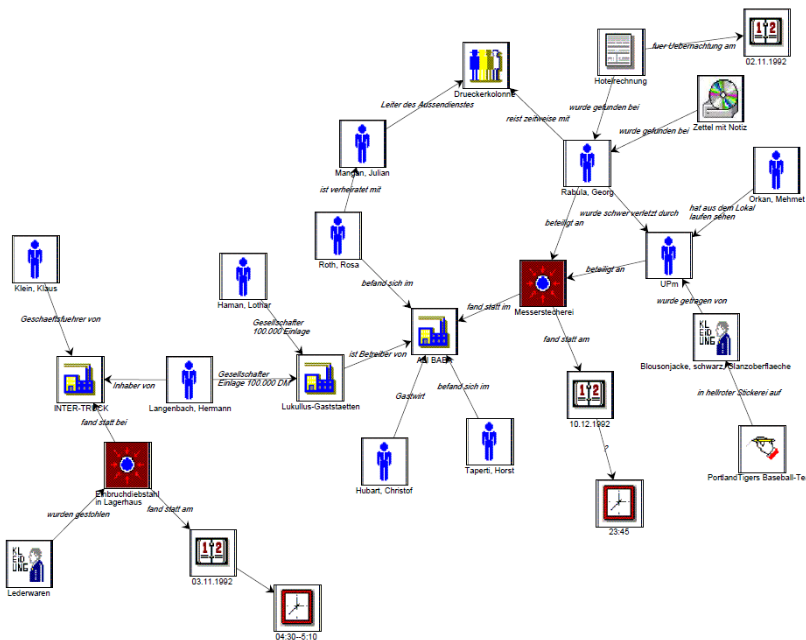


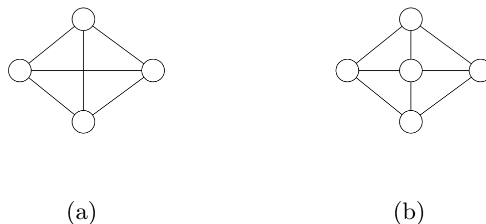
Abbildung 1.2: Die gleiche Abfrage neu gelayoutet

In ihrer Arbeit “Algorithmen zum automatischen Zeichnen von Graphen” [2] geben Brandenburg, Jünger und Mutzel eine Erklärung zum Thema Layout und Ästhetik:

> „Ästhetisch schöne Layouts zeichnen sich durch Kompaktheit, eine hohe Uniformität der Kantenlängen und gleichmäßige Winkel aus. Die Hervorhebung von Symmetrien ist bei einigen Anwendungen wünschenswert. Diese Optimalitätskriterien sollen unter Einhaltung von vorgegebenen Restriktionen, z. B. rechtwinkliges oder kreuzungsfreies Zeichnen, erreicht werden. Welches Ästhetikkriterium passend ist, hängt von der Anwendung und dem gewohnten Kontext ab.“

Der Schwerpunkt dieser Arbeit ist das im Zitat genannte kreuzungsfreie Zeichnen von Graphen, denn dies kann weiterhin Grundlage für weitere genannte Kriterien wie gleichmäßige Winkel sein. Außerdem ist eines der wichtigsten Kriterien zur Übersichtlichkeit von Diagrammen, die Anzahl der Kantenkreuzungen [20]. Ziel dieser Arbeit ist, ein Verfahren vorzustellen, welches erlaubt, sowohl Graphen als auch Hypergraphen ohne Überkreuzungen von Kanten zu zeichnen.

Die Grundidee des vorgestellten Verfahrens ist, dass auf solchen Kantenkreuzungen, die nicht vermieden werden können, Knoten einfügt werden. Dadurch entstehen zwar neue Knoten, allerdings werden Kantenkreuzungen komplett entfernt. Anschaulich ist dies in Abbildung 1.3 dargestellt.



(a) (b)

Abbildung 1.3: Ein Graph mit einer Kantenkreuzung (a) und dessen planarisierte Form (b)

Die Arbeit ist wie folgt aufgebaut: in Kapitel 2 werden die zum Verständnis notwendigen graphentheoretischen Grundlagen vorgestellt und erklärt. Im darauf folgenden Kapitel 3 werden verwendete Technologien und ein Ausschnitt von Arbeiten, die sich ebenfalls mit diesem Thema befassen, präsentiert. Nachdem diese Thematik behandelt wurde, befasst sich Kapitel 4 mit den eigentlichen Verfahren auf theoretischer Basis. Es teilt sich in zwei große Abschnitte. Der Erste davon befasst sich mit dem Fall, dass ein normaler Graph vorliegt, der Zweite mit Hypergraphen. Implementierungsdetails sind in Kapitel 5 zu finden, bevor in Kapitel 6 eine abschließende Zusammenfassung sowie ein Ausblick gegeben wird.

2 Graphentheoretische Grundlagen

In diesem Kapitel werden die graphentheoretischen Grundlagen und Definitionen vorgestellt, welche zum Verständnis der weiteren Arbeit notwendig sind, beginnend mit der Definition, wie Graphen zu verstehen sind, bis hin zu Planariätstests.

2.1 Graphen

Ein *Graph* $G = (V, E)$ besteht aus einer Menge von Knoten V und einer Menge von Kanten E, die jeweils ein Paar von Knoten repräsentieren. Sei z.B. $e = (v, w)$ die Kante, welche den Knoten v mit dem Knoten w verbindet, wobei $v, w \in V$. Wenn solch eine Kante existiert, dann ist v *adjazent* zu w. Eine Kante $e = (v, w)$ heißt *inzident* mit ihren beiden Endknoten u und v.
Es wird weiterhin zwischen gerichteten und ungerichteten Graphen unterschieden. Bei einem ungerichteten Graphen wird im Allgemeinen eine Kante e in der Form $e = \{v, w\}$ dargestellt. In gerichteten Graphen hat eine Kante einen Startknoten und einen Zielknoten. Die Schreibweise $e = (v, w)$ beschreibt eine Kante vom Knoten v nach w, während $e = (w, v)$ eine Kante von w nach v beschreibt. Da in dieser Arbeit nur ungerichtete Graphen betrachtet werden, wird zum Zweck der besseren Lesbarkeit die Schreibweise mit runden Klammern auch für ungerichtete Kanten verwendet.

Sei $G = (V, E)$ ein ungerichteter Graph und $W = (v_1, \ldots, v_n)$ eine Folge von Knoten aus V.

Ein *Weg* ist eine Folge von Knoten, mit der Eigenschaft, dass für alle i aus $\{1, \ldots, n-1\}$ gilt: Die Menge $\{vi, v_{i+1}\}$ ist Element von E.

Ein Weg in G wird *Pfad* genannt, falls alle Knoten in der Folge W voneinander verschieden sind, das heißt falls für alle i und j aus $\{1, \ldots, n\}$ gilt, dass $v_i \neq v_j$, falls $i \neq j$.

Ein Weg in G wird *Zykel* genannt, falls Start- und Endknoten von W identisch sind, das heißt, falls $v_1 = v_n$.

Ein Graph $G = (V, E)$ heißt *zusammenhängend*, wenn von jedem Knoten $v \in V$ ein Pfad zu jedem Knoten $w \in V$ existiert, wobei $v \neq w$.

Ein Graph $G = (V, E)$ heißt *vollständig*, wenn von jedem Knoten $v \in V$ eine Kante zu jedem Knoten $w \in V$ existiert, mit $w \neq v$. Für einen vollständigen Graphen mit

n Knoten wird die Schreibweise K_n verwendet. Die Schreibweise $K_{n,m}$ bezeichnet einen Graphen mit zwei disjunkten Knotenmengen, U mit m Knoten und V mit n Knoten, in dem jeder Knoten aus U mit jedem Knoten aus V verbunden ist.

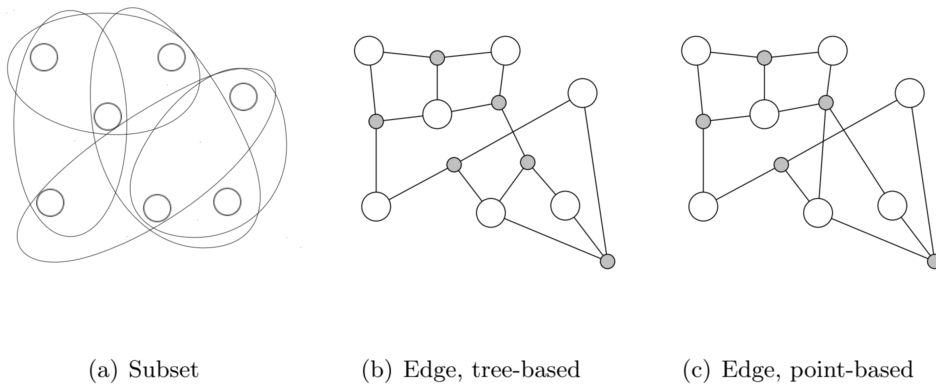
Ein Graph $G_1 = (V_1, E_1)$ heißt *Teilgraph* oder auch *Subgraph* eines Graphen $G_2 = (V_2, E_2)$, falls $V_1 \subseteq V_2$ und $E_1 \subseteq E_2$.

Ein Graph G_2 heißt *Unterteilung* eines Graphen G_1, wenn beliebig oft (auch null mal) Knoten auf Kanten von G_1 eingefügt werden. Diese Knoten dürfen nicht auf Kreuzungspunkten eingefügt werden.

Ein Graph heißt *Baum*, wenn er zusammenhängend ist und keine Zykel enthält.

2.2 Hypergraphen

Ein *Hypergraph* $H = (V, F)$ unterscheidet sich von einem normalen Graphen dahingehend, dass eine Kante mehr als zwei Knoten verbinden kann. Diese Kanten nennt man Hyperkanten. Eine Hyperkante $f \in F$ ist eine Teilmenge der Knotenmenge V, wobei $|f| \geq 2$. Nach Mäkingen [17] gibt es zwei Varianten zur Darstellung von Hyperkanten: Den *subset standard* und den *edge standard* (Abbildung 2.2). Die erste Variante wird schnell sehr unübersichtlich und es ist nicht einfach, klare Kantenkreuzungen zu definieren. Aus diesem Grund benutzen die meisten Anwendungen den *edge standard* [21, 10], der zwei Untervarianten erlaubt. Im *tree-based* Zeichenstil wird jede Hyperkante F mit einer baumartigen Linienstruktur dargestellt, deren Blätter jeweils die inzidenten Knoten von F sind. Wenn die baumartige Struktur nun als sternförmige Struktur mit jeweils nur einem Hyperknoten pro Hyperkante dargestellt wird, entsteht der *point-based* Zeichenstil. Bei Interesse wird der Artikel *Hypergraph planarity and the complexity of drawing venn diagrams* von D. S. Johnson und H. O. Pollak empfohlen [8].



(a) Subset (b) Edge, tree-based (c) Edge, point-based

Abbildung 2.1: Ein Hypergraph H in verschiedenen Darstellungsformen

2.3 Einbettung

Eine *Einbettung* eines Graphen beschreibt die Reihenfolge, in der die inzidenten Kanten an einem Knoten liegen. Solange in Abbildung 2.2 jede Kante in Richtung des grünen Pfeils die gleiche Nachfolgekante hat, handelt es sich um die gleiche Einbettung eines Knotens.

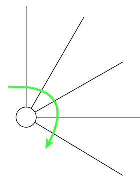


Abbildung 2.2: Grafische Veranschaulichung einer Einbettung

Ein Graph kann mehrere Einbettungen haben, die in Bezug auf Planarität verschiedene Ergebnisse liefern. Sei G solch ein Graph mit $V = \{1, 2, 3, 4, 5, 6\}$ und $E = \{(1, 2), (1, 3), (1, 5), (1, 6), (2, 4), (3, 4), (3, 5), (3, 6)\}$. Abbildung 2.3 zeigt zwei unterschiedliche Einbettungen des Graphen G. Auf den ersten Blick sieht dies nach zwei unterschiedlichen Graphen aus, doch alle Knoten- und Kanteninformationen sind identisch.

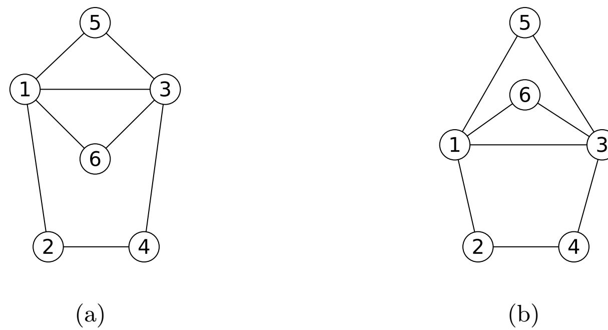


Abbildung 2.3: Zwei verschiedenen Einbettungen eines Graphen

2.4 Faces

Ein *Face* F ist eine Fläche in der planaren Zeichnung eines Graphen G, welche durch die anliegenden Kanten definiert ist. Anschaulich wird dies in Abbildung 2.4 verdeutlicht. Im Folgenden werden in Abbildungen Faces immer durch ein weißes Sechseck dargestellt.

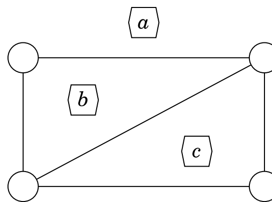


Abbildung 2.4: Ein Graph G mit seinen Faces a, b und c

Eulerscher Polyedersatz. *Seien E die Anzahl der Ecken, F die Anzahl der Flächen und K die Anzahl der Kanten eines beschränkten, konvexen Polyeders, dann gilt:* $E + F - K = 2$

Beweis:
Dieser Beweis zeigt mit struktureller Induktion die Gültigkeit des Satzes für planare Graphen. Der einfachste planare Graph besteht nur aus einem Knoten. Es gibt eine Face und keine Kanten. Es gilt also $E+F-K = 1+1-0 = 2$. Aus diesem einfachsten Graphen können alle weiteren ausschließlich durch die beiden folgenden Operationen konstruiert werden, welche die Gültigkeit des Satzes nicht verändern:

1. Hinzufügen eines Knotens, der über eine neue Kante mit dem Rest des Graphen verbunden ist. Die Anzahl der Knoten und Kanten steigt jeweils um

eins, während die Anzahl der Faces gleich bleibt. Galt für den alten Graphen $E + F - K = 2$, so gilt es auch für den neuen, da auf der linken Seite der Gleichung je eine Eins addiert und abgezogen wurde.

2. Hinzufügen einer Kante, die zwei bereits bestehende Knoten verbindet. Während die Anzahl der Knoten gleich bleibt, steigt die Anzahl der Faces und Kanten jeweils um eins. Wieder bleibt die Summe $E + F - K$ gleich, da je eine Eins addiert und abgezogen wurde.

Da der Satz für den ersten, einfachsten Graphen galt, muss er also auch für jeden Graphen gelten, der durch eine der beiden Operationen aus diesem entsteht. Jeder Graph, der durch eine weitere Operation aus einem solchen Graphen entsteht, muss den Satz ebenfalls erfüllen. Daher gilt der Satz für alle planaren Graphen und damit auch für alle konvexen Polyeder. □

Mit dem Eulerschen Polyedersatz ergibt sich für den Graphen aus Abbildung 2.4 eine Faceanzahl von 3, da $|E| = 5$ und $|V| = 4$ und $5 - 4 + 2 = 3$. Unterschiedliche Einbettungen eines Graphen führen auch zu unterschiedlichen Faces. Dies wird deutlich sichtbar, sobald beim Graphen aus Abbildung 2.3 die Faces eingezeichnet werden. Siehe dazu Abbildung 2.5.

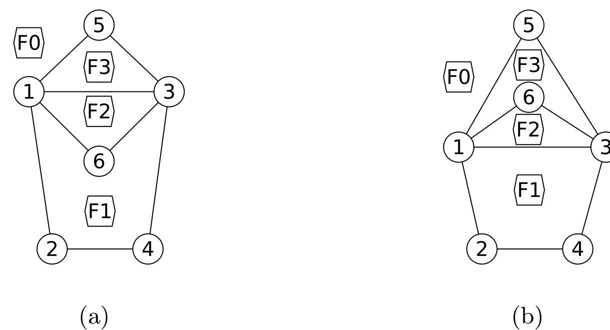


Abbildung 2.5: Unterschiedliche Faces bei unterschiedlichen Einbettungen eines Graphen G

In folgender Tabelle sind die zu den jeweiligen Einbettungen gehörenden Faces aufgelistet. Dabei bedeutet $F_0 = \{1, 5, 4, 3, 2\}$, dass Face 0 die Knoten 1, 2, 3, 4, 5 berührt.

Einbettung a	Einbettung b
$F_0 = \{1, 5, 4, 3, 2\}$	$F_0 = \{1, 5, 4, 3, 2\}$
$F_1 = \{1, 2, 3, 4, 6\}$	$F_1 = \{1, 2, 3, 4\}$
$F_2 = \{1, 6, 4\}$	$F_2 = \{1, 4, 6\}$
$F_3 = \{1, 4, 5\}$	$F_3 = \{1, 6, 4, 5\}$

2.5 Dualer Graph

Der duale Graph G' ergibt sich, indem für jede Face (siehe Abschnitt 2.4) aus dem Originalgraphen G ein Knoten erzeugt wird. Anschließend wird für jede Kante aus G eine neue Kante in G' erzeugt, welche die dualen Knoten verbinden. Ein dualer Graph hat die Eigenschaft, dass er zusammenhängend ist, d.h. es existiert von jedem Knoten ein Pfad zu jedem anderen Knoten. In Abbildung 2.6 wird veranschaulicht, wie zu einem gegebenen Graphen G der duale Graph G' aussieht.

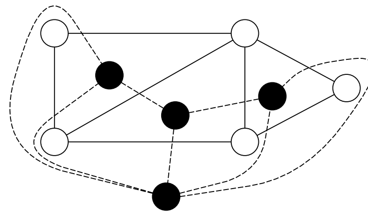


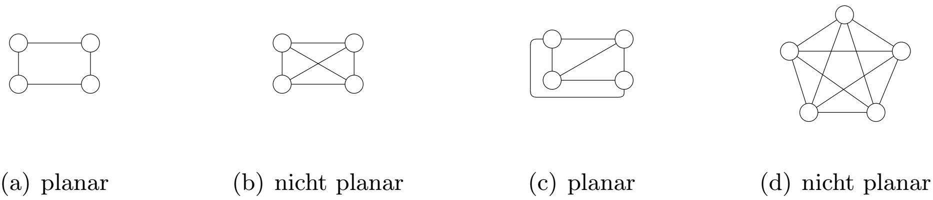
Abbildung 2.6: Ein Graph G und sein dualer Graph G' (schwarze Knoten mit gestrichelten Kanten)

Anmerkung: Da unterschiedliche Einbettungen eines Graphen G zu unterschiedlichen Faces führen, ergeben sie auch unterschiedliche duale Graphen.

2.6 Planarität

2.6.1 Definition der Planarität

Ein Graph $G = (V, E)$ ist genau dann *planar*, wenn eine Einbettung (siehe Abschnitt 2.3) existiert, so dass sich der Graph ohne Überkreuzungen von Kanten in einer Ebene zeichnen lässt. Abbildung 2.7(a) ist demnach ein Beispiel für eine planare Zeichnung eines Graphen. Es ist ersichtlich, dass sich keine Kanten schneiden. Der Graph in Abbildung 2.7(b) entspricht ebenfalls den Forderungen der Definition, auch wenn es nicht den Anschein hat. In der Zeichnung dieses Graphen schneiden sich zwar zwei Kanten, er lässt sich jedoch so zeichnen dass es keine Kantenkreuzungen gibt. Demnach existiert eine *planare Einbettung*. Dies zeigt Abbildung 2.7(c). Für den Graphen aus Abbildung 2.7(d) lässt sich jedoch keine planare Einbettung finden. In jeder Zeichnung dieses Graphen ist mindestens eine Kantenkreuzung vorhanden.



(a) planar (b) nicht planar (c) planar (d) nicht planar

Abbildung 2.7: Diverse Graphen

Die kleinsten nicht planaren Graphen sind der K_5 und der $K_{3,3}$ [16]. Der $K_{3,3}$ modelliert das Wasser-Gas-Strom Problem, d.h. es gibt drei Quellen und drei Häuser (zu sehen in Abbildung 2.8. Die Problemstellung besteht darin Leitungen von allen Anbietern zu allen Häusern zu legen, ohne dass sich zwei Leitungen kreuzen. Für dieses Problem existiert keine planare Lösung.

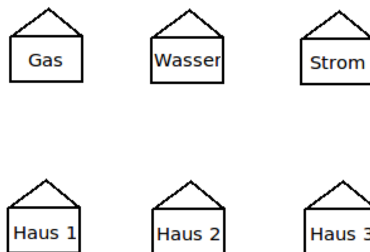
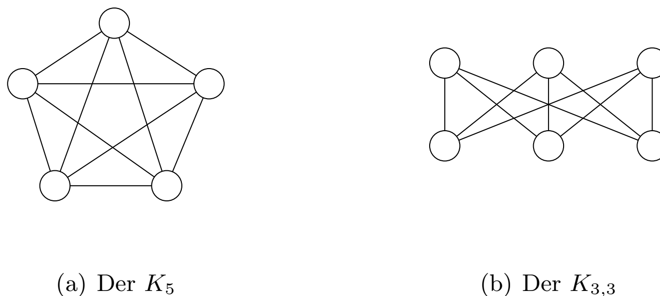


Abbildung 2.8: Das Gas-Wasser-Strom Problem

Satz von Kuratowski. *Ein Graph ist genau dann planar, wenn er keinen Teilgraphen enthält, der durch Unterteilung von K_5 oder $K_{3,3}$ entstanden ist [16].*



(a) Der K_5 (b) Der $K_{3,3}$

Abbildung 2.9: Die kleinsten nicht planaren Graphen

Dieses Kriterium ist zwar leicht verständlich, jedoch ist ein Algorithmus allerdings, welcher dieses Kriterium für den Planaritätstest und das Finden einer planaren Einbettung verwendet, sehr schwer zu implementieren.

2.6.2 Übersicht Planaritätstests

Ziel eines Planaritätstests ist es, einen gegebenen Graphen $G = (V, E)$ automatisiert auf Planarität zu testen und, falls G planar ist, eine planare Einbettung zu liefern. Es gibt weiterhin Planaritätstests oder Erweiterungen bestehender Planaritätstests [18] welche, wenn G nicht planar ist, einen planaren Subgraphen SG ermitteln. Um Graphen auf Planarität zu testen reicht es aus, ungerichtete und ungewichtete Graphen zu betrachten, da Richtung und Gewicht von Kanten bei der Kreuzungsfreiheit keine Rolle spielen.

Der Algorithmus von Hopcroft und Tarjan

Der erste Planaritätstest mit linearer Laufzeit, abhängig von der Anzahl der Knoten eines Graphen, also $O(V)$, wurde 1974 von John Hopcroft und Robert Tarjan entwickelt [14]. Dieser Algorithmus teilt sich in zwei Schritte ein. Im ersten Schritt wird mit Hilfe von Tiefensuche ein Gerüst von Kanten gebildet und die Kanten innerhalb dieses Gerüsts orientiert. Im zweiten Schritt werden die restlichen Kanten orientiert.

Der Boyer-Myrvold Alogrithmus

Dieser Algorithmus wurde 2004 von John Boyer und Wendy Myrvold entwickelt [1]. Er erzeugt mittels eines Tiefensuchdurchlaufs einen Teilgraphen, der aufgrund seiner Baumstruktur garantiert planar ist. Die dabei durchlaufenen Kanten werden separat als zweifach zusammenhängende Komponenten in eine spezielle Datenstruktur eingebettet. Dies ist notwendig, da der Algorithmus im Folgenden auf zweifach zusammenhängenden Komponenten, sogenannten Bicomps (*biconnected components*), operiert. Nach dieser Vorbereitung des Graphen liegt ein Tiefensuchbaum vor, dessen Baumkanten in (Single-)Bicomps eingebettet sind. Der Algorithmus beginnt nun mit der Hauptschleife. Dort, sofern dies möglich ist, werden in einer durch feste Regeln definierten Reihenfolge alle Rückwärtskanten der Tiefensuche planar eingebettet. Dazu hält der Algorithmus zu jeder Zeit die Invariante aufrecht, Knoten, die an späteren Einbettungen beteiligt sind, auf dem äußeren Gebiet zu halten. Dazu ist es notwendig, vor der Verschmelzung die Bicomps gegebenenfalls zu drehen. Sollte eine Kanteneinbettung fehlschlagen, folgt aus dem Vorgehen des Algorithmus, dass der Eingangsgraph nicht planar ist. Verlaufen alle Kanteneinbettungen erfolgreich, so ergibt sich eine planare Einbettung des Graphen. In beiden Fällen ergibt sich also die Lösung des Erkennungsproblems. Für eine ausführliche Beschreibung wird die Bachelorarbeit *Implementing an Algorithm for an Orthogonal Graph Layout* von Ole Claussen empfohlen, welche sich unter anderem mit diesem Test befasst [6].

Der Links-Rechts-Planaritätstest

Das Links-Rechts-Planaritätskriterium wurde von Hubert de Fraysseix, Pierre Rosenstiehl und Patrice Ossona Mendez entworfen [15]. Es liefert einen Algorithmus, der in Linearzeit einen Planaritätstest durchführt und eine planare Einbettung berechnet. Der auf dem Links-Rechts-Planaritätskriterium aufbauende Algorithmus lässt sich relativ einfach implementieren, wird allerdings selten genutzt, da er kaum dokumentiert ist.

3 Verwendete Technolgien und verwandte Arbeiten

In diesem Kapitel wird ein Überblick über die für diese Arbeit relevante Technologien gegeben. Diese teilen sich in Algorithmen, welche zur Lösung genutzt werden und Technologien, die zur Implementieren verwendet werden. Anschließend wird ein Ausschnitt von Arbeiten aus dem Themengebiet der Planarisierung vorgestellt.

3.1 Verwendete Algorithmen

3.1.1 Die Breitensuche (BFS)

Die *Breitensuche* ist ein Verfahren zum Durchsuchen bzw. Durchlaufen eines Graphen - genauer: seiner Knoten [7]. Sie arbeitet wie folgt: es sei ein Startknoten s gegeben. Von diesem Knoten s wird nun jede adjazente Kante betrachtet und geprüft, ob der dazugehörige adjazente Knoten schon entdeckt wurde oder schon der gesuchte Zielknoten ist. Wenn dies nicht der Fall ist, wird dieser adjazente Knoten in einer Warteschlange gespeichert und im darauf folgenden Schritt bearbeitet. Die Breitensuche bearbeitet immer zuerst alle Nachbarn des aktuellen Knotens. Dies unterscheidet sie von der Tiefensuche, welche jeweils die erste Kante wählt und so den Pfad ermittelt. Wenn alle Kanten des Ausgangsknotens betrachtet worden sind, wird der erste Knoten aus der Warteschlange entnommen und das Verfahren wiederholt.

Algorithmus 3.1: Breitensuche

```
1  Prozedur Breitensuche (G: ungerichteter Graph, s: Startknoten, t: Zielknoten)
2  setze V = Knotenmenge von G
3  für alle Knoten i ∈ V
4      besucht[i] = falsch
5  füge s zu Queue Q hinzu
6  besucht[s] = wahr
7  solange Q nicht leer
8      setze Knoten k = nächster Knoten in Q
9      entferne k aus Q
10     falls k = t
11         gib wahr zurück
12     für alle Nachbarn n von k
13         falls besucht[n] = falsch
14             füge n zu q hinzu
15             besucht[n] = wahr
16     gib falsch zurück
```

Der Algorithmus 3.1 ist als Pseudocode zu verstehen und gibt aus Gründen der Lesbarkeit nur an, ob der Zielknoten gefunden wurde. Weitere wichtige Informationen wie z.B. die aktuelle Pfadtiefe oder der bisherige Suchweg können zusätzlich eingefügt werden, sofern es der Anwendungsfall erfordert.

3.1.2 Der Dijkstra-Algorithmus

Der *Dijkstra-Algorithmus* (benannt nach seinem Entwickler Edsger W. Dijkstra) dient in einem Graphen mit Kantengewichten zur Ermittlung eines kürzesten Pfades von einem Startknoten zu einem oder mehreren Zielknoten [9]. Diese Gewichtungen dürfen aber nicht negativ sein. Sollte ein Graph nicht zusammenhängend sein und Start- und Zielknoten sind in dessen Knotenmenge V vorhanden, aber es existiert kein Pfad zwischen ihnen, kann der Abstand auch unendlich sein.

Der Algorithmus 3.2 arbeitet wie folgt: Es wird immer die Kante mit dem geringsten Gewicht vom Startknoten s aus gewählt. Erst wenn dies geschehen ist, werden die anderen Kanten verfolgt. Dadurch ist gewährleistet, dass beim Erreichen eines Knotens kein kürzerer Pfad existiert. Durch dieses Verfahren ist es nicht notwendig, einmal gesetzte Distanzen wieder zu ändern. Dieses Vorgehen wird nun fortgesetzt bis der Zielknoten erreicht wird und dessen Distanz zum Startknoten s berechnet. Alternativ werden die Distanzen aller Knoten zum Startknoten s berechnet.

Algorithmus 3.2: Dijkstra-Algorithmus

```
1  Prozedur Dijkstra−Algorithmus (G: ungerichteter Graph, s: Startknoten)
2  setze V = Knotenmenge von G
3  für alle Knoten n ∈ V
4      setze distanz[n] = ∞
5      setze vorgänger[n] = −1
6      setze besucht[n] = falsch
7  setze distanz[s] = 0
8  setze Q = V
9  solange Q nicht leer
10     setze u Knoten mit kleinstem Wert in distanz
11     entferne u aus Q
12     für alle Knoten v ∈ V adjazent zu u
13         falls v ∈ Q
14             setze alternativ = distanz[u] + Gewicht von (u, v)
15         falls alternativ < distanz[v]
16             distanz[v] = alternativ
17             vorgänger[v] = u
18 gib vorgänger zurück
```

3.2 Verwendete Technologien

3.2.1 Eclipse

Eclipse ist ein Open Source Projekt, welches als Programmierumgebung für Software verschiedenster Art dient. Ursprünglich wurde Eclipse als Entwicklungsumgebung für die Programmiersprache Java entwickelt. Inzwischen wird es aber für eine Vielzahl von Entwicklungsprojekten genutzt, da eine Stärke von Eclipse seine Erweiterbarkeit ist. Dabei wird zwischen Open Source Erweiterungen und kommerzielle Erweiterungen unterschieden. Eine weitere Besonderheit von Eclipse ist die *Rich Client Platform*, welche es Anwendungsentwicklern ermöglicht, basierend auf dem Eclipse Framework von der Eclipse-IDE unabhängige Anwendungen zu schreiben. Eine nähere Auflistung ist auf der Eclipse Homepage[1] zu finden. Eine als Rich Client verfügbare Applikationen ist KIELER.

3.2.2 KIELER

KIELER steht für Kiel Integrated Environment for Layout Eclipse Rich Client[2]. KIELER kann sowohl als Rich Client Applikation, als auch durch das Hinzufügen von Plug-Ins in eine vorhandene IDE genutzt werden. Dieses Open Source Projekt, welches von der Arbeitsgruppe (AG) Echtzeitsysteme und Eingebettete Systeme der Christian-Albrechts-Universität zu Kiel entwickelt wurde, dient dazu den modellbasierten Entwurf für komplexe Systeme zu verbessern. Dazu werden für alle grafischen Komponenten der Diagramme automatische Layouts angeboten. Dadurch eröffnen sich einerseits neue Möglichkeiten bei der Erstellung und Bearbeitung von Diagrammen und andererseits Methoden für die dynamische Visualisierung, wie zum Beispiel Simulationsdurchläufe. Zur näheren Betrachtung wird dem Leser das Paper *Taming graphical modeling* von Fuhrmann et al. [11] empfohlen

Ein Teilprojekt von KIELER ist KLoDD, was für KIELER Layout of Dataflow Diagrams steht. Das KLoDD Projekt hat sich zum Ziel gesetzt Layout-Algorithmen für Diagramme grafischer Datenfluss-Sprachen zu entwickeln. Es sind diese Algorithmen, welche dem Anwender das Arbeiten erleichtern, da nur ein paar Einstellungen getätigt werden müssen und anschließend ein Klick auf den Layout-Knopf genügt.

3.3 Verwandte Arbeiten

Da die Planarisierung nicht nur Selbstzweck ist, sondern auch die Grundlage für viele graphentheoretische Verfahren bildet, gibt es eine Fülle von Literatur zu diesem Thema. Das Hauptaugenmerk wird in dieser Ausarbeitung auf Werke von Petra Mutzel und ihrer Arbeitsgruppe gerichtet, welche als weltweit führend auf diesem Gebiet gelten.

[1]http://eclipse.org/community/rcp.php
[2]http://www.informatik.uni-kiel.de/rtsys/kieler/

Hauptreferenz dieser Arbeit ist *Algorithms for the Hypergraph and the Minor Crossing Number Problems* von Markus Chimani und Carsten Gutwenger [4]. Die Autoren betrachten hier die Problemkombination von Hypergraphen und Kreuzungsminimierung, welche vorher noch nicht zusammen betrachtet wurde. Sie präsentieren Ergebnisse für das optimale Einfügen von Kanten, sowohl für eine feste, als auch für eine variable Einbettung. Auf ihrem Werk basiert der Abschnitt 4.2.3, welcher die Kreuzungsminimierung und das anschließende Einfügen einer Kante in Hypergraphen behandelt.

In ihrer Arbeit *Zeichnen von Diagrammen – Theorie und Praxis* gibt Petra Mutzel eine praxisnahe Einführung in das Thema des automatischen Zeichnens von Diagrammen [19]. Neben dem planaren Zeichnen, welches in dieser Bachelorarbeit der Schwerpunkt ist, werden dort noch weitere Verfahren wie kräfte-basierte oder hierarchische Verfahren vorgestellt. Diese Arbeit Mutzels ist für all diejenigen zu empfehlen, welche sich erst seit kurzem mit dem Thema automatisches Zeichen befassen, da sie anschaulich an den Themenbereich heranführt.

Im bereits in der Einführung zitierten Werk *Algorithmen zum automatischen Zeichnen von Graphen* [2] beschreiben die Autoren die Anfänge des automatischen Zeichnens von Diagrammen. Hier wird die Entwicklung weg vom Freihandzeichnen hin zum automatischen Zeichnen beschrieben. Weiterhin werden diverse Algorithmen vorgestellt, die zum automatischen Zeichnen dienen.

Inserting an Edge Into a Planar Graph von Carsten Gutwenger, Petra Mutzel und René Weiskirchner [12] befasst sich mit dem Thema der Planarisierung von Graphen. Hier wird nicht nur analysiert, wie für eine Einbettung eines Graphen Kanten optimal eingefügt werden können. Ziel ist es über alle Einbettungen die optimale zu finden, so dass möglichst wenig Kantenkreuzungen entstehen. Viele Probleme, welche sich mit dem Finden einer Lösung über alle kombinatorischen Einbettung befassen, sind NP-schwer. Den Autoren ist es mit Hilfe von sogenannten SPQR-Bäumen gelungen, einen Algorithmus mit linearer Laufzeit zu entwickeln, der eine minimale Anzahl von Kreuzungen findet. Aus diesem Werk stammt die Idee, Kantenkreuzungen durch Knoten zu ersetzen, welche die Grundlage für die in dieser Arbeit vorgestellten Verfahren bildet.

Eine Erweiterung des Planaritätsproblems und deren Lösung wird in *Planarity Testing and Optimal Edge Insertion with Embedding Constraints* von Carsten Gutwenger, Karsten Klein und Petra Mutzel aufgeführt [13]. Inhalt ist, dass viele praktische Anwendungen Restriktionen bezüglich der Einbettung verlangen, insbesondere was die Anordnung der Kanten um einen Knoten betrifft. Die Autoren zeigen, dass auch unter diesen Restriktionen das Problem der Kreuzungsminimierung in linearer Zeit gelöst werden kann.

Ein weitere Arbeit, die sich mit der Vertiefung der Planarisierung auseinander setzt, ist *Layer-Free Upward Crossing Minimization* von Markus Chimani, Carsten Gutwenger, Petra Mutzel und Hoi-Ming Wong [5]. Die Vertiefung besteht hier darin, dass neben der Minimierung von Kreuzungen, alle Kanten des Graphen in vertikaler Richtung verlaufen. Das Ziel war es einen Algorithmus zu entwickeln, welcher deutlich bessere Ergebnisse als bisher bekannte Algorithmen liefert. Dies ist den Autoren gelungen und wird in dieser Arbeit präsentiert.

4 Vorstellung der Verfahren

Dieses Kapitel befasst sich mit dem Einfügen von Kanten in planare Graphen und planare Hypergraphen und stellt den Schwerpunkt dieser Arbeit dar. Es wird erst das Kernverfahren für normale Graphen vorgestellt und dieses dann anschließend auf Hypergraphen erweitert. Dazu werden die in Kapitel 1 vorgestellten graphentheoretischen Grundlagen verwendet. Neben der textuellen Erklärung und der Beschreibung der Algorithmen in Pseudocode werden zur Veranschaulichung Abbildungen verwendet.

4.1 Einfügen von Kanten in planare Graphen

Dieses Abschnitt handelt vom Einfügen von Kanten in einen planaren Graphen. Es werden die Themen der Bildung des dualen Graphen, das Finden eines kürzesten Pfades und das Einfügen einer neuen Kante behandelt.

4.1.1 Ermitteln von Faces und dem dualen Graphen

Im Folgenden wird ein Verfahren vorgestellt, mit dem zu einem gegebenen Graphen seine Faces (siehe Abschnitt 2.4) ermittelt werden können. Der Algorithmus 4.1 beschreibt die Prozedur für das Finden eines linken Face einer Kante.

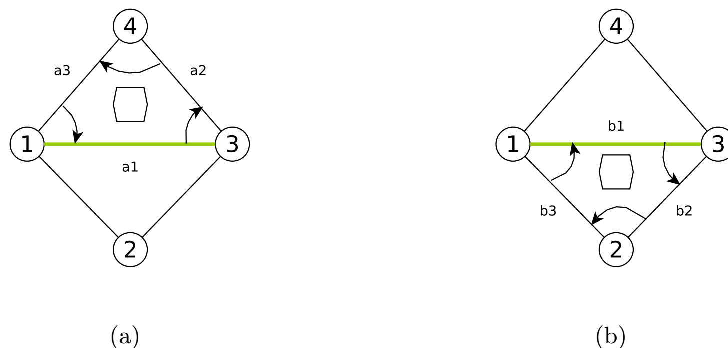


Abbildung 4.1: Finden eines linken (a) und eines rechten Face (b) in Schritten von 1 bis 3, beginnend am Knoten 1

Es wird über alle Kanten des Graphen iteriert und dabei geprüft, ob deren linkes oder rechte Faces noch unbekannt ist. Sollte dies für das linke Face der Fall sein,

wird vom Startknoten dieser Kante aus ein „Kreislauf“ gestartet und ein neues Face F initialisiert. Da es sich um das linke Face handelt, wird am Zielknoten die nächste ausgehende Kante im Uhrzeigersinn gewählt. In Abbildung 4.1 ist dies gut zu erkennen. Nun wird dies für alle nachfolgenden Kanten wiederholt, bis wieder der Startknoten der ursprünglichen Kante erreicht wurde. Für alle Kanten, durch die iteriert wurde, wird dieses neue Face F je nach Ausrichtung der Kante als linkes oder rechtes Face gesetzt. Gleichzeitig wird F ständig aktualisiert, d.h., dass diese durchlaufenen Kanten zur Menge der Kanten von F hinzugefügt werden. Dies geschieht analog für die durchlaufenen Knoten.

Algorithmus 4.1: Finden des linken Face

```
1  Prozedur (G: ungerichteter Graph)
2  setze E = Kantenmenge von G
3  für alle e ∈ E
4      falls e kein linkes Face hat
5          erzeuge neues Face f
6          setze linkes Face von e = f
7          setze n_0 = Startknoten von e
8          setze n_1 = n_0
9          setze e_1 = e
10         falls Graph nur eine Kante hat
11             füge n_1 zu f hinzu
12             füge e_1 zu f hinzu
13         tue
14             falls n_1 = Startknoten von e_1
15                 setze linkes Face von e = f
16             falls n_1 = Zielknoten von e_1
17                 setze rechtes Face von e = f
18             falls e_1 nicht die nächste Kante von n_1 gegen den Uhrzeigersinn
19             ist
20                 füge n_1 zu f hinzu
21                 füge e_1 zu f hinzu
22             setze n_1 = adjazenter Knoten über e_1
23             setze e_1 = nächste Kante von n_1 im Uhrzeigersinn
24         solange (n_1 ≠ n_0)
```

Falls beim Test das rechte Face einer Kante noch nicht gesetzt ist, wird analog verfahren. Der Unterschied ist, dass beim jeweiligen Zielknoten nicht die nächste, sondern die vorherige Kante im Uhrzeigersinn als nächste Kante gewählt wird, zu sehen in Abbildung 4.1(b).

Im Folgenden wird die Funktionsweise des Einfügens von Kanten erläutert. Die einzelnen Schritte für eine Kante werden dabei anhand eines Minimalbeispiels illustriert. In diesem soll eine Kante zwischen den beiden Knoten s und t eingefügt werden, siehe Abbildung 4.2. Es ist leicht zu erkennen, dass dies nicht möglich ist, ohne eine Kante zu kreuzen.

Wenn ein geeigneter Planaritätstest die Nicht-Planarität feststellt, liefert dieser nun eine Liste mit Kanten N, die man nicht mehr in den Graphen G einfügen kann,

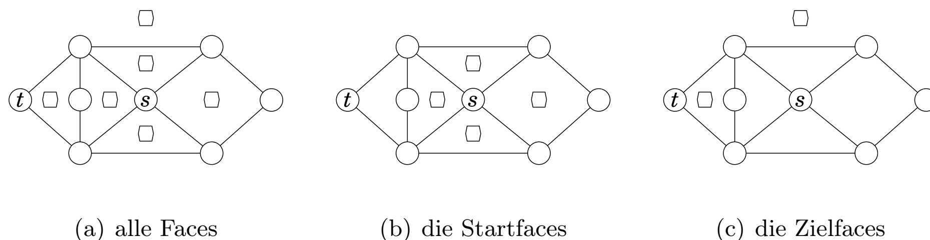
ohne dessen Planarität zu verletzen, sowie einen planaren Subgraphen $SG = (V', E')$ des ursprünglichen Graphen G. Dieser Subgraph enthält alle Knoten von G (d.h. $V' = V$), allerdings nicht zwangsläufig alle Kanten ($E' \subseteq E$). Nun gilt es diese Kanten N so in den Subgraphen SG einzufügen, dass man möglichst wenig Kantenkreuzungen erzeugt und somit die Anzahl der neuen Knoten minimiert.

Dazu wird zuerst der duale Graph SG' (siehe Abschnitt 2.5) des gegebenen Subgraphen SG gebildet, wie in Algorithmus 4.2 beschrieben.

Algorithmus 4.2: Erstellen des dualen Graphen

```
1 Prozedur Erzeuge−Dualen−Graphen (G: ungerichteter Graph)
2 setze F = Facemenge von G
3 erzeuge neuen Graphen G'
4 für alle Faces f ∈ F
5     erzeuge neuen Knoten v in G'
6 für alle Kanten e ∈ E
7     setze a = zum linken Face von e zugehöriger Knoten
8     setze b = zum rechten Face von e zugehöriger Knoten
9     erzeuge neue Kante (a, b) in G'
```

Aus der Eigenschaft eines dualen Graphen ist bekannt, dass SG' zusammenhängend ist. Aus der neu zu platzierenden Kante n wird der Startknoten $s \in V'$ und der Zielknoten $t \in V'$ in SG ermittelt. Zu diesen beiden Knoten gilt es nun die angrenzenden Faces zu finden und so im dualen Graphen SG' mögliche Start- und Zielpunkte für einen Suchalgorithmus zu bestimmen. Wie dies konkret aussieht zeigt Abbildung 4.2.



(a) alle Faces (b) die Startfaces (c) die Zielfaces

Abbildung 4.2: Ermitteln der allgemeinen und relevanten Faces

Es werden nun zwei verschiedene Verfahren vorgestellt, um den kürzesten Pfad im dualen Graphen SG' zu ermitteln.

4.1.2 Suche mit Breitensuche

Algorithmus 4.3: Ermitteln des optimalen Pfades mit BFS

```
1  Prozedur Finden−eines−Pfades−mit−Breitensuche (SG: ungerichtete Graph)
2  sei SG′ der duale Graph von SG
3  sei T ⊆ F die Menge der Zielfaces
4  für alle Kanten e ∈ N
5      für alle Startfaces f ∈ F
6          setze n = dualer Knoten von f in SG′
7          führe BFS in SG′ mit Starknoten n aus bis ein beliebiges t ∈ T erreicht wurde
8          setze P = Pfad von n nach t
9          falls Länge von P = 1
10             gib P zurück
11         falls S = leer oder Länge von S < Länge von P
12             setze S = P
13 gib S zurück
```

Ein möglicher Suchalgorithmus ist die Breitensuche (siehe Abschnitt 3.1.1). Die Laufzeit dieser Verfahrensweise 4.3 für einen Graphen $G = (V, E)$ beträgt in diesem Fall $O(|V| \cdot |S| \cdot |T|)$, mit S Menge der in Frage kommenden Startfaces und T Menge der möglichen Zielfaces. Die Laufzeit ist deshalb so hoch, weil für jedes Startface aus S Pfade zu allen Zielfaces aus T geprüft werden müssen, um den optimalen (kürzesten) Pfad zu finden.

4.1.3 Suche mit dem Dijkstra-Algorithmus

Algorithmus 4.4: Ermitteln des optimalen Pfades mit dem Dikstra-Algorithmus

```
1  Prozedur Finden−eines−Pfades−mit−dem−Dijkstra−Algorithmus (SG′: ungerichter
       Graph, s: Startknoten, t: Zielknoten)
2  setze S Liste der zum Startknoten s adjazenten Faces
3  setze T Liste der zum Zielknoten t adjazenten Faces
4  für alle Kanten e ∈ E in SG′
5      setze Gewicht von e auf 1
6  füge s in SG′ ein
7  füge t in SG′ ein
8  setze S′ ⊆ V′ Liste der dualen Knoten, welche den Faces aus S entsprechen
9  setze T′ ⊆ V′ Liste der dualen Knoten, welche den Faces aus T entsprechen
10 für alle Knoten n ∈ S
11     erzeuge neue Kante g = (s, n)
12     setzte Gewicht von g auf 0
13 für alle Knoten m ∈ T
14     erzeuge neue Kante h = (m, t)
15     setzte Gewicht von h auf 0
16 führe Dijkstra Algorithmus von s nach t in SG′ aus
17 setze P = Pfad von s nach t
18 gib P zurück
```

Da der Dijkstra-Algorithmus (siehe Abschnitt 3.1.2) auf gewichteten Kanten arbeitet, wird für jede Kante im dualen Graphen SG' ein Gewicht von 1 gesetzt. Dies dient dazu, dass der Dijkstra-Algorithmus alle Kanten gleichwertig behandelt (siehe dazu Algorithmus 4.4). Nun werden in den dualen Graphen SG' der ursprüngliche Startknoten s und Zielknoten t der neuen Kante eingefügt. Diese Verfahrenweise entstammt *Algorithms for the Hypergraph and the Minor Crossing Number Problems* [3] und hat den Zweck, dass der Algorithmus einen festen Start- und Zielknoten hat und dabei trotzdem alle Knoten berücksichtigt, welche als potenzielle Start- und Zielknoten im dualen Graphen SG' in Frage kommen. Diese entsprechen den Faces im gegebenen Graphen SG. Von diesen Knoten wird jeweils eine Kante zu denjenigen Knoten in SG' erzeugt, welche im Graphen SG die an s und t grenzenden Faces repräsentieren. Diese neuen Kanten erhalten ein Gewicht von 0, denn sie dienen wie vorher beschrieben nur als Start- und Zielknoten für den Algorithmus und haben keine Auswirkungen auf die Länge des Pfades.

Auf diesen erweiterten dualen Graphen wird nun der Dijkstra-Algorithmus angewandt, um den kürzesten Pfad von s nach t zu finden. Die Laufzeit beträgt $O(|V| \cdot \log |V| + |E|)$. Dieses Verfahren hat den Vorteil, dass Kanten die möglichst nicht gekreuzt werden sollen mit einem hohem Gewicht versehen werden können. So kann der Anwender Einfluss auf die Art der Planarisierung nehmen. Eine weitere Variablilität, die sich dadurch eröffnet, ist, dass Multikanten zu einzelnen Kanten mit einem höheren Gewicht zusammen gefasst werden können.

4.1.4 Das Einfügen

Durch Ausführen des jeweiligen Suchalgorithmus entsteht eine Liste von Kanten $L \subseteq E'$, die gekreuzt werden müssen. Siehe dazu Abbildung 4.3. Auf diesem Pfad P sollte im Normalfall mindestens eine Kante gekreuzt werden, denn wenn die Kante ohne Kreuzung eingefügt werden kann, liegt ein Fehler im Planaritätstest vor. Es wurde dann nicht der maximal planare Subgraph ermittelt. Eine solche Kante hätte zu dem planaren Subgraphen SG hinzugefügt werden können, ohne die Eigenschaft der Planarität zu verletzten. Das Einfügen einer solchen Kante stellt allerdings kein Hindernis da, denn sie kann ohne weitere Bearbeitung zwischen Start- und Zielknoten eingefügt werden.

Das Einfügen beginnt vom Startknoten s im Graphen SG aus. Die erste (und in diesem Beispiel schon letzte) Kante, die es zu kreuzen gilt, ist die erste Kante in L. Auf dieser Kante $l \in L$ wird ein neuer Knoten k_0 eingefügt, wobei zu beachten ist, diesen korrekt mit dem Startknoten und dem Zielknoten der Kante l zu verbinden (Algorithmus 4.5). Dabei ist besonders wichtig, dass die Einbettung in korrekter Form erhalten bleibt. Algorithmus 4.6 beschreibt, wie die neue Kante an der richtigen Position mit dem Knoten verbunden wird.

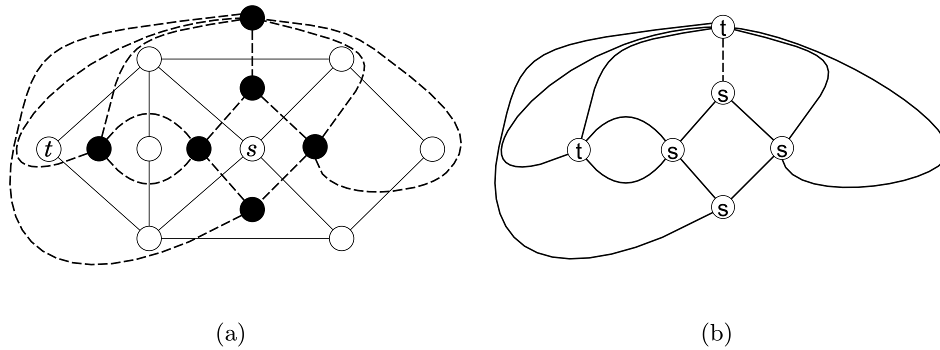


Abbildung 4.3: Der Graph G und sein dualer Graph G', in dem der kürzeste Pfad (gestrichelte Kante) von einem Startknoten der s zu einem der Zielknoten t gewählt wird.

Algorithmus 4.5: Einfügen eines Knotens auf einer Kante

1 **Prozedur** Kantenspalten (G: ungerichteter Graph, e: Kante in G)
2 setze V = Knotenmenge von G
3 setze $s \in V$ Startknoten von e
4 setze $t \in V$ Zielknoten von e
5 erzeuge neuen Knoten k
6 entferne Kante (s, t)
7 erzeuge neue Kante $f = (s, k)$
8 erzeuge neue Kante $g = (k, t)$
9 füge f so an s an, dass die Einbettung erhalten bleibt
10 füge g so an t an, dass die Einbettung erhalten bleibt

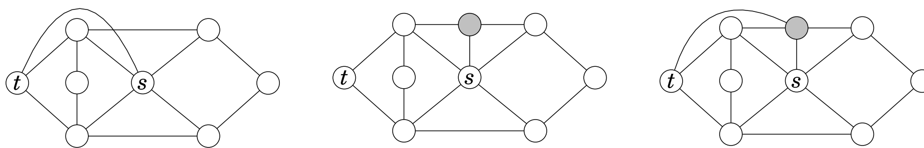
Algorithmus 4.6: Einbettung aktualisieren

1 **Prozedur** Einbettung−erhalten (G: ungerichteter Graph, e: Kante in G)
2 setze V = Knotenmenge von G
3 sei $n \in V$ der Knoten an den e gebunden wird
4 **falls** n Startknoten von e ist
5   sei f das rechte Face von e
6   setze Kante $b \in E$ die an n und f liegt
7   füge e im Uhrzeigersinn hinter b ein
8 **falls** n Zielknoten von e ist
9   sei f das linke Face von e
10   setze Kante $b \in E$ die an n und f liegt
11   füge e im Uhrzeigersinn hinter b ein

Algorithmus 4.7: Einfügen der neuen Kanten mit Hilfe neuer Knoten

1 **Prozedur** Kanteneinfügen (G: ungerichter Graph, N: Liste von Kanten)
2 setze V = Knotenmenge von G
3 **für alle** Kanten $e \in N$
4   erzeuge neue Liste von Knoten K
5   setze $s \in V$ Startknoten von e
6   setze $t \in V$ Zielknoten von e
7   füge s zu K hinzu
8   **für alle** Kanten $l \in L$
9   erzeuge neuen Knoten k auf l
10   füge n zu K hinzu
11   füge t zu K hinzu
12   **für alle** Knoten $k \in K$
13   erzeuge neue Kante (k, Nachfolger von k in K)

Der Startknoten s wird nun mit dem neuen Knoten k_0 durch eine neue Kante $e_0 = (s, k_0)$ verbunden. So wird weiter verfahren für alle Kanten $l \in L$, die es zu kreuzen gilt. Von dem letzten neuen Knoten k_n, der durch das Aufspalten der letzten Kante aus der Liste L entsteht, wird nun eine Kante $e_n = (k_n, t)$ zum Zielknoten t erzeugt. Siehe dazu Abbildung 4.4.



(a) Graph mit einzufügender Kante (b) Startknoten mit neuem Knoten verbunden (c) Neuer Knoten mit Zielknoten verbunden

Abbildung 4.4: Einfügen des neuen Knotens und der neuen Kante

Damit wurde durch das Einfügen von neuen Knoten und die Verbindung eben dieser eine Kante in den planaren Graphen SG eingefügt, ohne dessen Planarität zu verletzten. Dies gilt es nun, wie in Algorithmus 4.7 beschrieben, für alle Kanten $n \in N$ durchzuführen.

4.2 Einfügen von Kanten in planare Hypergraphen

Dieser Abschnitt befasst sich mit der Erweiterung des Verfahrens aus Abschnitt 4.1.4. Entwickelt und in seiner Korrektheit bewiesen wurde dieses Verfahren von Markus Chimani und Carsten Gutwenger [3].

4.2.1 Die Point-based Form herstellen

Um das in Abschnitt 4.1 für reguläre Graphen vorgestellte Verfahren auf Hypergraphen anwenden zu können, muss, falls dies nicht schon der Fall ist, die in Kapitel 1 vorgestellte *point-based* Form hergestellt werden. Als Voraussetzung sei gegeben, dass der gegebene Subgraph SG nicht im *subset standard* dargestellt ist, sondern im *edge standard*. Nun kann es aber sein, dass dieser Graph in *tree-based* Form gegeben ist. Um diesen in die *point-based* Form zu überführen, wird folgendes Verfahren angewendet: es werden alle Hyperknoten $v \in V$ des Graphen SG durchlaufen und dabei verifiziert, ob sich an einer Hyperkante ein weiterer Hyperknoten $w \in V$ anstelle eines normalen Knotens befindet. Sollte dies der Fall sein, werden alle Hyperknoten, die an einer Hyperkante liegen, zu einem Hyperknoten verschmolzen. Dies bedeutet, dass alle Kanten, die vormals auf die verschiedene Hyperknoten verteilt waren, nun mit einem einzigen Hyperknoten verbunden werden. Anschließend werden die anderen Hyperknoten innerhalb dieser Hyperkante aus dem Graphen entfernt.

Algorithmus 4.8: Verschmelzen der Hyperknoten

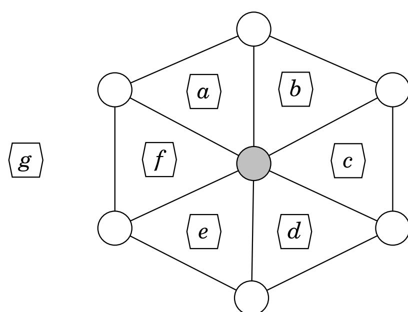
```
1 Prozedur Hyperknoten−Verschmelzen (G: ungerichtert Graph)
2 für alle Hyperknoten h1 in G
3     für alle Hyperknoten h2 in G
4         falls h1 adjazent zu h2
5             lösche Kante e = (h1, h2)
6             für jede zu h2 adjazente Kante f
7                 löse f von h2
8                 verbinde f mit h1
9             entferne h2
```

4.2.2 Den dualen Graphen erweitern

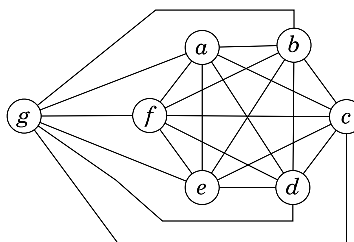
Nachdem der gegebene Subgraph SG in die *point-based* Form überführt wurde, wird analog das in Abschnitt 4.1 vorgestellte Verfahren angewendet und der duale Graph SG' berechnet. Doch hier bedarf es einer Erweiterung. Da das Kreuzen einer Hyperkante enormes Einsparungspotential beim Einfügen neuer Knoten bietet, muss dies bei der Pfadsuche im dualen Graphen SG' berücksichtigt werden. Unabhängig von der Anzahl der Kanten in einer Hyperkante, bedarf es nur eines neuen Knoten. Dies wird wie folgt realisiert: Zuerst werden alle zu einem Hyperknoten adjazenten Faces erfasst. Zu diesen lassen sich die entsprechenden Knoten im dualen Graphen SG' ermitteln. Diese Knoten werden untereinander alle jeweils durch eine Kante in SG' verbunden, sofern noch keine Kante zwischen ihnen existiert. Dies wird für alle Hyperknoten in SG durchgeführt. Auch diesmal wird die Vorgehensweise anhand grafischer Ausschnitte verdeutlicht. Abbildung 4.5 zeigt, wie aus Faces, welche um einen Hyperknoten (in folgenden Abbildungen immer grau dargestellt) liegen, im dazugehörigen dualen Graphen ein zusammenhängender Subgraph entsteht.

Algorithmus 4.9: Erweitern des dualen Graphen

```
1 Prozedur Erweiterung−des−dualen−Graphen (H: Hypergraph)
2 setze V = Knotenmenge von G
3 sei H′ der duale Graph von H
4 für alle Hyperknoten h ∈ V
5     sei F Menge der an h anliegenden Faces
6     sei F′ Menge der dualen Knoten in H′, die den Faces aus F in H entsprechen
7     für alle Knoten n, m ∈ F′
8         falls keine Kante zwischen n und m existiert
9             erzeuge neue Kante (n, m)
```



(a) Hypergraph H mit seinen Faces

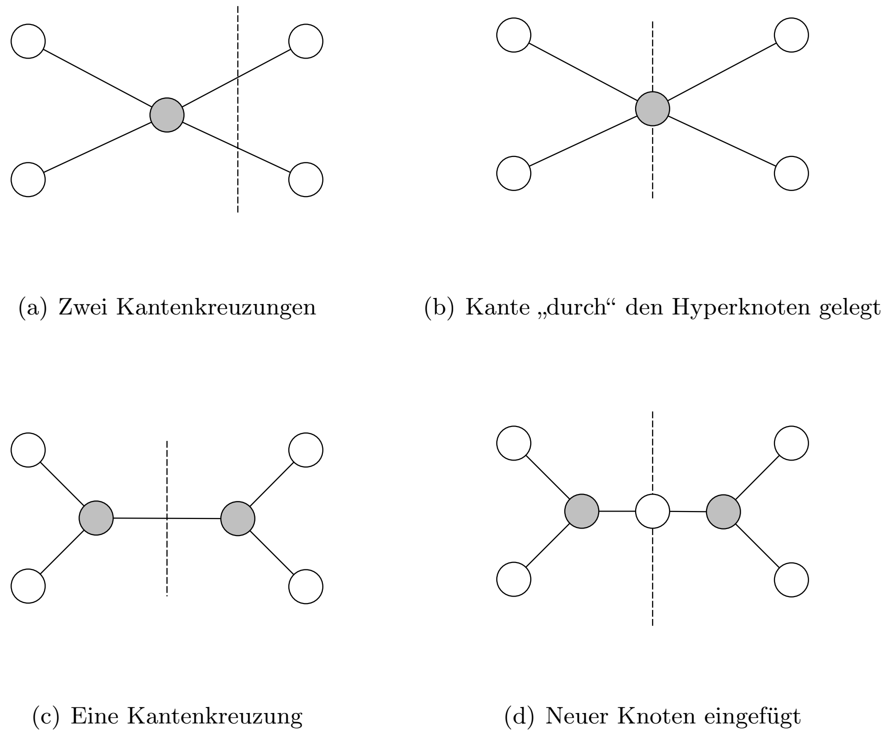


(b) Der duale Graph von H

Abbildung 4.5: Faces a bis f, welche am Hyperknoten von Hypergraph H liegen, werden im dualen Graphen zu einem vollständigen Teilgraphen

4.2.3 Das Einfügen

In diesem erweiterten Hypergraphen wird nun auf die gleiche Art und Weise wie in Abschnitt 4.1 ein kürzester Pfad vom Startknoten zum Zielknoten der neuen Kante ermittelt. Nachdem dieser Pfad bekannt ist, werden wie in Abschnitt 4.1.4 die zu kreuzenden Kanten auf dem Pfad betrachtet. Sollte es sich dabei um Kanten handeln, die zwischen normalen Knoten liegen, wird analog zu Abschnitt 4.1.4 verfahren. Neu hinzu kommt nun der Fall, in welchem solch eine zu kreuzende Kante einen Hyperknoten als Start- oder Zielknoten hat. In Abbildung 4.6(a) ist zu sehen, wie die neu einzufügende Kante zwei Kanten einer Hyperkante kreuzt. Auf diesen Fall wird im Folgenden eingegangen und dessen Lösung erklärt.



(a) Zwei Kantenkreuzungen (b) Kante „durch“ den Hyperknoten gelegt

(c) Eine Kantenkreuzung (d) Neuer Knoten eingefügt

Abbildung 4.6: Einfügen einer neuen Kante (gestrichelt)

Um nun die Anzahl der Kantenkreuzungen minimal zu halten wird wie folgt verfahren: Die einzufügende Kante wird zuerst „durch“ den Hyperknoten $h \in V$ gelegt (Abbildung 4.6(b)), welcher der zu den gekreuzten Kanten zugehörige Hyperknoten ist. Da vorher bereits die *point-based* Form hergestellt wurde, ist dieser Hyperknoten h eindeutig identifizierbar.

Algorithmus 4.10: Aufspalten eines Hyperknotens

```
1 Prozedur Hyperknoten−spalten (h: Hyperknoten)
2 erzeuge neuen Hyperknoten h2
3 setzte n Anzahl der Kanten an h
4 erzeuge neue Kante e = (h, h2)
5 für die ersten n/2 Kanten f an h hinter e
6         löse f von h
7         verbinde f mit h2
```

Der nun ermittelte Hyperknoten h wird in zwei Hyperknoten h_1 und h_2 aufgespalten. Anschließend wird eine neue Kante (h_1, h_2) erzeugt, wie Algorithmus 4.10 beschreibt. Somit liegt nicht mehr die *point-based* Form, sondern wieder die *tree-based* Form vor. Es bleibt aber nach wie vor die originale Hyperkante erhalten, da sich deren Knotenmenge F nicht ändert, sondern nur die Darstellung durch die Hyperknoten.

Zusätzlich kommt hier nun hinzu, dass die Kanten, die vorher alle an dem ursprünglichen Hyperknoten h lagen, nun auf die beiden Hyperknoten h_1 und h_2 aufgeteilt werden.

Hierbei ist wieder darauf zu achten, dass die Einbettung in korrekter Form erhalten bleibt. Damit ist auch leicht zu ersehen, dass die Anzahl der notwendigen Kreuzungen von zwei in Abbildung 4.6(a) auf eine in Abbildung 4.6(c) verringert werden konnte.

Nun wird, analog zu Abschnitt 4.1, die neue entstandene Kante zwischen den Hyperknoten h_1 und h_2 gekreuzt, indem ein neuer Knoten n auf dieser eingefügt wird (siehe Algorithmus 4.7). Diese eine Kreuzung wird dann wie vorher auch durch einen neuen Knoten ersetzt, indem die neue kreuzende Kante in zwei Kanten aufgespalten wird, von denen eine den ursprünglichen Startknoten mit n verbindet und die andere n mit dem ursprünglichen Zielknoten. Abbildung 4.6(d) zeigt das endgültige Ergebnis.

4.3 Änderung der Graphstruktur

Die klassische Aufgabe eines Layouters für Graphen ist, die Elemente durch Verschieben in der zweidimensionalen Ebene neu anzuordnen. Er verändert dabei jedoch nicht die Graphenstruktur, d.h. die Anzahl der Knoten und Kanten, sowie die Einbettung der Kanten (Startknoten, Zielknoten).

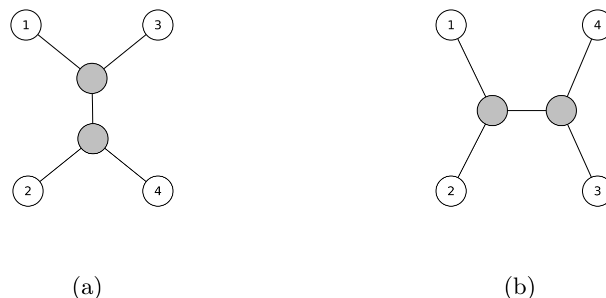
Die in Kapitel 4 vorgestellten Verfahren (sowohl das für normale Graphen als auch das für Hypergraphen) erweitern diese Aufgabe. Sollte beim Planaritätstest die Nichtplanarität festgestellt werden, ändern sie die Graphenstruktur. Dies geschieht dadurch, dass erst Kanten entfernt und anschließend bei der Planarisierung neue Knoten und Kanten eingefügt werden. Diese Änderungen müssen allerdings nicht an das Originalmodell übertragen werden, da sie nach der Durchführung von Orthogonalisierung und Konmpaktierung wieder rückgängig gemacht werden. Was hingegen verankert werden muss, ist die Veränderung der Hyperkanten F von H aus Abschnitt 4.2.

Diese Änderungen müssen an das Graphenmodell von H übertragen werden. Dieses Problemstellung wurde bisher kaum behandelt und kann auf unterschiedlichen Ebenen behandelt werden.

Ersetzen aller Hyperkanten ist in radikaler Ansatz, alle Hyperknoten und somit Hyperkanten $f \in F$ aus dem Hypergraphen H zu entfernen und anschließend wieder einzufügen. Vorteil dieses Ansatzes ist, dass nicht überprüft werden muss, ob sich etwas an den Hyperkanten geändert hat. Der Nachteil besteht allerdings darin, dass auch Hyperkanten neu erzeugt werden, an denen keine Änderung stattgefunden hat.

Ersetzen der geänderten Hyperkanten wäre eine andere Variante, bei welcher nur genau die Hyperkanten neu erzeugt werden, welche vom Algorithmus betroffen sind. Innerhalb dieser Variante gäbe es verschiedene Ansätze um betroffene Hyperkanten zu ermitteln. Es wäre zu prüfen, ob eine Hyperkante f nach der Planarisierung noch genau einen Hyperknoten hat, also in *point-based* Form vorliegt. Wenn dies auch vor der Planarisierung der Fall ist, wurde die Hyperkante f nicht verändert (Hyperknoten verschmolzen oder gekreuzt) und es bedarf keiner Änderung. Nach diesem Ansatz würden alle Hyperkanten mit mehr als einem Hyperknoten neu erzeugt werden.

Originale Hyperkanten nur modifizieren ist die Variante, welche die Anzahl der Umformungen minimieren würde. Idee wäre hier bei gleichgebliebener Anzahl von Hyperknoten die Kanten zwischen Knoten und Hyperknoten so „umzubiegen", dass der Graph dem Ergebnis der Planarisierung entspricht. Einen möglicher Fall zeigt Abbildung 4.7: hier müssten die Kanten von Knoten 1 und 2 mit dem einem und die Kanten von Knoten 3 und 4 mit dem anderem Hyperknoten verbunden werden. Nachteil dieser Verfahrensweise ist, dass ein *matching* zwischen der alten und der geänderten Form der Hyperkante stattfinden muss. Eine Lösung Wenn sich die Anzahl der Hyperknoten in f ändert, kompliziert sich das Problem noch weiter.



(a) (b)

Abbildung 4.7: Änderung einer Hyperkante

Ein offenes Problem, welches weiterer Forschung bedarf, ist, von diesen Varianten diejenige mit der optimalen Laufzeit zu finden. Dazu muss der Aufwand von Prüfung und Neu-Erzeugen abgewogen werden, um ein optimales Maß zu ermitteln. Dies gestaltet sich insofern schwierig, da erst geeignete Verfahren ermittelt werden müssen.

5 Implementierung

Die Implementierung der in dieser Arbeit vorgestellten Verfahren wurde in der Programmiersprache Java vorgenommen.
Sie ist im KIELER Plug-In de.cau.cs.kieler.klay.planar zu finden. Mit der Entwicklung dieses Plug-Ins wurde in der Veranstaltung *Layout Algorithms* im Sommersemester 2010 begonnen. Es stellt einen weiteren Layoutalgorithmus zur Verfügung. Es unterteilt sich in Planaritätstests, Planarisierung, Orthogonalisierung sowie Kompaktierung. Es beinhaltet weiterhin eine eigene Datenstruktur: die PGraph Struktur.

5.1 Datenstruktur

Für alle nach außen relevanten Elemente der PGraph Struktur gibt es entsprechende Interfaces, wie Abbildung 5.1 zeigt.

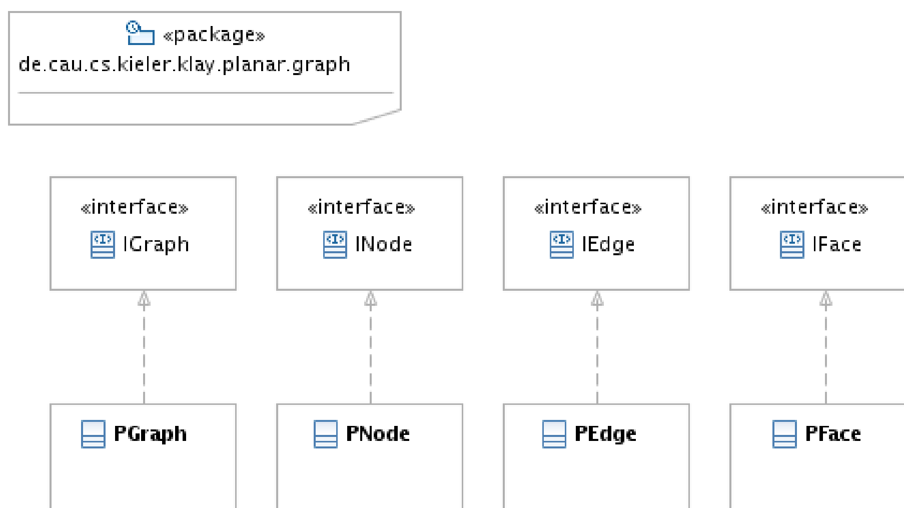


Abbildung 5.1: Die Interfaces der PGraph Struktur

Die Datenstruktur dient primär dazu, dass auf ihr theoretische Graphalgorithmen ausgeführt werden können. Sie enthält keine Informationen wie X- oder Y-Koordinaten, die zum Zeichnen der Elemente des Graphen dienen. Ein gegebener Graph liegt in der KGraph[1] Struktur vor und wird zum Bearbeiten in die PGraph

[1] http://rtsys.informatik.uni-kiel.de/trac/kieler/browser/trunk/plugins/de.cau.cs.kieler.core.kgraph

Struktur umgewandelt. Nachdem alle Operationen durchlaufen wurden, wird der erzeugte PGraph wieder in einen KGraph umgewandelt, um ihn visuell darstellen zu können.

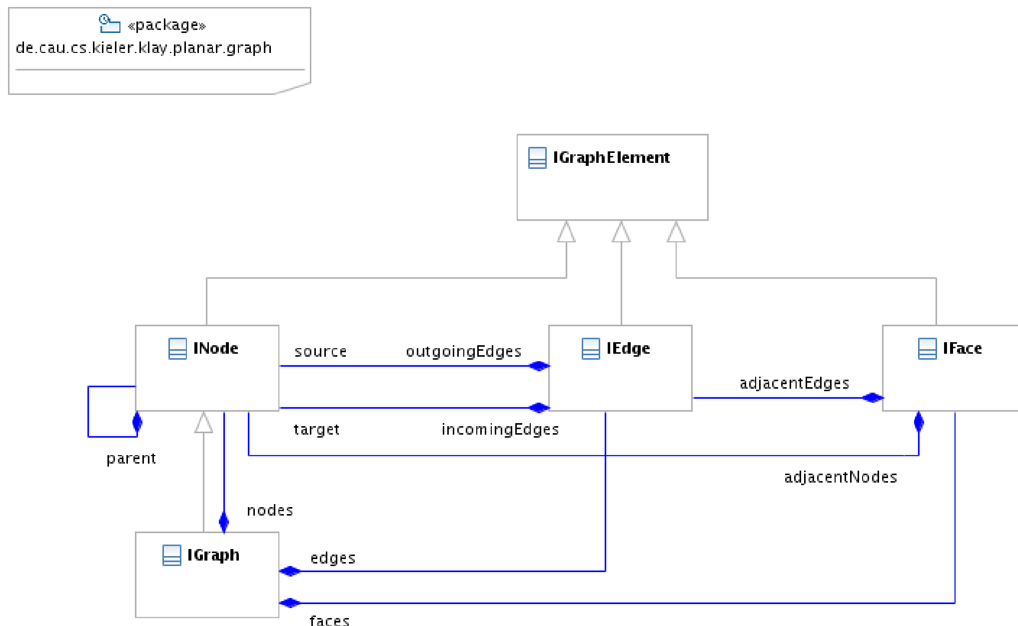


Abbildung 5.2: Die Datenstruktur

Der Typ eines Knotens wird durch eine Enumeration beschrieben. Es gibt drei Typen von Knoten:

- NORMAL, den normalen Knoten. Die meisten Knoten in einem Graphen sind von diesem Typ.
- HYPER, den Hyperknoten. Dieser dient zur Verbindung (wie im Kapitel 1 beschrieben) mehrerer Knoten mit einer Hyperkante.
- COMPOUND, ein Compoundknoten ist ein Knoten, der in sich wieder einen Graphen darstellt. Er enthält in sich wieder Knoten, Kanten und sogar weitere Compoundknoten.

Ein Knoten besitzt weiterhin Informationen über benachbarte Kanten und Knoten, sowie eingehende und ausgehende Kanten.
Eine Kante besitzt die Informationen, welches ihr Start- und Zielknoten ist, sowie ihr linkes und rechtes Face ist.
Ähnlich wie bei dem Knoten hat auch ein Face diverse Iteratoren, um Informationen über angrenzende Knoten, Kanten und auch Faces zu liefern.

5.1.1 Einbettung

In der Implementierung ist eine Einbettung in der Art dargestellt, dass die zu einem Knoten inzidenten Kanten in einer sortierten Liste gespeichert werden. Auf diese Art kann auch ohne grafische Ausgabe leicht verifiziert werden, ob die Einbettung korrekt ist. Da nur die Reihenfolge, aber nicht die erste Kante durch eine Einbettung beschrieben wird, können unterschiedliche Listen die gleiche Einbettung darstellen.

5.1.2 Wahl des Suchalgorithmus

Bei der Wahl des Suchalgorithmus fiel die Wahl auf den Dijkstra-Algorithmus. Dieser ist bereits im Rahmen des Plug-Ins implementiert und liefert eine Liste von Kanten, welche den Pfad vom Startknoten zum Zielknoten beschreiben, zurück. Es ist leicht möglich diesen durch die Breitensuche zu ersetzen, welche ebenfalls durch dieses Plug-In zur Verfügung gestellt wird. Beide Algorithmen liefern einen Kantenzug im dualen Graphen, so dass das restliche Verfahren kompatibel zu beiden ist.

5.1.3 Testen der Implementierung

Die Implementierung der Planarisierung ist ein Teilprojekt, des Plug-Ins *de.cau.cs.kieler.klay.planar*. Da sich dieses noch in seiner abschließenden Entwicklung befindet, war ein auf grafischer Darstellung basierendes Testen nicht möglich, denn erst nach der letzten Phase in diesem Plug-In wird der PGraph in einen KGraph überführt. Aus diesem Grund wurde die Ergebnisse per Konsole ausgegeben und von Hand verifiziert. Durch die Ausgabe der Einbettung kann solch ein Ergebnisgraph auch ohne konkrete X- und Y-Koordinaten erstellt werden. Da diese Arbeitsweise jedoch sehr zeitaufwändig ist, wurde beim Testen der Schwerpunkt auf relativ kleine Testgraphen gelegt, welche möglichst viele Spezialfälle abdecken.

6 Zusammenfassung und Ausblick

Automatische Layouts von Diagrammen erleichtern das Arbeiten in vielen Bereichen und sind dort nicht mehr weg zu denken. Da solche Diagramme auf Graphen basieren, ist für den Layoutprozess die Entwicklung graphenbasierter Verfahren notwendig. In den letzten Jahren wurden immer mehr, vormals nur theoretisch betrachtete Fragestellungen, praxisnah betrachtet und eine Lösung umgesetzt. Da die Anzahl der Kantenkreuzungen in einem Diagramm eine wichtige Rolle bei Verständnis dieser haben [20], ist die Planarisierung des dem Diagramm zu Grunde liegenden Graphen ein wichtiges Problem.

Ein wichtiges Kriterium bei der Planarisierung ist, die Anzahl der zu kreuzenden Kanten zu minimieren. In dieser Arbeit wurde ein Verfahren vorgestellt, welches es ermöglicht, die Anzahl der Kantenkreuzungen zu minimieren und die verbleibenden notwendigen Kreuzungen durch neue Knoten zu ersetzten und das sowohl für normale Graphen, als auch für Hypergraphen. Dazu können bekannte Verfahren wie die Breitensuche oder Dijkstra-Algorithmus verwendet werden.

Der Ansatz dieser Arbeit ging von einer festen gegebenen Einbettung aus. Von daher wäre eine Erweiterung, die eine optimale Lösung über alle Einbettungen, unter Zuhilfenahme von SPQR-Bäumen (nach dem Verfahren von Gutwenger et al. [12]) liefert, denkbar. Wie die Arbeiten von Chimani, Gutwenger et al. [13, 5] aufzeigen, gibt es viele Richtungen, in welche weiter geforscht und entwickelt wird. Ein anderer wichtiger Aspekt der Planarisierung ist, dass sie die Grundlage für Verfahren wie Orthogonalisierung und Kompaktierung bildet [6] und von daher ein eine Ausgangsbasis für anderweitige Verfahren verschiedener Art ist.

Zu klären ist weiterhin, welcher Lösungsansatz aus Abschnitt 4.3 für das aktualisieren des Hypergraphen der optimale ist.

Literaturverzeichnis

[1] John Boyer and Wendy Myrvold. On the cutting edge: Simplified $\mathcal{O}(n)$ planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004.

[2] Franz J. Brandenburg, Michael Jünger, and Petra Mutzel. Algorithmen zum automatischen Zeichnen von Graphen. *Research Report Number: MPI-I-97-1-007*, Informatik-Spektrum 20(4):199–207, 1997.

[3] Markus Chimani and Carsten Gutwenger. Algorithms for the hypergraph and the minor crossing number problems. In *18th International Symposium on Algorithms and Computation (ISAAC'07)*, volume 4835 of *LNCS*, pages 184–195. Springer, 2007.

[4] Markus Chimani, Carsten Gutwenger, Michael Jünger, Karsten Klein, Petra Mutzel, and Michael Schulz. The Open Graph Drawing Framework. Poster at the 15th International Symposium on Graph Drawing (GD07), 2007.

[5] Markus Chimani, Carsten Gutwenger, Petra Mutzel, and Hoi-Ming Wong. Layer-free upward crossing minimization. In *WEA 2008: Proceedings of the 7th International Workshop on Experimental Algorithms*, volume 5038 of *LNCS*, pages 55–68. Springer-Verlag, 2008.

[6] Ole Claußen. Implementing an algorithm for orthogonal graph layout, 2010. to appear.

[7] Thomas H. Cormen, Charles Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 2001. ISBN 0-262-53196-8.

[8] H. O. Pollak D. S. Johnson. Hypergraph planarity and the complexity of drawing venn diagrams. *Journal of Graph Theory*, 11:309–325, 1987.

[9] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[10] Thomas Eschbach, Wolfgang Guenther, and Bernd Becker. Orthogonal hypergraph drawing for improved visibility. *Journal of Graph Algorithms and Applications*, 10(2):141–157, 2006.

[11] Hauke Fuhrmann and Reinhard von Hanxleden. Taming graphical modeling. Technical Report 1003, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2010.

[12] Carsten Gutwenger, Petra Mutzel, and René Weiskircher. Inserting an edge into a planar graph. In *SODA '01: Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 246–255. SIAM, 2001. ISBN 0-89871-490-7.

[13] Carsten Gutwenger, Karsten Klein, and Petra Mutzel. Planarity testing and optimal edge insertion with embedding constraints. In Michael Kaufmann and Dorothea Wagner, editors, *GD 2006: Proceedings of the 14th International Symposium on Graph Drawing*, volume 4372 of *LNCS*, pages 126–137. Springer-Verlag, 2007.

[14] John Hopcroft and Robert Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974. ISSN 0004-5411.

[15] Daniel Kaiser. Das Links-Rechts-Planaritätskriterium. http://www.inf.uni-konstanz.de/algo/lehre/ws08/projekt/ausarbeitungen/kaiser.pdf.

[16] Casimir Kuratowski. Sur le probleme des courbes gauches en topologie. *Fund. Math.*, 15:271–283, 1930.

[17] Erkki Mäkinen. How to draw a hypergraph. *International Journal of Computer Mathematics*, 34:177–185, 1990.

[18] Kurt Mehlhorn and Petra Mutzel. On the embedding phase of Hopcroft and Tarjan planarity testing algorithm. *Algorithmica*, 16:233–242, 1996.

[19] Petra Mutzel. Zeichen von Diagrammen - Theorie und Paxis.

[20] Helen C. Purchase. Which aesthetic has the greatest effect on human understanding? In *Proceedings of Graph Drawing Symposium, Di Battista, G. (ed)*, volume 1353 of *LNCS*. Springer Verlag, 1997.

[21] Georg Sander. Layout of directed hypergraphs with orthogonal hyperedges. In *GD 2003: Proceedings of the 11th International Symposium on Graph Drawing*, volume 2912 of *LNCS*, pages 381–386. Springer-Verlag, 2004. ISBN 978-3-540-20831-0.