Universität Konstanz
FB Informatik und Informationswissenschaft

Bachelor-Studiengang Information Engineering

Bachelorarbeit

Implementation und Animation des Links-Rechts-Planaritätstests

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

von

Daniel Kaiser

(Matrikelnummer: 01/640934)

Studienfach:	Information Engineering
Schwerpunkt:	Informatik
Themengebiet:	Angewandte Informatik

Erstgutachter:	Prof. Dr. Ulrik Brandes
Zweitgutachter:	Dr. Sven Kosub
Betreuer:	Prof. Dr. Ulrik Brandes
Einreichung:	19. Oktober 2009

Zusammenfassung

Diese Bachelorarbeit beschäftigt sich mit dem Links-Rechts-Planaritätstest, einem noch relativ unbekannten Algorithmus, welcher in linearer Zeit einen Planaritätstest durchführt und die planare Einbettung des getesteten Graphen bestimmen kann. Dieser Algorithmus läuft sehr schnell und ist im Vergleich zu anderen linearen Planaritätstestalgorithmen leicht verständlich. Vor allem die Einbettung ist sehr einfach und kommt mit den aus dem Test verwendeten Datenstrukturen aus.
Neben einer Einführung, welche die Grundlagen erklärt und der Herleitung des Links-Rechts-Planaritätskriteriums, welchem der Links-Rechts-Planaritätstest zu Grunde liegt, wird hauptsächlich auf den Algorithmus selbst, eine eigene Implementierung mit C++ unter Verwendung der Algorithmenbibliothek LEDA und auf ein selbst geschriebenes Animationsprogramm eingegangen. Dieses Animationsprogramm ermöglicht das schrittweise Durchgehen der 2. Phase des Algorithmus, während die einzelnen Schritte visualisiert werden. Eine Erklärung der Implementierung dieses Programms ist ebenfalls Teil dieser Arbeit. Die Ausführungen zum Planaritätskriterium, zum Algorithmus und die Implementierung des Algorithmus basieren auf [2].[1]

[1] Ebenfalls auf [2] aufbauend wurden eine Implementierungen in Java mit yFiles [4] von Martin Mader entwickelt.

Inhaltsverzeichnis

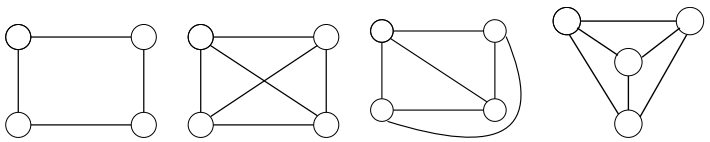
1 Einführung

1.1 Was ist Planarität?

Planarität ist eine Grapheneigenschaft. Sie ist folgendermaßen definiert:

Definition 1 *Ein Graph ist dann planar, wenn er sich so zeichnen lässt, dass sich keine Kanten des Graphen schneiden.*

Abbildung 1(a) ist demnach ein Beispiel für einen planaren Graphen. Man sieht, dass sich keine Kanten schneiden.
Auch Abbildung 1(b) entspricht den Forderungen der Definition. Im Graphen schneiden sich zwar zwei Kanten, er lässt sich jedoch so zeichnen, dass es keine Kantenschnitte gibt. Dies zeigt Abbildung 1(c). Man kann eine Kante, die in der Mitte war, außen herum zeichnen. Ebenso zeigt Abbildung 1(d) den gleichen Graphen. Er ist wie der in Abbildung 1(c) so gezeichnet, dass man sofort sehen kann, dass er planar ist. Seine Kanten sind jedoch alle gerade gezeichnet. Jeder Graph, der planar ist, kann so gezeichnet werden, dass alle Kanten gerade sind.



(a) offensichtlich planar (b) planar? (c) planar! (d) gerade Kanten

Abbildung 1: planar

Satz 1 *Eine Zeichnung eines planaren Graphen, bei der sich keine Kanten schneiden und die Informationen, die man benötigt um solch eine Zeichnung anzufertigen, bezeichnet man als planare Einbettung des Graphen.*

Als Information, die man dafür benötigt, genügt es für jeden Knoten zu wissen, in welcher Reihenfolge man die zu ihm inzidenten Kanten zeichnen muss. Abbildung 2 zeigt eine Einbettung von Kanten um einen Knoten. Solange jede Kante in Richtung des grünen Pfeils die gleiche Nachfolgekante hat, handelt es sich um die gleiche Einbettung.
Die Planarität findet Anwendung zur besseren Visualisierung von Netzwerken (z.B. U-Bahn- oder Schaltpläne).

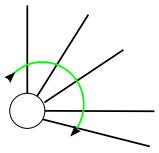


Abbildung 2: Einbettung

1.2 Der Satz von Kuratowski

Beispielhaft soll nun ein anderes Planaritätskriterium[2] - der Satz von Kuratowski - erwähnt werden, um später Vergleiche ziehen zu können.

Satz 2 *Ein endlicher Graph ist genau dann planar, wenn er keinen Teilgraphen enthält, der durch Unterteilung*[3] *von* K_5 *oder* $K_{3,3}$ *entstanden ist. [5]*

K_5 (Abbildung 3) und $K_{3,3}$ (Abbildung 4) sind die kleinsten nicht planaren Graphen. Ein Graph ist offensichtlich nicht planar, wenn er einen der kleinsten nicht planaren Graphen enthält. Dieses Planaritätskriterium ist leicht zu verstehen. Ein Algorithmus für den Planaritästest und die planare Einbettung, welcher dieses Kriterium verwendet, ist allerdings sehr schwer zu implementieren.

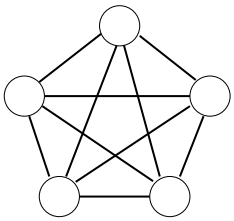


Abbildung 3: K_5

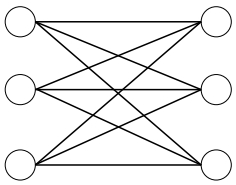


Abbildung 4: $K_{3,3}$

1.3 LR-Planaritätskriterium - Entstehung und Motivation

Das Links-Rechts-Planaritätskriterium wurde von Hubert de Fraysseix, Pierre Rosenstiehl und Patrice Ossona Mendez entworfen. Es liefert einen Algo-

[2]Ein Planaritätskriterium ist eine Menge von Regeln, die genau dann gelten muss, wenn ein Graph planar ist. Es ist also eine Charakterisierung von planaren Graphen.

[3]Unterteilung bedeutet hier das beliebig oft wiederholbare (auch nullmalige) Einfügen von neuen Knoten auf Kanten.

rithmus, der in Linearzeit einen Planaritätstest durchführt und eine planare Einbettung berechnet. Es gibt zwar andere Algorithmen, die auf anderen Planaritätskriterien basieren und ebenfalls in linearer Zeit laufen; der Algorithmus, dem das Links-Rechts-Planaritätskriterium von de Fraysseix zu Grunde liegt, ist aber der schnellste zur Zeit bekannte Algorithmus für den Planaritätstest und die planare Einbettung. Im Gegensatz zum oben vorgestellten Satz von Kuratowski, lässt sich der auf dem LR Kriterium aufbauende Algorithmus relativ leicht implementieren. Trotz dieser Vorteile wird dieser Algorithmus kaum verwendet. Dies liegt daran, dass es nur eine unzureichende Dokumentation und nur eine bekannte Implementierung gibt. Aus diesem Grund wurde [2] geschrieben. Dieses Paper erklärt das Planaritätskriterium und den Algorithmus verständlich. Diese Arbeit baut darauf auf.

2 Das LR-Planaritätskriterium

Mit Hilfe eines beispielhaften planar eingebetteten Graphen (Abbildung 5) soll im folgenden das LR-Planaritätskriterium hergeleitet werden.

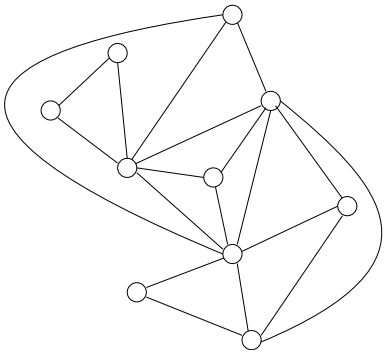


Abbildung 5: planar eingebetteter Graph

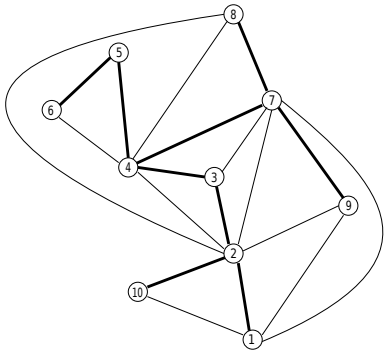


Abbildung 6: Tiefensuchbaum

Zur Vereinfachung werden ohne Einschränkung der Allgemeinheit nur Graphen mit folgenden Eigenschaften betrachtet:

- einfach (simpel)
- ungerichtet
- zusammenhängend

Diese Einschränkung hat keinen Einfluss auf die Planarität. Einfache Graphen haben weder Loops (Kanten von einem Knoten zu sich selbst) noch Mehrfachkanten. Loops und Mehrfachkanten muss man für die Planarität nicht beachten, da man Loops beliebig klein zeichnen kann. Das heißt man kann eine Kante immer um einen Loop herum zeichnen. Mehrfachkanten kann man beliebig nahe nebeneinander zeichnen. Das bedeutet, wenn man eine Kante von einem Knoten zu einem anderen schnittfrei zeichnen kann, kann man dies auch mit mehreren. Die Richtung von Kanten macht für ihr Schnittverhalten offensichtlich nichts aus. Es genügt zusammenhängende Graphen zu betrachten, weil ein Graph dann planar ist, wenn alle seine Zusammenhangskomponenten planar sind. Es gibt zwischen Zusammenhangskomponenten keine Kanten, also können sich zwischen Zusammenhangskomponenten auch keine Kanten schneiden.

2.1 Tiefensuche & LR-Planarität

Das LR-Planaritätskriterium benötigt die Tiefensuche. Die Tiefensuche liefert einen *aufspannenden Baum*. Wenn man eine Tiefensuche über den gewählten Beispielgraphen laufen ließe, könnte diese das in Abbildung 6 gezeigte Ergebnis liefern, wobei die fett gezeichneten Kanten die des Tiefensuchbaumes sind und die Knotennummerierung die Tiefensuchdurchlaufreihenfolge der Knoten aufzeigt.
Mit dem Tiefensuchbaum liefert die Tiefensuche auch eine *Tiefensuchorientierung*. Diese entsteht dadurch, dass jede Kante im Graphen nach ihrer Durchlaufrichtung orientiert wird. (Die Kanten waren davor auf Grund der durchgeführten Vereinfachung ungerichtet.) Abbildung 7 zeigt die durch die vorhin durchgeführte Tiefensuche entstehende Tiefensuchorientierung des Beispielgraphen. Die Kanten des aufspannenden Baumes, die in Abbildung 7 solid gezeichnet sind, bezeichnet man als *Baumkanten*, die übrigen als *Rückwärtskanten*. Diese sind in der Zeichnung gepunktet dargestellt. Die Kanten eines Graphen $G = (V, E)$ kann man also in Baumkanten und Rückwärtskanten einteilen.

$$E = T \uplus B \ ^{4}$$

Hierbei steht das T (wie tree egde) für die Baumkanten und das B (wie back edge) für die Rückwärtskanten.
Da der Tiefensuchbaum ein maximal zykelfreier Teilgraph des betrachteten Graphen ist, führt das Hinzufügen jeder Kante zu einem Zykel. Daraus folgt, dass jede Rückwärtskante einen Kreis im Graphen induziert. Weiter gilt, dass

[4] $\uplus$ bedeutet Vereinigung zweier disjunkt partitionierter Mengen; d.h. kein Element der einen Menge ist in der anderen und umgekehrt

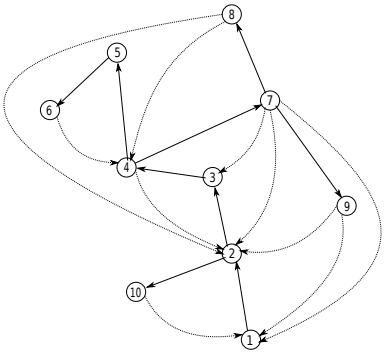


Abbildung 7: Tiefensuchorientierung

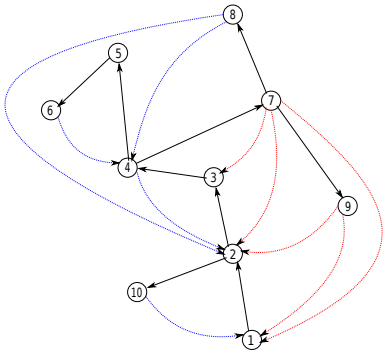


Abbildung 8: Einteilung der Kreise

die einzigen in einem Graphen vorkommenden, gerichteten Kreise von genau einer Rückwärtskante induziert werden.
Daraus kann schon eine wichtige Erkenntnis für die Planarität gezogen werden. Man sieht, dass bei dem angegebenen Beispielgraphen folgendes gilt:

- Jeder Kreis im Graphen ist entweder mit oder gegen den Uhrzeigersinn orientiert.
- Zwei Kreise, die eine Kante teilen, sind gleich orientiert, genau dann wenn einer innerhalb des anderen liegt.

Die Richtigkeit der ersten Aussage ist leicht ersichtlich. In Abbildung 8 sind die Rückwärtskanten der mit dem Uhrzeigersinn orientierten Kreise des Beispielgraphen rot gefärbt, diejenigen Rückwärtskanten, die zu gegen den Uhrzeigersinn laufenden Kreisen gehören, blau. Abbildung 8 genügt offensichtlich auch der 2. Forderung. Vom Beispielgraphen ausgehend kann man verallgemeinernd jede Tiefensuchorientieung eines beliebigen planar eingebetteten Graphen in Betracht ziehen. Auch für diese müssen die beiden Aussagen gelten.
Mit Hilfe dieses Wissens kann man nun ein intuitives Planaritätskriterium formulieren:
Ein Graph ist dann planar, wenn sich seine Kreise so in die Gruppen

- *mit dem Uhrzeigersinn*
- *gegen den Uhrzeigersinn*

einteilen lassen, dass die Kreise in den beiden Gruppen schnittfrei schachtelbar sind.

2.2 LR Partitionierung

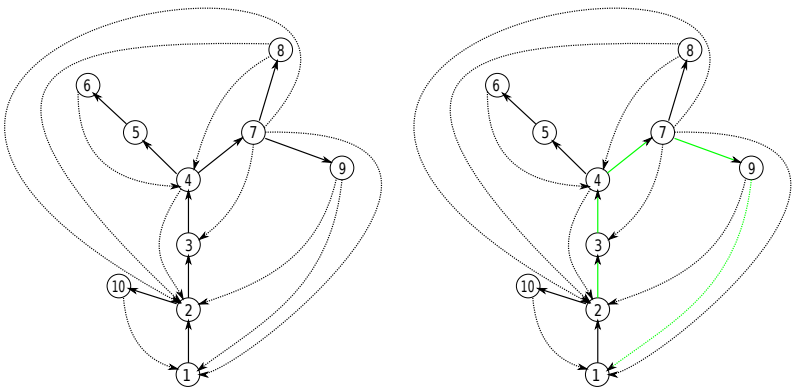
In diesem Abschnitt soll folgende Leitfrage beantwortet werden:
Wann ist es bei einem gegebenen Graphen möglich, die Kreise auf diese Weise anzuordnen? Das Ziel ist es, das Planaritätskriterium zu formulieren. Es kann an dieser Stelle noch eine weitere Vereinfachung vorgenommen werden. Da jede Rückwärtskante genau einen Kreis induziert und jeder Kreis somit genau eine Rückwärtskante enthält, genügt es, statt der Kreise die Rückwärtskanten zu betrachten. Diese werden in die Gruppen linke und rechte eingeteilt. In Abbildung 8 kann man erkennen, dass die Kreise im Uhrzeigersinn von Rückwärtskanten die rechts liegen induziert werden (rot), die Kreise, die gegen den Uhrzeigersinn laufen von Rückwärtskanten die links liegen (blau).

$$B = L \uplus R$$

Hierbei steht B (wie back edge) für die Rückwärtskanten, L für die Menge der linken und R für die Menge der rechten Rückwärtskanten. Eine solche Einteilung der Rückwärtskanten nennt man Links-Rechts-Partitionierung (kurz LR-Partitionierung). Die Leitfrage kann nun auf die Rückwärtskanten angepasst werden:
Wann kann man die Rückwärtskanten in einem gegebenen Graphen so in linke und rechte einteilen, dass keine Kreuzungen entstehen? D.h. wann gibt es eine nicht kreuzende LR-Partitionierung? Um diese Frage beantworten zu können, und damit das Links-Rechts-Planaritätskriterium zu formulieren, werden noch ein paar Definitionen benötigt. Zu den Definitionen sollen auch formale Schreibweisen eingeführt werden.
Zur Erklärung soll der schon oben eingeführte, planar eingebettete Beispielgraph verwendet werden. Zum besseren Erkennen der folgenden Definition wird er übersichtlicher dargestellt (Abbildung 9(a)).
Folgende Bezeichnungen sollen definiert werden:

Baumkanten und Rückwärtskanten:
Baumkanten und Rückwärtskanten wurden schon definiert. Die Schreibweise für eine Baumkante $(u, v) \in T$ ist $u \to v$, für eine Rückwärtskante $(v, w) \in B$ ist sie $v \hookrightarrow w$.

Baumpfad:
Ein Baumpfad von v nach w ist ein (gerichteter) Pfad von u nach w, der nur Baumkanten verwendet. Für die reflexive und transitive Hülle von $\to$ wird $\to^*$ geschrieben. Damit kann dieser Baumpfad als $u \to^* w$ geschrieben



(a) Beispielgraph

(b) Beispiel Rückkehrpfad, Rückkehrkante, Rückkehrpunkt

Abbildung 9: Beispielgraph

werden.

(strikt) höher und (strikt) tiefer:
Ein Knoten $v \in V$ ist tiefer als ein Knoten $u \in V$, wenn es einen Baumpfad $u \rightarrow^* v$ von u nach v gibt. Der Knoten v ist in diesem Fall höher als u. Strikt höher und strikt tiefer setzen einen Baumpfad $u \rightarrow^+ v$ voraus. Strikt, setzt also zusätzlich voraus, dass u und v verschieden sind.

Rückkehrpfad, Rückkehrkante und Rückkehrpunkt:
Als mit $q \in (V \cup E)$ startenden Rückkehrpfad bezeichnet man einen Baumpfad, der mit q startet gefolgt von einer Rückwärtskante deren Zielknoten strikt tiefer liegt als q, falls $q \in V$ und strikt tiefer als der Quellknoten von q, falls $q \in E$. Ein Rückkehrpfad von u nach w über v $(u, w, v \in V)$ wird also $u \rightarrow^* v \hookrightarrow w$ geschrieben, wobei w tiefer als u liegen muss. Eine Rückkehrkante von $q \in (V \cup E)$ ist eine Rückwärtskante, die Teil eines Rückkehrpfades ist, der mit q beginnt. Ein Rückkehrpunkt ist der Zielknoten einer Rückkehrkante. In Abbildung 9(b) ist der grün eingezeichnete Pfad ein Rückkehrpfad von Knoten 2 über Knoten 9 nach Knoten 1. Knoten 1 liegt strikt tiefer als Knoten 2. Die Kante $9 \hookrightarrow 1$ ist die Rückkehrkante und Knoten 1 der Rückkehrpunkt.

lowpoint und lowpoint2:
Der lowpoint von $q \in (V \cup T)$ ist der tiefste über einen mit q startenden Rückkehrpfad erreichbare Knoten. Der lowpoint2 von $q \in (V \cup T)$ ist der zweit tiefste, vom lowpoint verschiedene über einen mit q startenden Rück-

kehrpfad erreichbare Knoten. Wenn der lowpoint oder lowpoint2 undefiniert ist, so ist er q falls $q \in V$ oder u, falls $q = (u, v) \in T$. Bei einer Rückwärtskante ist der lowpoint ihr Zielknoten und der lowpoint2 ihr Quellknoten. In Abbildung 9(a) ist der lowpoint von Knoten 3 Knoten 1. Der lowpoint 2 von Knoten 3 ist Knoten 2.

von x begrenzt:
Eine Rückkehrkante ist von x begrenzt, $x \in V$, wenn ihr Zielknoten höher liegt als x. In Abbildung 10 sind die vom Knoten 1 begrenzten Rückkehrkanten von Knoten 3 grün gefärbt. Knoten 3 hat keine von Knoten 2 begrenzten Rückkehrkanten, da alle seine Rückkehrkanten höchstens bei Knoten 2 zurückkehren dürfen.

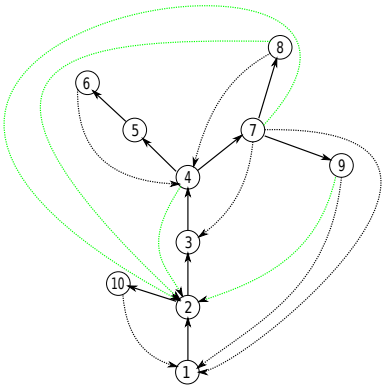


Abbildung 10: Beispiel: von x begrenzt

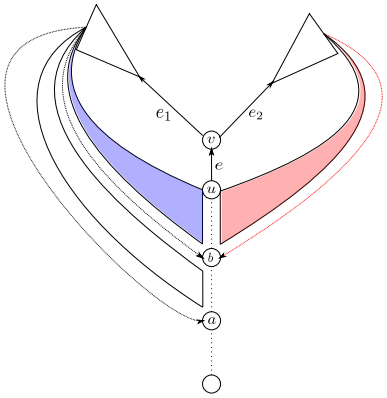


Abbildung 11: nicht kreuzende LR Partitionierung

Mit diesen Definitionen lässt sich die nicht kreuzende LR-Partitionierung formal definieren.

Satz 3 *Eine LR-Partitionierung ist genau dann nicht kreuzend, wenn für jedes Kantenpaar $e_1, e_2 \in E$ welches von einem Knoten $v \in V$ ausgeht gilt:*

1. *die vom lowpoint von e1 begrenzten Rückkehrkanten von e2 sind in einer Klasse*
2. *die vom lowpoint von e2 begrenzten Rückkehrkanten von e1 sind in der anderen Klasse*

Damit lässt dich das Links-Rechts-Planaritätskriterium formulieren:

Satz 4 *Ein Graph ist genau dann planar, wenn es eine nicht kreuzende Links-Rechts-Partitionierung seiner Rückwärtskanten gibt.*

Zur Veranschaulichung der beiden Bedingungen soll Abbildung 11 dienen. Sie zeigt Pakete von Rückkehrkanten aus den Unterbäumen von e_1 und e_2. Der lowpoint von e_1 ist a, der von e_2 ist b. Da b höher liegt als a, müssen nach der ersten Regel alle Rückkehrkanten von e_2 auf eine bestimmte Seite. Sie sind in Abbildung 11 rot gefärbt. Die Kanten, die nach der zweiten Regel in die andere Klasse müssen, sind blau gefärbt. Alle Kanten die nicht gefärbt sind, können beim Betrachten von e_1 und e_2 gemäß beiden Regeln auf eine wählbare Seite.

Definition 2 *Eine Konfliktkante $f \in E$ von $e \in E$ ist eine Kante, die nicht auf der gleichen Seite wie e angeordnet werden darf.*

In dem Beispielgraphen sind folglich alle roten Kanten Konfliktkanten jeder blauen Kante und umgekehrt.

Definition 3 *Eine Rückwärtskante $e' \in B$ heißt lowpoint-Kante einer Baumkante $e \in T$, falls e' zum lowpoint von e zurückkehrt.*

Definition 4 *Eine LR-Partitionierung heißt kanonisch, wenn für alle Baumkanten $e \in T$ gilt, dass ihre lowpoint-Kanten jeweils auf der gleichen Seite sind.*

Satz 5 *Jede nicht kreuzende LR-Partitionierung kann kanonisch gemacht werden, ohne dass Kreuzungen entstehen.*

Dies ist deswegen so, weil für alle lowpoint-Kanten eines Knotens $v \in V$ die gleichen Bedingungen aus Satz 3 gelten.

3 Anwenden des Kriteriums für den Planaritätstest

Das Links-Rechts-Planaritätskriterium ist nun formuliert. Die Leitfragen für den folgenden Abschnitt sind:

1. Wie kann man bei einem gegebenen Graphen testen, ob er dieses Kriterium erfüllt?

2. Wie kann man bei einem gegebenen planaren Graphen mit Hilfe des Kriteriums eine planare Einbettung desselben berechnen?

Ziel ist es, einen Algorithmus zu finden, der in linearer Zeit einen Planaritätstest durchführt und eine planare Einbettung liefert.

3.1 Verschachtelungsreihenfolge der Kanten

3.1.1 Erklärung

Die Verschachtelungsreihenfolge sagt, welche Kanten außerhalb, beziehungsweise innerhalb welcher Kanten liegen, wenn man sie überhaupt schnittfrei auf einer Seite zeichnen kann. Die Verschachtelungsreihenfolge kann man über die lowpoints bestimmen:

Satz 6 *Die Verschachtelungsreihenfolge für 2 Kanten $e_1, e_2 \in E$ ist folgendermaßen definiert:*

1. *e_1 ist außerhalb von e_2, wenn der lowpoint von e_1 tiefer ist als der von e_2.*
2. *e_1 ist außerhalb von e_2, wenn beide den gleichen lowpoint haben, e_2 aber einen lowpoint2 hat.*

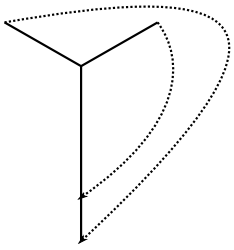


Abbildung 12: Regel 1 Beispiel

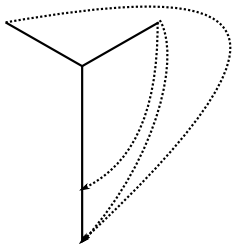


Abbildung 13: Regel 2 Beispiel

Wichtig ist, dass die erste Regel (aus Satz 6) nicht besagt, dass wenn ihre Bedingung erfüllt ist, e_1 immer außerhalb von e_2 schnittfrei gezeichnet werden kann! Sie besagt bei erfüllter Bedingung nur, dass wenn die Kanten e_1 und e_2 schnittfrei gezeichnet werden können, e_1 immer außerhalb von e_2 gezeichnet werden kann. Selbiges gilt auch für die zweite Regel.
Zusammenfassend lässt sich sagen, dass die Verschachtelungsreihenfolge eine Art größer-kleiner Relation auf den Kanten definiert. Diese Relation soll mit

$\prec$ beziehungsweise $\succ$ geschrieben werden. Für die Kanten e_1, e_2 (aus Satz 6) gilt, wenn die Bedingung einer der beiden Regeln erfüllt ist : $e_1 \preceq e_2$. [5]

3.1.2 Planaritätstest

Für den Links-Rechts-Planaritätstest muss nun für alle von einem Knoten $v \in V$ ausgehenden Kanten geprüft werden, ob sie paarweise die oben, für eine nicht kreuzende LR Partitionierung gemachten Bedingungen erfüllen. Würde man die Kanten wirklich paarweise testen, würde das eine quadratische Laufzeit zur Folge haben. Der gesuchte Algorithmus soll aber in linearer Zeit laufen. Möglich ist dies, wenn man die oben definierte Verschachtelungsreihenfolge der Kanten kennt. Dies soll am Beispiel dreier, vom Knoten $v \in V$ ausgehender Kanten $e_1, e_2, e_3 \in E$ gezeigt werden. Die Kanten sollen bereits nach der Verschachtelungsordnung sortiert sein. Es soll $e_1 \preceq e_2 \preceq e_3$ gelten.

Als erstes werden alle Kanten auf der rechten Seite angeordnet. Das hat zur Folge, dass alle Kanten die nach den Regeln für die nicht kreuzende LR-Partitionierung aus Satz 3 keine Konflikte [6] haben, auf der rechten Seite sind. Zuerst werden nun die Kanten e_1 und e_2 betrachtet und auf diese die Regeln (aus Satz 3) angewendet. Aufgrund der Verschachtelungsreihenfolge weiß man nun, dass wenn man die beiden schnittfrei zeichnen kann, e_1 außen liegen muss. Das heißt alle Rückkehrkanten von e_2 müssen nun auf die eine Seite (es soll rechts gewählt werden), da der lowpoint von e_1 tiefer liegt. Entweder gibt es nun Rückkehrkanten von e_1, die vom lowpoint von e_2 begrenzt sind. Diese müssen dann auf die andere Seite (es soll die linke gewählt werden). Oder es gibt keine solche Rückkehrkanten und alle Kanten können rechts bleiben. Als nächstes vergleicht man die Rückkehrkanten von e_3 mit denen von e_1 und e_2. Man betrachtet die Rückkehrkanten von e_1 und e_2 jetzt als einen Block. Da e_3 wegen der Verschachtelungsreihenfolge einen höheren lowpoint als e_1 und e_2 hat, müssen alle Rückkehrkanten von e_3 und diejenigen Rückkehrkanten von e_1 und e_2 welche vom lowpoint von e_3 begrenzt sind, auf verschiedene Seiten. Wenn eine der Rückkehrkanten auf beiden Seiten Konfliktkanten hat, gibt es keine nicht kreuzende LR-Partitionierung. Der Graph ist dann nicht planar. Durch dieses Verfahren kann man eine lineare Laufzeit garantieren, weil für jede Kante nur geprüft wird, ob sie sich mit allen ihren Vorgängern, welche als eine Gruppe gesehen werden, integrieren

[5] Sie können auch den gleichen Platz in der Verschachtelungsreihenfolge haben, denn die Bedingung von Regel eins ist auch wahr, wenn der lowpoint gleich ist (siehe Definition von tiefer).

[6] Eine Kante hat einen Konflikt, wenn es eine andere Kante gibt, mit der sie nicht auf der gleichen Seite angeordnet werden darf (also wenn sie eine Konfliktkante hat).

lässt. Ohne die Verschachtelungsreihenfolge wäre dies nicht möglich. Im oben gemachten Beispiel müsste man dann ebenfalls testen, ob e_3 zwischen e_1 und e_2 oder ganz außen liegen könnte.

Nun soll eine schematische Darstellung der Funktionsweise des Planaritätstest folgen, die auf den gerade gewonnenen Erkenntnissen aufbaut:

- ordne alle Rückwärtskanten rechts an
- sortiere alle Kanten nach der Verschachtelungsordnung
- betrachte alle von einem Knoten v ausgehenden Kanten $e_i \in E_v = \{e_1, ..., e_d\}$
- prüfe ob e_i sich in $e_1, ..., e_{i-1}$ integrieren lässt
- wenn ja mache mit e_{i+1} weiter
- wenn nein, lässt sich keine nicht kreuzende LR-Partitionierung finden

Abbildung 14 soll dies veranschaulichen.

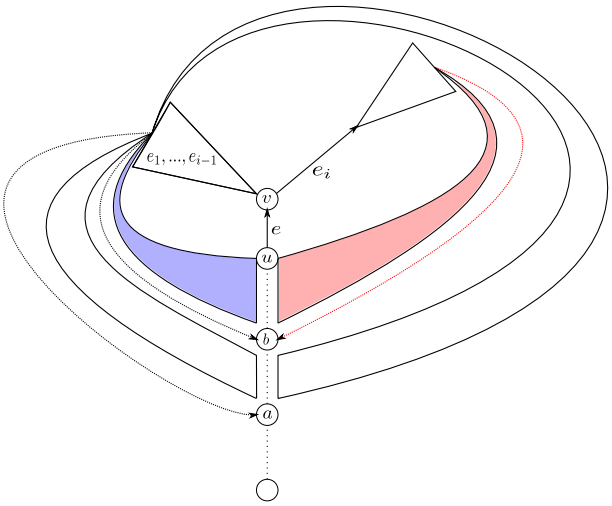


Abbildung 14: Planaritätstest Beispiel

3.1.3 planare Einbettung

Die Wichtigkeit der Verschachtelungsreihenfolge für die planare Einbettung ist intuitiv leicht ersichtlich. Wenn ein planarer Graph gegeben ist, kann mit Hilfe der Verschachtelungsreihenfolge bestimmt werden, welche Kanten außerhalb beziehungsweise innerhalb welcher anderen Kanten gezeichnet werden müssen. Im folgenden soll erklärt werden, wie man mit Hilfe der Verschachtelungsreihenfolge eine formale Einbettung des Graphen generieren kann. Wie in der Einführung bereits erwähnt, benötigt man dafür die Information, in welcher Reihenfolge die zu einem Knoten inzidenten Kanten gezeichnet werden müssen. Ferner soll die LR-Partitionierung zu diesem Zweck um die Baumkanten erweitert werden. Eine Baumkante ist auf der gleichen Seite, wie ihre Rückkehrkanten zu ihrem höchsten Vorfahren. Wenn sie keine Rückkehrkanten hat, kann sie auf eine beliebige Seite[7].

Zuerst sollen nur die von einem Knoten ausgehenden Kanten betrachtet werden. In Abbildung 15 sieht man einen Teil[8] des oben verwendeten Beispielgraphen. Alle Kanten sind rechts und entsprechend der Verschachtelungsordnung gezeichnet. Zur Veranschaulichung sollen nun die von Knoten 7 ausgehenden Kanten $e_1, ..., e_5$ betrachtet werden. Für diese gilt in Richtung des grünen Pfeils die oben definierte Relation $\prec$[9]. Wie man sehen kann, sind aber noch rechte(rot) und linke(blau) Kanten auf der rechten Seite. Um dies zu vermeiden, müssen - bildlich erklärt - lediglich die linken Kanten auf die linke Seite gebracht werden [10] (Abbildung 16). Wie man sieht, ist die Reihenfolge der von Knoten 7 ausgehenden Kanten in Richtung des grünen Pfeils nun die, in der diese für die planare Einbettung gezeichnet werden müssen.
Formal kann nun aus der Verschachtelungsreihenfolge die Zeichenreihenfolge generiert werden, in der die von einem Knoten ausgehenden Kanten gezeichnet werden müssen. Dazu soll die Verschachtelungsreihenfolge in ausgehende linke $e_1 \preceq \cdots \preceq e_l$ und ausgehende rechte Kanten $e_1^{'} \preceq \cdots \preceq e_r^{'}$ unterteilt werden. Die gesuchte Zeichenreihenfolge ist nun folgende: $e_l, ..., e_1, e_1^{'}, ..., e_r^{'}$.
Dies soll am am Beispiel der Abbildungen 15 und 16 veranschaulicht werden. Die Kanten e_3 und e_4 müssen auf die linke Seite, weswegen sie für die Zeichenreihenfolge in umgekehrter Reihenfolge vor die Kanten gestellt werden,

[7] In Abbildung 16 ist die Kante e_2 auf der rechten Seite, weil die Rückkehrkante $9 \hookrightarrow 2$, welche die Rückkehrkante zu ihrem höchsten Vorfahre ist, auf der rechten Seite ist. Sie ist deshalb auch rot gefärbt. Analoges gilt für die Kante e_4. Da sie auf die linke Seite muss, ist sie blau gefärbt.

[8] Im folgenden sollen speziell Knoten 7 und seine Rückkehrpfade betrachtet werden.

[9] Es gilt also $e_1 \prec e_2 \prec e_3 \prec e_4 \prec e_5$

[10] Welche Kanten linke und welche rechte sind, ist aus dem Test, der im Abschnitt davor erklärt wurde, bekannt.

die auf die rechte Seite müssen (e_1,e_2 und e_4). Diese Kanten müssen also in der Reihenfolge e_4, e_3, e_1, e_2, e_5 gezeichnet werden. Wie man sehen kann entspricht diese genau der Reihenfolge entlang des grünen Pfeils aus Abbildung 16, welche durch den Wechsel der linken Kanten auf die richtige (linke) Seite entstanden ist.

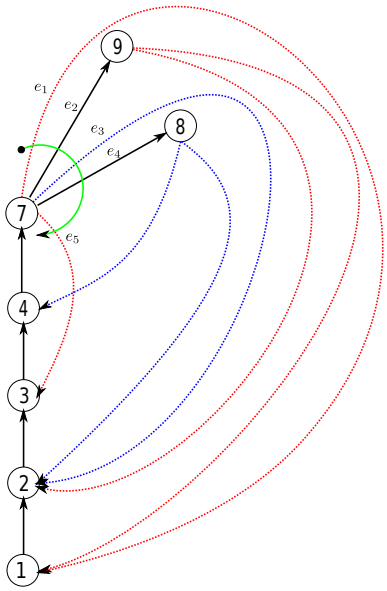


Abbildung 15: Einbettung, alle Kanten rechts

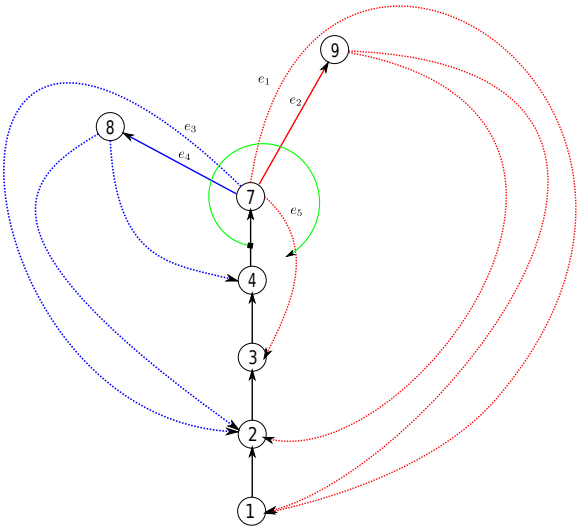


Abbildung 16: Einbettung, blaue Kanten links

Im folgenden sollen zusätzlich die eingehenden Kanten betrachtet werden. Dazu wird ein Knoten $v \in V$ betrachtet. Alle Kanten, die bei v eingehen, müssen Rückkehrkanten aus dem Unterbaum einer von v ausgehenden Kante e_i sein. Auch diese Rückkehrkanten sind in linke und rechte eingeteilt [11]. $L(f)$ bezeichne die linken Rückkehrkanten aus dem Unterbaum einer Kante $f \in E$ welche v als Rückkehrpunkt haben. $R(f)$ soll entsprechend die rechten bezeichnen. Damit keine Schnitte entstehen, müssen leicht einsehbar zuerst die Kanten L(f), dann die Kanten R(f) einer von v ausgehenden Kante $f \in E$ bei v eingehend gezeichnet werden.[12]
Nun ist bekannt, wie sowohl die ausgehenden als auch die eingehenden Kanten jeweils untereinander gezeichnet werden müssen. Um nun eine Zeichenreihenfolge für alle zu einem Knoten inzidenten Kanten zu finden, müssen diese beiden einerseits für die ausgehenden andererseits für die eingehenden Kanten gemachten Zeichenreihenfolgen zusammen geführt werden. Die daraus

[11] Deren Einteilung in linke und rechte wurde bereits durchgeführt
[12] im Uhrzeigersinn, wie der grüne Pfeile bei den Abbildungen 15 und 16

entstehende Reihenfolge soll LR-Reihenfolge genannt werden.

Definition 5 *Sei e die Elternkante eines Knotens $v \in V$ seien $e_1, ..., e_l$ die linken, $e_1', ..., e_r'$ die rechten ausgehenden Kanten des Knotens v, so ist die LR-Reihenfolge der zu ihm inzidenten Kanten:*
$e, L(e_l), e_l, R(e_l), ..., L(e_1), e_1, R(e_1), L(e_1'), e_1', R(e_1'), ..., L(e_r'), e_r', R(e_r')$

Satz 7 *Die LR-Reihenfolge der inzidenten Kanten für jeden Knoten ist eine hinreichende Information für die planare Einbettung.*

Es ist zu erkennen, dass die beiden Einzelreihenfolgen, die oben erklärt worden waren, bei der LR-Reihenfolge eingehalten wurden. Abbildung 17 veranschaulicht die LR-Reihenfolge.

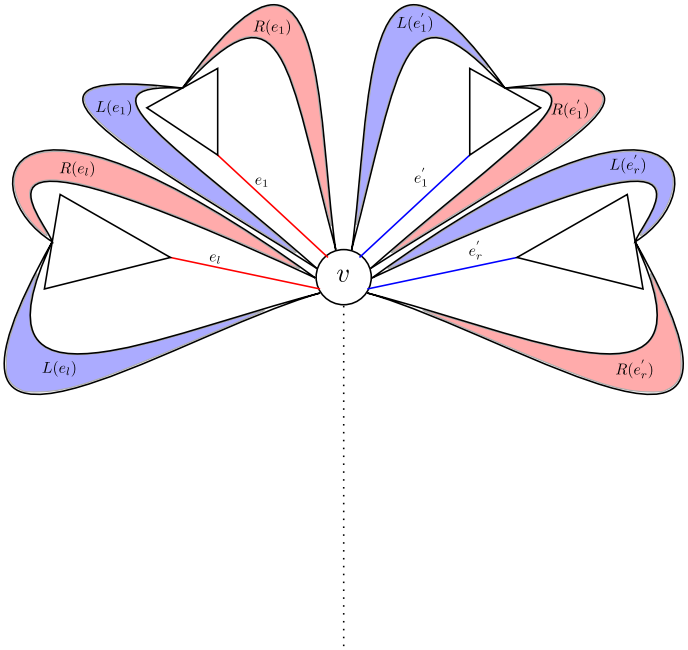


Abbildung 17: Beispiel LR-Ordnung

3.2 Übersicht über den Algorithmus

Der folgende Abschnitt soll eine Übersicht über den Algorithmus von de Frayssaix, der das Links Rechts Planaritätskriterium und damit die oben gemachten Erkenntnisse verwendet um den Planaritätstest und die planare Einbettung durchzuführen. Der Algorithmus ist in drei Phasen eingeteilt,

wobei in jeder Phase eine angepasste Tiefensuche durchgeführt wird. Die Phasen liefern folgende Ergebnisse:

1. Tiefensuche (Phase1):

- Tiefensuch-Orientierung
- lowpoints (lowpoint und lowpoint2)
- Verschachtelungsreihenfolge
- außerdem: Tiefensuchnummer und Elternkante jedes Knotens, Kantenart (Baum- oder Rückwärtskante)

2. Tiefensuche (Phase 2):

- Planaritätstest
- kanonische, nicht-kreuzende LR-Partitionierung (falls der Graph planar ist)
- Referenzkante

3. Tiefensuche(Phase 3):

- Einbettung

3.3 Funktionsweise des Algorithmus

Im folgenden Abschnitt soll anhand vereinfachtem Pseudocode[13] die Funktionsweise des Algorithmus erklärt werden.

3.3.1 Phase 1 - Orientierung

Die erste Tiefensuche liefert, wie in der Übersicht bereits erwähnt, die Tiefensuchorientierung, die lowpoints und die Verschachtelungsordnung. Algorithmus 1 zeigt an welchen Stellen in der Tiefensuche, die entsprechenden Ergebnisse geliefert werden. Wie lowpoints genau bestimmt werden, kann in [2] nachgelesen werden.

[13]Eine implemtierungsnaher Pseudocode kann in [2] gefunden werden.

Algorithm 1: Orientierungstiefensuche

```
Tiefensuchzähler ← 0;
integer Tiefensuchnummer[e], e ∈ E, initialisiere mit 0;
integer lowpt[e], ∀e ∈ E;
integer lowpt2[e], ∀e ∈ V;
bool Baumkante[e], ∀e ∈ E;
Kante Elternkante[v], ∀v ∈ V;
Orientierungstiefensuche(Knoten v) begin
    Tiefensuchzähler ← Tiefensuchzähler + 1;
    Tiefensuchnummer[v] ← Tiefensuchzähler;
    while es gibt nicht orientierte Kanten {v, w} ∈ E do
        orientiere {v, w} als (v, w);
        if Tiefensuchnummer[v] = 0 then
            Baumkante[(v, w)] ← wahr;
            Elternkante[w] ← (v, w);
            Orientierungstiefensuche(w);
            aktualisiere falls nötig lowpt[e], lowpt2[e] und
            Verschachtelungsordnung;
        end
        else //es handelt sich um eine Rückwärtskante
            aktualisiere falls nötig lowpt[e], lowpt2[e] und
            Verschachtelungsordnung;
    end
end
```

Nach dieser Tiefensuche werden die Adjazenzlisten nach der Verschachtelungsreihenfolge sortiert. Damit dies in linearer Zeit (z.B. mit Bucketsort) möglich ist, muss die Verschachtelungsreihenfolge als numerische Zahlenfolge vorliegen. Um jeder Kante eine solche Verschachtelungszahl zuzuordnen, wird die Tiefensuchnummer ihres lowpoints mit 2 multipliziert. Falls die Kante einen lowpoint 2 hat, wird dazu 1 addiert. Nun ist die Kante mit der kleinsten Zahl, diejenige, die außen liegt. Wenn man die Kanten nach diesen Zahlen sortiert, sind sie genau in der oben definierten Verschachtelungsreihenfolge. Im Algorithmus geschieht diese Berechnung an den im Pseudocode angegebenen Stellen der Tiefensuche. Auch diese kann genauer in [2] nachgelesen werden.

3.3.2 Phase 2 - Planaritätstest

Die Hauptdatenstruktur, welche in der zweiten Phase verwendet wird ist ein Stack von Kantenlistenpaaren. Ein Kantenlistenpaar hat eine rechte und ein linke Kantenliste. Die Kantenlistenpaare stellen dabei Konfliktgruppen von Kanten dar.
Es gelten folgende Invarianten:

1. Jede Kante in der linken Kantenliste eines Kantenlistenpaars ist Konfliktkante jeder Kante aus der rechten Kantenliste desselben Kantenlistenpaars.

2. Die Kanten sind in den Kantenlisten entsprechend ihrer Verschachtelungsreihenfolge sortiert. Am Anfang jeder Liste ist die Kante, die am weitesten innen liegt.

3. Oben auf dem Stack ist das Kantenlistenpaar, welches die höchsten Rückkehrkanten hat. Ganz unten im Stack ist das mit der tiefsten.

Da nach der ersten Tiefensuche die Adjazenzlisten nach der Verschachtelungsreihenfolge sortiert worden sind, traversiert die Tiefensuche der zweiten Phase, wenn sie sich bei einem Knoten $v \in V$ befindet zuerst diejenige von v ausgehende Kante, welche nach der Verschachtelungsreihenfolge als nächste kommt. Dies garantiert, wie oben beim schematischen Verfahren schon erwähnt, eine lineare Laufzeit.
Die zweite Tiefensuche wird mit der Wurzel des Tiefensuchbaumes[14] aus der ersten Tiefensuche aufgerufen.

[14] Wenn der eingegebene Graph mehrere Zusammenhangskomponenten hat, müssen in der ersten Phase die Wurzeln der Tiefensuchbäume in einem Array gespeichert werden. Die zweite Tiefensuche wird dann für alle diese Wurzeln aufgerufen.

Algorithm 2: Test-Tiefensuche

```
Stack von Kantenlistenpaaren S;
Hilfs-Kantenlistenpaare P, Q;
TestTiefensuche(Knoten v) begin
    if v ist Wurzel then
        rufe für alle Ziele w ausgehender Kanten von v
        TestTiefensuche(w) auf
    end
    else
        e ← Elternkante[v];
        E_v^+ ← alle von v ausgehenden Kanten;
        foreach e_i ∈ E_v^+ do
            if Baumkante[e_i] then TestTiefensuche(Zielknoten[e_i]);
            else push (∅, ⟨e_i⟩) → S;
            if e_i ≠ e_1 then
                fügehinzu (siehe Algorithmus 3)
            end
        end
        Entferne alle Rückkehrkanten, die höher als als Quellknoten[e]
        zurückkehren;
        Speichere Referenzkante und Seite der entfernten Kanten;
    end
end
```

Die Tiefensuche der zweiten Phase ruft sich rekursiv für jeden Zielknoten von Kanten, die von der Wurzel ausgehen auf. Sie traversiert, wie man am Pseudocode sehen kann, nur ausgehende Kanten.
Für jeden Knoten[15] v mit dem sie aufgerufen wird, merkt sie sich dessen Elternkante e. Nun geht sie alle von v ausgehenden Kanten $e_i \in \{e_1, ..., e_d\}$ durch. (Da die Kanten nach der Verschachtelungsreihenfolge aufgerufen werden, weiß man, dass der lowpoint von e_1 tiefer ist als der von e_2 etc...) Handelt es sich um eine Baumkante, ruft sie sich rekursiv deren Zielknoten auf. Handelt es sich um eine Rückwärtskante wird ein neues Paar von Kantenlisten auf den Stack gelegt, welches links leer ist und rechts als einziges Element diese Rückwärtskante enthält. Dies entspricht dem anfänglichen rechts Anordnen der Rückwärtskanten im schematischen Verfahren.
Die erste Kante, die von v ausgeht (egal ob Baum- oder Rückwärtskante) muss noch nicht auf Konflikte überprüft werden. Alle anderen von v ausgehenden Kanten $(e_2, ..., e_d)$ werden nun mit den bisher behandelten Kanten

[15] außer der Wurzel

integriert.
D.h. es wird geschaut, ob es möglich ist, die aktuelle Kante e_i mit ihrem Unterbaum und den Rückkehrkanten daraus zusammen mit den Kanten $e_1, ..., e_{i-1}$ und deren Unterbäumen und Rückkehrkanten so zu zeichnen, dass es keine Kreuzungen gibt und zu schauen, welche Kanten dafür auf verschiedene Seiten müssen.
Das Hinzufügen der Kante e_i zu den bisher behandelten Kanten $e_1, ..., e_{i-1}$ geschieht im Pseudocode in Algorithmus 3.

Algorithm 3: fügehinzu

```
fügehinzu begin
    repeat
        Q ← pop(S);
        if Q.L ≠ ∅ then vertausche Q.L, Q.R;
        if Q.L ≠ ∅ then NICHT PLANAR!;
        else hänge Q.R an P.R;
    until der Unterbaum abgearbeitet ist ;
    while es gibt Kanten auf S die über lowpt(e_i) zurückkehren do
        Q ← pop(S) ;
        if Kanten in Q.R verursachen Konflikte then vertausche
        Q.R, Q.L;
        if Kanten in Q.R verursachen Konflikte then NICHT
        PLANAR!;
        else
            hänge Q.L an P.L;
            hänge Q.R an P.R;
        end
    end
    if P ≠ ∅ then push P → S
end
```

Das Ziel von fügehinzu ist es ein neues Kantenlistenpaar für die von v ausgehenden Unterbäume zu generieren, das den Forderungen für eine nicht kreuzende LR-Partitionierung genügt oder eine Meldung auszugeben, falls der Graph nicht planar ist. In diesem Kantenlistenpaar sollen rechts alle Rückkehrkanten von e_i sein, die vom lowpoint von e_1[16] beschränkt sind und diejenigen (oder ein Teil von denen), die auf beide Seite dürfen. Auf der linken Seite sollen die Rückkehrkanten von $e_1, ..., e_{i-1}$ welche vom lowpoint von e_i beschränkt werden. Dies kann auch noch einmal an Abbildung 14, welche schon für die schematische Erklärung des Tests verwendet wurde, nachvollzo-

[16]Der lowpoint von e_1 ist der tiefste von $e_1, ..., e_{i-1}$. Er ist also sozusagen der lowpoint dieser Kantenmenge

gen werden. Anhand dieser Abbildung kann auch das folgende nachvollzogen werden.
Das neue Kantenlistenpaar, was erzeugt werden soll, ist im Pseudcode das Hilfskantenlistenpaar P. In der ersten Schleife des Pseudocodes soll die rechte Seite von P nach den eben beschriebenen Bedingungen gefüllt werden. Dazu werden solange Kantenlistenpaare vom Stack genommen bis alle zum Unterbaum von e_i gehörigen Kantenlistenpaare abgearbeitet sind. (Wie man diese Grenze auf dem Stack erkennt wird weiter unten erklärt.) Diese werden in jedem Schleifendurchlauf in der zweiten Hilfsvariablen Q gespeichert. Es wird geprüft, ob die linke Seite frei ist. Wenn dies der Fall ist, werden die Kantenlisten getauscht, weil diese Kanten, wie oben beschrieben, auf die rechte Seite sollen. Ist nun die linke Seite immer noch nicht frei, sind beide Seiten bereits belegt. Ist dies der Fall, ist der Graph nicht planar. Warum dies so ist, soll nun erklärt werden. Zuerst soll klar gestellt werden, warum auf der linken Seite schon Kanten enthalten sein können. Dies liegt daran, dass der Unterbaum von e_i bereits abgearbeitet wurde[17] und dort schon Kanten auf die linke Seite mussten. Wenn beide Seiten schon voll waren, heißt das, dass e_i linke und rechte Rückkehrkanten hat. Da aber e_1 auf jeden Fall tiefer als e_i zurückkehrt, müssen die Rückkehrkanten von e_1 außen liegen. Damit werden sie aber sicher von den linken oder rechten Rückkehrkanten von e_i geschnitten. Dieser Fall wird in Abbildung 18 veranschaulicht. Die beschriebenen Problemkanten sind in Abbildung 18 hell-violett dargestellt.
Wenn die Qs spätestens nach dem ersten Tauschen auf der linken Seite leer sind, werden deren rechte Seiten unten an die rechte Seite von P angehängt. Nach dem die Schleife beendet wurde, ist die rechte Seite des zu erzeugenden Kantenlistenpaares fertig.

In der zweiten Schleife des Pseudocodes wird die linke Seite von P mit den Rückkehrkanten von $e_1, ..., e_{i-1}$, die vom lowpoint von e_i begrenzt werden, gefüllt. Es werden solange Kantenlistenpaare von Stack genommen und in Q gespeichert, bis alle zu $e_1, ..., e_{i-1}$ gehörigen Kantenlistenpaare, welche Kanten enthalten, die höher als der lowpoint von e_i zurückkehren, abgearbeitet sind. (Auch das Finden dieser Grenze auf dem Stack wird unten beschrieben.) Für die Qs wird nun geprüft, ob ihre rechte Seite Kanten enthält, die vom lowpoint von e_i begrenzt werden. Wenn dies der Fall ist, werden die Kantenlistenpaare getauscht. Gibt es nun immer noch Kanten auf der rechten Seite, die vom lowpoint von e_i begrenzt sind, so gibt es diese Kanten

[17] Zum Verständnis: An dieser Stelle befindet sich der rekursive Algorithmus im Backtracking, da der rekursive Aufruf vor fügehinzu stattgefunden hat. Deshalb wurden die Unterbäume der betrachteten Kanten schon behandelt.

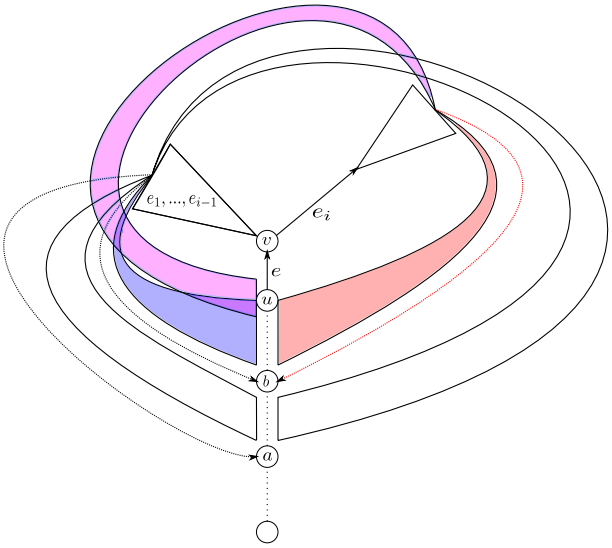
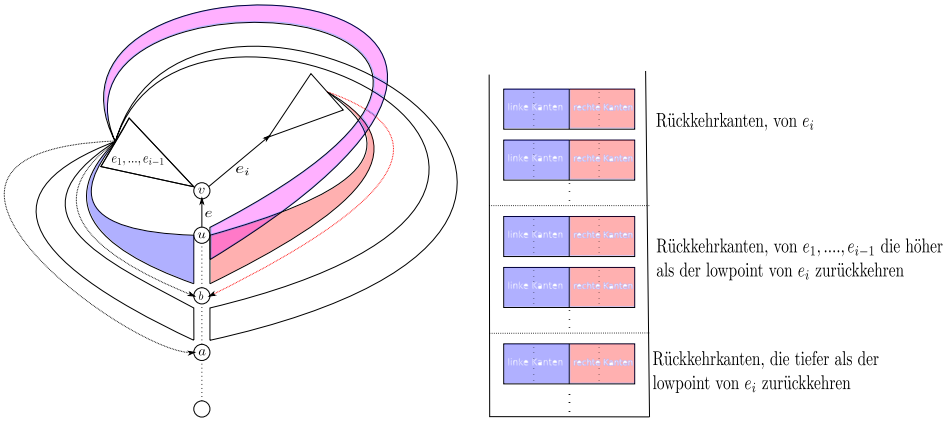


Abbildung 18: Beispiel LR-Ordnung

auf beiden Seiten und der Graph ist nicht planar, da sich in diesem Fall jene Kanten mit den schon rechts befindlichen von e_i schneiden würden. Abbildung 19(a) veranschaulicht diesen Fall. Die beschriebenen Problemkanten sind in Abbildung 19(a) hell-violett dargestellt. (Der Grund dass es überhaupt auf der rechten Seite Kanten geben darf ist gleiche, der oben für die linken beschrieben wurde.)
Nun werden zu dem Hilfspaket P, welches auf der rechten Seite schon die Rückkehrkanten von e_i enthält, auf der linken Seite die linken Seiten der Qs angehängt. Dies sind die blauen Kanten in Abbildung 14. Die rechten Seiten der Qs werden an die rechte Seite von P angehängt. Dies sind die farblosen Kanten in Abbildung 14, also diejenigen, die (noch) keine Konfliktkanten haben. Dieses Kantenlistenpaar P wird, wenn es Kanten enthält, nach der zweiten Schleife auf den Stack gelegt. Es entspricht dann den oben geforderten Bedingungen.
Nachdem fügehinzu ausgeführt wurde, müssen noch diejenigen Kanten aus den Kantenlisten paaren auf dem Stack entfernt werden, die als Zielknoten den Quellknoten von e, der Elternkante von v, haben. Dies ist deswegen notwendig, weil diese Kanten keine Konflikte mehr mit Rückkehrkanten der folgenden Knoten v haben können, da diese Rückkehrkanten immer tiefer zurückkehren. Dies kann man sich auch an Abbildung 14 veranschaulichen. Dort würden alle Kanten, die v als Rückkehrpunkt hätten keine Konflikte mehr machen.



(a) Beispiel LR-Ordnung (b) Stack

Abbildung 19: LR-Ordnung

Beim Entfernen dieser Kanten wird ihre Referenzkante und ihre Seite gespeichert. Diese Information braucht man später in Phase 3 für die Einbettung. Wenn die Seite 1 ist heißt das, dass die Kante auf die gleiche Seite wie ihre Referenzkante muss. Wenn die Seite -1 ist, muss die auf die andere Seite.
Nun soll erklärt werden, wie man die oben erwähnten Grenzen auf dem Stack findet. Abbildung 19(b) veranschaulicht die Grenzen im Stack.
Wichtig dafür ist die Existenz der oben definierten (2. und 3.) Invarianten. Wenn beim oberstern Kantenlistenpaar auf dem Stack die Tiefensuchnummer der obersten Kante links oder rechts kleiner als die Tiefensuchnummer von e_i[18] ist, ist der Unterbaum von e_i abgearbeitet. Es wird also die Tatsache benutzt, dass der Unterbaum von e_i dann abgearbeitet ist, wenn es keine Kanten mehr gibt, die nach e_i durchlaufen worden sind. Die zweite Grenze ist dann erreicht, wenn beim obersten Kantenlistenpaar die höchsten Kanten sowohl rechts, als auch links, tiefer als der lowpoint von e_i zurückkehren.

In der aktuellsten Version von [2] und ebenfalls in der weiter unten vorgestellten Implementierung wurden statt der hier erklärten Kantenlistenpaare Kantenintervallpaare verwendet. Ein Kantenintervall speichert nur die beiden Kanten, die als erste und als letzte Kante in einer Kantenliste wären. Den Kanten dazwischen wird jeweils die Kante, die in der Kantenliste unter ihr wäre, als Referenzkante zugewiesen. Eine neues Kantenlistenpaar, welches anfangs rechts eine Rückwärtskante hält und links leer ist, wird als Kantenintervall, das rechts oben und unten diese eine Rückwärtskante hält und links oben und unten leer ist, realisiert. So wird insgesamt weniger Speicher

[18] Die Tiefensuchnummer einer Kante ist ihre Nummer in der Durchlaufreihenfolge

verbraucht und der Algorithmus läuft schneller. Die im folgenden Abschnitt erklärte dritte Phase geht bereits davon aus, dass Kantenlistenintervalle verwendet wurden und damit die Referenzkanten auf die gerade beschriebene Weise festgesetzt wurden.

3.3.3 Phase 3 - Einbettung

In der Einbettungsphase sollen die Adjazenzlisten der Knoten so sortiert werden, dass sie der oben beschriebenen LR-Reihenfolge (s. Definition 5) entsprechen. Zuerst werden die ausgehenden Kanten in die richtige Reihenfolge gebracht. Dies geschieht vor der eigentlichen Tiefensuche. Unter den rechten Kanten soll die Kante mit der größten Verschachtelungsordnungszahl, welche in Phase 2 berechnet wurde, zuerst kommen. Unter den linken Kanten soll die Kante mit der kleinsten Verschachtelungsordnungszahl die erste sein. Ferner sollen alle linken ausgehenden Kanten vor den rechten sein. Dies ist möglich indem man der Verschachtelungsordnungszahl der linken Kanten ein negatives Vorzeichen gibt. Dazu muss wird sie mit -1 multipliziert.
Um zu bestimmen, auf welche Seite eine Kante gezeichnet werden soll, muss man die in Phase 2 berechnete Referenzkante und das ebenfalls in Phase 2 berechnete Vorzeichen der Kante betrachten. Eine Kante soll auf der gleichen Seite wie ihre Referenzkante gezeichnet werden, sofern das Vorzeichen positiv ist, auf der anderen sofern es negativ ist. Da man aber zuerst wissen muss, auf welche Seite die Referenzkante gezeichnet werden soll, muss man rekursiv durch die Referenzkantenhierarchie gehen, bis man auf eine Kante stößt, welche keine Referenzkante hat. Solch eine Kante ist links, sofern das Vorzeichen negativ ist und anderenfalls rechts. Die Funktion $Vorzeichen(Kante\ e)$ in Algorithmus 4 veranschaulicht das.

Algorithm 4: Reihenfolge für die ausgehenden Kanten

```
for e ∈ E do
Verschachtelung[e] = Vorzeichen(e) · Verschachtelung[e];
Vorzeichen(Kante e) begin
    if Referenzkante[e] ≠ ⊥ then
        Seite[e] ← Seite[e] · Vorzeichen(Referenzkante[e]);
        Referenzkante[e] ← ⊥;
    end
    return Seite[e]
end
```

Danach werden die Adjazenzlisten wie schon vor Phase 2 nochmals nach der - nun um die Vorzeichen erweiterten - Verschachtelungsreihenfolge mit Bucketsort sortiert. Erst nach diesem Schritt wird die dritte Tiefensuche

gestartet, welche die Aufgabe hat, die eingehenden Kanten in diese Ordnung zu integrieren. Algorithmus 5 zeigt die Einbettungstiefensuche[19].

Algorithm 5: Einbettungstiefensuche

```
linkeReferenz[v], ∀v ∈ V;
rechteReferenz[v], ∀v ∈ V;
Einbettungstiefensuche(Knoten v) begin
    E_v^+ ← alle von v ausgehenden Kanten;
    foreach e_i ∈ E_v^+ do
        w ← Zielknoten[e];
        if e_i = Elternkante[w] then //Baumkante
            positioniere die Kante e_i an erster Stelle der Adjazenzliste
            von w;
            linkeReferenz[v] ← e_i;
            rechteReferenz[v] ← e_i;
            Einbettungstiefensuche(w);
        end
        else //Rückwärtskante
            if Seite[e_i] = 1 then
                positioniere e_i direkt hinter rechteReferenz[w] in der
                Adjazenzliste von w;
            end
            else
                positioniere e_i direkt vor linkeReferenz[w] in der
                Adjazenzliste von w;
                linkeReferenz[w] ← e_i;
            end

    end
end
```

Die Einbettungstiefensuche wird mit allen Wurzelknoten aufgerufen und traversiert alle ausgehenden Kanten e_i der Verschachtelungsreihenfolge entsprechend. Falls es sich um eine Baumkante handelt, wird sie in in der Adjazenzliste ihres Zielknotens an erster Stelle positioniert. Zusätzlich wird diese Kante für ihren Quellknoten v als linke und rechte Referenz[20] gespeichert. Damit ist die linke und rechte Referenz eines Knotens v zunächst diejenige von ihm ausgehende Kante e_d, in deren Unterbaum die Tiefensuche sich

[19]Der Pseudocode der Phase 3 wurde nur leicht verändert aus [2] übernommen, da sich eine Kürzung hier nicht lohnen würde, weil die Phase 3 auch implementierungsnah wenig Code enthält.

[20]nicht zu verwechseln mit den Referenzkanten aus der 2. Phase

gerade befindet. Diese Referenz ist für die spätere Positionierung der aus diesem Unterbaum zurückkehrenden Rückwärtskanten wichtig. Wenn e_i eine Rückwärtskante ist, ist garantiert, dass für ihren Zielknoten w die linke und rechte Referenz gesetzt ist, weil der Zielknoten einer Rückwärtskante von der Tiefensuche zuerst durchlaufen wird, da nur ausgehende Kanten traversiert werden. Handelt es sich bei e_i um eine Rückwärtskante wird geprüft, ob die Seite 1 (rechts) oder -1 (links) ist.
Falls sie 1 ist, muss die Kante auf die rechte Seite. Da die Tiefensuchdurchlaufreihenfolge garantiert, dass die erste durchlaufene rechte Kante diejenige ist, welche rechts ganz außen positioniert werden muss, wird diese rechts neben der rechten Referenz von w, welche die von w ausgehende, diesem Unterbaum zugehörige Baumkante e_d ist, gezeichnet. Die nächste rechte Kante ist wieder direkt hinter (rechts von) e_d zu positionieren.
Falls die Seite -1 ist, muss die Kante auf die linke Seite. Für diese Kanten ist garantiert, dass die erste diejenige ist, welche links ganz innen positioniert werden muss. Die nächste ist links neben der letzten durchlaufenen linken, zu w zurückkehrenden Rückkehrkante zu positionieren, weswegen die linke Referenz jeweils auf die gerade durchlaufene linke Rückwärtskante gesetzt wird. Ist der Unterbaum einer von einem Knoten v ausgehenden Kante e_d abgearbeitet, ist die Reihenfolge des entsprechenden Teils seiner adjazenten Kanten wie erwünscht $L(e_d), e_d, R(e_d)$. Wenn der Knoten v komplett abgearbeitet ist, sind seine Kanten in LR-Ordnung:
$e, L(e_l), e_l, R(e_l), ..., L(e_1), e_1, R(e_1), L(e_1'), e_1', R(e_1'), ..., L(e_r'), e_r', R(e_r')$.

4 Implementierung des LR-Planaritätstests

4.1 Einleitung und Übersicht

Wie bereits oben erwähnt wurde für die Implementation die in C++ geschriebene Algorithmenbibliothek LEDA[6] verwendet[21]. LEDA stellt viele Datenstrukturen und Algorithmen bereit, von denen einige aus dem Graphenbereich stammen. Durch diese Datenstrukturen ist es möglich leicht lesbare, pseudocodenahe Programme zu schreiben. In der im folgenden beschriebenen Implementation des Links-Rechts-Planaritätstests werden folgende LEDA-Datentypen verwendet[22].:

[21] Eine LEDA API-Referenz findet sich hier: [7]

[22] Diese Liste soll nur einen groben Überblick über die Datentypen geben. Einige Member und Methoden werden nicht erwähnt. Bei der Erklärung der einzelnen Phasen wird genauer auf bestimmte Anwendungsmöglichkeiten eingegangen. Für eine detaillierte Auflistung der Methoden dieser Datentypen siehe die o.a. API-Referenz [7]

- *graph*, *node* und *edge*:
  graph repräsentiert einen Graphen in LEDA. Er kann gerichtet oder ungerichtet sein. *graph* besteht aus einer Liste von Knoten und einer Liste von Kanten. Die Knoten werden durch den Datentyp *node* repräsentiert, die Kanten durch *edge*. *node* und *edge* sind Pointer auf *node_struct* beziehungsweise *edge_struct*. In LEDA haben die Knoten eines gerichteten Graphen getrennte Adjazenzlisten für ein- und ausgehende Kanten. Dies wird in der 3. Phase eine Rolle spielen. *node_struct* enthält jeweils einen Pointer auf die Anfänge und Enden seiner beiden Adjazenzlisten. Ferner enthält *node_struct* einen Pointer auf den Graphen zu dem der repräsentierte Knoten gehört. Ein Konten ist damit immer genau einem Graphen zugeordnet. *edge_struct* enthält den Start- sowie den Endknoten der repräsentierten Kante. Ferner hält *edge_struct* die zyklischen Vorgänger und Nachfolger der Adjazenzlisten, in denen sich die Kante befindet. Für Graphen gibt es einige Iterationsmakros, mit deren Hilfe der Code übersichtlich bleibt. `forall_nodes(v, G) {}` führt den Block, welcher von den dem Makro folgenden geschweiften Klammern umgeben wird, für jeden Knoten $v \in V$ aus, wobei V die Knotenmege des Graphen sei und die Variable v innerhalb des Blockes verwendet werden kann.

- *node_array* $< E >$
  Diese Datenstruktur bildet die Knotenmenge eines Graphen auf einen Typ E ab. Bei der Initialisierung, die durch den Konstruktor oder die Methode `init()` durchgeführt werden kann, muss der entsprechende Graph angegeben werden. Ferner kann ein Parameter vom Typ E übergeben werden, mit dem dann alle Knoten initialisiert werden. Der Array-Zugriffsoperator wurde überladen, so dass man auf den Knoten v in einem *node_array* A wie bei Arrays gewohnt zugreifen kann: $A[v]$. Der Typ dieses Ausdrucks ist E. Es wird garantiert, dass die Initialisierung in $O(n)$ abläuft. Ein Zugriff wird in konstanter Zeit durchgeführt. Dies ist wichtig um die lineare Laufzeit des Algorithmus auch in der Implementierung garantieren zu können.

- *edge_array* $< E >$
  Analog zu *node_array* $< E >$

- *list* $< E >$
  Es handelt sich um eine doppelt verkettete Liste von items. Jedes item

enthält ein Element des Typs E. Die LEDA Liste hat sehr viele Memberfunktionen. Die o.a. API-Referenz[7] zählt alle auf. In dieser Implementierung wurde $list < E >$ verwendet um Knoten- und Kantenlisten zu bilden. Wie für *graph* gibt es auch für Listen ein Iterationsmakro. Mit `forall(e, L) {}` kann man über alle Elemente e vom Typ E einer Liste L iterieren. Ferner kann man mit `forall_items(it, L) {}` über alle items *it* einer Liste L iterieren.

- *node_list*
  Diese spezielle Knotenliste ist effizienter als $list < node >$. Sie hat aber den Nachteil, dass einerseits jeder Knoten nur in einer *node_list* sein darf, andererseits hat sie deutlich weniger Memberfunktionen als $list < node >$.

- *GraphWin*
  GraphWin repräsentiert ein Fenster, in welchem Graphen gezeichnet werden können. Das *GraphWin* wird in dem hier vorgestellten Programm zur Visualisierung und Animation des Planaritätstests verwendet.

Wenn eigene Datentypen als Element generischer LEDA Collection-Typen verwendet werden sollen, müssen diese bestimmte Voraussetzungen erfüllen. In Phase 2 werden solche eigenen Typen verwendet. Die wichtigste Voraussetzung ist, dass der eigene Typ T einen Copyconstructor besitzt.

$$T :: T(const\ T\&)$$

Dieser wird verwendet um das Element in die Collection einzusortieren. Die Elemente werden also beim Einsortieren kopiert. Zusätzlich sollten die selbst definierten Parametertypen einen Konstruktor ohne Parameter besitzen sowie den Zuweisungsoperator und die beiden Streamoperatoren überladen. In diesem Anwendungsfall waren diese zusätzlichen Bedingungen allerdings nicht nötig.

In LEDA gibt es bereits Implementierungen zweier anderer Planaritätstests. Zum einen der von Hopcroft und Tarjan zum anderen der von Booth und Lueker. Diese beiden Planaritätstests können über die Funktionen `bool HT_PLANAR(graph&, bool embed=false)` bzw. `bool BL_PLANAR(graph&, bool embed=false)` aufgerufen werden. Ferner gibt es eine Funktion `bool PLANAR(graph&, bool embed=false)`, welche den (einkompilierten) Standardplanaritätstest aufruft[23]. Diese Funktionen wurden über das Schlüsselwort `extern` in der Datei plane_graph_alg.h verfügbar gemacht. Diese Datei

[23] standardmäßig der Booth und Lueker Planaritätstest

muss man folglich einbinden, um jene Funktionen verwenden zu können. Da der Rechts-Links-Planaritätstest vielleicht einmal in LEDA integriert werden soll, wurde eine ähnliches Format gewählt. Die Funktion heißt `bool LR_PLANAR(graph& G, bool embed = false, LR_observer OB = nil)`[24]. Sie könnte ebenfalls über `extern` in plane_graph_alg.h verfügbar gemacht werden.
Zur Verwaltung der global benötigten Variablen wurde die Struktur *gblvars* erstellt. Sie wurde aus folgenden Gründen verwendet: Globale Variablen sollten vermieden werden, um thread savety zu gewährleisten. Da in LEDA keine Klassen zum Kapseln der Planaritätstests verwendet wurden, wurde diese Möglichkeit thread savety zu gewährleisten hier ebenfalls nicht verwendet. In den LEDA Implementierungen werden die von (fast) allen Funktionen benötigten Variablen über die Parameterliste mitgegeben. Weil das bei dieser Implementierung jedoch zu viele wären, wird jedes Mal statt all dieser Variablen eine Instanz dieser Struktur übergeben. Diese Vorgehensweise bietet auch für den Observer Vorteile, welche im Abschnitt 5.2.1 erläutert werden.

Das hier vorgestellte Programm besteht aus 6 Quelldateien:

- lr_planar.cpp
  In dieser Datei befindet sich die bereits oben erwähnte Funktion `LR_PLANAR()`. Ferner sind dort die Funktionen, welche die einzelnen Phasen repräsentieren und die von ihnen benötigten Hilfsfunktionen.

- data.h, data.cpp
  Hier sind zum einen die für die Phase 2 benötigten eigenen Datentypen definiert, zum anderen die Struktur, welche die global benötigten Variablen hält.

- LR_observer.h, LR_observer.cpp
  In diesen Dateien befindet sich die Observerklasse `LR_Observer`. Sie wird für die Animation verwendet. In der Obeserverklasse sind auch das Stack- und das Messagewindow definiert.

- main.cpp
  Hier werden das Graphwindow und der Observer für die Animation initialisiert und gestartet. In der Hauptschleife des Graphwindow wird der Links-Rechts-Planaritätstest auf dem eingegebenen Graphen ausgeführt.

[24] Der letzte, optionale Parameter ist für die Animation wichtig und wird weiter unten erklärt.

Im Folgenden wird auf die Punkte eingegangen, die für die Implementierung der einzelnen Schritte des Links-Rechts-Planaritätstests in LEDA wichtig sind. Diese Schritte wurden in Abschnitt 3.3 beschrieben. Wie dort bereits erwähnt, gibt es einen implementierungsnahen Pseudocode in [2]. Wie der Pseudocode sind auch die folgenden Erklärungen zur Implementierung in die entsprechenden Abschnitte aufgeteilt. Weiter soll erklärt werden, warum an laufzeitkritischen Stellen die jeweilige Implementierungsmöglichkeit gewählt wurde. Es wird auch auf strukturelle Unterschiede zwischen Pseudocode und Implementierung hingewiesen, damit es leichter ist, sich im Code zurechtzufinden.
Zwei sofort auffallende strukturelle Unterschiede, welche sich durch alle Phasen ziehen, sind zum einen die Observeraufrufe zum anderen das Referenzieren der vom Algorithmus gebrauchten Variablen über die gblvars Struktur und das Übergeben dieser Struktur bei Funktionsaufrufen.
Folgende Schrifthervorhebungsarten wurden gewählt: `Codefragmente`; *LEDA-Datentypen* oder *Formelfragmente*.

4.2 Hauptfunktion des Algorithmus

Der Name der Hauptfunktion ist aus oben genannten Gründen `LR_PLANAR()`. Wie im Pseudocode dient die Hauptfunktion zum Aufrufen der Funktionen für die einzelnen Phasen[25], zum Richten der Kanten[26] und zum Sortieren der Adjazenzlisten nach der Verschachtelungsordnung [27]. Die Aufrufe der Observerfunktionen werden in Abschnitt 5.2.1 behandelt. Wenn man LR_PLANAR() ohne einen Observer zu übergeben aufruft, werden die entsprechenden Aufrufe nicht ausgeführt, damit LR_PLANAR() wie die oben erwähnten anderen Planaritätstestimplementierungen von LEDA verwendbar ist[28].
Zuerst wird eine Instanz der der Struktur gblvars erstellt und initialisiert, welche die global benötigten Variablen enthält. Am Ende der Funktion werden mit `free_gblvars(gblvars& gbl)` die in der Struktur vorhandenen Knoten- und Kantenlisten geleert. Es ist wichtig, dass dies gemacht wird, solange der Graph, zu dem die Knoten und Kanten in den Listen gehören, noch existiert.

[25] jeweils für jede Wurzel, falls der Graph mehrere Zusammenhangskomponenten hat
[26] nach Phase 1
[27] jeweils nach Phase 1 und Phase 2
[28] Wenn Laufzeittests durchgeführt werden, sollten die Observeraufrufe auskommentiert oder die entsprechenden if-Abfragen durch Präprozessordirektiven ersetzt werden, damit zur Laufzeit nicht jedes Mal überprüft werden muss, ob es einen Observer gibt. In dieser Implementierung wird dies zur Laufzeit geprüft, damit die Funktion möglichst allgemein bleibt.

Versucht man eine Liste zu leeren, deren Knoten oder Kanten zu einem nicht mehr existenten Graphen gehören, führt das zu einem schwer auffindbaren Laufzeitfehler.
Um Mehrfachkanten und Loops zu entfernen muss der Graph mit `G.make_undirected()` ungerichtet gemacht werden, damit danach beim Aufruf von `Make_Simple(G)`[29], falls zwei Kanten $(v, w),\ (w, v) \in E$ zwischen zwei Knoten $v, w \in V$ existieren, eine der beiden gelöscht wird. Danach wird der Graph mittels `G.make_directed()` wieder gerichtet gemacht, damit in Phase 1 eine Tiefensuchorientierung aufgebaut werden kann. Die beiden zusätzlichen Funktionsaufrufe vor und nach dem Aufruf der Einbettungsfunktion, werden im Abschnitt zur Einbettungsphase näher erläutert. Wie man im Profiler-Ergebnis[30] sieht, verbraucht die Sortierung der Adjazenzlisten relativ viel Laufzeit. Auch die Zeit für `leda::node_struct::del_adj_edge` (hauptsächlich) und für `leda::graph::sort_edges` gehören zu Bucketsort. `bucket_sort_edges` fügt alle Kanten des Graphen einer einzigen Liste hinzu. Diese wird dann mit Bucketsort sortiert. Dann werden die Adjazenzlisten aller Knoten gelöscht, weswegen auch auch die von `del_adj_edge` benötigte Zeit hauptsächlich `bucket_sort_edges` zuzuordnen ist. *sort_edges* fügt danach die Kanten aus der sortierten Liste in die entsprechende Adjazenzliste ein. In dieser Implementierung wurde jedoch darauf verzichtet eine eigene Bucketsortmethode zu erstellen[31], da erstens die Lösung alle Kanten in einer Liste zu sortieren und danach wieder auf die einzelnen Adjazenzlisten aufzuteilen schneller ist, als jede Adjazenzliste einzeln zu sortieren. Zweitens wird Anzahl der Buckets durch LEDA auf maximal $2n + 1,\ n = \#V$ eingeschränkt, da das kleinste und größte Element des übergebenen Kantenarrays ($edge\_array < int >$) bestimmt wird und nur Buckets für die beiden Grenzfälle und die dazwischenliegenden Ganzzahlen erstellt werden. Somit ist die Bucketgröße minimal, sodass an dieser Stelle nicht optimiert werden kann.

4.3 Phase 1 - Orientierung

Die Schleife, in der im Pseudocode alle noch nicht orientierten, von einem Knoten v ausgehenden Kanten durchlaufen werden, wird in der Implementierung anders ausgedrückt. Das Iterationsmakro `forall_inout_edges(vw,v)` wird verwendet um über alle Kanten (v, w), die von einem Knoten v ausgehen zu iterieren. Innerhalb dieser Iterationsschleife muss getestet werden, ob die Kante bereits orientiert wurde. Algorithmus 6 veranschaulicht den Aufbau

[29] macht einen Graphen einfach

[30] s. Anhang)

[31] Für diesen Zweck; für die layout()-Funktion wurde eine eigene Bucketsortmethode geschrieben. Warum dies nötig war, wird weiter unten erklärt.

der ersten Tiefensuche der LEDA-Implementation. Die Höhe wurde in der Implementierung mit -1 initialisiert.

Algorithm 6: DFS1 - Struktur der LEDA-Implementierung

```
Kantenliste U
dfs1_impl(Knoten v) begin
    Knoten w
    forall_inout_edges(v, vw) begin
        w ← opposite(v, vw);
        if height[w] < 0 then
            //Baumkante
            if v ist nicht der Startknoten von vw then
                füge (vw) zu U hinzu (Orientierung)
            end
            Elternkante, Lowpoints und Nestingorder
        end
        else if height[w] < height[v] and vw ≠ parentedge[v] then
            //Rückwärtskante
            if v ist nicht der Startknoten von vw then
                füge (vw) zu U hinzu (Orientierung)
            end
            Lowpoints und Nestingorder
        end
    end
end
```

Was bei der Orientierungstiefensuche beachtet werden muss und bereits in Algorithmus 6 angedeutet wurde ist, dass die zu orientierenden Kanten nicht direkt wie im Pseudocode orientiert werden, sondern einer Kantenliste mit umzuorientierenden Kanten hinzugefügt wird. Die Richtung, der sich in dieser Liste befindlichen Kanten, wird in der Hauptfunktion (`LR_PLANAR()`) nach der Orientierungstiefensuche umgekehrt. Dies ist deswegen nötig, weil das Iterationsmakro `forall_inout_edges(vw,v)` nicht erlaubt, dass sich während des Durchlaufs die Adjazenzlisten von v verändern. Durch die Richtungsänderung einer Kante vw, die v enthält, würden sich die Adjazenzlisten von v insofern ändern, als dass vw entweder aus der Adjazenzliste der eingehenden Kanten gelöscht wird und in die Adjazenzliste der ausgehenden Kanten eingefügt wird, oder umgekehrt.

4.4 Phase 2 - Planaritätstest

Die Struktur der Blöcke des Pseudocodes unterscheidet sich auch in Phase 2 von der der Implementierung[32]; jedoch nur in einer Kleinigkeit. In der Implementierung werden die Wurzelknoten getrennt behandelt um für jede Wurzel ein Vergleich zu sparen. Im Pseudocode werden sie wie alle anderen Knoten behandelt, damit der Code kompakter bleibt. Es wird für die Wurzelknoten deswegen zusätzlich geprüft, ob sie Rückkehrkanten haben, was bei zur Wurzel adjazenten, ausgehenden Kanten nie der Fall sein kann. Da der Unterschied nur bei Wurzeln auftritt ist der Laufzeitunterschied nicht messbar. Er wurde nur der Vollständigkeit halber hier aufgeführt.
Für den in dieser Phase verwendeten Stack wurde nicht der nahe liegende LEDA-Datentyp $stack < E >$ verwendet sondern eine Liste. Dies wurde deshalb gemacht, weil für die Animation in jedem Schritt der ganze Stack angezeigt werden soll. Bei einem Stack wäre dies nur möglich, indem man ihn komplett abbaut, den Inhalt zwischenspeichert und ihn dann wieder aufbaut. Es wäre möglich gewesen, es so auf diese Art und Weise zu lösen mit der Begründung, dass die Funktion `LR_PLANAR()` so schnell wie möglich laufen soll und vom Observer so wenig wie möglich beeinflusst werden soll. In dieser Implementierung wurde die Möglichkeit mit der Liste gewählt, da sie bequemer ist und der Observer so oder so die Funktion `LR_PLANAR()` beeinflusst und es deswegen eine Version ohne Observer gibt, die möglichst schnell laufen soll. Diese Version liegt ebenfalls auf der CD bei. Der Stack ist etwas schneller, weil er durch eine einfach verkettete Liste repräsentiert wird. $list < E >$ wird durch eine zweifach verkettete Liste repräsentiert. So spart man sich durch die Verwendung von $stack < E >$ zusätzliche Pointer.

Für die auf dem Stack liegenden Intervallpaare wurden zwei eigene Datentypen `s_interval` und `pair` eingeführt. `s_interval` wurde als Struct, `pair` als Klasse realisiert. Wegen der kürzeren Schreibweise wurde der Typ `interval` definiert, welcher gleichbedeutend mit `struct s_interval` ist. `s_interval` hat zwei Elemente `low` und `high` vom Typ *edge*. Ferner besitzt es einen Copyconstructor und eine Methode `empty()`. Sie gibt true zurück, falls `low` leer ist. Das spart einen Vergleich, da ein Intervall mit einem Element keinen Sinn macht[33]. Die Klasse `pair` hat zwei Member L und R vom Typ `interval`. Sie besitzt einen Copyconstructor und einen Initialisierungskonstruktor, welchem die Kanten, die in den Intervallen gespeichert werden sollen, übergeben werden. Ferner eine Methode `empty()` und eine Metho-

[32] Im oben erklärten vereinfachten Pseudocode (Algorithmus 2) wurde die gleiche Struktur wie in der Implementierung gewählt.

[33] In diesem Kontext gibt es nur endliche Intervalle.

de `reset()`, welche dafür gebraucht wird, die Hilfsintervallpaare Q und P in `add_constraints()` zurücksetzen zu können. Diese beiden Datenstrukturen sind ein Kompromiss aus lesbarem Code und Geschwindigkeit. Ferner sollten sie für diesen Zweck nur das nötigste enthalten. Es sollen noch zwei weitere Möglichkeiten aufgezeigt und mit der verwendeten verglichen werden, von denen eine ohne eigene Datentypen auskommt und die andere nur einen eigenen Datentyp benötigt. Die Methode ohne eigene Datentypen verwendet den LEDA Datentyp $two\_tuple < A, B >$ verschachtelt. Das heißt, ein Intervall wird als $two\_tuple < edge, edge >$ dargestellt, ein Paar als $two\_tuple < two\_tuple < edge, edge >, two\_tuple < edge, edge > >$. Der Vorteil wäre, keine eigene Struktur implementieren zu müssen. Die unschöne, pseudocodeferne Schreibweise mit den verschachtelten Templates könnte man mittels typedef umgehen:

```
typedef two_tuple<edge, edge> interval;
typedef two_tuple<interval, interval> pair;
```

Es gibt jedoch zwei entscheidende Nachteile. Der Zugriff auf die Elemente im Tupel geschieht über die Getter-Methoden $two\_tuple < A, B >:: first()$ und $two\_tuple < A, B >:: second()$. Das entfernt das Codebild weiter vom Pseudocode. Um auf die untere Kante des rechten Intervalls eines Intervallpaares P zugreifen zu können, müsste man `P.second().second()`[34] aufrufen. Bei Verwendung der oben vorgestellten eigenen Datenstruktur wäre der Aufruf `P.R.low`. Damit entspräche er dem Pseudocode und wäre leichter zu lesen. Der zweite große Nachteil ist, dass dem LEDA-Tupel eine `swap()` Methode fehlt. Diese wird aber gebraucht um in `add_constraints()` die Intervalle eines Paares tauschen zu können. Lösen könnte man das Problem, indem man für jedes Tupel ein bool Wert für die Seite mitführen würde. Diese Methode wäre sehr unelegant. Weiter könnte man einen einen neuen Datentyp von $two\_tupel < A, B >$ erben lassen, der die fehlende Funktionalität implementiert. Dies wäre aber fast der gleiche Aufwand wie einen einfachen eigenen Typen zu erstellen. Mit einem eigenen Typen hat man auch den oben beschriebenen Nachteil nicht. Aus diesen Gründen wurde von der Verwendung des LEDA-Typs $two\_tupel < A, B >$ abgesehen. Die zweite Möglichkeit, die ebenfalls mit der gewählten verglichen werden soll, ist die Verwendung eines Arrays mit fünf Elementen, in dem die ersten beiden Elemente den Kanten des einen Intervalls entsprechen, die zweiten beiden Elemente den Kanten des anderen Intervalls. Das fünfte Element enthält ein Side-Flag. Der Vorteil wäre, dass man swap extrem schnell realisieren könnte. Es müsste nur das

[34] Man könnte das Intervallpaar auch anders auf die Tupel aufteilen. Hier soll nur die unschöne Schreibweise gezeigt werden. Auf String-Ersetzung mittels Präprozessordirektiven wurde verzichtet, weil dies sehr schlechter Stil wäre. Es könnte auch an anderen Stellen im Code noch `second` oder `first` vorkommen.

Flag verändert werden. Da man in C++ eigentlich keine Arrays mit verschiedenen Typen in einer Dimension machen kann, könnte man das Flag auch durch den Typ *edge* repräsentieren. Ein Nullpointer würde z.B. heißen, dass die ersten beiden Elemente das linke Intervall repräsentieren, eine beliebige Kante das Gegenteil. Der offensichtliche Nachteil dieser Methode ist die unschöne und nicht abstrakte Zugriffsschreibweise. Ferner bräuchte man eine Wrapperklasse, welche die oben erwähnten Bedingungen erfüllt, damit ein so dargestelltes Paar als Teil einer LEDA-Liste verwendet werden kann. Deswegen wurde auch von dieser Repräsentation des Intervallpaares abgesehen.
Eine weitere Methode, welche wahrscheinlich später in einer noch mehr auf Geschwindigkeit optimierten Version verwendet werden wird, wäre eine Kombination aus der Arraymethode und der verwendeten Klasse `Pair`. Man könnte in der Klasse einen Member `bool side` halten, welcher bestimmt welches der beiden Memberintervalle das rechte bzw. das linke ist. Ferner könnten die Methoden `Pair::getL()` bzw. `Pair::getR()` den Zugriff abstrahieren. Damit wäre `swap()` schnell und die Zugriffsschreibweise akzeptabel.

Für beide eigenen Datentypen wurde das Makro `LEDA_MEMORY(T)` verwendet. Es definiert für einen eigenen Typen T `new` und `delete` um und schaltet für ihn das LEDA-Speichermanagement ein. Der LEDA-Speichermanager reserviert 255 Byte große Speicherblöcke in Listen. Für jeden Aufruf von `new`, teilt er einen vorreservierten Speicherbereich zu, für jeden Aufruf von `delete` gibt er ihn wieder frei. Somit spart man bei vielen kleinen Objekten den Aufwand für das Reservieren und Freigeben kleiner Speicherbereiche, was Rechenzeit spart. *Pair* verbraucht 32 Bytes, *interval* 16 Bytes. Für diese kleinen Objekte, welche oft gelöscht und erzeugt werden, lohnt sich der LEDA-Speichermanager.

4.5 Phase 3 - Einbettung

Das Problem bei der Implementierung der Einbettungsphase ist, dass - wie oben beschrieben - die Knoten getrennte Adjazenzlisten für ein- bzw. ausgehende Kanten haben. In der Einbettungsphase soll aber für jeden Knoten eine Ordnung auf allen, zu einem Knoten adjazenten Kanten bestimmt werden. Aus diesem Grund muss der Graph in der Hauptfunktion `LR_PLANAR()` zunächst bidirektional gemacht werden. Das heißt, dass für jede Kante $e = (v, w);\ v, w \in V$ eine Umkehrkante $e' = (w, v)$ eingefügt wird. Danach muss jeder Kante die Umkehrinformation zugewiesen werden, was bedeutet, dass jede Kante e ihre Umkehrkante e' kennen muss. Wenn diese Umkehrinformation bei einem bidirektionalen Graphen gesetzt ist, wird er Map genannt. In LEDA wird aus einem Graphen G durch die Methode `G.make_bidirected()`

ein bidirektionaler Graph. `G.make_map()` erzeugt aus einem bidirektionalen Graphen eine Map. In dieser Implementierung wurde die überladene Methode `G.make_map(list<edge>& R)` verwendet, welche beide Schritte direkt ausführt und die hinzugefügten Kanten in der Liste R speichert. Diese Liste ist für die Animation wichtig, damit man nach der Einbettungsphase die Umkehrkanten für die Anzeige wieder löschen kann. Dafür gibt es keine spezielle Methode.
Wie in Abschnitt 3.3.3 bereits erwähnt, wurden die von einem Knoten ausgehenden Kanten mittels der Verschachtelungsreihenfolge bereits in die richtige Ordnung gebracht. Innerhalb der Einbettungstiefensuche sollen die eingehenden Kanten in diese Ordnung integriert werden. Deshalb werden nun die Adjazenzlisten, welche die ausgehenden Kanten halten, verwendet. In ihnen sind nun die Umkehrkanten der eingehenden Kanten[35]. Wenn man diese Umkehrkanten der eingehenden Kanten richtig in die Ordnung der ausgehenden Kanten einfügt, befindet sich die für die planare Einbettung gebraucht Information in der Map. Man erhält eine planare Map.
Wegen der getrennten Adjazenzlisten muss man auch innerhalb der Einbettungstiefensuche-Funktion Dinge beachten, die nicht im Pseudocode stehen. Listing 1 zeigt die Implementierung dieser Funktion. Die folgenden Zeilennummern sind auf dieses Listing bezogen.

Listing 1: Phase3 Implementierung

```
1  static void embedding_dfs(node v, gblvars& gbl) {
2    graph& G = *gbl.G;
3    edge ei;
4    forall_out_edges(ei,v) {
5      node w = G.target(ei);
6      edge f = G.reverse(ei);
7      if(ei == gbl.parentedge[w]) {  //real tree edge
8        G.move_edge(f, G.first_out_edge(w), v, leda::before);
9        gbl.rightRef[v] = ei;
10       gbl.leftRef[v] = ei;
11       embedding_dfs(w, gbl);
12     }
13     else if(gbl.height[w] < gbl.height[v]-1) {  //real back edge
14       if(gbl.side[ei] == 1) {
15         G.move_edge(f, gbl.rightRef[w], v, leda::behind);
16       }
17       else {
18         G.move_edge(f, gbl.leftRef[w], v, leda::before);
19         gbl.leftRef[w] = f;
20       }
21     }
22   } //end forall_out_edges(ei,v)
23 }
```

[35] In den Adjazenzlisten, welche die eingehenden Kanten enthalten, sind nun auch die Umkehrkanten der ausgehenden Kanten. Diese werden zwar nicht gebraucht, es ist aber nicht möglich Umkehrkanten nur für eine Art von Kanten zu erstellen.

In Zeile 6 wird zunächst für jede Kante, die von Knoten v ausgeht, die Umkehrkante in f gespeichert. In Zeile 7 wird geprüft, ob es sich bei der Kante e_i um eine wirkliche Baumkante handelt[36]. Ist dies der Fall, wird in Zeile 8 die Umkehrkante f, welche in der Adjazenzliste der ausgehenden Kanten von w die eingehende Elternkante repräsentiert, vor alle wirklichen ausgehenden Kanten geschoben. Zeile 13 prüft, ob es sich bei e_i um eine echte Rückwärtskante handelt. Ist dies der Fall, wird die Umkehrkante f repräsentativ für die Kante e_i in der Adjazenzliste der ausgehenden Kanten von w an der richtigen Stelle eingefügt.

5 Das Animationsprogramm

5.1 Funktionsumfang des Animationsprogramms

Das Animationsprogramm ermöglicht es, die Testphase des oben erklärten Algorithmus schrittweise auszuführen. Wie das Hauptfenster, welches ein LEDA-GraphWin ist, bedient wird, kann in [6] und [7] nachgelesen werden. Hier soll veranschaulicht werden, was mit dem Animationsprogramm möglich ist. Nachdem der Benutzer einen Graphen ausgewählt hat, wird nach der Bestätigung des Benutzers die Orientierungsphase ausgeführt. Danach wird der Graph gelayoutet. Dieses Layout soll dem Layout der oben zur Erklärung verwendeten Beispielgraphen nahekommen. Der Tiefensuchbaum ist deutlich zu erkennen. Die Rückwärtskanten sind kurvenförmig gezeichnet und liegen zu Beginn alle auf der rechten Seite[37]. Dies ist der Ausgangszustand vor der zweiten Phase. Die zweite Phase wird in drei Ebenen eingeteilt. In jeder dieser Ebenen gibt es gewisse Ereignisse. In der ersten Ebene befinden sich die Ereignisse, die die Elternkante e des Knoten v, mit dem die Test-Tiefensuche aufgerufen wurde, betreffen. In der zweiten Ebene befinden sich diejenigen Ereignisse, die die vom Zielknoten der Kante e ausgehenden Kanten e_i betreffen und in Ebene 3 befinden sich die in `add_constraints()` auftretenden Ereignisse. Folgende Übersicht zeigt die einzelnen Ereignisse[38].

Ebene 1:

- Der Unterbaum der Kante e wurde abgeschlossen.

[36] e_i könnte auch eine der Umkehrkanten sein, welche aber nicht beachtet werden sollen.

[37] Das jetzige Layout eignet sich nur für kleinere Graphen. Genaueres zum Layout findet sich in Abschnitt 5.2.3

[38] In einer zukünftigen Version könnte man noch mehr Ereignisse definieren.

- Die zum Quellknoten von zu e zurückkehrenden Kanten wurden getrimmt.

Ebene 2:

- Für die Kante e_i wurde ein neues Intervallpaar auf den Stack gelegt.
- Die Kante e_i wurde samt Unterbaum in $e_1, ..., e_{i-1}$ integriert.

Ebene 3:

- Das oberste, zum Unterbaum von e_i gehörige Intervallpaar wurde vom Stack geholt und in Q gespeichert.
- $Q.L$ und $Q.R$ wurden vertauscht, weil $Q.L$ nicht leer war.
- Die rechte Seite von Q wurde in die rechte Seite von P integriert.
- Der Lowpoint von $Q.R.low$ war gleich dem Lowpoint von e [39].
- Der Unterbaum von e_i wurde abgearbeitet. In den Intervallpaaren auf dem Stack gibt es keine Kanten mehr, welche zum Unterbaum von e_i gehören.
- Das oberste, zum Unterbaum von $e_1, ..., e_{i-1}$ gehörige Intervallpaar wurde vom Stack geholt und in Q gespeichert.
- $Q.L$ und $Q.R$ wurden vertauscht, weil $Q.R$ mit e_i im Konflikt stand.
- Die rechte Seite von Q wurde an die rechte Seite von P angehängt.
- Die linke Seite von Q wurde in die linke Seite von P integriert.
- P wurde auf den Stack gelegt.

Jedes mal, wenn nun eine Bestätigung des Benutzers erfolgt, wird der nächste Schritt ausgeführt, was heißt, dass das Programm bis zum nächsten Auftreten eines dieser Ereignisse läuft und auf erneute Bestätigung wartet. Bei Ereignissen aus der dritten Ebene hält das Programm nur an, wenn es mit dem Parameter -l3 gestartet wurde. In jedem Schritt wird eine Meldung im Messagewindow gezeigt, welche beschreibt, was gerade geschehen ist. Im Stackwindow wird der aktuelle Zustand des Stacks angezeigt und die gerade aktuellen Kanten e und e_i[40]. Falls das Programm mit -l3 gestartet wurde, werden dort auch die von `add_constraints()` benötigten Hilfskantenlistenpaare

[39] wurde kanonisiert

[40] e_i wird nur angezeigt, wenn es im gezeigten Schritt eine Kante e_i gab.

angezeigt. Die Kanten werden im Stackfenster in der Form (<$Quellknoten\text{-}Tiefensuchnummer$>, <$Zielknoten\text{-}Tiefensuchnummer$>) benannt. Die angezeigten Tiefensuchnummern entsprechen bereits den Tiefensuchnummern der 2. Phase. Warum diese Tiefensuchnummern für die Animation bereits vor der 2. Phase berechnet wurden, wird in Abschnitt 5.2.3 erklärt. Das Hauptfenster zeigt den Graphen, in dem die von dem gerade durchgeführten Schritt betroffenen Kanten eingefärbt sind. Hier wurde Colorlinking eingesetzt. Das heißt, die Kante e und die Kanten in P und Q haben im Stackwindow die gleiche Farbe wie im Graphen. Die einzelnen Farben bedeuten folgendes:

- braun : Die aktuelle Kante e
- schwarz: Eine im aktuellen Schritt neu hinzugefügte Rückwärtskante e_i, eine Kante e_i welche kanonisiert wurde. Falls die gerade abgearbeitete Kante e_i eine Baumkante ist, wird sie ebenfalls schwarz gezeichnet.
- rot : Eine Kante welche im aktuellen Schritt auf die rechte Seite gelegt wurde.
- blau: Eine Kante welche im aktuellen Schritt auf die linke Seite gelegt wurde.
- grün: Eine Kante welche im aktuellen Schritt getrimmt wurde.
- grau: Kanten, welche im aktuellen Schritt nicht wichtig waren. Kanten die grau gestrichelt sind, sind Kanten, die nie mehr angefasst werden, also Kanten welche in einem früheren Schritt bereits getrimmt wurden.

Wenn ein Kante im aktuellen Schritt auf die linke Seite gebracht wurde, so wird sie auch im Hauptfenster links herum gezeichnet. Sie bleibt dort links, solange sie sich im Stack auf der linken Seite eines Intervallpaares befindet. Ist der Graph planar, so werden am Ende alle Kanten, welche endgültig nach links kommen links herum gezeichnet und blau eingefärbt. Die anderen sind dann rot eingefärbt und werden rechts herum gezeichnet. Dafür wird die nach Phase 2 berechnete, vorzeichenbehaftete Verschachtelungsordnung verwendet. Es werden auch die Baumkanten dieser Verschachtelungsordnung entsprechend blau (negatives Vorzeichen) oder rot (positives Vorzeichen) gefärbt und in die richtige Reihenfolge gebracht. Die Rückwärtskanten werden jedoch nur entweder ganz links (für linke) oder ganz rechts (für rechte) von den Baumkanten gezeichnet. Diese in der richtigen Reihenfolge zu zeichnen

und damit die vollständige planare Einbettung anzuzeigen, wird in einer späteren Version realisiert werden. Ist der Graph nicht planar, so soll ein im Graphen enthaltener Kuratowski-Untergraph angezeigt werden. Dies ist in der jetzigen Version noch nicht vollständig implementiert, aber bereits in Arbeit.

5.2 Implementierung

Im folgenden soll beschrieben werden, wie das oben erläuterte Animationsprogramm implementiert wurde. Bei der Animation wurde der Laufzeit eine geringere Bedeutung beigemessen. Es war nur wichtig, dass der Benutzer das Programm flüssig bedienen kann.

5.2.1 Observer

Wie schon oben erwähnt, wurde die schrittweise Ausführung des Algorithmus für die Animation mittels eines abgewandelten Obseverpatterns gelöst. Diese Methode wurde gewählt, weil sie ebenfalls bei bereits in LEDA enthaltenen Algorithmenanimationen, bei denen der Algorithmus auch schrittweise ausgeführt werden kann, verwendet wurde. Es handelt sich aus mehreren Gründen nicht um einen klassischen Observer. Es wird kein Objekt sondern eine Funktion[41] beobachtet. Es soll nicht auf Änderungen einer Datenstruktur reagiert werden, sondern auf Zustandsänderungen eines Algorithmus. Weiter ist der Observer nicht generisch, da er nur eine spezielle Funktion[42] beobachten soll. Die Funktion `LR_PLANAR()` kann den Observer nicht im eigentlichen Sinne registrieren. Der Observer wird, falls man eine Animation möchte, als drittes Argument in Form seiner Adresse an `LR_PLANAR()` übergeben. Falls kein Observer übergeben wird, werden die Observerbefehle nicht ausgeführt und die Funktion funktioniert analog zu `HT_PLANAR()` und `BL_PLANAR()`[43]. Bei der Erstellung des Observers in der main()-Funktion muss diesem eine Referenz auf das Graphwindow übergeben werden, damit später der angezeigte Graph auf den aktuellen Zustand angepasst werden kann. Ob die Ereignisse in der dritten Ebene beachtet werden sollen, kann man mittels einer boolschen Variablen, welche dem Observer als zweiter Parameter übergeben wird, bestimmen. Sofern der Hauptfunktion des Algorithmus `LR_PLANAR()` ein Observer übergeben wurde, wird diesem die Adresse der davor erzeugten Instanz der `gblvars` Struktur übergeben. Damit hat der Observer durch die Über-

[41] Es handelt sich eigentlich um eine Menge von Funktionen, da vom Observer auch von `LR_PLANAR()` aufgerufene Funktionen beobachtet werden.

[42] und von dieser Funktion aufgerufene Funktionen

[43] Allerdings - wie oben bereits erwähnt - nicht so schnell wie die ebenfalls mitgelieferte schnelle Version, da oft geprüft werden muss ob ein Observer übergeben wurde.

gabe einer einzigen Variablen Zugriff auf alle für den Algorithmus relevanten Daten. Dies ist ein weiterer Vorteil der gblvars Struktur.
Damit im Stackwindow die aktuellen Kanten e und e_i angezeigt werden können, muss an den entsprechenden Stellen im Algorithmus `LR_Observer::set_e(edge e)` bzw. `LR_Observer::set_ei(edge ei)` aufgerufen werden. `set_e()` kann direkt nach der Definition von e in der 2. Phase aufgerufen werden. `set_ei()` wird als erstes in der Schleife, welche für alle von v ausgehenden Kanten e_i durchlaufen wird, aufgerufen.
Wichtig ist, dass sowohl e als auch e_i vor dem ersten Schritt des Bachtrackings, das heißt direkt nach dem rekursiven Aufruf, nochmals gesetzt werden müssen, da man sonst möglicherweise in den im Backtracking stattfindenden Schritten falsche Kanten e und e_i angezeigt bekäme. Nach der Schleife, in der die Kanten e_i durchlaufen werden, wird `set_ei(nil)` aufgerufen, damit e_i in Schritten, in denen es kein e_i gibt, nicht angezeigt wird.
Für jedes der oben erwähnten Ereignisse gibt es eine Funktion, welche an der entsprechenden Stelle im Algorithmus ausgeführt wird. Da es in Ebene 3 viele Ereignisse gibt, welche viele Gemeinsamkeiten haben, wurde für die dritte Ebene nur eine Funktion erstellt, die einen Parameter des selbst erstellten Typs `level3_states` erwartet. `level3_states` ist eine Enum, welche für jedes Ereignis aus der dritten Ebene eine Konstante definiert.
Nach Eintreten eines jeden dieser Ereignisse, soll das Animationsprogramm den Algorithmus anhalten, um den aktuellen Zustand des Stacks und des Graphen zu zeigen. Durch ein Klicken des `done` Buttons[44] soll der Algorithmus bis zum nächsten Ereignis laufen.
Das Anhalten wird dadurch erreicht, dass am Ende jeder Ereignisfunktion die Funktion `GraphWin::edit()` auf dem bei der Erzeugung des Observers übergeben `GraphWin` aufgerufen wird, welche so lange wartet, bis der done Button geklickt wird. Wenn diese Funktion `false` zurück gibt, wurde das Programm vom Benutzer beendet. Deswegen werden in diesem Fall die vom Observer gehaltenen Ressourcen freigegeben.
In jeder Ereignisfunktion wird das Stackwindow mit Hilfe von `LR_observer::print_stack()` aktualisiert, eine Meldung im Messagewindow mittels `LR_observer::set_message(std::string msg)` ausgegeben und der Graph durch `LR_observer::layout()` neu gelayoutet, da es sein kann, dass während diesem Schritt eine Kante die Seite gewechselt hat. Diese drei Funktionen werden in den folgenden Abschnitten erläutert. Weiter werden in den Ereignisfunktionen die Kanten im Hauptfenster entsprechend eingefärbt und dargestellt. Dazu hält der Observer fünf Kantenlisten. Jeweils eine

[44] Der Name dieses Button wurde gelassen, da er in den mit LEDA mitgelieferten Animationen immer ebenfalls so heißt.

für blaue (`blue`), rote (`red`), schwarze (`black`), gerade getrimmte (`trimmed`) und in einem früheren Schritt getrimmte Kanten (`trimmed_all`). Für jede dieser Listen gibt es eine Funktion, mit der sich Kanten hinzufügen lassen. Diese Kantenlisten werden an den entsprechenden Stellen im Algorithmus gefüllt. Alle neu hinzugefügten Rückwärtskanten werden an die Liste `black` angehängt. Innerhalb von `add_constraints` werden die Kanten der oben aufgeführten Farbübersicht entsprechend zu den Listen für schwarze, rote und blaue Kanten hinzugefügt. Beim Trimming werden die getrimmten Kanten der Liste `trimmed` hinzugefügt. Wichtig ist, dass die Listen nach dem Zeichnen zurückgesetzt werden, damit immer nur die im aktuellen Schritt relevanten Kanten angezeigt werden. Beim Zurücksetzen der Liste `trimmed` werden die darin enthaltenen Kanten der Liste `trimmed_all` hinzugefügt. `trimmed_all` wird nie zurückgesetzt. Die darin enthaltenen Kanten werden grau gepunktet dargestellt. Ferner muss vor dem Löschen der Listen `blue`, `red` und `black` die Funktion `LR_observer::save_side_info()` aufgerufen werden. Diese speichert alle Kanten, welche gerade auf der linken Seite gezeichnet worden sind. Dazu hängt sie alle Kanten aus der Liste für blaue Kanten an eine weitere Kantenliste (`left`); die Liste für Kanten, welche links gezeichnet werden sollen. Ist eine Kante, die im letzten Schritt noch links war aktuell in der Liste der roten oder schwarzen Kanten, so wird sie aus `left` gelöscht. Die Kanten in dieser Liste werden nicht alle eingefärbt. Die weiter unten erklärte layout()-Funktion benötigt diese Liste. Nach sämtlichen Operationen, welche das Aussehen des Graphen im GraphWin verändern sollen, muss `GraphWin::update_graph()` aufgerufen werden, damit die Änderungen im GraphWin angezeigt werden. Falls die gerade abgearbeitete Kante e_i eine Baumkante ist, wird sie ebenfalls schwarz gezeichnet, obwohl sie in keiner dieser Listen ist.

5.2.2 Messagewindow und Stackwindow

Die beiden zusätzlichen Fenster, das Message- und das Stackwindow, wurden mit GTK+ (siehe [1]) und GtkHTML realisiert. Die Entscheidung diese weitere Bibliothek mit einzubeziehen wurde deshalb gefasst, weil es sehr angenehm ist, die Inhalte der beiden Fenster im Observer als HTML-Stream zusammen zu bauen und diesen dann dem Fenster zu übergeben. Die oben bereits erwähnten, in jeder Ereignisfunktion aufgerufenen Funktionen
`LR_observer::print_stack()` und
`LR_observer::set_message(std::string msg)` tun genau dies.

In LEDA wäre es viel umständlicher ein solches Stackwindow zu realisieren[45].
Zuerst müssen mittels `LR_observer::init_gtk_threads()` GTK-Threads aktiviert werden, weil die beiden Fenster in eigenen Threads laufen sollen. Die Lösung mit Threads wurden deshalb gewählt, weil die Hauptanwendung so lange blockiert bis done geklickt wurde. Man könnte also, wenn alles in einem Thread laufen würde, nur mit dem Graphwindow interagieren. Es wäre nicht möglich im Stackwindow falls nötig zu scrollen. Außerdem hinterlegen manche Windowmanager Fenster, die blockiert sind, grau. Wenn man ein solches Fenster bewegt, wird der Inhalt nicht mehr oder verzerrt dargestellt. Die beiden Fenster werden danach durch die Funktionen
`LR_observer::init_stack_window(int* argc, char*** argv)` bzw.
`LR_observer::init_msg_window(int* argc, char*** argv)` erstellt. Diese Funktionen starten jeweils einen neuen Thread mit `pthread_create()`. `pthread_create()` wird eine der Funktionen
`stackwin_create_and_event_dispatcher(void* data)` bzw.
`msgwin_create_and_event_dispatcher(void* data)` als Funktionspointer übergeben, diese Funktion läuft dann im neu erzeugten Thread. Die Daten, welche der Thread benötigt, müssen über einen void-Pointer an `pthread_create()` übergeben werden. Sie werden in der Struktur `win_data` gehalten, welche vor der Übergabe in einen void-Pointer gecastet wird. Die Threads benötigen `int* argc` und `char*** argv` um das GTK-Fenster starten zu können. Ferner benötigen sie einen Zeiger auf den aufrufenden Observer, damit die Fenster Zugriff auf die boolsche Variable haben, in der gespeichert wird, ob die Ereignisse der dritten Ebene visualisiert werden sollen. Ist dies der Fall, müssen die Fenster entsprechend größer initialisiert werden. Ferner hat der Observer für jedes der beiden Fenster eine Membervariable vom Typ `GtkWidget*`, in welcher der HTML-Inhalt gespeichert wird. Diese Variable wird innerhalb der entsprechenden, oben erwähnten create-Funktion mit einem HTML-Widget initialisiert. Da der Observer die beiden HTML-Widgets kennt, kann am Ende der Funktion `LR_observer::print_stack()`, bzw. `LR_observer::set_message(std::string msg)` der in dieser Funktion erzeugte HTML-Code in das entsprechende Widget geschrieben werden. `LR_observer::print_stack()` benutzt die Hilfsfunktion `std::string LR_observer::get_pair_html(const pair& P, string lc, string rc)`, welche für ein übergebenes Intervallpaar P, eine Tabelle zurück gibt, deren Inhalt der linken Seite die HTML-Farbe `lc` hat, rechts entsprechend `rc`.

[45] Allein wegen des Messagewindows würde GtkHTML nicht verwendet werden. Da aber das Stackwindow bereits damit realisiert wurde, wurde auch das Messagewindow so erstellt.

Auf eine Aufzählung und Erklärung aller verwendeten GTK+-Befehle wurde verzichtet, da dies zu weit vom Thema wegführen würde. Der Lösungsweg sollte nur schematisch erklärt werden.

5.2.3 Graphenlayout

Das Layouten des Graphen im Hauptfenster geschieht - wie oben bereits erwähnt - in der Funktion `LR_observer::layout()`. Diese ruft die rekursive Funktion `LR_observer::layout_traversal(node v, bool root)` auf, welche den Tiefensuchbaum layoutet. Das Layout der Rückwärtskanten wird nach dem Aufruf von `LR_observer::layout_traversal(node v, bool root)` in der Layout-Hauptfunktion `LR_observer::layout()` erstellt. Das Layout des Tiefensuchbaums wird nach einem in [3] beschriebenen Algorithmus für ein radiales Baumlayout bestimmt. Jeder Knoten bekommt dabei einen bestimmten Keil als Platz für seinen Unterbaum zugeteilt. Der Wurzel[46] wird ein Keil von 180°[47] zugewiesen, da das Layout annähernd den obigen Beispielzeichnungen entsprechen soll. Für die Verteilung des Keilplatzes an die Kinder wurde hier eine sehr einfache Heuristik verwendet: Der Elternkegel wird gleichmäßig an die Kinder verteilt[48].
Für die anfallenden Daten hat der Observer die Member `coords`, `wedge` und `angle`. `coords` ist vom Typ $node\_array < point >$ und speichert für jeden Knoten die Koordinaten. `wedge` und `angle` sind vom Typ $node\_array < double >$. `wedge` speichert die einem Knoten für sich und seinen Unterbaum zur Verfügung stehende Keilgröße. `angle` speichert für jeden Knoten den Abstand seines Keils vom rechten Rand des Elternkegels.
Zur Veranschaulichung: Der Knoten v sei die Wurzel mit einem Keil von 180° und habe 4 Kindknoten. Jeder dieser Kindknoten bekommt als `wedge` den Wert 45. Der Wert für `angle` des Kindes, welches ganz links gezeichnet werden soll ist 0. Für die anderen von links nach rechts: 45, 90, 135.
Die Funktion `LR_observer::layout_traversal(node v, bool root)` lässt sich in 2 Teile einteilen:

[46] Bis jetzt können nur Graphen mit einer Zusammenhangskomponente gelayoutet werden.

[47] Im Paper bekommt die Wurzel 360° zugeteilt.

[48] Hier könnte später auch eine bessere Heuristik verwendet werden. Man könnte jedem Kind Platz relativ zu der Anzahl der Blätter in seinem Unterbaum zuweisen.

1. Bestimmung der Koordinaten des Knotens v[49]

2. Bestimmung von `wedge` und `angle` für jeden Kindknoten w von v

Listing 2: Layout - Bestimmung der Knotenkoordinaten

```
1  if(!root) {
2    node u = G.source(this->gbl->parentedge[v]);
3    point p = this->coords[u];
4    double newx= p.xcoord()+EDGE_LEN*(cos(this->angle[v]+(this->wedge[v]/2)));
5    double newy= p.ycoord()+EDGE_LEN*(sin(this->angle[v]+(this->wedge[v]/2)));
6    this->coords[v] = point(newx, newy);
7  }
```

Listing 2 zeigt die Bestimmung der Koordinaten für die Knoten[50]. Mittels `EDGE_LEN` lässt sich die Kantenlänge einstellen. Zu `angle[v]` wird `wedge[v]/2` addiert, damit der Knoten genau in der Mitte (auf der Winkelhalbierenden) des für ihn reservierten Keils gezeichnet wird.
Damit der zweite Schritt durchgeführt werden kann, muss zunächst die Liste `adj_tree`, welche die von Knoten v ausgehenden Baumkanten enthält, bestimmt werden. Diese Liste muss mittels `list::reverse()` umgekehrt werden, da sonst die erste ausgehende Baumkante der entsprechenden Adjazenzliste ganz rechts gezeichnet würde, die letzte ganz links und die anderen der Reihenfolge entsprechend dazwischen, weil dieser Layoutalgorithmus - wie in Listing 2 und 3 zu erkennen - die erste Kante der Liste `adj_tree` ganz rechts zeichnet. Diese Kanten sollen aber in umgekehrter Reihenfolge gezeichnet werden, da dies in den Beispielzeichnungen ebenfalls so gemacht wurde.

Listing 3: Layout - Bestimmung von wedge und angle

```
1  double angle = this->angle[v];
2  int cc = adj_tree.length();
3  forall(e, adj_tree) {
4    w = G.target(e);
5    this->wedge[w] = this->wedge[v] / cc;
6    this->angle[w] = angle;
7    angle = angle + this->wedge[w];
8    this->layout_traversal(w, false);
9  }
```

Listing 3 zeigt den zweiten Schritt von `LR_observer::layout_traversal(node v, bool root)`.

Die Rückwärtskanten werden als Bézierkurven gezeichnet. Auf welche Seite eine Rückwärtskante gezeichnet werden muss, wird mit Hilfe der oben erwähnten Liste `left` bestimmt.

[49] Falls v nicht die Wurzel ist. Die Koordinaten der Wurzel werden vor dem Aufruf von `LR_observer::layout_traversal(node v, bool root)` festgesetzt, da alles relativ zur Wurzel gelayoutet wird.

[50] this bezieht sich in diesem Listing auf den Observer. Die gesamte Implementierung befindet sich auf der beiliegenden CD. Die Definition dieser Funktion ist in der Datei LR_observer.cpp.

Das Layouten der Rückwärtskanten ist in folgende Schritte eingeteilt, welche jede Rückwärtskante durchläuft:

1. Bestimmung der Knoten, um die die Rückwärtskante gezeichnet werden muss
2. Sortieren dieser Knoten
3. Festsetzen der Bézierpunkte für diese Knoten
4. Bestimmung eines letzten Bézierpunktes, welcher in der Nähe des Zielknotens der Rückwärtskante ist

Zunächst werden für alle Rückwärtskanten diejenigen Knoten bestimmt, um die sie herum gezeichnet werden sollen. Für Rückwärtskanten, die rechts herum gezeichnet werden sollen, sind das alle Blätter, die rechts von ihrem Startknoten liegen. Für die linken gilt entsprechendes. Würde man für das Verteilen der Keile eine andere Gewichtung verwenden, müsste man unter Umständen die konvexe Hülle aller, auf der jeweiligen Seite vom Startknoten liegender Knoten berechnen. Die Koordinaten aller Knoten sind an dieser Stelle bereits bekannt. Aus diesem Grund könnte man die rechts- bzw. linksliegenden Blätter geometrisch bestimmen. Dies ist aber nicht nötig, denn man hat die Information, dass die Baumkanten der Adjazenzlisten der Reihe nach von links nach rechts gezeichnet wurden. Aus diesem Grund wurde das Finden der relevanten Blätter folgendermaßen gelöst: Die Kanten werden vom Startknoten jeder Rückwärtskante aus über die Elternkantenzeiger bis zum Knoten vor dem Zielknoten durchlaufen. Für jede dieser Kanten wird geschaut, ob sie Vorgänger (falls die Rückwärtskante nach links gezeichnet werden muss) bzw. Nachfolger (falls die Rückwärtskante nach rechts gezeichnet werden muss) in der Adjazenzliste ihres Startknotens hat. Für die Zielknoten aller Vorgänger bzw. Nachfolger erfolgt der Aufruf: `add_subtree_leaf_coords(node v, list<node>& l)`. Als zweiter Parameter wird dieser Funktion die Kantenliste `relevant_nodes` übergeben. Sie durchläuft den Unterbaum von v rekursiv und speichert alle Blätter in der durch l referenzierten Liste. Wichtig ist, dass hierbei nur der Unterbaum vom Zielknoten der Rückwärtskante beachtet wird, aus dem die Rückwärtskante zurückkehrt. Sonst würde sie möglicherweise unnötig um einen gesamten weiteren Unterbaum ihres Zielknotens herum gezeichnet werden. Deswegen wird die Elternkantenkette nur bis zum Knoten vor dem Zielknoten durchlaufen. Die Kante vom Zielknoten zu diesem Knoten wird nicht beachtet.
Als nächstes sollen für jede Rückwärtskante Bézierpunkte gesetzt werden, sodass die Kante um alle Blätter herumläuft. Da es auf die Reihenfolge

der Bézierpunkte ankommt, müssen die relevanten Blätter, welche sich nun in `relevant_nodes` befinden, entsprechend sortiert werden. Die Sortierung muss die zyklische Reihenfolge der Blätter um den Quellknoten der Rückwärtskante liefern. Das heißt die Sortierung ist von der x und y Koordinate der Blätter abhängig. Hier ist wie oben kein geometrischer Ansatz nötig. Es reicht die Blätter nach der Tiefensuchnummer, welche aus der Umsortierung der Kanten nach der Verschachtelungsreihenfolge entsteht, zu sortieren. Aus diesem Grund mussten diese Tiefensuchnummern schon vor der 2. Phase berechnet werden. Da sie der Durchlaufreihenfolge der 2. Phase entsprechen, heißt das sie speichernde node_array `dfs_num2`.
Die nach diesen Tiefensuchnummern sortierten Blätter entsprechen deswegen der oben erwähnten zyklischen Reihenfolge, weil durch dieses Tiefensuchlayout garantiert ist, dass für die Tiefsuchnummern von von einem Knoten v ausgehenden Unterbäumen $U_1, ..., U_n$ [51] folgendes gilt: $T_{U_1} < \cdots < T_{U_n}$, wobei T_U die Menge der Tiefensuchnummern eines Unterbaumes U bezeichnet und $A < B$ aussagt, dass alle Elemente der Menge A kleiner sind als das kleinste der Menge B. Die Unterbäume sind durch das Layout bedingt, zyklisch um den Knoten v angeordnet[52].
Weil in LEDA keine Möglichkeit gefunden wurde, eine Liste von Knoten nach deren Werten in einem node_array zu sortieren, wurde dazu die Funktion `list<node> bucketsort_list_node_map(const list<node>& l, node_array<int>, bool asc)` implementiert[53]. Sie wurde absichtlich nicht generisch erstellt, da der Typ *node_array* verwendet werden sollte, welcher nur Knoten als Schlüsselmenge zulässt.
Die Bézierpunkte sollen über den Blättern in Richtung der Elternkante gezeichnet werden. Das heißt, für ein Blatt an Position q dessen Elternknotenposition p ist, soll für den dazugehörigen Bézierpunkt b folgendes gelten: Der Vektor $\overrightarrow{pb}$ soll ein k-faches des Vektors $\overrightarrow{pq}$ sein. Dieser Faktor k soll so gewählt werden, dass die Anzahl der Kantenschnitte möglichst gering bleibt. Die Bézierpunkte der Rückwärtskante sollen desto weiter vom zu umlaufenden Blatt entfernt sein, desto größer der Höhenunterschied von ihrem Start- zu ihrem Endknoten ist. Das bezweckt, dass die Rückwärtskanten desto weiter außen gezeichnet werden können, desto größer der Höhenunterschied ist. Weiter soll bei jedem Blatt, was von einer Rückwärtskante zusätzlich umlaufen werden muss, der Bézierpunkt um eine gewisse Einheit weiter außen

[51] Wobei U_1 am weitesten links gezeichnet wurde, U_n am weitesten rechts und alle anderen der Reihenfolge entsprechend dazwischen.

[52] Sie sind auf einem Kreissegment mit einer `wedge[v]` entsprechenden Länge angeordnet.

[53] Implementierungsdetails können in der Datei data.cpp nachgeschaut werden, welche sich auf der beigefügten CD befindet.

gesetzt werden. Das soll vor allem bezwecken, dass von zwei Rückwärtskanten, welche auf der gleichen Seite gezeichnet werden müssen und den gleichen Höhenunterschied haben, diejenige außen gezeichnet wird, welche um mehr Blätter herum gezeichnet werden muss. Man kann diese beiden Faktoren unterschiedlich gewichten. Außerdem könnte man noch weitere Faktoren verwenden. Diese hier beschriebene Heuristik ist nur für kleinere Graphen geeignet. Was hier noch nicht beachtet wurde ist, dass wenn eine Rückwärtskante um einen sehr großen Unterbaum herum gezeichnet werden muss, müssen die zu den Blättern dieses Unterbaumes gehörigen Bézierpunkte relativ zu der Höhe dieses Unterbaumes weiter weg sein.
Listing 4 zeigt den Quellcode für die Bézierpunktebestimmung[54]. Die Liste `points` hält die Bézierpunkte einer Rückwärtskante.

Listing 4: Layout - Bestimmung der Bézierpunkte

```
1  int height_differential = this->gbl->height[v] - this->gbl->height[w];
2  double hdc = height_differential / 1.5; //this can be any term using height_differential.
3  node t;
4  int count = 2;
5  forall(t, relevant_nodes) {
6    point p = this->coords[G.source(this->gbl->parentedge[t])];
7    point q = this->coords[t];
8    double newx = (q.xcoord() - p.xcoord()) * hdc * (count / 2) + p.xcoord();
9    double newy = (q.ycoord() - p.ycoord()) * hdc * (count / 2) + p.ycoord();
10   points.append(point(newx, newy));
11   ++count;
12 }
```

Ein weiterer Bézierpunkt soll dafür sorgen, dass Rückwärtskanten mit einem größeren Höhenunterschied in einem weniger steilen Winkel beim Zielknoten ankommen. Dieser soll auf dem linken bzw. rechten Rand des Keils jenes Kindes x vom Zielknoten w einer Rückwärtskante liegen, welches die Wurzel des Unterbaumes ist, aus welchem die Rückwärtskante zurückkehrte. Er soll den Grundabstand einer Baumkantenlänge (`EDGE_LENGTH`) von w haben. Zusätzlich soll dieser Abstand mit dem Höhenunterschied gewichtet werden.Zur Bestimmung von x werden wieder die Tiefensuchnummern und die Tatsache, dass die Ordnung der Adajazenzlisten bekannt ist, verwendet. x ist das Kind von w mit der größten Tiefensuchnummer, die kleiner als die Tiefensuchnummer des Startknotens v der Rückwärtskante ist. Wegen der Ordnung der Adjazenzlisten muss man lediglich alle Kinder von w durchlaufen und beim ersten Kind, welches eine kleinere Tiefensuchnummer als v hat, anhalten. Das zuvor durchlaufene Kind ist das gesuchte Kind x.
Listing 5 zeigt den dazugehörigen Quellcode. Wie man dort erkennen kann, muss man bei der Implementierung darauf achten, dass in der Adjazenzliste

[54]Das Listing ist kompakter als der eigentliche Quellcode. Es wurden nur die für die oben zur Bézierpunktbestimmung relevanten Stellen aufgeführt. Der gesamte Quellcode befindet sich auf der beigelegten CD.

auch Rückwärtskanten sind. Das heißt, x kann der Zielknoten einer von w ausgehenden Rückwärtskante sein. In diesem Fall muss man x auf das zuletzt durchlaufene x, welches zuvor in $x2$ gespeichert wurde, zurücksetzen.

Listing 5: Layout - Bestimmung von x

```
1  list<node> adj_nodes = G.adj_nodes(w);
2  node x, x2;
3  forall(x, adj_nodes) {
4    if(this->dfs_num2[w] > this->dfs_num2[x]) {
5      //x is no child of w; it is the target of the backedge (w,x)
6      x = x2;
7      continue;
8    }
9    if(this->dfs_num2[v] < this->dfs_num2[x]) {
10     x = x2;
11     break;
12   }
13   x2 = x;
14 }
```

Nachdem Knoten x bekannt ist, kann man die mit dem Höhenunterschied gewichtete Position des letzten Bézierpunktes bestimmen. Listing 6 zeigt dies.

Listing 6: Layout - Bestimmung des letzten Bezierpunktes

```
1  point wp = this->coords[w];
2  double wpx;
3  double wpy;
4  if(!is_left) {
5    wpx = wp.xcoord() + EDGE_LEN * (hdc) * (cos(this->angle[x]));
6    wpy = wp.ycoord() + EDGE_LEN * (hdc) * (sin(this->angle[x]));
7  }
8  else {
9    wpx = wp.xcoord() + EDGE_LEN * (hdc) * (cos(this->angle[x] + this->wedge[x]));
10   wpy = wp.ycoord() + EDGE_LEN * (hdc) * (sin(this->angle[x] + this->wedge[x]));
11 }
12 points.append(point(wpx, wpy));
```

Danach werden die Kontrollpunkte der aktuellen Rückwärtskante mit `this->gw.set_bends(e, points)` gesetzt. Zuvor muss `this->gw.set_shape(e, leda::bezier_edge)` aufgerufen werden, damit die Kontrollpunkte als Bézierpunkte interpretiert werden. Als letztes wird in der Schleife, die alle Rückwärtskanten durchläuft, die Liste der Punkte zurückgesetzt, damit die nächste Kante nicht zusätzlich um die für die aktuelle Kante relevanten Blätter herum gezeichnet wird. Am Ende der Layoutfunktion fokusiert die Funktion `this->gw.zoom_graph()` den Graphen im Hauptfenster. Ein möglicher Nachteil hierbei ist, dass LEDA so zoomt, dass sämtliche Bézierpunkte noch im Bild sind. Das kann bei Bézierpunkten, die weit weg sind unerwünscht sein, da so weniger Platz für den Graphen verwendet wird.

6 Ausblick

Die hier vorgestellte Implementierung wird weiterentwickelt und kann an einigen Stellen noch verbessert werden. Ein paar Verbesserungen sind bereits in Arbeit. Wenn ein Graph nicht planar ist, wird am Ende ein sich im untersuchten Graphen befindlicher Kuratowski-Untergraph angezeigt werden, welcher beweist, dass der Graph nicht planar ist. Dies soll, wie auch die Einbettung, mit den in der Testphase erzeugten Datenstrukturen geschehen. Es wird nicht nötig sein, einen komplett anderen Algorithmus für die Extraktion von Kuratowski-Untergraphen zu verwenden. Die Implementierung dazu ist schon in Arbeit. Teile dieses Quelltextes sind bereits in der sich auf der CD befindlichen Animationsversion des Programms, sodass man sich ein Bild von der Funktionsweise machen kann.[55]
Weiter wird das Layout des Graphen, welches während der Animationsschritte zur Visualisierung des Graphen dient, verbessert werden. Für das Layout der Baumkanten könnte die oben erwähnte Heuristik, welche die Blattanzahl eines Unterbaumes als Kriterium für den diesem Unterbaum zugewiesenen Platz verwendet, benutzt werden werden. In das Layout der Rückwärtskanten sollen noch mehr Faktoren mit einfließen, um die Anzahl der Schnitte der Rückwärtskanten weiter minimieren zu können. Falls der Graph planar ist, soll eine schnittfreie Zeichnung des selben gezeigt werden.
Die Implementierung des Algorithmus selbst soll weiter beschleunigt werden. Die eigenen Datenstrukturen könnten dazu auf die oben beschriebene Weise angepasst werden. Genauere Pläne für weitere Optimierungen wurden noch nicht gefasst.

7 Anhang

7.1 Kompilieren des Animationsprogramms

Um das Anwendungsprogramm kompilieren zu können, werden die Bibliotheken LEDA und GTK+ samt GtkHTML benötigt. LEDA kann man auf der in [6] angegeben Seite herunter laden. Das Hauptverzeichnis, in dem sich LEDA befindet, muss über die Umgebungsvariable `LEDAROOT` zugänglich gemacht werden. Für diese Implementierung wurde LEDA in Version 6.0 verwendet. Die verwendete GTK+ Version 2.0 und GtkHTML Version 3.14-19 können unter Ubuntu oder Debian mit apt-get install libgtkhtml3.14-19 installiert werden. Sie können auch auf der unter [1] aufgeführten Seite

[55] Die Aufrufe der Extraktionsfunktion sind an den entsprechenden Stellen noch auskommentiert.

herunter geladen werden. Weiter müssen gcc in der Version 4.1 und make installiert sein. Danach kann im Hauptverzeichnis dieser Implementierung `make` ausgeführt werden. Für Animationsversion ist das auf der CD der Ordner lr_planar_anim, für die schnellere Version ohne Animation heißt der Ordner lr_planar_algo. Sollen andere Versionen als die oben angegebenen verwendet werden, muss das Makefile angepasst werden.

7.2 Schwer auffindbare Fehler bei der Implementierung

Zwei Punkte, die speziell bei der Verwendung von LEDA und GTK+,GtkHTML wichtig sind, sollen hier noch erwähnt werden, da sie zu sehr schwer auffindbaren Fehlern führen. GTK+ setzt die Lokaleinstellungen (locales). Die Locales definieren unter anderem die Schreibweise für Zahlen, welche festsetzt, ob ein Dezimalpunkt oder -komma verwendet wird und ob der Punkt oder das Komma als Tausendertrennzeichen zu verwenden ist. Diese Schreibweise verwendet LEDA für das Graphen-Ausgabeformat gml. Man kann deswegen mit bestimmten Lokaleinstellungen,welche GTK+ unter Umständen setzt, fremde .gml-Dateien nicht mehr einlesen und andere Programme können mit den eigens erzeugten .gml-Dateien nichts mehr anfangen. Deswegen muss man als Sprache wieder `C` einstellen, was der Standardeinstellung entspricht. Dies macht man am Anfang der main()-Funktion über die Environmentvariable `LANG` mittels `setenv(''LANG'', ''C'', 1)`. Die `1` als letztes Argument sagt, dass falls schon eine Variable `LANG` gesetzt ist, diese überschrieben wird, was in diesem Fall gewünscht ist.
Der zweite Punkt betrifft nur GtkHTML. Da er aber ebenfalls viel Zeit in Anspruch genommen hat, soll er auch erwähnt werden. Das Inkludieren der Header für GTK+ und GtkHTML muss innerhalb von `extern ''C''{}` geschehen. Für den Code, welcher sich in diesem Block befindet, wird das C++ spezifische Überladen ausgestellt, bei dem der Compiler den Funktionen eindeutige Namen geben muss. Wird dies nicht verwendet, findet der Linker die GtkHTML-Funktionen nicht. Die Fehlermeldungen sehen so aus, als hätte man die Header gar nicht eingebunden. Eigentlich sollte es hier nicht nötig sein, `extern ''C''{}` selbst schreiben zu müssen.

7.3 Profiler-Ergebnis

Folgendes Profil der Implementierung ohne Animation wurde mit gprof erstellt. `LR_PLANAR()` wurde mit einem zufälligen Graphen mit einer Million Knoten und zwei Millionen Kanten aufgerufen. Auf der CD befindet sich im Ordner lr_planar_algo das Shellscript profile.sh mit dem ein solches Profile erstellen kann ohne das Makefile zu verändern, sofern gprof installiert ist und

die oben erklärten Voraussetzungen für das Kompilieren erfüllt sind. Das erstellte Profil befindet sich danach im gleichen Ordner in der Datei profile.txt und wird mit automatisch mit gedit geöffnet.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
time   seconds   seconds    calls   s/call   s/call  name
12.32     4.29     4.29                             leda::graph::copy_graph(leda::graph const&)
8.73      7.33     3.04                              leda::node_struct::del_adj_edge(leda::edge_struct*, int, int)
5.92      9.39     2.06                              leda::dlist::bucket_sort(int, int)
4.94     11.11     1.72                              leda::dlist::clear()
4.40     12.64     1.53                              leda::graph::del_all_edges()
4.37     14.16     1.52                              leda::graph::add_edge(leda::node_struct*, leda::node_struct*, void*)
3.02     15.21     1.05                              leda::memory_manager::clear()
2.96     16.24     1.03    37086     0.00     0.00  test_dfs(leda::node_struct*, gblvars&)
2.93     17.26     1.02                              leda::random_planar_graph(leda::graph&, int, int)
2.79     18.23     0.97                              leda::node_struct::insert_adj_edge(leda::edge_struct*, leda::edge_struct*, int, int, int)
2.79     19.20     0.97                              leda::target_num(leda::edge_struct* const&)
2.76     20.16     0.96    37086     0.00     0.00  preprocessing_dfs(leda::node_struct*, gblvars&)
2.44     21.01     0.85                              leda::maximal_planar_map(leda::graph&, int)
2.41     21.85     0.84                              leda::node_struct::append_adj_edge(leda::edge_struct*, int, int)
2.41     22.69     0.84                              leda::dlist::permute()
2.36     23.51     0.82                              leda::random_planar_map(leda::graph&, int, int)
2.23     24.29     0.78  8000000     0.00     0.00  leda::list<leda::edge_struct*>::copy_elem(leda::dlink*) const
2.18     25.05     0.76                              leda::graph::sort_edges(leda::list<leda::edge_struct*> const&)
2.01     25.75     0.70                              leda::graph::del_edge(leda::edge_struct*)
2.01     26.45     0.70                              leda::graph::all_edges() const
1.90     27.11     0.66                              leda::source_num(leda::edge_struct* const&)
1.77     27.72     0.62 70653010     0.00     0.00  leda::graph_map<leda::graph>::array_access(leda::edge_struct*) const
1.74     28.33     0.61                              leda::dlist::dlist(leda::dlist const&)
1.41     28.82     0.49                              leda::graph::make_directed()
1.39     29.30     0.49 51006187     0.00     0.00  leda::graph_map<leda::graph>::array_access(leda::node_struct*) const
```

Abbildungsverzeichnis

Literatur

[1] Gtk+, see
http://www.gtk.org

[2] Brandes, U.: The left-right planarity test Unpublished

[3] Christian Bachmaier, Ulrik Brandes, B.S.: Optimal algorithms for radial and circular drawings of phylogenetic trees

[4] yWorks GmbH: yfiles online documentation,
see http://www.yworks.com/en/products_yfiles_about.htm

[5] Kuratowski, C.: Sur le probleme des courbes gauches en topologie pp. 271–283 (1930)

[6] algorithmic solutions: Leda library,
see http://www.algorithmic-solutions.com/leda/about/index.htm

[7] algorithmic solutions: Leda manual,
http://www.algorithmic-solutions.info/leda_manual/MANUAL.html