Praktikum „Algorithm Engineering“

Wintersemester 06/07

Universität Karlsruhe

Der Menger-Algorithmus

Florian Böhl, Simon Friedberger, Boris Groß-Hardt

Zusammenfassung

Ein planarer Graph ist ein Graph, der in der Ebene gezeichnet werden kann, ohne dass sich Kanten kreuzen. Planare Graphen haben viele schöne Eigenschaften, die benutzt werden können, um für Probleme, die sich auf planaren Graphen modellieren lassen, besonders einfache und schnelle Algorithmen zu entwerfen. Oft können sogar Probleme, die auf allgemeinen Graphen (NP-)schwer sind, auf planaren Graphen sehr effizient gelöst werden. Als eine von vier Gruppen war es unsere Aufgabe, den Menger-Algorithmus nicht nur effizient zu implementieren, sondern auch mit Hilfe einer am Institut entwickelten Bibliothek, die auf Java3D aufsetzt, zu visualisieren. Anschlieend haben wir im Rahmen einer experimentellen Analyse das Laufzeitverhalten unserer Implementierung untersucht.

Inhaltsverzeichnis

1 Einleitung

Der kantendisjunkte Menger-Algorithmus soll schnell implementiert und die einzelnen Schritte nachvollziehbar visualisiert werden. Zur internen Repräsentation der bearbeiteten Graphen in unserer Implementierung wird die Jung-Graphenbibliothek für Java verwendet[1]. Zur Visualisierung wird AlgoVis3d verwendet, ein vom Institut zur Verfügung gestelltes Werkzeug, das auch im Verlauf des Praktikums weiter von Mitarbeitern angepasst wurde.

2 Der Algorithmus von Menger

Der Algorithmus beruht auf dem *Satz von Menger (1927)* [2]:

Seien s und t zwei Knoten eines Graphen G, s und t nicht adjazent bei der knotendisjunkten Version

- $\kappa_G(s,t) \geq k$ *genau dann, wenn es k paarweise intern knotendisjunkte Wege zwischen s und t in G gibt.*
- $\lambda_G(s,t) \geq k$ *genau dann, wenn es k paarweise intern kantendisjunkte Wege zwischen s und t in G gibt.*

Hierbei stehen $\kappa_G(s,t)$ und $\lambda_G(s,t)$ für den Knoten- bzw Kantenzusammenhang der Knoten s und t in G. Das zugehörige *Menger-Problem* besteht darin, eine solche maximale Anzahl von Wegen zu finden und der von uns behandelte Algorithmus löst es für den kantendisjunkten Fall.

Wir haben den Menger-Algorithmus in vier Schritte unterteilt, die im Folgenden näher beschrieben werden.

2.1 Schritt 1

Im Gegensatz zur Theorie ist die Vorverarbeitung bei uns Teil von Schritt 1 und wird deshalb auch hier beschrieben.

2.1.1 Theorie

Der Menger-Algorithmus arbeitet auf einer bigerichteten Entsprechung $\overrightarrow{G}$ des Ausgangsgraphen G, in der jede Kante durch zwei entgegengesetzt laufende Doppelkanten ersetzt wird. In Schritt 4 wird dann das Ergebnis (s-t-Wege im gerichteten Graphen) zurück auf den ursprünglichen Graphen überführt.

2.1.2 Implementierung

Die Graphen werden in unserer Implementierung aus *.graphml Dateien eingelesen und in ein JUNG-Graphobjekt geladen. Das Graphobjekt übergeben wir dann dem Planar-Embedder, der freundlicherweise von einem Mitarbeiter des Instituts zur Verfügung gestellt wurde. Wie der Name vermuten lässt handelt es sich um ein Programm, das aus geometrischen Einbettungen die entsprechenden planaren Einbettungen gewinnt und in Form eines PlanarGraph-Objektes zurückgibt.
In diesem PlanarGraph-Objekt hat der Planar-Embedder alle Kanten aus der

geometrischen Einbettung durch ungerichtete Doppelkanten ersetzt. Da wir gerichtete Kanten benötigen, löschen wir von diesen jeweils eine und übergeben das Ergebnis an den eigentlichen Schritt 1.

Als Schritt 1 implementiert wurde gab es noch keine „MixedMode“-Graphen (Graphen zur Repräsentation von gerichteten sowie ungerichteten Graphen), also legen wir, anstatt gerichtete Kanten in den vom PlanarEmbedder zurückggebenen Graphen einzufügen, einen neuen gerichteten Graphen an und bauen diesen entsprechend auf. Dann fügen wir die benötigten entgegengesetzt laufenden Doppelkanten ein. Die Referenzen zu den alten Kanten verwalten wir in Maps aus java.util.

2.2 Schritt 2

Im zweiten Schritt des Algorithmus sollen auf dem gerichteten planaren Graphen Rechtkreise gefunden werden, welche die Facetten nach ihrem Abstand zur äußeren Facette separieren. „Rechts“ sind die Kreise insofern, als dass sich die im Graphen weiter innen liegende Facette rechts von der „Laufrichtung“ des Kreises befindet.

2.2.1 Dualgraph erzeugen

Zu dem planaren Graphen G wird nun zunächst der Dualgraph ermittelt. Hierbei ist zu beachten, dass die ungerichteten Kanten – wie in Schritt 1 erwähnt – eine implizite Richtung haben (man beachte: „linke“ und „rechte“ Facette im Pseudocode), die notwendig ist, um später in Schritt 3 die passende gerichtete Kante zu finden.

Algorithmus 1 : Dualgraph erzeugen

Eingabe : Ungerichteter Graph G
Ausgabe : Dualgraph G^* zu G

```
1  Erstelle neuen leeren Graphen G*;
2  für alle Facetten f von G tue
3      Füge neuen Knoten k in G* ein;
4      Setze korrespondierende Facette zu k auf f;
5      Setze korrespondierenden Knoten zu f auf k;
6  für alle Kanten k aus G tue
7      Setze k'_quelle auf korrespond. Knoten in G* zur linken Facette von k;
8      Setze k'_ziel auf Korrespond. Knoten in G* zur rechte Facette von k;
9      Erstelle ungerichtete Kante k' von k'_quelle nach k'_ziel;
10     Füge k' in G* ein;
```

Laufzeit: $O(|V| + |E|)$

2.2.2 Distanzen der Facetten bestimmen

Um die Facettendistanzen zu bestimmen, muss zunächst eine äußere Facette gewählt werden (gehalten in 'aeussere_facette'). Da ein an der äußeren Facette liegender Zielknoten vom Algorithmus gefordert wird, sind Kandidaten nur die zu t (aus G) benachbarten Facetten. Im Beispielgraphen (vgl. Abb. 1) wurde die blaue Facette als „äußere“gewählt.

Die Facettendistanzen werden durch eine Breitensuche auf dem Dualgraphen G^* bestimmt.

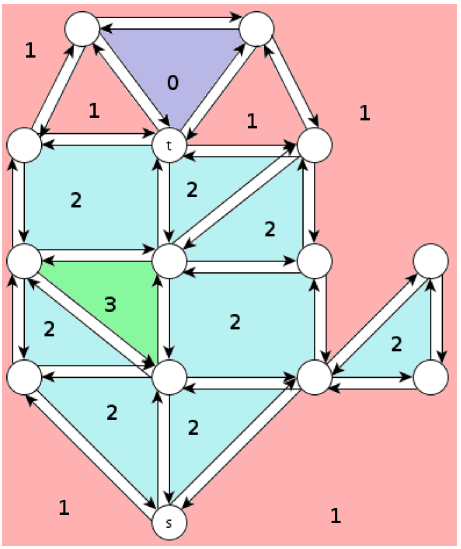


Abbildung 1: Facettendistanzen bestimmt

Algorithmus 2 : Facettendistanzen bestimmen

Eingabe : Dualgraph zu G G^*, gewählte äußere Facette $aeussere\_facette$

Ausgabe : Zuordnung der Knoten des Dualgraphen G^* bzw. der korrespondierenden Facetten in G zu ihrer Distanz zur äußeren Facette

```
1  Erstelle neue Queue q;
2  für alle Knoten k aus G* tue
3      Setze Distanz von k auf ∞;
4  Setze s auf aeussere_facette;
5  q.push(s);
6  solange q nicht leer tue
7      Setze k auf q.pop();
8      für alle Benachbarten Knoten n von k tue
9          wenn Distanz von k < ∞ dann
10             Setze Distanz von n auf Distanz von k + 1;
11             Setze Distanz von zu n korrespond. Facette auf Distanz von n;
               q.push(n);
12 Setze max_distanz auf k.dist;
```

Laufzeit: $O(|V| + |E|)$ (Breitensuche)

2.2.3 Kreise finden

Nun sollen die Kreise auf dem gerichteten Graphen $\overrightarrow{G}$ gefunden werden, wobei die Facetten aus dem ungerichteten Graphen zugrundegelegt werden (in dem gerichteten Graphen entsteht beim Einfügen der Kanten zwischen jedem Paar zweier gerichteter Kanten eine neue Facette - diese wollen wir explizit nicht berücksichtigen). Die Anzahl der Kreise entspricht der Distanz der am weitesten von der äußeren entfernten Facette (max_distanz). Im Beispielgraphen (vgl. Abb. 2) sind die gefundenen Kreise mit je einer Farbe (blau, rot und grün) markiert.

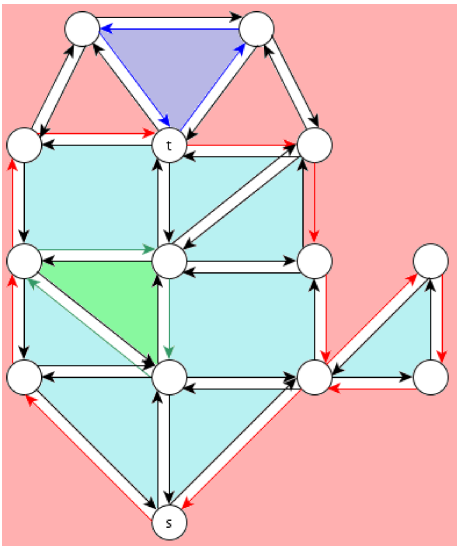


Abbildung 2: Kreise gefunden

Algorithmus 3 : Kreise finden

Eingabe : Ungerichteter Graph G, Facettendistanzen aus Algorithmus 2
Ausgabe : Rechtskreise in $\overrightarrow{G}$

```
1 Initialisiere Feld kreise[0..max_dist];
2 für alle Kanten k aus G tue
3     wenn Distanz der linken Facette zu k < Distanz der rechten Facette
      zu k dann
4         Die gerichtete Kante aus G⃗ gleicher Richtung wie k zu
          kreise[Distanz der linken Facette zu k] hinzufügen;
5     wenn Distanz der linken Facette zu k > Distanz der rechten Facette
      zu k dann
6         Die gerichtete Kante aus G⃗ entgegengesetzter Richtung wie k zu
          kreise[Distanz der rechten Facette zu k] hinzufügen;
```

Laufzeit: $O(|E|)$

2.2.4 Kanten umdrehen

Alle auf einem Kreis liegenden Kanten müssen nun noch umgedreht und als Kreiskanten markiert werden (für Schritt 3). Da das Umdrehen der Kanten nicht von der Jung-Bibliothek unterstützt wird, haben wir einfach eine neue Kante in entgegengesetzter Richtung angelegt und eingefügt.

Abb. 3: Der Beispielgraph mit den drei zuvor gefundenen Kreisen (blau, rot und grün), die nun umgedreht wurden.

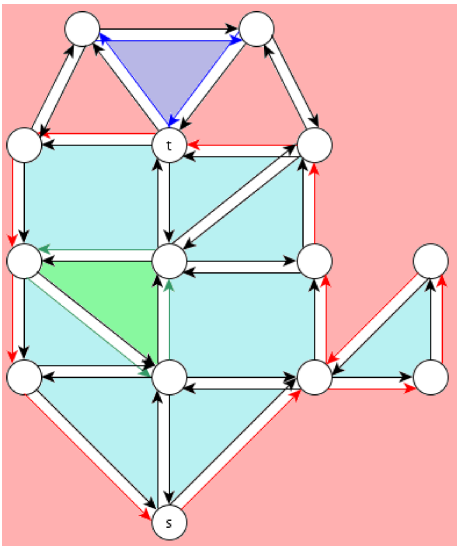


Abbildung 3: Kanten umgedreht

Algorithmus 4 : Kanten umdrehen

Eingabe : Gerichteter Graph $\overrightarrow{G_C}$, *kreise*[] aus Algorithmus 3
Ausgabe : Gerichteter Graph $\overrightarrow{G_C}$ mit gedrehten Kreiskanten

```
1 G_C ← G;
2 für alle i ← 0 bis max_dist tue
3     für alle Kanten k aus Kreis kreise[i] tue
4         k aus G_C entfernen;
5         Erstelle neue Kante k', als k gedreht;
6         Füge k' in G_C ein;
7         Markiere k' als Kreiskante;
8         Alle weiteren Attribute bzw. Korrespondenzen von k auf k'
          übertragen;
```

Laufzeit: $O(|E|)$

2.3 Schritt 3

2.3.1 Theorie

Im dritten Schritt wird zuerst mit einer Right-First-Search (RFS) eine Menge von kantendisjunkten, gerichteten s-t-Wegen $\overrightarrow{P_C}$ in $\overrightarrow{G_C}$ (dem Graph mit den umgedrehten Kreisen) mit maximaler Anzahl berechnet. Als Nächstes wird die Menge $\overrightarrow{P}$ der Wege in $\overrightarrow{G}$ aus $\overrightarrow{P_C}$ berechnet, wie unten genauer beschrieben wird.

Die RFS folgt von jeder ausgehenden Kante von s aus jeweils der rechtesten, freien, auslaufenden Kante bis t oder wieder s erreicht werden. Hierbei bedeutet „rechteste“ dass die nächste Kante gegen den Uhrzeigersinn benutzt wird. Jede Kante, der einmal gefolgt wurde, gilt dann als nicht mehr frei. In Algorithmus 5 wird eine mögliche Implementierung dargestellt.

Algorithmus 5 : Right-First-Search

Eingabe : Graph mit umgedrehten Kreisen $\overrightarrow{G_C}$, Startknoten s, Zielknoten t
Ausgabe : Wege $\overrightarrow{P_C}$

```
1  P_C = ∅;
2  e_1, ..., e_r ← ausgehende Kanten von s in G_C;
3  für i := 1 bis r tue
4      e = e_i;
5      p_i = {e_i};
6      solange e ≠ s und e ≠ t tue
7          v ← Endknoten von e;
8          e' ← rechteste, freie, ausgehende Kante von v bezüglich e;
9          p_i ← p_i ∪ {e'};
10         e ← e';
11     wenn p_i ein s-t-Weg ist dann
12         P_C = P_C ∪ {p_i};
```

Um nun $\overrightarrow{P}$ zu berechnen entfernen wir zunächst alle Kanten aus dem Graphen, die weder auf einem der $\overrightarrow{C_i}$, noch in $\overrightarrow{P_C}$ liegen. Danachs werden die Kanten entfernt, die in $\overrightarrow{P_C}$ sind und auf einem der $\overrightarrow{C_i}$ liegen, da diese in $\overrightarrow{G}$ nicht vorhanden sind. Von den übrigen Kanten werden diejenigen, die auf einem der $\overrightarrow{C_i}$ liegen, wieder umgedreht. Es entsteht also ein Teilgraph von $\overrightarrow{G}$. Um nun aus der Menge der Kanten dieses Teilgraphen, die $\overrightarrow{P}$ entspricht, eine Menge von Wegen zu erstellen wird darauf wieder die RFS angewandt (siehe Algorithmus 6).

2.3.2 Visualisierung

Während der Algorithmus läuft wird der aktuelle Pfad in gelb dargestellt. Abgeschlossene s-t-Wege werden in unterschiedlichen Farben eingefärbt, solange dies (mit den vorhandenen 14 Farben) möglich ist. s-s-Wege werden wieder entfärbt.

Algorithmus 6 : Schritt 3

1 Right-First-Search($\overrightarrow{G_C}$, s, t);
2 $\overrightarrow{E'} = \overrightarrow{E} \setminus \{e | (e \in \overrightarrow{C_i}) \vee (e \notin \overrightarrow{C_i} \wedge e \notin \overrightarrow{P_C})\}$ // Entfernt alle Kanten, die auf einem Kreis iegen, sowie Kanten die weder auf einem Kreis noch auf einem Weg liegen.
3 $\overrightarrow{E'} = \overrightarrow{E'} \cup \{(v, v') | (v', v) \in \overrightarrow{E_C}\}$ // Fügt die Kanten, die auf einem Kreis, aber nicht auf einem Weg lagen umgedreht wieder ein.
4 $\overrightarrow{G_C} = (V, \overrightarrow{E'})$;
5 Right-First-Search($\overrightarrow{G_C}$, s, t);

2.4 Schritt 4

2.4.1 Theorie

Nun, da wir die maximale Anzahl s-t-Wege in dem gerichteten Graphen gefunden haben, bleibt nur noch, diese Wege zurück in eine maximale Anzahl Wege im ungerichteten Ausgangsgraphen zu überführen.

Dies wird genau dann nicht-trivial, wenn im gerichteten Graphen sowohl die Hin- als auch Rückkante durch Wege belegt sind. Das kann auf zwei Arten geschehen:

1. Ein Weg läuft eine Schleife, beginnend und endend bei einem Knoten v, und läuft dann von diesem Knoten weiter.

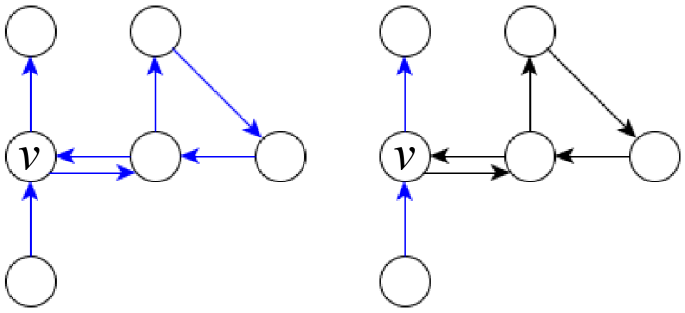


Abbildung 4: Fall 1

2. 2 Wege kreuzen die selbe Kante

Im ersten – einfacheren – Fall löscht man alle Kanten nach v, bis man wieder zu v gelangt. Dieses Vorgehen ändert nicht die Anzahl der Wege, entfernt aber eine Doppelbelegung.

Im zweiten Fall bilden zwei Wege einen Mehrfachkreis wie in Abbildung 5 dargestellt. Man nimmt alle Kanten von Weg A nach w und hängt sie an Weg B vor v, entsprechend hängt man alle Kanten nach v von Weg B an Weg A vor w. Auch dieses Vorgehen verringert die Zahl der doppelt belegten Kanten um eins.

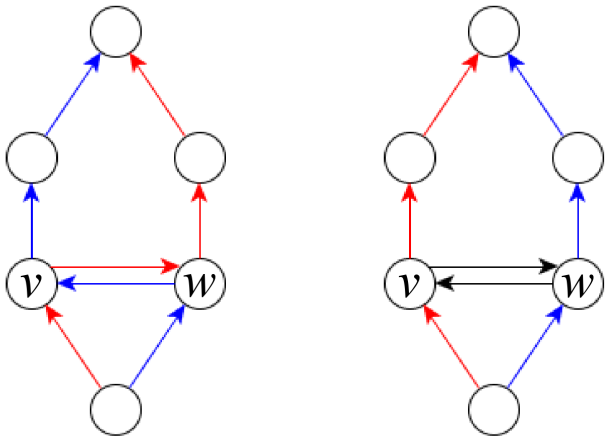


Abbildung 5: Fall 2

Hat man auf diese Weise alle doppelt belegten Kanten aus den Wegen entfernt, kann man die Wege 1:1 in den ursprünglichen Graphen überführen.

2.4.2 Implementierung

Bei der Implementierung von Schritt 4 ist zu beachten, dass zunächst die Doppelkanten nach Fall 2 entfernt werden und dann die nach Fall 1.

Algorithmus 7 : Schritt 4

Eingabe : s-t Wege mit (möglicherweise) doppelt benutzten Kanten in $Wege[]$

Ausgabe : s-t Wege ohne doppelt benutzte Kanten in $NeueWege[]$

```
1  NeueWege[] = leer;
2  neuerWeg = leer;
3  für alle Weg in Wege[] tue
4      solange Ende von Weg nicht erreicht tue
5          kante = nächste Kante in Weg;
6          gegenKante = die zu kante entgegengesetzt laufende Kante;
7          wenn gegenKante wird benutzt von beliebigem Weg dann
8              Weg = der Weg von gegenKante;
9          sonst
10             fuege kante an neuerWeg an;
11     fuege neuerWeg in NeueWege[] ein;
12     neuerWeg = leer;
```

Das Entfernen einer Doppelkante nach Fall 2 kann eine Doppelkante nach Fall 1 erzeugen. Wir müssen den Algorithmus also, nachdem er für alle Wege einmal durchgelaufen ist, ein weiteres Mal durchlaufen lassen, um die neu entstandenen Doppelkanten zu entfernen.

Da beim ersten Durchlauf nur Doppelkanten nach Fall 1 neu entstehen können, und das Entfernen von Doppelkanten nach Fall 1 keine neuen Doppelkanten erzeugen kann sind wir nach 2 Durchläufen fertig. Man ersetzt einfach in den beim Ausführen des Algorithmus neu angelegten Wegen jede Kante durch ihre Entsprechung im Ursprungsgraphen und erhält das Ergebnis.

2.4.3 Visualisierung

Man erkennt alle Stellen, an denen Schritt 4 arbeiten muss, an Kanten mit unterschiedlich eingefärbten Kegeln oder Knoten mit mehr als 2 Eingangskanten derselben Farbe. Wie schon erklärt, verschwindet im ersten Fall einfach die Problemkante aus den Wegen, im zweiten Fall verschwindet die Problemkante und alle nachfolgenden Kanten.

3 Das fertige Programm

Man startet AlgoVis, lädt einen planaren Graphen aus einer *.graphml Datei, in der s und t gekennzeichnet sind sowie den Algorithmus. Die Größe der Graphen, die verarbeitet werden können, ist proportional zur Größe des zur Verfügung stehenden Arbeitsspeichers. Auf einem Rechner mit 1GB Arbeitsspeicher kann unsere Implementierung Graphen bis circa 5000 Knoten verarbeiten.
Die ersten Schritte zeigen den Graphen, der aus der Datei geladen wurde und das Hinzufügen der entgegenlaufenden Kanten. Dann werden die Kreise gefunden und eingefärbt. Sollte die zufällig gewählte äußere Facette nicht mit der äußeren Facette der Visualisierung übereinstimmen, können diese Kreise ein wenig unintuitiv liegen. Im nächsten Schritt werden die Kanten umgedreht die auf Kreisen liegen, visualisiert durch die ihre Richtung ändernden Kegel. Dann folgt Schritt 3: Die gefundenen Wege werden eingefärbt und die nicht mehr betrachteten Kanten werden aus der Visualisierung entfernt. Schritt 4 markiert eine Problemkante und zeigt die neuen Wege, die entstehen, wenn man Schritt 4 die Problemkante entfernen lässt.

Der letzte Schritt der Visualierung zeigt die gefundenen Wege im Ursprungsgraphen.

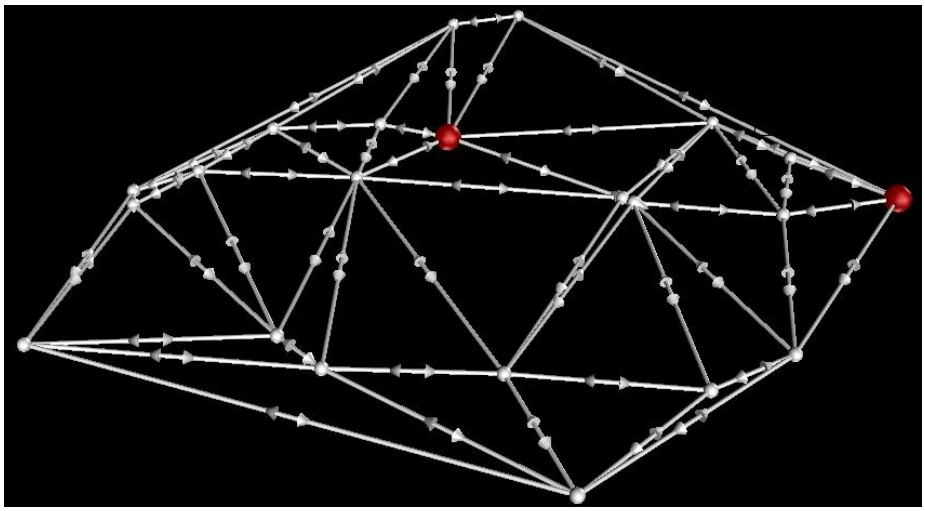


Abbildung 6: Start

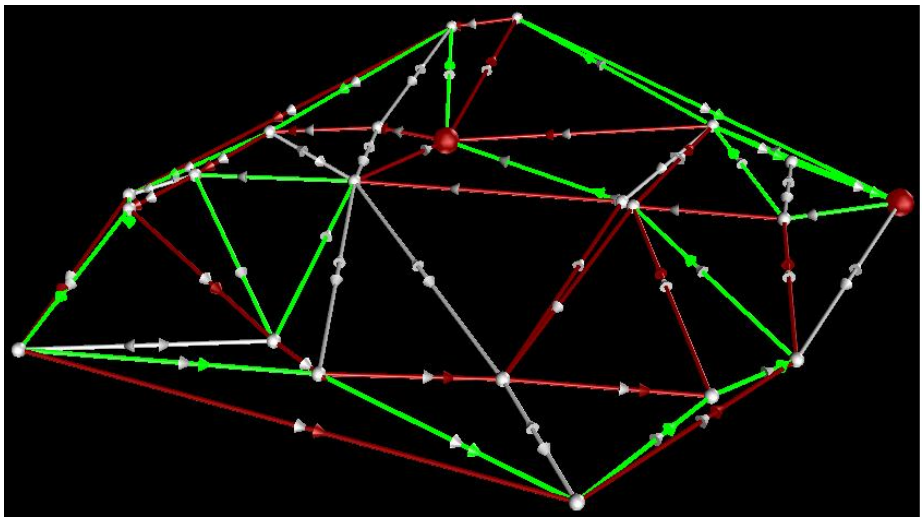


Abbildung 7: eingefärbte Kreise, die Kanten sind schon gedreht

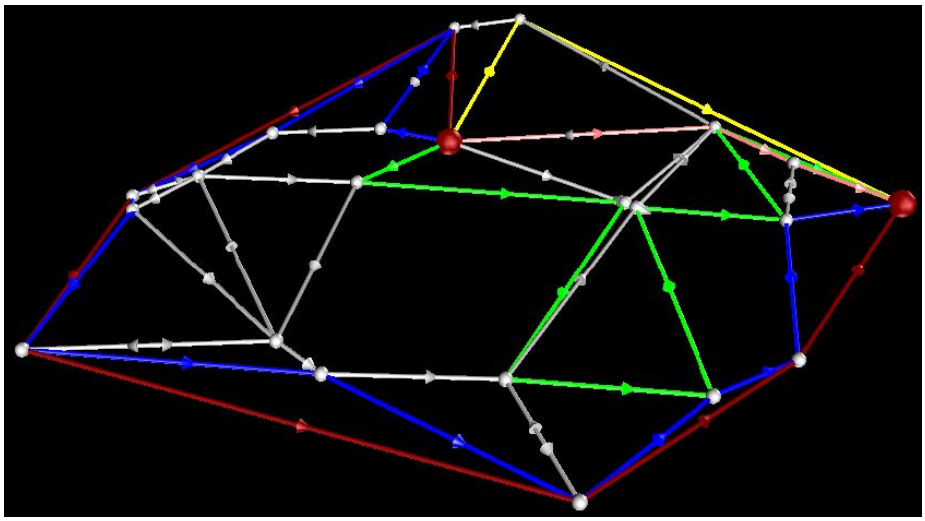


Abbildung 8: fertig

4 Weiterführendes

4.1 remove-loops

RemoveLoops ist ein einfache Funktion zum Entfernen von Schleifen in Pfaden, um die Übersichtlichkeit zu erhöhen. Hierzu wird jeder Pfad verfolgt und markiert. Trifft man hierbei auf einen bereits markierten Knoten, liegt offenbar eine Schleife vor und der Teil des Pfades seit der Markierung des Knotens wird gelöscht. Es sei noch darauf hingewiesen, dass die LinkedList Klassen aus java.util nicht geeignet sind um dies in linearer Zeit zu tun. Trotz des Namens ist hier ein Navigieren zum nächsten oder vorherigen Eintrag von einem gegebenen Element aus nicht möglich.

Algorithmus 8 : remove-loops

Eingabe : Menge von Wegen W (als Listen von Kanten)

```
1  NW = ∅;
2  für alle w ∈ W tue
3      Knoten = ∅;
4      nw = ∅;
5      für alle k ∈ w tue
6          wenn Ausgangsknoten von k ∈ Knoten dann
7              lösche alle alle Kanten nach k aus nw;
8          sonst
9              nw ∪ k;
10             Knoten ∪ Ausgangsknoten von k;
11     NW ∪ nw;
```

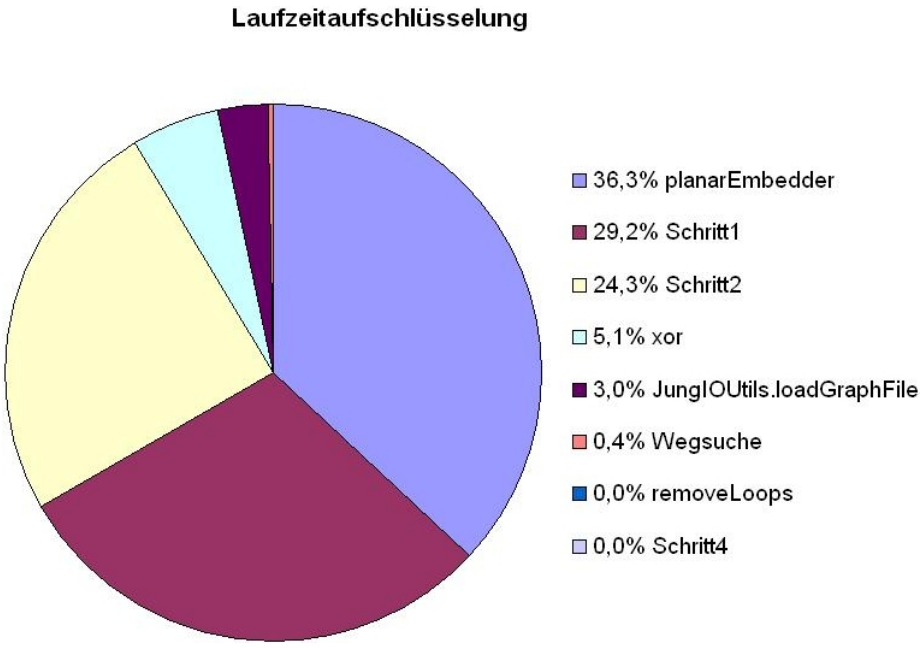
4.2 Experimentelle Analyse

Drei Aspekte wurden von uns untersucht:

- Die Laufzeitanteile der verschiedenen Teile des Programms
- Die Gesamtlaufzeit in Abhängigkeit von der Graphgröße
- Verschiedene Charakteristika eines Durchlaufes für einen Graphen, wie z.B. die Anzahl der entfernten Doppelkanten oder die gefundenen Wege nach Graphgröße

4.2.1 Laufzeitanteile

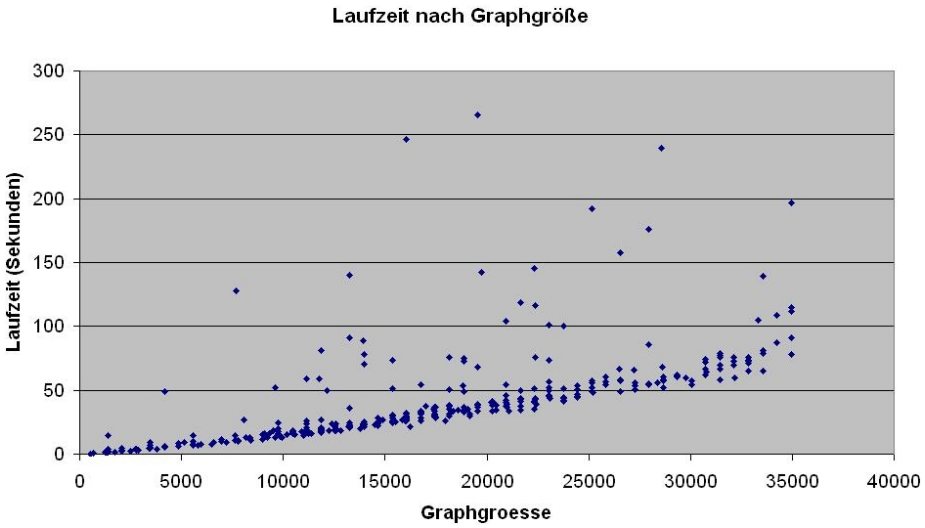
Um die Messungen ins rechte Licht zu rücken, betrachten wir zuerst die Laufzeitanteile der verschiedenen Phasen.



Mit Hintergrundwissen über die in den einzelnen Funktionen vorgenommenen Operationen sieht man sofort, dass der eigentliche Algorithmus verschwindend geringen Anteil an der Laufzeit hat. Die Verwaltung der Jung-Graphen nimmt den Löwenanteil der Rechenzeit in Anspruch:
PlanarEmbedder, der nur die interne Repräsentation aufbaut schluckt alleine schon 36 Prozent der Rechenzeit und Schritt 1, der eigentlich triviale Vorgang jede Kante durch 2 entgegengesetzt laufende Kanten zu ersetzen, kostet weitere 29 Prozent.

Der Aufwand für die eigentliche Arbeit des Algorithmus – die Wegsuche und das Finden der Kreise – ist verschwindend gering oder geht komplett in den Graphoperationen von Schritt 2 unter. Nichtsdestoweniger werden wir die anderen Ergebnisse besprechen, beginnend mit der Gesamtlaufzeit in Abhängigkeit von der Graphgröße

4.2.2 Laufzeit

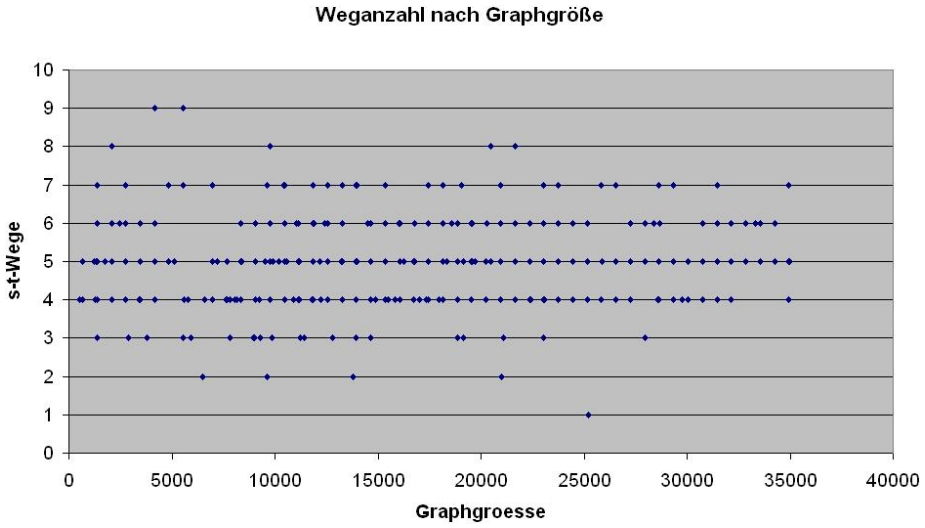


Die Ausreisser lassen sich zum einen durch den Java-Garbage-Collector und zum anderen durch den begrenzten Hauptspeicher des Testrechners erklären. Wenn das Programm anfängt, Speicher auf die Festplatte auszulagern, steigt die Laufzeit. Abgesehen davon sind die Ergebnisse dieser Messung wenig überraschend: je größer der Graph desto länger dauert seine Bearbeitung. Ein linearer Verlauf lässt sich erkennen.

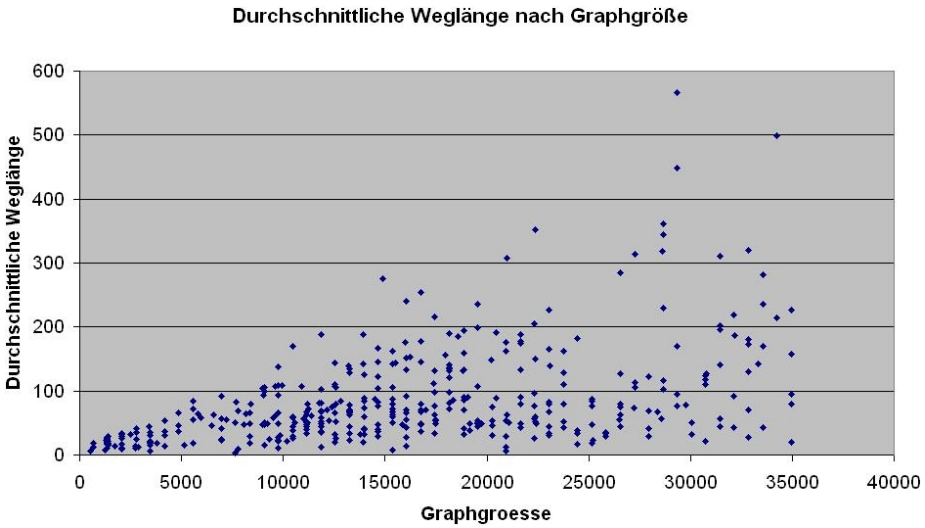
Nun noch ein Blick auf die möglicherweise verschiedenen Abläufe im Algorithmus.

4.2.3 Häufigkeiten

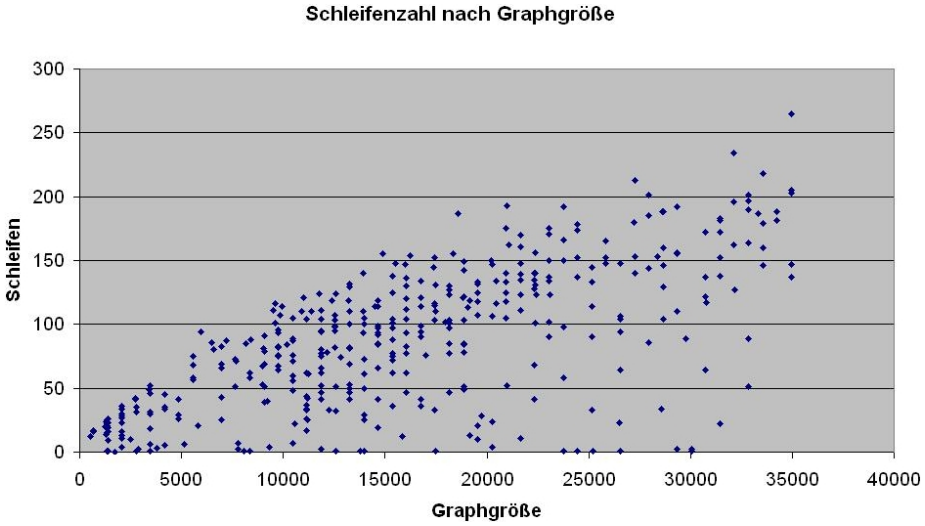
Untersucht wurde unter anderem, wie oft sich Wege überscheiden, oder wieviele Wege gefunden werden.



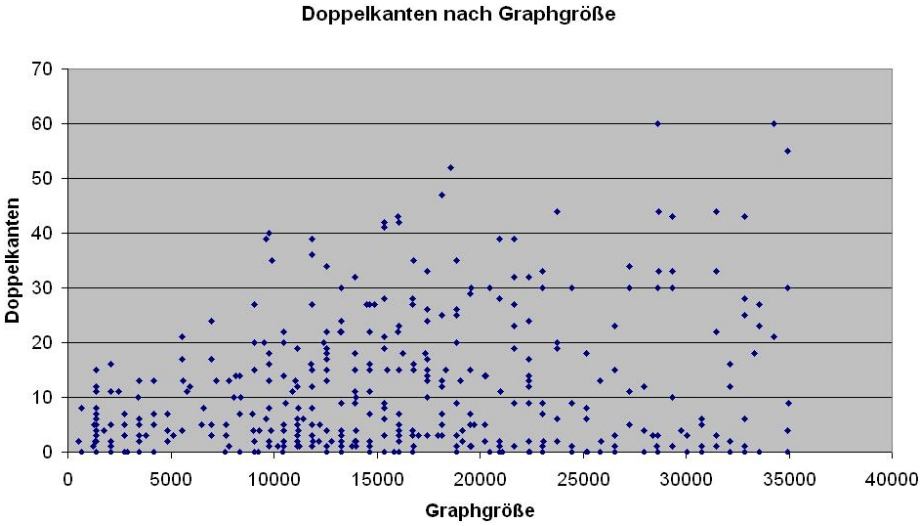
Die Wegzahl hängt vom Zusammenhang von s und t ab, nicht von der Graphgröße. Sie wird natürlich durch diese begrenzt, was aber nur bei kleinen Zahlen relevant ist.



Bei zufälligen s und t provozieren größere Graphen längere Wege.



Ein paar Durchläufe des Algorithmus zeigen, dass dieser nicht sehr zielgerichtet vorgeht. In großen Graphen müssen also mehr Doppelkanten und Schleifen entfernt werden.



Literatur

[1] JUNG Java Universal Network/Graph Framework - http://jung.sourceforge.net/

[2] Wagner, Dorothea „Algorithmen für Planare Graphen “ - http://i11www.iti.uni-karlsruhe.de/teaching/SS_06/planalgo/skript/algo-plan.pdf