Universität Konstanz
Fachbereich Mathematik und Statistik

Kantenkreuzungen in Kreislayouts

Michael Baur

DIPLOMARBEIT

Betreuer: Prof. Dr. D. Wagner

Konstanz, im Juli 2003

Inhaltsverzeichnis

Kapitel 1

Einführung

In dieser Arbeit wird ein Teilproblem der Visualisierung von Graphen, die Kreuzungsminimierung in Kreislayouts, betrachtet. Kreislayouts sind eine Darstellung von Graphen, bei der alle Knoten auf einem einzigen Kreis platziert werden. Solche Layouts treten in vielen verschiedenen Anwendungsbereichen auf, beispielsweise bei der Visualisierung von Computer-, oder allgemeiner, Telekommunikationsnetzwerken. Im nächsten Abschnitt werde ich ein nicht-technisches Anwendungsgebiet vorstellen, das für mich die Motivation zur Betrachtung von Kreislayouts war, nämlich soziale Netzwerke.

Durch die Visualisierung als Graph sollen gewöhnlich gewisse Strukturen der repräsentierten Daten hervorgehoben werden. Damit sind meistens Einschränkungen der möglichen Darstellungen gegeben. In diesem Rahmen soll die Darstellung des Graphen möglichst übersichtlich und *schön* sein.

Es gibt verschiedene Kriterien, mit denen diese *Schönheit* ausgedrückt werden kann, davon ist die Anzahl der sich kreuzenden Kanten wahrscheinlich das offensichtlichste und auch wichtigste. Darstellungen mit vielen Kreuzungen machen es eindeutig schwer, die Struktur des zugrunde liegenden Graphen zu erfassen (siehe Abb. 1.1).

Wir betrachten nun ein konkretes Beispiel für die Visualisierung von Informationen

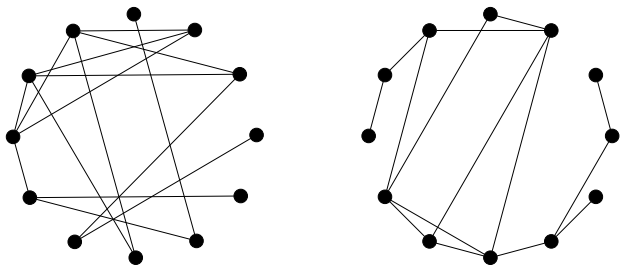


Abbildung 1.1: Zwei Darstellungen des gleichen Graphen.

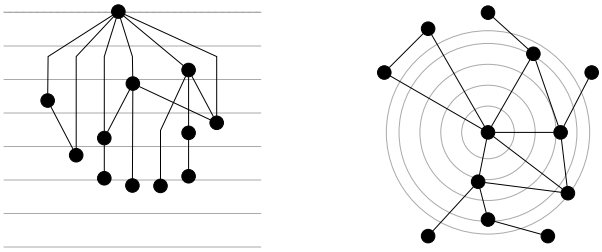


Abbildung 1.2: Beispiele für Status- und Zentralitätslayouts.

durch Kreislayouts, bei dem die Anzahl der Kreuzungen minimiert werden soll.

1.1 Soziale Netzwerke

Eine interessante Anwendung, in der Graphen zu Visualisierung von Informationen verwendet werden, ist die soziale Netzwerkanalyse. Dabei werden die Beziehungen innerhalb einer Gruppe von sozialen Akteuren betrachtet und ihre Struktur analysiert. Ein Netzwerk induziert damit offensichtlich ein Graph, in dem die Knoten die Akteure und die Kanten die Beziehungen repräsentieren. Beispiele für soziale Netzwerke sind die Mitglieder eines Vereines mir der Beziehung *A ist mit B befreundet* und die Beamten einer Behörde und die Beziehung *A ist Vorgesetzter von B*.

Wie im richtigen Leben, stellt sich auch bei der Analyse dieser Netzwerke die Frage, wie *wichtig* oder *bedeutend* die einzelnen Akteure in der Gruppe sind. Dies lässt sich auf verschiedene Arten definieren und berechnen. Steht die Wichtigkeit einmal fest, soll das Netzwerk (d.h. der entsprechende Graph) so dargestellt werden, dass er die *Bedeutung* der Knoten widerspiegelt. Dazu werden zwei verschiedene Layoutarten benutzt, Statuslayouts und Zentralitätslayouts. Abbildung 1.2 zeigt dafür jeweils ein Beispiel.

In Statuslayouts werden wichtigere Akteure weiter oben, in Zentralitätslayouts näher an der Mitte platziert. Welches Layout für ein Netzwerk gewählt wird, hängt von der Art der Beziehung ab. Für die beiden obigen Beispiele wird für die Freundschaftsbeziehung zwischen den Vereinsmitgliedern eher ein Zentralitätslayouts gewählt, für die Vorgesetztenbeziehung zwischen Beamten eher ein Statuslayout.

Die Layouts sollen nicht nur die Bedeutung der einzelnen Akteure (durch den Abstand zur Mitte bzw. die Höhe) sondern auch die Beziehung der Akteure untereinander darstellen. Dazu ist es wichtig, dass die Kanten gut zu erkennen sind, und dafür, dass die Anzahl der Kreuzungen klein ist.

Für Zentralitätslayouts ist es schwierig, die Kreuzungen zu reduzieren, da jeder Knoten beliebig auf dem Kreis mit dem Radius, der seiner Bedeutung entspricht, verschoben werden kann. Nimmt man stattdessen an, das alle Knoten auf dem

gleichen Kreis liegen, wird die Aufgabe deutlich übersichtlicher und man steht vor dem Problem, Kantenkreuzungen in Kreislayouts betrachten zu müssen.

1.2 Inhalt

Diese Arbeit lässt sich in zwei Teile einteilen, der erste beschäftigt sich mit dem theoretischen Hintergrund der Kreuzungsminimierung, der zweite mit der praktischen Anwendung.

Im ersten Teil werden zuerst die grundlegenden Begriffe definiert und einige Optimierungskriterien für Kreislayouts vorgestellt. Ein solches Kriterium ist auch die Anzahl der Kreuzungen, was uns zur Kreuzungsminimierung führt. Danach werden die Graphen charakterisiert, für die es ein kreuzungsfreies Kreislayout gibt und ein Algorithmus zu ihrer Erkennung vorgestellt. Schließlich werden wir sehen, dass die Kreuzungsminimierung für beliebige Graphen ein $\mathcal{NP}$-schweres Problem ist.

Mit Kapitel 4 beginnt der praktische Teil. Zuerst werden Algorithmen betrachtet, die die Anzahl der Kreuzungen in einem gegebenen Layout zu berechnen. Danach werden mehrere aus der Literatur bekannte Heuristiken zur Kreuzungsreduzierung besprochen und zwei neue Verfahren vorgestellt. Schließlich werden die von den Heuristiken erzeugten Layouts miteinander verglichen.

Kapitel 2

Kreislayouts und lineare Layouts

In diesem Kapitel werden zunächst die grundlegenden Begriffe erklärt. Ausgehend von einer Knotenordnung werden lineare Layouts und das eigentliches Thema dieser Arbeit, Kreislayouts, definiert.

Im zweiten Teil werden verschiedene Kriterien für schöne Layouts besprochen. Dies vermittelt einen Eindruck von den Problemen, die bei der Layoutoptimierung auftreten. Das wichtigste Kriterium, die Kreuzungsanzahl, wird erst ab dem nächsten Kapitel besprochen.

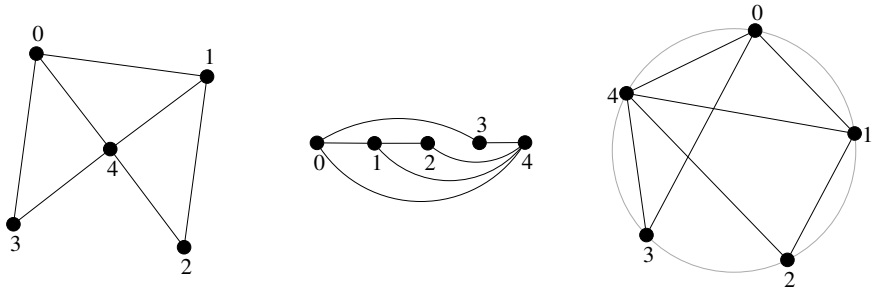
2.1 Grundlagen

Graph

Sei $G = (V, E)$ ein ungerichteter Graph mit $n := |V|$ Knoten und $m := |E|$ Kanten. Zu einem Knoten $v \in V$ bezeichnet $N(v) := \{u \in V : \{u, v\} \in E\}$ die Nachbarn, $d(v) := |N(v)|$ den Knotengrad und $E(v) := \{\{v, u\} \in E\}$ die inzidenten Kanten. Eine Knotenanordnung bzw. Knotenordnung von G ist eine Bijektion $\pi : V \to \{0, \ldots, n-1\}$.

Layout

Ein *Layout* ist eine visuelle Darstellung des Graphen. Diese Arbeit beschäftigt sich mit rein kombinatorischen Gütekriterien von Layouts, im besonderen mit der Anzahl der Kantenkreuzungen. Nicht beachtet werden die eher graphischen Aspekte einer Darstellung, zum Beispiel Größe der Knoten oder Farbe der Kanten. Deshalb ist es zweckmäßig, Knoten und Kanten möglichst uniform darzustellen, die Knoten als Kreise, die Kanten als möglichst einfache Linien.



(a) Graph (b) Lineares Layout (c) Kreislayout

Abbildung 2.1: Ein Beispielgraph und zwei Layouts.

Lineare Layouts

Mit diesen Voraussetzungen induziert jede Knotenanordnung π ein *lineares Layout* des Graphen G. Dabei werden alle Knoten auf einer Geraden platziert, die Knotenordnung gibt die Reihenfolge der Knoten an. Der Abstand zwischen je zwei nebeneinander liegenden Knoten soll dabei gleich sein. $\pi(v)$ gibt also direkt die Position eines Knotens v an. Zwei Knoten heißen *nebeneinander liegend*, wenn sich ihre Positionen nur um eins unterscheiden.

Die einfachste Möglichkeit, um die Kanten darzustellen, sind Geraden. Dies ist in linearen Layouts natürlich nicht sinnvoll. Stattdessen werden oft nur die Kanten zwischen benachbarten Knoten als Geraden gezeichnet, alle anderen als Halbkreise bzw. Kreisbögen oder Bezierkurven. Diese Darstellung ist gut zur Visualisierung der weiter unten besprochenen Probleme geeignet. Abbildung 2.1(b) zeigt ein solches Layout für den Graphen aus Abb. 2.1(a). Oft wird für solche linearen Layouts noch verlangt, dass alle Kanten auf der gleichen Seite der Einbettungsgeraden verlaufen.

Viele Fragestellungen hängen nur von der relativen Lage der Knoten zueinander ab, nicht von ihrer genauen Position. In diesen Fällen ist das Ergebnis für eine Knotenordnung π und ihre Umkehrung $\bar{\pi} : v \mapsto (n-1) - \pi(v)$ gleich. Mathematisch lässt sich dies durch eine Äquivalenzrelation auf der Menge der Knotenordnungen eines Graphen ausdrücken. Zwei Anordnungen seien äquivalent, wenn sie jeweils die gleichen Knoten nebeneinander platzieren. Für die meisten Probleme genügt es dann, die Äquivalenzklassen zu betrachten. In linearen Layouts enthält jede Klasse genau eine Anordnung und ihre Umkehrung, also existieren $\frac{n!}{2}$ unterschiedliche Äquivalenzklassen.

Kreislayouts

Wählt man als Grundlage für die Platzierung der Knoten statt einer Geraden einen Kreis, induziert eine Knotenanordnung π ein *Kreislayout*. Die Überlegungen für li-

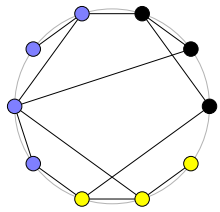


Abbildung 2.2: Gleichfarbige Knoten bilden *fortlaufend angeordnete* Teilmengen.

neare Layouts können auf Kreislayouts übertragen werden, müssen allerdings entsprechend angepasst werden.

Da die Knoten auf einem Kreis liegen, können die Kanten als Geraden gezeichnet werden. Ein solches Layout heißt *geradliniges Kreislayout*. Es ist durch die Knotenordnung bereits vollständig festgelegt. In dieser Arbeit werden nur solche Kreislayouts betrachtet, deshalb ist mit Kreislayout immer implizit auch geradlinig gemeint. Abbildung 2.1(c) zeigt ein Kreislayout des Beispielgraphen aus Abb. 2.1(a).

Der fundamentale Unterschied von Kreislayouts zu linearen Layouts besteht darin, dass die Knoten an den Positionen 0 und $n-1$ nebeneinander liegen. Die Ordnung hat dadurch keinen Anfang und kein Ende und es ergeben sich nicht nur zwei sondern $2n$ Knotenordnungen mit gleicher relativer Reihenfolge der Knoten – zu π die Drehungen um i Positionen $\pi_i : \pi_i(v) = \pi(v) + i \pmod n$ für $1 \leq i < n$ und alle Umkehrungen dieser Anordnungen.

Definiert man wie oben eine Äquivalenzrelation der Knotenordnungen, enthält jede Äquivalenzklasse genau diese $2n$ Elemente. Es gibt also $\frac{(n-1)!}{2}$ unterschiedliche (bzgl. der Äquivalenzrelation) Kreislayouts.

Durch die Wahl eines Startknotens s erhält man aus einem Kreislayout π eine lineare Ordnung, indem die Knoten, beginnend bei s, in der Reihenfolge ihrer Lage auf dem Kreis betrachtet werden. Mathematischer ausgedrückt: die linear Ordnung $\preceq_s^\pi$, $s \in V$, wird definiert durch

$$v \preceq_s^\pi w \iff (\pi(v) - \pi(s) \mod n) \leq (\pi(w) - \pi(s) \mod n)$$

für alle Knoten $v, w \in V$. Auf die Nennung von π wird verzichtet, wenn sich das Kreislayout aus dem Kontext ergibt.

Fortlaufend angeordnete Teilmengen

Eine Teilmenge $V_1 \subseteq V$ heißt *fortlaufend angeordnet* bezüglich einer Knotenordnung π, wenn Knoten von V_1 nebeneinander liegende Positionen belegen, d.h. die Positionen ein Intervall der Länge $|V_1| - 1$ bilden (siehe Abb. 2.2).

In linearen Layouts ist V_1 genau dann fortlaufend angeordnet, wenn ein i existiert, so dass für alle $v \in V_1$ gilt: $\pi(v) \in [i, i + |V_1|[$. In Kreislayouts kann sich das

Intervall über die Position $n-1$ erstrecken, d.h. es gilt $|V_1|+i > n-1$. In diesem Fall muss die Bedingung zu $\pi(v) \in [i, n-1] \cup [0, i+|V_1|-n[$ erweitert werden.

Positionsangaben, die größer als $n-1$ sind, treten in Formeln im Zusammenhang mit Kreislayouts häufig auf. Diese Werte müssen Modulo n interpretiert werden. Dadurch wird die Formulierung von Aussagen erschwert und unübersichtlicher – u.u. sind mehrere Fallunterscheidungen nötig – bereitet aber keine konzeptionellen Probleme.

Zusammenhangskomponenten

Viele der im Zusammenhang mit Layouts interessanten Probleme lassen sich auf die Zusammenhangskomponenten des Graphen einschränken. Damit ist gemeint, dass sich aus Lösungen für die einzelnen Komponenten des Graphen einfach eine Lösung für den kompletten Graphen zusammensetzen lässt. Da die meisten interessanten Probleme $\mathcal{NP}$-vollständig sind und durch Heuristiken nur näherungsweise gelöst werden, wird da das Aufteilen des Graphen die Laufzeit verringert und die Qualität der Ergebnisse verbessert.

Einige Probleme sind sogar auf noch kleinere Komponenten einschränkbar, das Kreuzungsminimierungsproblem in Kreislayouts beispielsweise auf die zweifachen Zusammenhangskomponenten des Graphen.

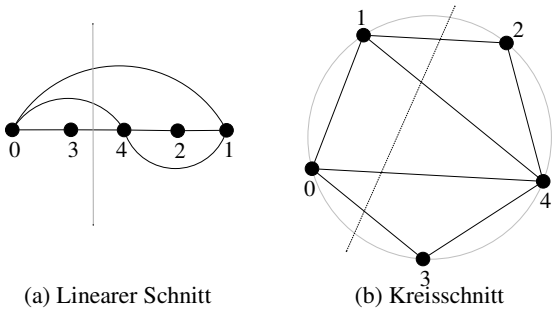
Die Komponenten eines Graphen können mit Erweiterungen des Tiefensuchalgorithmus in $\mathcal{O}(m)$ berechnet werden, es geht also keine Laufzeit verloren.

2.2 Optimierungskriterien

Durch eine konkrete Anwendung wird meistens die Art des Layouts eines Graphen vorgegeben, aber das genaue Layout, d.h. im Fall von linearen Layouts und Kreislayouts die Knotenordnung, ist frei wählbar. Diese soll dann natürlich nicht zufällig gewählt werden, sondern so, dass die Darstellung schön aussieht. Dazu ist es nötig, *schön* durch mathematisch erfassbare Kriterien auszudrücken.

Die Anzahl der Kantenkreuzungen ist wahrscheinlich das wichtigste Kriterium für die Güte eines Layouts. Existieren viele Kreuzungen, ist es schwierig, dem Verlauf der Kanten zu folgen und die Struktur des Graphen zu erkennen. Aber auch andere Kriterien sind für die Qualität eines Layouts wichtig und bereits gut untersucht. Um einen Eindruck von den verschiedenen Möglichkeiten zu vermitteln, werden drei davon in diesem Abschnitt näher betrachtet. Dabei wird auffallen, dass Layoutoptimierung eine schwierige Aufgabe ist, alle Kriterien führen zu $\mathcal{NP}$-vollständigen Problemen.

Natürlich kann man nicht erwarten, ein Layout zu finden, das alle Kriterien gut erfüllt. Trotzdem ist es etwas überraschend, dass sich manche Kriterien sogar ge-



(a) Linearer Schnitt (b) Kreisschnitt

Abbildung 2.3: Schnittweiten des Beispielgraphen.

gensätzlich zueinander verhalten, d.h. ein gutes Layout nach dem einen Kriterium liefert ein extrem schlechtes Layout nach dem anderen Kriterium. Ein Beispiel dafür sind die Kriterien *kurze längste Kante* und *Kantenkreuzung*.

2.2.1 Kleine maximale Schnittweite

Die Idee zu diesem Kriterium ist, dass man nur dem Verlauf einer begrenzte Anzahl von Kanten gleichzeitig folgen kann. Über jede Position soll deshalb nur eine begrenzte Anzahl von Kanten hinweg führen. Das zugehörige Problem heißt *Cutwidth Minimization*.

Anschaulicher und genauer formuliert: Zu einem Graphen $G = (V, E)$ und einer Konstanten k ist ein lineares Layout π gesucht, so dass jede zur Einbettungsgeraden des Graphen senkrecht stehende Gerade höchstens k Kanten schneidet.

Ein lineares Layout für den Beispielgraphen aus Abb. 2.1(a) ist in Abb. 2.3(a) dargestellt. Die ursprüngliche (numerische) Knotenanordnung hat eine Schnittweite von vier, während die geänderte (optimale) Anordnung zu einem Wert von drei führt.

Jede Knotenposition induziert eine zu obiger Beschreibung äquivalente Aufteilung der Knotenmenge in zwei bezüglich der Anordnung π zusammenhängende Teilmengen $V_{i-} := \{v \in V : \pi(v) \leq i\}$ und $V_{i+} := \{v \in V : \pi(v) > i\} = V - V_{i-}$. Der dadurch definierte (lineare) Schnitt ist die Menge der Kanten zwischen diesen beiden Knotenmengen: $S(i) := \{\{u, v\} \in E : u \in V_{i-}, v \in V_{i+}\}$. Das Problem lautet damit:

$$\text{Ist } \max\{|S(i)| : 0 \leq i < n - 1\} \leq k \text{ ?}$$

Nach Garey und Johnson [5] ist dieses Problem $\mathcal{NP}$-vollständig.

Es lässt sich kanonisch auf Kreislayouts übertragen und heißt dann *Circular Cutwidth Minimization Problem* (CCMP). Zu einem Graphen und einer gegebenen Konstanten k ist ein Kreislayout gesucht, in dem jede den Kreis schneidende Gerade höchstens k Kanten kreuzt.

Eine Aufteilung eines Kreislayouts wird im Gegensatz zu linearen Layouts erst durch zwei Knotenpositionen $0 \leq i < j < n$ induziert. Diese liefern die bezüglich der Anordnung π zusammenhängenden Teilmengen $V_{i,j} := \{v \in V : i \leq \pi(v) < j\}$ und $V_{j,i} := V - V_{i,j}$. Der dadurch definierte (Kreis-)Schnitt ist $S(i,j) := \{\{u,v\} \in E : u \in V_{i,j}, v \in V_{j,i}\}$. Damit lautet das *kleine maximale Schnittweite*-Problem folgendermaßen:

Problem 2.1 *Circular Cutwidth Minimization Problem (CCMP)*
Gegeben: Ein Graph $G = (V, E)$ und eine positive ganze Zahl k.
Gesucht: Eine Knotenordnung π, für die gilt:

$$\max\{|S(i,j)| : 0 \leq i < j < n\} \leq k.$$

Für bestimmte Graphen und k ist dieses Problem einfach zu entscheiden. Es gibt keine Lösung, wenn der maximale Knotengrad größer als k ist. In diesem Fall kann immer einer der Knoten mit diesem Grad eine der Teilmengen bilden. Für $k = 1$ ist das Problem offensichtlich nur zu erfüllen, wenn der Graph nur eine Kante besitzt. Für $k = 2$ ist das Problem genau für die Graphen lösbar, bei denen jeder Knoten einen Grad ≤ 2 besitzt. Wird das Layout so gewählt, dass adjazente Knoten nebeneinander liegen, schneidet jede Gerade höchstens zwei Kanten. Im allgemeinen Fall ist das Problem $\mathcal{NP}$-vollständig, wie Erkki Mäkinen [1] durch Reduktion auf *Simple Max Cut* nachweist.

Abbildung 2.3(b) zeigt eine optimale Lösung für den Beispielgraphen. Da der maximale Knotengrad vier beträgt, gibt es sicher keine bessere Lösung.

2.2.2 Kleine Summe der Kantenlängen

Ein weiteres Optimierungskriterium für das Layout eines Graphen ist die Gesamtlänge seiner Kanten. Diese soll möglichst klein sein. Um nicht von der konkreten Darstellung der Kanten, etwa als Halbkreise oder Bezierkurven, abhängig zu sein, betrachtet man statt der Kantenlänge die Differenz der Positionen der Endknoten.

Das zugehörige Problem ist als *Dilation Minimization Problem* und als *Arrangement Problem* bekannt. Für lineare Layouts lautet die Frage:

$$\text{Gilt} \sum_{\{u,v\} \in E} |\pi(u) - \pi(v)| \leq k \ ?$$

Daraus erhält man direkt die Version für Kreislayouts.

Problem 2.2 *Optimal Circular Arrangement Problem (OCAP)*
Gegeben: Ein Graph $G = (V, E)$ und eine positive ganze Zahl k.

Gesucht: Eine Knotenordnung π, für die gilt:

$$\sum_{\{u,v\}\in E} \min\{|\pi(u) - \pi(v)|, n - |\pi(u) - \pi(v)|\} \leq k.$$

Beide *Arrangement*-Probleme sind $\mathcal{NP}$-vollständig und können einfach aufeinander reduziert werden (siehe [1] und [6]).

Eine Heuristik für OCAP kann auch zur Kreuzungsreduzierung benutzt werden. Die Endknoten jeder Kante e teilen das Layout in zwei Teilmengen. Da in einer guten Lösung von OCAP adjazente Knoten tendenziell nahe beieinander liegen, sind die zwei resultierenden Teilmengen unterschiedlich mächtig, eine sehr klein, die andere mit fast allen Knoten sehr groß. Dadurch verlaufen zwischen den beiden Teilmengen normalerweise weniger Kanten als zwischen etwa gleichgroßen Teilmengen, was zu wenigen Kreuzungen, an denen die Kante e beteiligt ist, führt. Da dies für alle Kanten gilt, ergeben sich auch insgesamt relativ wenige Kreuzungen.

2.2.3 Kurze längste Kante

Als letztes Kriterium betrachten wir die Länge der längsten Kante im Layout. Da lange Kanten schwierig zu verfolgen sind, soll diese möglichst kurz sein. Wie bereits oben argumentiert, wird nicht die tatsächliche Länge der Kanten betrachtet, sondern die Differenz der Positionen der Endknoten.

Für lineare Layouts erhält man das gut untersuchte *Linear Bandwidth Minimization Problem*. Zu einem gegebenen Graphen $G = (V, E)$ und einer Konstante k ist eine Knotenanordnung gesucht, in der die längste Kante kürzer als k ist. Die Frage lautet also: Existiert eine Sortierung π, für die gilt:

$$\max\{|\pi(u) - \pi(v)| : \{u, v\} \in E\} \leq k\ .$$

In dieser allgemeinen Form ist das Problem $\mathcal{NP}$-vollständig (siehe [3]). Für jedes feste k existiert aber ein polynominaler Lösungsalgorithmus mit Laufzeit $\mathcal{O}(n^k)$ (siehe [4]).

Die Übertragung auf Kreislayouts ist vollkommen kanonisch.

Problem 2.3 *Circular Bandwidth Minimization Problem (CBMP)*
Gegeben: Ein Graph $G = (V, E)$ und eine positive ganze Zahl k.
Gesucht: Eine Knotenordnung π, für die gilt:

$$\max\{\min\{|\pi(u) - \pi(v)|, n - |\pi(u) - \pi(v)|\} : \{u, v\} \in E\} \leq k.$$

Für $k = 1$ hat das Problem die gleichen einfach zu überprüfenden Lösungen wie $CCMP$ für $k = 2$, nämlich die Graphen mit maximalem Knotengrad zwei. Im Gegensatz zur linearen Version ist es bereits für jedes feste $k > 1$ $\mathcal{NP}$-vollständig.

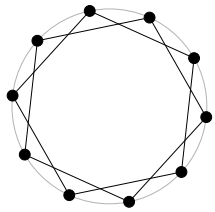


Abbildung 2.4: Beispiel zum Circular Bandwidth Minimization Problem.

CBMP ist ein Beispiel für ein Problem, das sich nicht auf die Zusammenhangskomponenten einschränken lässt, obwohl dies beim *Linear Bandwidth Minimization Problem* sehr wohl möglich ist. Dies sieht man leicht an einem Graphen, der aus zwei disjunkten Kreisen besteht. Für jeden der Kreise lässt sich das Problem für $k = 1$ lösen, für den Gesamtgraphen allerdings nur für $k = 2$ (Abb. 2.4). Dies ist auch der Grund, dass sich die polynominalen Lösungsalgorithmen des linearen Problems nicht auf die Kreisversion übertragen lassen.

Abb. 2.4 zeigt auch, dass *Bandwidth* und *Kreuzungszahl* als Optimierungskriterien in gewisser Weise gegensätzlich sind. Die optimale Lösung der Kreuzungsminimierung teilt jeder Zusammenhangskomponente eines Graphen einen Kreisbogen exklusiv zu und produziert dadurch, wie im Beispiel, sehr lange Kanten, die fast die Maximallänge $\frac{n}{2}$ erreichen. Andererseits verschiebt die optimale Lösung für *Bandwidth* verschiedene Zusammenhangskomponenten stark ineinander, so dass viele Kreuzungen zwischen Kanten verschiedener Komponenten entstehen. Solche Kreuzungen treten in einer optimalen Lösung des Kreuzungsminimierungsproblems nicht auf.

Kapitel 3

Kantenkreuzungen in Kreislayouts

Lässt sich ein gegebener Graph so auf ein Blatt Papier zeichnen, dass sich keine seiner Kanten schneiden? Diese Frage ist wahrscheinlich eines der bekanntesten und für Nicht-Informatiker verständlichsten Probleme der Graphentheorie. Das Studium dieser – planaren – Graphen ist ein wichtiges Teilgebiet der Graphentheorie. Bereits 1974 entwickelten Hopcroft und Tarjan [8] einen Algorithmus, der das Problem in Linearzeit löst.

Für diese Arbeit sind Graphen interessant, die nicht nur in beliebigen Layouts kreuzungsfrei sind, sondern in (geradlinigen) Kreislayouts. Dies sind die *außenplanaren* Graphen, sie werden im zweiten Abschnitt dieses Kapitels ausführlich besprochen. Wir werden sehen, dass nicht jeder planare Graph auch ein kreuzungsfreies Kreislayout besitzt. Deshalb werden wir einen Linearzeitalgorithmus betrachten, der entscheidet, ob ein Graph außenplanar ist und gegebenenfalls ein entsprechendes Layout liefert.

Für nicht-außenplanare Graphen suchen wir ein Kreislayout mit möglichst wenigen Kreuzungen. Im dritten Abschnitt werden wir sehen, dass dieses nicht effizient berechnet werden kann, da das Kreuzungsminimierungsproblem $\mathcal{NP}$-schwer ist.

Zuerst einige grundlegende Definitionen und Überlegungen zu Kantenkreuzungen.

3.1 Grundlagen

Für die Kreuzungsminimierung betrachten wir schlichte, ungerichtete Graphen, da Mehrfachkanten die Beschreibungen oft unnötig komplizieren würden. Die Algorithmen zum Kreuzungszählen in Kapitel 4 und zur Kreuzungsminimierung in Kapitel 6 lassen sich einfach auf gewichtete Graphen übertragen. Dabei wird jede Kreuzung mit dem Produkt der Gewichte der beiden Kanten gezählt. Mehrfach-

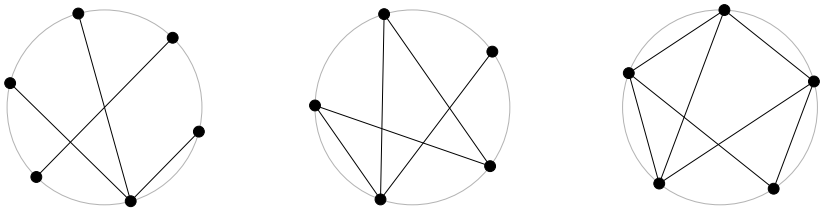


Abbildung 3.1: Beispiele für Kantenkreuzungen.

kanten können damit durch entsprechende Kantengewichte repräsentiert werden.

In geradlinigen Kreislayouts hängt die Anzahl der Kantenkreuzungen, wie auch die anderen Optimierungskriterien, nur von der Knotenordnung ab. Zwei Kanten $e_1 = \{u_1, v_1\}$ und $e_2 = \{u_2, v_2\}$ schneiden sich genau dann, wenn die Endknoten von e_1 und e_2 durch das Layout abwechselnd auf dem Kreis angeordnet werden (siehe Abb. 3.1). Diese Bedingung lässt sich schnell, in konstanter Zeit, überprüfen.

Die konkrete Implementierung ist etwas unübersichtlich, da einige Vergleiche und Fallunterscheidungen nötig sind. Der Algorithmus *IsCrossing* überprüft, ob sich zwei Kanten e_1 und e_2 im Layout π kreuzen. Dazu werden zuerst die Startknoten $s(e_i)$ und die Zielknoten $t(e_i)$ der Kanten ermittelt. Danach wird getestet, welcher der beiden Startknoten die kleinere Position besitzt, um die zu überprüfende Bedingung auswählen zu können.

Algorithm 1: IsCrossing

Eingabe: Ein Kreislayout π und die Kanten $\{u_1, v_1\}$ und $\{u_2, v_2\}$.
Ausgabe: Kreuzen sich die Kanten?

1. **if** $(\pi(u_1) < \pi(v_1))$ **then** $s_1 \leftarrow u_1, t_1 \leftarrow v_1$
2. **else** $s_1 \leftarrow v_1, t_1 \leftarrow u_1$
3. **if** $(\pi(u_2) < \pi(v_2))$ **then** $s_2 \leftarrow u_2, t_2 \leftarrow v_2$
4. **else** $s_2 \leftarrow v_2, t_2 \leftarrow u_2$

5. **if** $(s_1 < s_2)$ **then** **return** $(s_2 < t_1 < t_2)$
6. **else** **return** $(s_1 < t_2 < t_1)$

3.1.1 Kreuzungsanzahl

In diesem Abschnitt werden Bezeichnungen für oft benötigte Anzahlen von Kreuzungen (kurz: *Kreuzungsanzahlen*) definiert. Dazu sei $G = (V, E)$ ein ungerichteter, schlichter Graph, für den die Knotenordnung π ein Kreislayout definiert.

Die Anzahl aller Kreuzungen im Layout π von G wird mit $\chi_\pi(G)$ bezeichnet. Für zwei disjunkte Teilmengen $E_1, E_2 \subseteq E, E_1 \cap E_2 = \emptyset$ bezeichnet $\chi_\pi(E_1, E_2)$

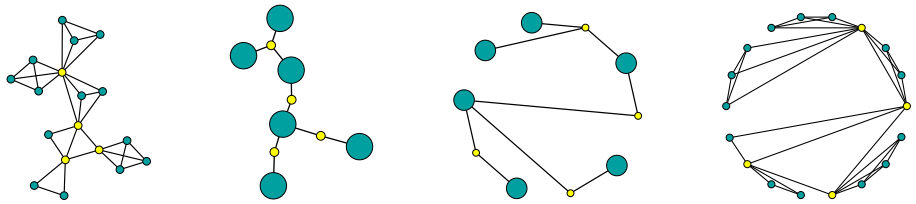


Abbildung 3.2: Konstruktion eines Kreislayouts mit Hilfe des Komponenten-Schnittknoten-Baums der zweifachen Zusammenhangskomponenten (Schnittknoten sind heller dargestellt).

die Anzahl der Kreuzungen, die von je einer Kante aus E_1 und aus E_2 verursacht werden. $\chi_\pi(E_1)$ ist eine Abkürzung für $\chi_\pi(E_1, E - E_1)$, also die Anzahl von Kreuzungen zwischen Kanten aus E_1 und allen anderen Kanten.

Häufig sind die Kreuzungen zwischen den zu einem Knoten v inzidenten Kanten $E(v)$ und allen anderen interessant. Man nennt sie *die Kreuzungen des Knoten* v und schreibt $\chi_\pi(v) := \chi_\pi(E(v)) = \chi_\pi(E(v), E - E(v))$. Entsprechend gilt für zwei Knoten $u, v \in V$: $\chi_\pi(u, v) := \chi_\pi(E(u), E(v))$. Wenn sich das betrachtete Kreislayout aus dem Kontext ergibt, wird π bei diesen Bezeichnungen weggelassen.

Die *Kreuzungszahl* (englisch: *crossing number*) $cr(G)$ des Graphen ist die minimale Kreuzungsanzahl aller Kreislayouts: $cr(G) := \min_\pi \chi_\pi(G)$.

Einfache obere Schranken für die Kreuzungsanzahl sind $\frac{1}{24}n(n-1)(n-2)(n-3)$, die Anzahl der Kreuzungen des vollständigen Graphen K_n, und $\frac{1}{2}m(m-1)$, die Kreuzungsanzahl eines Layouts, in dem jede Kante jede andere kreuzt.

3.1.2 Zusammenhangskomponenten

Viele Layoutoptimierungsprobleme können auf die Zusammenhangskomponenten des betrachteten Graphen eingeschränkt werden. Dies gilt auch für die Kreuzungsminimierung. In einem bezüglich Kreuzungsanzahl optimalen Layout schneiden sich die Kanten verschiedener Komponenten offensichtlich nicht.

Das Problem kann sogar noch weiter eingeschränkt werden, auf die zweifachen Zusammenhangskomponenten. Der Komponenten-Schnittknoten-Baum eines Graphen besitzt, wie jeder Baum, ein kreuzungsfreies Layout. Aus diesem erhält man mit den Layouts für die einzelnen Komponenten ein Layout für den gesamten Graphen, in dem sich genau die Kanten kreuzen, die sich schon in einem der Teillayouts kreuzen. Abbildung 3.2 zeigt dieses Vorgehen.

Im Folgenden werden deshalb nur noch zweifach zusammenhängende Graphen

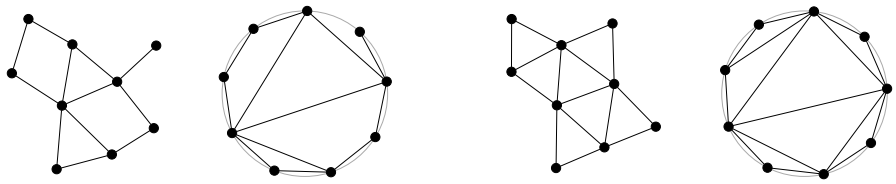


Abbildung 3.3: Außenplanare Graphen.

betrachtet.

3.2 Außenplanare Graphen

In diesem Abschnitt betrachten wir Graphen, die Kreislayouts ohne Kantenkreuzungen besitzen, sogenannte außenplanare Graphen. Ziel ist ein Algorithmus, der in Linearzeit entscheidet, ob ein Graph außenplanar ist und gegebenenfalls ein kreuzungsfreies Layout berechnet. Dazu werden maximale außenplanare Graphen betrachtet, die, analog zu maximalen planaren Graphen, einige schöne Eigenschaften besitzen.

3.2.1 Definition

Graphen, die sich so in die Ebene einbetten lassen, dass alle Knoten an eine ausgezeichnete Facette grenzen und sich keine Kanten kreuzen, heißen *außenplanare* Graphen (englisch: *outerplanar*). Der Name kommt daher, dass die ausgezeichnete Facette normalerweise als die äußere, den Graph umschließende Facette gezeichnet wird.

3.2.2 Beobachtungen

Die außenplanaren Graphen sind offensichtlich genau die Graphen mit kreuzungsfreien Kreislayouts – ein kreuzungsfreies Kreislayout ist eine außenplanare Einbettung, andererseits gibt eine außenplanare Einbettung direkt eine Reihenfolge der Knoten auf einem Kreis vor, bei der sich geradlinige Kanten nicht kreuzen. Abb. 3.3 zeigt für zwei Beispielgraphen jeweils eine außenplanarer Einbettung und ein entsprechendes Kreislayout.

Die Menge der außenplanaren Graphen ist eine echte Teilmenge der planaren Graphen – das kleinste Beispiel für einen planaren Graphen, der kein kreuzungsfreies Kreislayout besitzt, ist der K_4 (siehe Abb. 3.6).

Interessant ist, dass jeder planare Graph sowohl mit geradlinigen Kanten als auch als nicht-geradliniges Kreislayout planar zu zeichnen ist. Erst durch die Kombination der beiden Eigenschaften *geradlinig* und *Kreislayout* wird eine wirkliche

Einschränkung erreicht.

Der Test auf Außenplanarität kann mit der vorherigen Argumentation auf die zweifachen Zusammenhangskomponenten eines Graphen eingeschränkt werden. In einem zweifach zusammenhängenden außenplanaren Graphen bilden die zur äußeren Facette inzidenten Kanten offensichtlich einen Hamilton-Kreis, sie heißen deshalb *Kreiskanten*. Die anderen Kanten sind die *Sehnen* dieses Kreises.

3.2.3 Maximale außenplanare Graphen

Planare Graphen, in die sich keine Kanten zwischen (nicht-adjazenten) Knoten einfügen lassen, so dass sie planar bleiben, heißen *maximale* planare Graphen. Diese Eigenschaft lässt sich direkt auf außenplanare Graphen übertragen. Ein *maximaler außenplanarer* Graph (MAP) ist ein außenplanarer Graph, in den sich keine Kante zwischen zwei beliebigen nicht-adjazenten Knoten einfügen lässt, so dass er außenplanar bleibt. Der rechte Graph in Abb. 3.3 ist maximal außenplanar.

Maximale außenplanare Graphen besitzen einige schöne und leicht zu überprüfende Eigenschaften. Wie bei maximalen planaren Graphen ist auch in einem MAP durch die Knotenanzahl die Anzahl der Kanten und Facetten bereits festgelegt. Es gilt:

Lemma 3.1 *Jeder maximale außenplanare Graph mit n Knoten besteht aus genau $2n - 3$ Kanten und $n - 1$ Facetten.*

Zusätzlich besitzt jeder MAP mindestens zwei Knoten mit Grad 2, sogenannte *2-Knoten*. Da ein MAP trianguliert ist, sind die beiden adjazenten Knoten eines 2-Knotens miteinander verbunden. Wird ein 2-Knoten aus einem Graphen entfernt, erhält man wieder einen MAP mit zwei 2-Knoten und $2(n - 1) - 3$ Kanten. Da MAPs planar sind, begrenzt keine Kante mehr als zwei Facetten. Damit besitzt man genug Eigenschaften, um maximale außenplanare Graphen zu charakterisieren.

Satz 3.2 *Ein Graph G mit n Knoten ist genau dann maximal außenplanar, wenn er entweder ein K_3 ist oder die folgenden vier Eigenschaften erfüllt:*

1. *G hat genau $2n - 3$ Kanten,*
2. *G hat mindestens zwei 2-Knoten,*
3. *keine Kante begrenzt mehr als zwei Facetten und*
4. *durch Löschen eines beliebigen 2-Knotens erhält man wieder einen MAP.*

Mit Abb. 3.3 ist der Satz anschaulich klar. Einen ausführlichen Bweis liefert Mitchell [9].

3.2.4 Algorithmus MAP-Test

Der Algorithmus testet nicht direkt auf Außenplanarität, sondern auf maximale Außenplanarität. Dies ist keine Einschränkung, da sich jeder außenplanare Graph in einen maximalen außenplanaren Graph triangulieren lässt, analog zur Triangulierung planarer Graphen. Der rechte Graph in Abb. 3.3 ist eine Triangulierung des linken. Eine Triangulierung kann in linearer Zeit berechnet werden. Außerdem gilt das folgende Lemma:

Lemma 3.3 *Ein Graph G ist genau dann außenplanar, wenn er durch Triangulierung in einen maximalen außenplanaren Graphen transformiert werden kann.*

Idee

Das Vorgehensweise des Algorithmus orientiert sich an der rekursiven Struktur des vorhergehenden Satzes. Sukzessive werden 2-Knoten aus dem Graphen entfernt. Jeder der dabei entstehenden Teilgraphen muss wieder ein MAP sein, also die Bedingungen erfüllen. Sind nur noch drei Knoten übrig, muss der Teilgraph ein K_3 sein. Entsteht bei einem der Schritte ein Graph, der eine der Bedingungen verletzt oder bleibt am Ende kein K_3 übrig, war der Graph kein MAP.

Korrektheit

Die Korrektheit des Algorithmus ergibt sich folgendermaßen. Tritt während der Abarbeitung des Algorithmus kein Fehler auf, bleibt am Ende ein Graph mit drei Knoten und drei Kanten übrig, also ein K_3, und die Endbedingung ist erfüllt. Die Zeilen 5 und 12 garantieren die zweite Bedingung des Satzes. In Zeile 4 wird die korrekte Kantenanzahl des Ausgangsgraphen überprüft. Da in jedem Schritt mit einem 2-Knoten auch zwei Kanten gelöscht werden, besitzt jeder der Teilgraphen induktiv die richtige Kantenanzahl.

Die dritte Bedingung wird mit der Liste von Knotenpaaren *pairs* überprüft. Wenn der Graph ein MAP war, enthält sie am Ende alle Sehnen genau einmal (Zeilen 14-16). Kommt ein Knotenpaar in der Liste mehrmals vor, begrenzt die entsprechende Kante mindestens drei Facetten, nämlich jeweils eine dreieckige Facette mit dem 2-Knoten, bei deren Entfernung sie in die Liste eingefügt wurde, und eine im Restgraphen nach der Entfernung aller dieser 2-Knoten noch bestehende Facette. Ein Beispiel für einen solchen Fall ist der Graph in Abbildung 3.4, die Kante $(0, 4)$ wird zweimal in die Liste eingefügt. Kanten die nie in die Liste *pairs* eingetragen wurden, sind im Laufe des Algorithmus mit einem inzidenten 2-Knoten gelöscht worden. Zu diesem Zeitpunkt begrenzten sie höchstens zwei Facetten. Da sie nie in die Liste eingetragen wurden, können sie auch früher nicht an mehr Facetten gegrenzt haben.

Algorithm 2: MAP-Test

Eingabe: Ein Graph G.
Ausgabe: Ist G ein MAP?

1. **list of (pair of nodes)** pairs $\leftarrow \emptyset$
2. **stack of nodes** 2verts $\leftarrow$ Alle 2-Knoten von G
3. **list of edges** edges $\leftarrow$ Alle Kanten von G

4. **if** (edges.length()) $\neq 2n - 3$) **then return** FALSE
5. **if** (2verts.length() < 2) **then return** FALSE

6. **while** (G.numberOfNodes() > 3) **do**
7. Sei v ein 2-Knoten von G
8. Seien w_1, w_2 die beiden zu v adjazenten Knoten
9. Füge das Paar $\{w_1, w_2\}$ zu *pairs* hinzu
10. Lösche v aus G
11. Falls neue 2-Knoten entstanden sind, füge sie zu *2verts* hinzu
12. **if** (2verts.length()) < 2) **then return** FALSE

13. **if** (G.numberOfEdges() $\neq 3$) **then return** FALSE
14. Sortiere *edges*
15. Sortiere *pairs*
16. Falls es ein Element in *pairs* gibt, das nicht in *edges* ist
    return FALSE
17. Sonst
    return TRUE

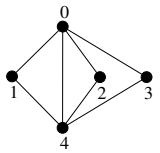


Abbildung 3.4: Ein nicht maximal-außenplanarer Graph.

Laufzeit

Die ersten fünf Schritte besitzen offensichtlich lineare Laufzeit, ebenso die Schritte 13 bis 16, wenn die Listen mit *Bucket Sort* sortiert werden. Die Schleife von Zeile sechs bis Zeile 12 wird n-mal durchlaufen. Die kritische Stelle ist hier das Löschen des Knoten und der Test, ob neue 2-Knoten entstanden sind. Beim Löschen des 2-Knotens müssen die entsprechenden Einträge aus der Adjazenzliste seiner beiden benachbarten Knoten gelöscht werden, dies ist in konstanter Zeit möglich. Danach müssen nur diese beiden Knoten überprüft werden, ob sie zu 2-Knoten geworden sind, was ebenfalls in konstanter Zeit möglich ist. Damit liegt die Gesamtlaufzeit des Algorithmus in $\mathcal{O}(n)$.

3.2.5 Außenplanare Einbettungen finden

Mit dem Algorithmus MAP-Test erhält man zunächst nur eine Antwort auf die Frage, ob eine kreuzungsfreie (außenplanare) Einbettung existiert. Durch eine kleine Erweiterung erhält man auch eine zugehörende Einbettung. Die Liste *pairs* enthält alle Sehnen. Entfernt man diese aus der Liste *edges*, bleiben alle *Kreiskanten* übrig. Diese ergeben direkt die Einbettung.

Algorithm 3: MAP

17. Lösche die Elemente von *pairs* aus *edges*
18. Erstelle aus den Kanten in *edges* die Einbettung

3.2.6 Zusammenfassung

Es gibt einen einfachen Linearzeitalgorithmus, um Graphen auf Außenplanarität zu testen und sie gegebenenfalls kreuzungsfrei in ein Kreislayout einzubetten. Damit ist die Kreuzungsminimierung für diese Graphen effizient gelöst. Als nächstes stellt sich die Frage, wie mit nicht-außenplanaren Graphen verfahren werden soll.

3.3 Kreuzungsminimierung ist $\mathcal{NP}$-schwer

In Kreislayouts erscheint die Kreuzungsminimierung zuerst als einfaches Problem, einfacher zumindest als Kreuzungsminimierung in allgemeinen oder geradlinigen Layouts, da die Kantenkreuzungen nur von der Knotenordnung abhängen.

Leider ist es das nicht: 1987 wurde bewiesen, dass das Problem $\mathcal{NP}$-schwer ist [10]. Für den Beweis betrachten wir das zur Kreuzungsminimierung gehörende Entscheidungsproblem:

Problem 3.1 *Circular Crossing Reduction Problem (CCRP)*
Gegeben: Ein Graph $G = (V, E)$ und eine positive ganze Zahl k.
Gesucht: Eine Kreislayout π, so dass es höchstens k Kantenkreuzungen gibt.

Eine modifizierte Version des bereits bekannten und $\mathcal{NP}$-vollständigen *Optimal Linear Arrangement Problem* wird auf dieses Problem reduziert. Überraschenderweise ist der Beweis für Multigraphen einfacher und anschaulicher zu führen, er wird deshalb zuerst ausgeführt und danach für gewöhnliche Graphen angepasst.

3.3.1 Modifiziertes OLAP

Beim *Optimal Arrangement Problem* soll die Summe der Kantenlängen minimiert werden. Für lineare Layouts ist es in Kapitel 2.2.2 folgendermaßen definiert worden:

Problem 3.2 *Optimal Linear Arrangement Problem (OLAP)*
Gegeben: Ein Graph $G = (V, E)$ und eine positive ganze Zahl k.
Gesucht: Eine Knotenordnung π, für die gilt:

$$\sum_{\{u,v\}\in E} |\pi(u) - \pi(v)| \leq k.$$

Da jede Kante mindestens die Länge eins und höchstens die Länge $n - 1$ besitzt, ist das Problem für $k < m$ und $k > m(n - 1)$ einfach zu entscheiden – im ersten Fall gibt es keine Lösung, im zweiten ist jede Reihenfolge eine Lösung. Da also nur $k >= m$ betrachtet werden müssen, kann die Bedingung umgeformt werden zu:

$$\sum_{\{u,v\}\in E} (|\pi(u) - \pi(v)| - 1) \leq k - m.$$

Problem 3.3 *Modifiziertes OLAP*
Gegeben: Ein Graph $G = (V, E)$ und eine positive ganze Zahl $k \leq m(n - 1)$.

Gesucht: Eine Knotenordnung π, für die gilt:

$$\sum_{\substack{\{u,v\}\in E,\\ \pi(u)<\pi(v)}} (\pi(u) - \pi(v) - 1) \leq k.$$

3.3.2 Beweis für Multigraphen

Durch einen gewöhnlichen Graph $G_0 = (V_0, E_0)$ mit $n := |V_0|$ und eine positive ganze Zahl k_0 wird eine Instanz des modifizierten OLAP definiert. Daraus konstruieren wir den Multigraphen $G = (V, E)$, der sich von G_0 durch einen zusätzlichen Knoten $\bar{v}$, der mit jedem Knoten von V_0 durch jeweils n^4 Kanten verbunden ist, unterscheidet, und eine positive Konstante k:

- $V := V_0 \cup \{\bar{v}\}$,
- $E := E_0 \cup \bar{E}$, wobei $\bar{E} := \bigcup_{v\in V_0}$ (n^4 Kanten zwischen v und $\bar{v}$),
- $k := n^4 \cdot k_0 + n^4 - 1$.

Lemma 3.4 *Es seien die Graphen und Konstanten wie beschrieben gegeben. Dann ist die Knotenordnung $\pi_0 : V_0 \to \{0, \ldots, n-1\}$ eine Lösung des modifizierte OLAP für G_0 und k_0 genau dann, wenn ein Kreislayout π von G existiert mit $\chi_\pi(G) \leq k$, wobei $\pi_{|V_0} = \pi_0$.*

Beweis
$\Longrightarrow$: Seien ein Graph $G_0 = (V_0, E_0)$ mit $n := |V_0|$, eine positive ganze Zahl k_0 und eine Knotenordnung $\pi_0 : V_0 \to \{0, \ldots, n-1\}$ gegeben, so dass die Bedingung

$$\sum_{\substack{\{u,v\}\in E,\\ \pi_0(u)<\pi_0(v)}} (\pi_0(u) - \pi_0(v) - 1) \leq k_0$$

erfüllt wird. Daraus seien der Graph $G = (V, E)$ und k wie beschrieben konstruiert. Durch $\pi : V \to \{0, \ldots, n\}$ mit $\pi_{|V_0} = \pi_0$ und $\pi(\bar{v}) := n$ wird ein Kreislayout für G induziert.

Für jede Kante $e = \{u, v\} \in E_0$ mit $\pi(u) < \pi(v)$ gilt:

$$\chi_\pi(\{e\}, \bar{E}) = n^4 \cdot (\pi(v) - \pi(u) - 1).$$

Daraus folgt:

$$\chi_\pi(E_0, \bar{E}) = n^4 \cdot \sum_{\substack{\{u,v\}\in E,\\ \pi_0(u)<\pi_0(v)}} (\pi_0(u) - \pi_0(v) - 1) \leq n^4 \cdot k_0$$

Die neuen Kanten aus $\bar{E}$ schneiden sich nicht, also gilt $\chi_\pi(\bar{E}) = 0$. Da es weniger als n^2 Kanten in E_0 gibt, gilt $\chi_\pi(E_0) \leq n^4 - 1$. Insgesamt gilt für die Anzahl von Kreuzungen in G:

$$\begin{aligned} \chi_\pi(G) &= \chi_\pi(\bar{E}) + \chi_\pi(E_0) + \chi_\pi(E_0, \bar{E}) \\ &\leq (n^4 - 1) + n^4 \cdot k_0 = k \end{aligned}$$

$\Longleftarrow$: Seien die Graphen $G_0 = (V_0, E_0)$ und $G = (V, E)$ sowie die Parameter k_0 und k wie oben gegeben. Weiter sei die Knotenordnung π mit $\pi(\bar{v}) = n$ eine Lösung des CCRP für G und k, also $\chi_\pi(G) \leq k$. Dann ist $\pi_0 := \pi_{|V_0}$ eine Lösung des modifizierten OLAP für G_0 und k_0.

Sonst gilt:

$$\sum_{\substack{\{u,v\} \in E, \\ \pi_0(u) < \pi_0(v)}} (\pi_0(u) - \pi_0(v) - 1) \geq k_0 + 1$$

Daraus folgt:

$$\begin{aligned} \chi_\pi(E) \geq \chi_\pi(E_0, \bar{E}) &= \sum_{e \in E_0} \chi_\pi(\{e\}, \bar{E}) \\ &= n^4 \cdot \sum_{\substack{\{u,v\} \in E, \\ \pi_0(u) < \pi_0(v)}} \pi_0(u) - \pi_0(v) - 1 \\ &\geq n^4 \cdot k_0 + n^4 > k \\ &\quad \text{Widerspruch!} \end{aligned}$$

□

Satz 3.5 *Multi-CCRP ist $\mathcal{NP}$-vollständig.*

Beweis
Multi-CCRP gehört zur Klasse $\mathcal{NP}$. Die beschriebene Transformation einer Instanz des modifizierten OLAP nach Multi-CCRP ist in polynominaler Zeit durchführbar. Das Lemma 3.4 zeigt schließlich, dass eine Lösung für eine Instanz des einen Problems direkt zu einer Lösung für die entsprechende Instanz des anderen Problems führt. □

3.3.3 Modifikationen für gewöhnliche Graphen

Wenn das obige Vorgehen für gewöhnliche Graphen übernommen werden soll, müssen die n^4 Kanten, die von jedem Knoten aus V_0 zu $\bar{v}$ führen, ersetzt werden. Die erste Idee ist, $\bar{v}$ durch n^4 neue Knoten zu ersetzen. Dadurch ist allerdings nicht mehr garantiert, dass V_0 im Kreislayout fortlaufend angeordnet wird. Also ist ein raffinierterer Ansatz notwendig.

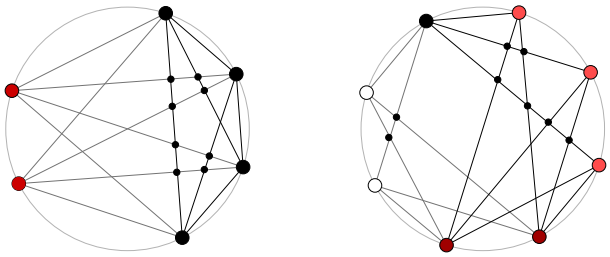


Abbildung 3.5: $g3(4,2) = 8$ und $g4(1,3,2,2) = 8$.

Als erstes betrachten wir vier Funktionen $g_1, g_2, g_3, g_4 : \mathbf{N} \to \mathbf{N}$, die die Kreuzungsanzahl bestimmter Graphen beschreiben. Mit diesen Funtionen kann im nächsten Abschnitt der Beweis für gewöhnliche Graphen geführt werden.

g_1 ist die Anzahl von Kreuzungen, die ein vollständiger Graph in einem Kreislayout erzeugt, also $g_1(p) := \chi(K_p)$. Diese Definition ist aufgrund der Symmetrie vollständiger Graphen unabhängig von einer bestimmten Knotenordnung. Abb. 3.6 zeigt die Graphen K_4 und K_7 als Beispiel.

Für g_2 betrachten wir einen vollständig bipartiten Graphen $K_{p,q} = (V_1 \cup V_2, E)$ mit $p := |V_1|$ und $q := |V_2|$ und eine Knotenordnung π_2, die beide Knotenmengen fortlaufend anordnet. Dann ist $g_2(p,q) := \chi_{\pi_2}(K_{p,q})$.

Als nächstes betrachten wir den Graphen $H_3 := (V_1 \cup V_2, E_1 \cup E_2)$ mit $p := |V_1|$ und $q := |V_2|$, der durch seine Teilgraphen $H_1 := (V_1, E_1)$ und $H_2 := (V_1 \cup V_2, E_2)$ definiert wird. H_1 ist ein vollständiger Graph K_p, d.h. $E_1 := \{\{u,v\} : u,v \in V_1\}$, H_2 ein vollständig bipartiter Graph $K_{p,q}$ und damit $E_2 := \{\{u,v\} : u \in V_1, v \in V_2\}$. Damit ist $g_3(p,q) := \chi(E_1, E_2)$. Wie g_1 ist auch g_3 unabhängig von der Knotenordnung, da jeder Knoten $v \in V_2$ an jeder Position immer gleichviele Kreuzungen mit E_1 erzeugt (da H_1 symmetrisch ist). Abb. 3.5 zeigt den $g3(4,2)$ und die entsprechenden Kreuzungen.

Für g_4 wird der Graph $H_4 := (V_1 \cup V_2 \cup V_3 \cup V_4, E_1 \cup E_2 \cup E_3 \cup E_4)$ benötigt. Die vier Kantenmengen sind durch $E_i := \{\{u,v\} : u \in V_i, v \in V_{i+1 \pmod 4}\}$ für $i = 1, \ldots 4$ definiert, d.h. die Subgraphen $(V_i \cup V_{i+1}, E_i)$ sind bipartite Graphen. Mit $p := |V_1|$, $q := |V_2|$, $r := |V_3|$, $s := |V_4|$ und einer Knotenordnung π_4, die die Knotenmengen in dieser Reihenfolge fortlaufend anordnet, ist $g_4(p,q,r,s) := \chi_{\pi_4}(E_1, E_2) + \chi_{\pi_4}(E_3, E_4)$ (siehe Abb. 3.5 für ein Beispiel).

Lemma 3.6 *Für die so definierten Funktionen $g_1, \ldots, g_4$ gilt:*

$$g_1(p) = \frac{1}{24}p(p-1)(p-2)(p-3). \tag{3.1}$$

$$g_2(p,q) = \frac{1}{4}pq(p-1)(q-1). \tag{3.2}$$

$$g_3(p,q) = \frac{1}{6}pq(p-1)(p-2). \tag{3.3}$$

$$g_4(p,q,r,s) = \frac{1}{2}pr(q(q-1)+s(s-1)). \tag{3.4}$$

Zum Beweis benötigen wir das folgende Korollar.

Korollar 3.7 *Seien $K_n = (V,E)$ ein vollständiger Graph und u ein zusätzlicher Knoten, der durch die Kanten $E(u) := \{\{u,v\} : v \in V\}$ mit allen Knoten aus V verbunden ist. Dann ist in jedem Kreislayout von $(V \cup \{u\}, E \cup \{E(u)\})$ die Anzahl der Kreuzungen zwischen E und $E(u)$*

$$\chi(E,u) = \frac{1}{6}n(n-1)(n-2). \tag{3.5}$$

Beweis
Mit Induktion lässt sich dies leicht beweisen. Für $n = 3$ ist die Gleichung erfüllt, es gibt genau eine Kreuzung (siehe Abb. 3.6).

Für den Induktionsschritt betrachten wir zunächst einen der beiden Knoten $w \in V$ des K_n, die im Layout neben u liegen. Um die Notation zu vereinfachen, bezeichne v_i, $0 \leq i < n$, den Knoten an Position i, u sei an Position n platziert und w sei der Knoten v_0 . Nach Induktionsvoraussetzung haben seine inzidenten Kanten $E(w) := \{\{w,v\} : v \in V - w\}$ mit den Kanten $E - E(w)$ genau

$$\chi(E - E(w), E(w)) = \chi(w) = \frac{1}{6}(n-1)(n-2)(n-3)$$

Kreuzungen. $E(u)$ hat gleichviele Kreuzungen mit $E - E(w)$ wie $E(w)$. Außerdem kreuzen sich die Kanten von $E(u)$ und $E(w)$. Die Kante $\{w,v_1\} \in E(w)$ hat keine Kreuzungen mit Kanten aus $E(u)$, $\{w,v_2\}$ wird von genau einer Kante aus $E(u)$ geschnitten, $\{w,v_3\}$ bereits von zweien, usw. Also gilt:

$$\begin{aligned}
\chi(E,E(u)) &= \chi(w) + \chi(E(u),E(w)) \\
&= \frac{1}{6}(n-1)(n-2)(n-3) + \sum_{i=1}^{n-2} i \\
&= \frac{1}{6}(n-1)(n-2)(n-3) + \frac{1}{2}(n-1)(n-2) \\
&= \frac{1}{6}n(n-1)(n-2).
\end{aligned}$$

□

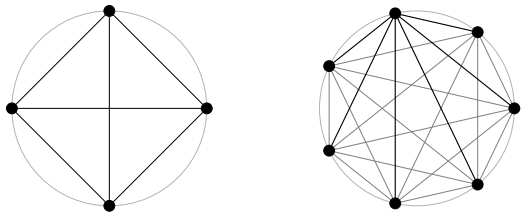


Abbildung 3.6: K_4 und K_7.

Beweis (Von Lemma 3.6.)
Gleichung (3.3) für g_3 erhält man direkt aus (3.5). Offensichtlich erzeugen die inzidenten Kanten $E(u)$ jedes Knotens $u \in V_2$ gleich viele Kreuzungen mit den Kanten E_1 des vollständigen Teilgraphen. Nach dem Korollar sind das für jeden der q Knoten in V_2 $\frac{1}{6}p(p-1)(p-2)$ Kreuzungen.

Gleichung (3.1) für g_1 erhält man mit (3.5) durch Induktion. Für $p = 3$ ist die Gleichung erfüllt. Weiter gilt:

$$\begin{aligned} g_1(p) &= g_1(p-1) + g_3(p-1,1) \\ &= \frac{1}{24}(p-1)(p-2)(p-3)(p-4) + \frac{1}{6}(p-1)(p-2)(p-3) \\ &= \frac{1}{24}p(p-1)(p-2)(p-3). \end{aligned}$$

Sei $K_{p,q} = (V_1 \cup V_2, E)$ ein vollständig bipartiter Graph mit $p := |V_1|$ und $q := |V_2|$. O.E. seien $V_1 := \{v_0, \ldots, v_{p-1}\}$, $V_2 := \{v_p, \ldots, v_{p+q-1}\}$ und die Knotenordnung π definiert durch $\pi(v_i) := i$, $0 \leq i < p+q$. Wie in der Definition von g_2 beschrieben, ordnet π die Knotenmengen V_1 und V_2 fortlaufend an. Der Beweis läuft per Induktion über p. Da die Argumente von g_2 vertauschbar sind ($g_2(p,q) = g_2(q,p)$), ist keine separate Induktion über q nötig. Ist $p = 1$ oder $q = 1$ gibt es keine Kreuzungen, $g_2(p,q) = 0$. Für $p, q > 1$ betrachten wir den Knoten v_0 und seine inzidenten Kanten $E(v_0) = \{e_i := \{v, v_{p+i} : 0 \leq i < q\}$. Da sich die Kanten aus $E(v_0)$ logischerweise untereinander nicht schneiden, gilt für die Kreuzungsanzahl:

$$g_2(p,q) = \chi_\pi(E - E(v_0)) + \chi_\pi(v_0)$$

.

Der Graph $((V_1 - v_0) \cup V_2\,,\, E - E(v_0))$ ist wieder vollständig bipartit und hat nach Induktionsvoraussetzung

$$\chi_\pi(E - E(v_0)) = g_2(p-1,q) = \frac{1}{4}(p-1)q(p-2)(q-1)$$

Kreuzungen. Für $0 \leq i < q-1$ schneidet jede Kante $e_i \in E(v_0)$ die Kanten aus $E - E(v_0)$, die zu den Knoten $v_{p+i+1}, \ldots, v_{p+q-1}$ inzident sind, dies sind

$(p-1)(q-1-i)$ viele Kanten. Die Kante e_{q-1} schneidet keine Kanten. Also gilt:

$$\begin{aligned} g_2(p,q) &= \chi_\pi(E-(E(v_0))+\chi_\pi(v_0) \\ &= \frac{1}{4}(p-1)q(p-2)(q-1)+\sum_{i=0}^{q-2}(p-1)(q-1-i) \\ &= \frac{1}{4}(p-1)q(p-2)(q-1)+(p-1)\sum_{i=1}^{q-1}(q-i) \\ &= \frac{1}{4}(p-1)q(p-2)(q-1)+(p-1)\cdot\frac{1}{2}(q-1)(q-2) \\ &= \frac{1}{4}(p-1)q(p-2)(q-1)+(p-1)\cdot\frac{1}{2}q(q-1) \\ &= \frac{1}{4}pq(p-1)(q-1) \end{aligned}$$

Gleichung (3.4) ist auf (3.2) zurückführbar. Die Graphen $(\,(V_1 \cup V_3) \cup V_2\,,\, E_1 \cup E_2\,)$, $(V_1 \cup V_2, E_1)$ und $(V_2 \cup V_3, E_2)$ sind vollständig bipartite Graphen und ihre Kreuzungsanzahlen damit durch Gleichung (3.2) gegeben. Für den ersten Teil der Kreuzungen von g_4 gilt mit diesen Überlegungen:

$$\begin{aligned} \chi_{\pi_4}(E_1,E_2) &= \chi_{\pi_4}(E_1 \cup E_2)-\chi_{\pi_4}(E_1)-\chi_{\pi_4}(E_2) \\ &= g_2(p+r,q)-g_2(p,q)-g_2(r,q) \\ &= \frac{1}{2}pqr(q-1). \end{aligned}$$

Analog gilt für den zweiten Teil

$$\chi_{\pi_4}(E_1,E_2) = \frac{1}{2}prs(s-1).$$

□

3.3.4 Beweis für gewöhnliche Graphen

Durch einen gewöhnlichen Graph $G_0 = (V_0, E_0)$ mit $n := |V_0|$ und eine positive ganze Zahl k_0 wird eine Instanz des modifizierten OLAP definiert. Daraus wird der Graph $G = (V, E)$ und eine positive Konstante k der Instanz von CCRP folgendermaßen konstruiert:

- $V := V_0 \cup V_1 \cup V_2$, wo
  $V_1 := w_0, w_1, \ldots, w_{2n^4-1}$ und
  $V_2 := u_0, u_1, \ldots, u_{3n^{10}-1}$,
- $E := E_0 \cup E_{01} \cup E_{11} \cup E_{12} \cup E_{22}$, wo
  $E_{01} := \{\{v, w\} : v \in V_0, w \in V_1\}$,

$E_{11} := \{\{v, w\} : v, w \in V_1\}$,
$E_{12} := \{\{v, w\} : v \in V_1, w \in V_2\}$ und
$E_{22} := \{\{v, w\} : v, w \in V_2\}$,

- $k := g_1(3n^{10}+2n^4)+g_2(2n^4, n)+g_3(2n^4, n)+g_4(n, n^4, 3n^{10}, n^4)+n^4-1+2n^4 \cdot k_0$.

Die Teilgraphen $(V_1 \cup V_2, E_{11} \cup E_{12} \cup E_{22})$, (V_1, E_1) und (V_2, E_2) bilden die vollständige Graphen $K_{2n^4+3n^{10}}$, K_{2n^4} und $K_{3n^{10}}$, $(V_0 \cup V_1, E_{01})$ und $(V_1 \cup V_2, , E_{12})$ sind vollständig bipartite Graphen.

Das Kreislayout $\pi : V \to \{0, \ldots, 3n^{10} + 2n^4 + n - 1\}$, das G erhält, ist folgendermaßen definiert:

$$\begin{aligned}
\pi(v) &:= \pi_0(v) && \text{falls } v \in V_0,\\
\pi(w_i) &:= n + i && \text{falls } w_i \in V_1 \text{ und } 0 \le i < n^4,\\
\pi(u_i) &:= n + n^4 + i && \text{falls } u_i \in V_2,\\
\pi(w_i) &:= n + 3n^{10} + i && \text{falls } w_i \in V_1 \text{ und } n^4 \le i < 2n^4.
\end{aligned}$$

Die Knotenmengen V_0 und V_2 werden also fortlaufend angeordnet, zwischen ihnen liegen jeweils genau n^4 Knoten aus V_1. Layouts, die zu π äquivalent sind, heißen *kanonisch*.

Lemma 3.8 *Es seien die Graphen und Konstanten wie beschrieben gegeben. Dann ist die Knotenordnung $\pi_0 : V_0 \to \{0, \ldots, n-1\}$ eine Lösung des modifizierten OLAP für G_0 und k_0 genau dann, wenn ein Kreislayout π von G existiert mit $\chi_\pi(G) \le k$, wobei $\pi_{|V_0} = \pi_0$.*

Beweis
$\Longrightarrow$: Seien ein Graph $G_0 = (V_0, E_0)$ mit $n := |V_0|$, eine positive ganze Zahl k_0 und eine Knotenordnung $\pi_0 : V_0 \to \{0, \ldots, n-1\}$ gegeben, so dass die Bedingung

$$\sum_{\substack{\{u,v\}\in E,\\ \pi_0(u)<\pi_0(v)}} \pi_0(u) - \pi_0(v) - 1 \le k_0$$

erfüllt wird. Daraus seien der Graph $G = (V, E)$ und k wie beschrieben konstruiert und seine Knoten *kanonisch* angeordnet.

Bei diesem Layout kreuzen sich die Kanten aus E_0 nicht mit Kanten aus E_{11}, E_{12} und E_{22}, genauso wie Kanten aus E_{01} nicht von Kanten aus E_{22} geschnitten werden. Damit ist die Gesamtkreuzungsanzahl

$$\begin{aligned}
\chi_\pi(G) &= \chi_\pi(E_0) + \chi_\pi(E_{01}) + \chi_\pi(E_{11} \cup E_{12} \cup E_{22})\\
&\quad + \chi_\pi(E_{01}, E_{11}) + \chi_\pi(E_{01}, E_{12}) + \chi_\pi(E_0, E_{01}).
\end{aligned}$$

Für die Teilterme gilt:

$$\begin{aligned}
\chi_\pi(E_0) &< n^4 - 1, \text{ da } |E_0| < n^2 \\
\chi_\pi(E_{01}) &= g_2(2n^4, n) \\
\chi_\pi(E_{11} \cup E_{12} \cup E_{22}) &= g_1(3n^{10} + 2n^4) \\
\chi_\pi(E_{01}, E_{11}) &= g_3(2n^4, n) \\
\chi_\pi(E_{01}, E_{12}) &= g_4(n, n^4, n, 3n^{10}).
\end{aligned}$$

Es fehlt noch noch $\chi_\pi(E_0, E_{01})$. Jede Kante $\{u, v\} \in E_0$ mit $\pi(u) < \pi(v)$ schneidet $2n^4 \cdot (\pi(v) - \pi(u) - 1)$ Kanten aus E_{01}. Daraus folgt:

$$\chi_\pi(E_0, E_{01}) = 2n^4 \cdot \sum_{\substack{\{u,v\} \in E, \\ \pi_0(u) < \pi_0(v)}} (\pi_0(u) - \pi_0(v) - 1) \leq 2n^4 \cdot k_0.$$

Zusammen gilt:

$$\begin{aligned}
\chi_\pi(G) &\leq (n^4 - 1) + g_2(2n^4, n) + g_1(3n^{10} + 2n^4) + g_3(2n^4, n) \\
&\quad + g_4(n, n^4, n, 3n^{10}) + (2n^4 \cdot k_0) \\
&= k.
\end{aligned}$$

$\Longleftarrow$: Seien die Graphen $G_0 = (V_0, E_0)$ und $G = (V, E)$ sowie die Parameter k_0 und k wie oben gegeben. Weiter sei eine Knotenordnung π als Lösung des CCRP für G und k gegeben. Das folgende Lemma 3.10 zeigt, dass eine solche Anordnung ein *kanonisches* Layout sein muss. Sei also π eine *kanonische* Anordnung mit $\pi(V_0) = \{0, \ldots, n-1\}$. Damit ergibt sich direkt die Lösung $\pi_0 := \pi_{|V_0}$ des modifizierten OLAP für G_0 und k_0.

Sonst würde gelten:

$$\sum_{\substack{\{u,v\} \in E, \\ \pi_0(u) < \pi_0(v)}} (\pi_0(u) - \pi_0(v) - 1) \geq k_0 + 1$$

Jede Kante $\{u, v\} \in E_0$ mit $\pi(u) < \pi(v)$ schneidet $2n^4 \cdot (\pi(v) - \pi(u) - 1)$ Kanten aus E_{01}. Daraus folgt:

$$\begin{aligned}
\chi_\pi(E_0, E_{01}) &= 2n^4 \cdot \sum_{\substack{\{u,v\} \in E, \\ \pi_0(u) < \pi_0(v)}} (\pi_0(u) - \pi_0(v) - 1) \\
&\geq 2n^4(k_0 + 1) \\
&> 2n^4 \cdot k_0 + n^4 - 1
\end{aligned}$$

Damit erhält man eine untere Schranke für die Anzahl der Kreuzungen

$$\begin{aligned}\chi_\pi(G) &\geq \chi_\pi(E_{01}) + \chi_\pi(E_{11} \cup E_{12} \cup E_{22}) + \chi_\pi(E_{01}, E_{11}) \\ &\quad + \chi_\pi(E_{01}, E_{12}) + \chi_\pi(E_0, E_{01}) \\ &> g_2(2n^4, n) + g_1(3n^{10} + 2n^4) + g_3(2n^4, n) \\ &\quad + g_4(n, n^4, 3n^{10}, n^4) + 2n^4 \cdot k_0 + n^4 - 1 \\ &= k \\ &\quad \text{Widerspruch!}\end{aligned}$$

□

Es fehlt noch das wichtige Lemma 3.10, das zeigt, dass nur *kanonische* Layouts von G zu Kreuzungsanzahlen führen, die kleiner als k sind. Dazu zuerst eine Erweiterung des Lemma 3.6, in dem die Funktionen g_i beschrieben sind.

Lemma 3.9 *Seien G und k_0 wie oben definiert und sei π eine Knotenordnung von G. Dann gilt:*

- $\chi_\pi(E_{11} \cup E_{12} \cup E_{22}) = g_1(3n^{10} + 2n^4)$ *und* $\chi_\pi(E_{01}, E_{11}) = g_3(2n^4, n)$,
- $\chi_\pi(E_{01}, E_{12}) \geq g_4(n, n^4, n, 3n^{10}) + 3n^{10}$, *falls π nicht* kanonisch *ist, und*
- $\chi_\pi(E_{01}, E_{22}) > g_2(2n^4, n) + 2n^4 \cdot k_o + n^4 - 1$, *falls V_2 nicht fortlaufend angeordnet ist.*

Beweis
Die erste Behauptung ist eine Wiederholung des Lemma 3.6. Die beiden anderen Behauptungen ergeben sich schnell, wenn man die optimale (d.h. *kanonische*) Anordnung betrachtet und darin einen Knoten so verschiebt, dass die neue Ordnung nicht mehr *kanonisch* ist. Die vollständigen Beweise werden in [11] ausgeführt. □

Lemma 3.10 *Seien G und k wie oben definiert und sei π eine Knotenordnung von G. Gilt für die Kreuzungsanzahl $\chi_\pi(G) \leq k$, dann ist π* kanonisch.

Beweis
Annahme: V_2 wird durch π nicht fortlaufend angeordnet.
Nach Lemma 3.9 gilt dann:

$$\begin{aligned}\chi_\pi(G) &\geq \chi_\pi(E_{11} \cup E_{12} \cup E_{22}) + \chi_\pi(E_{01}, E_{11}) + \chi_\pi(E_{01}, E_{12}) + \chi_\pi(E_{01}, E_{22}) \\ &> g_1(3n^{10} + 2n^4) + g_3(2n^4, n) + g_4(n, n^4, 3n^{10}, n^4) \\ &\quad + g_2(2n^4, n) + 2n^4 \cdot k_o + n^4 - 1 \\ &= k \\ &\quad \text{Widerspruch!}\end{aligned}$$

Annahme: V_2 ist fortlaufend angeordnet, trotzdem ist π nicht *kanonisch*.
Mit Lemma 3.9 gilt: $\chi_\pi(E_{01}, E_{12}) \geq g_4(n, n^4, n, 3n^{10}) + 3n^{10}$. Außerdem ist $3n^{10} > g_2(2n^4, n) + 2n^4 \cdot k_o + n^4 - 1$. Damit gilt:

$$\begin{aligned}
\chi_\pi(G) &\geq \chi_\pi(E_{11} \cup E_{12} \cup E_{22}) + \chi_\pi(E_{01}, E_{11}) + \chi_\pi(E_{01}, E_{12}) \\
&> g_1(3n^{10} + 2n^4) + g_3(2n^4, n) + g_4(n, n^4, 3n^{10}, n^4) \\
&\quad + g_2(2n^4, n) + 2n^4 \cdot k_o + n^4 - 1 \\
&= k \\
&\quad \text{Widerspruch!}
\end{aligned}$$

□

Satz 3.11 *CCRP ist $\mathcal{NP}$-vollständig.*

Beweis
CCRP gehört zur Klasse $\mathcal{NP}$. Die beschriebene Transformation einer Instanz des modifizierten OLAP nach CCRP ist in polynominaler Zeit durchführbar. Das Lemma 3.8 zeigt schließlich, dass eine Lösung für eine Instanz des einen Problems direkt zu einer Lösung für die entsprechende Instanz des anderen Problems führt. □

Kapitel 4

Algorithmen zum Kreuzungszählen

Da die Kreuzungsminimierung in Kreislayouts $\mathcal{NP}$-vollständig ist, existieren wahrscheinlich keine effizienten Algorithmen zur Lösung des Problems. Deshalb ist es interessant, Heuristiken zu entwickeln, die Layouts mit relativ wenigen Kreuzungen erzeugen. Um die Ergebnisse solcher Heuristiken vergleichen zu können, müssen die in einem Layout vorhandenen Kreuzungen gezählt werden.

Nach einigen Definitionen wird ein von Janet Six und Ioannis Tollis [12] entworfener Algorithmus besprochen, der die Kreuzungen, d.h. die Liste der sich kreuzenden Kantenpaare, eines Kreislayouts in $\mathcal{O}(\chi(G))$ berechnet. Diesen werden wir so modifizieren, dass er in weniger Zeit nur die Anzahl der Kreuzungen berechnet.

Danach betrachten wir einen Algorithmus, der die von einem Knoten erzeugten Kreuzungen berechnet, d.h. die Kreuzungen, an denen Kanten aus $E(v)$ beteiligt sind. Dieser Algorithmus wird von manchen der Heuristiken der Kapitel 5 und 6 benützt. Er ist außerdem nützlich, um Knoten, die das Layout sehr stark verschlechtern, zu identifizieren.

Da sich die Kreuzungszählung immer auf eine feste Knotenordnung bezieht, werden in diesem Kapitel die Knoten mit ihrer Position identifiziert.

4.1 Grundlagen

Durch die Knotenordnung erhält jede Kante eine *Richtung* vom kleineren zum größeren Endknoten. Der Endknoten einer Kante e an der kleineren Position heißt *Startknoten* $s(e)$, der andere *Zielknoten* $t(e)$.

Zu einer Position i sind die *offenen* Kanten diejenigen mit Startknoten kleiner als i und Zielknoten größer als i. Ist ein Algorithmus an Position i angelangt, hat er von den offenen Kanten bereits die Startknoten besucht, aber noch nicht die Zielknoten,

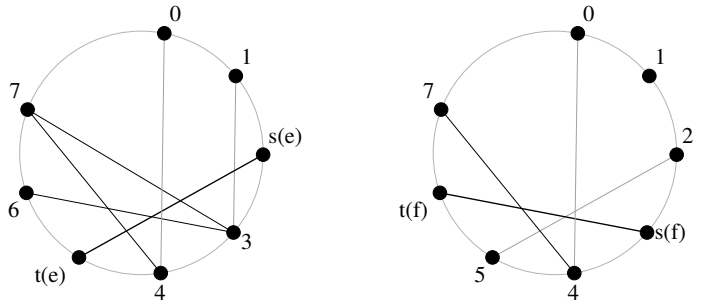


Abbildung 4.1: Offene (schwarz) und geschlossene Kanten (grau) von e und f.

ihr Ende ist also noch *offen*. Eine Kante e ist genau dann *offen* bezüglich Position i, wenn gilt: $s(e) < i < t(e)$.

Die *geschlossenen* Kanten bezüglich Position i sind diejenigen mit (Start- und) Zielknoten kleiner als i. Diese sind vom Algorithmus bereits endgültig bearbeitet. Kanten, die bis zu Position i noch nicht gefunden wurden, also mit Startknoten größer als i, sind für die Beschreibung nicht wichtig und bekommen keine Bezeichnung.

Damit lässt sich die Menge der Kanten, die von einer ausgezeichneten Kante e gekreuzt werden, in zwei Teile teilen, die Menge der bezüglich des Zielknotens $t(e)$ *offenen* Kanten $O(e)$ und die Menge der *geschlossenen* Kanten $C(e)$. $O(e)$ enthält die Kanten $\bar{e}$, die e kreuzen und nach $t(e)$ enden. Dies sind genau die Kanten mit $s(e) < s(\bar{e}) < t(e) < t(\bar{e})$. Die Menge $C(e)$ enthält die Kanten $\bar{e}$, die e kreuzen und bereits vor $t(e)$ enden. Für diese Kanten gilt $s(\bar{e}) < s(e) < t(\bar{e}) < t(e)$ (siehe auch Abb. 4.1).

4.2 Kreuzungen eines Graphen

4.2.1 Kanonischer Algorithmus

Der einfachste Algorithmus, um die Anzahl der Kantenkreuzungen zu zählen, testet für jedes mögliche Kantenpaar, ob sie sich schneiden. Dieser Test ist in konstanter Zeit möglich, wie wir in Kapitel 3.1 gesehen haben. Damit liegt die Laufzeit dieses Algorithmus in $\mathcal{O}(m^2)$.

4.2.2 Die Kreuzungen berechnen

Der kanonische Algorithmus ist besonders bei wenigen existierenden Kreuzungen sehr ineffizient, da dann viele Kantenpaare überprüft werden, die sich nicht kreuzen. Der folgende Algorithmus betrachtet die Kanten in der Reihenfolge der Knotenordnung und erreicht dadurch eine Laufzeit, die im wesentlichen durch die Kreuzungsanzahl bestimmt wird. Er berechnet eigentlich die Liste der sich kreu-

zenden Kanten, also wirklich die Kreuzungen, und nicht nur ihre Anzahl. Da ihn Janet Six und Ioannis Tollis in [12] aber nur zum Kreuzungszählen verwenden, wird er in ihrer Version beschrieben und die für die Kreuzungsliste nötige Änderung am Ende beschrieben.

Vorgehensweise

Die Idee ist, reihum, beginnend bei 0, alle Knotenpositionen zu besuchen. An jeder Position wird für jede dort endende Kante e die Anzahl der Kreuzungen von e mit ihren offenen Kanten $O(e)$ berechnet. Die Kreuzungen mit den geschlossenen Kanten $C(e)$ wurden bei diesem Vorgehen bereits früher gezählt, da e für alle diese Kanten eine offene Kante ist und ihre Zielknoten natürlich bereits vor der aktuellen Position besucht wurden. Jede Kreuzung wird also genau einmal gezählt, am kleineren Zielknoten der beteiligten Kanten.

Um die offenen Kanten einfach und effizient zu bestimmen, wird eine Kantenliste *openEdges* geführt. An jeder Position i werden zu dieser Liste die Kanten mit Startknoten i hinzugefügt und die Kanten mit Zielknoten i gelöscht. Dadurch sind die Kanten in *openEdges* zu jedem Zeitpunkt monoton wachsend nach den Positionen ihrer Startknoten sortiert. Zu einer Kante e, die in i endet und deren Kreuzungen berechnet werden sollen, sind die offenen Kanten genau die Kanten in *openEdges*, deren Startknoten größer als $s(e)$ ist und die nicht in $t(e)$ starten oder enden. Diese stehen durch die Sortierung nach den Startknoten am Ende der Liste und sind einfach zu zählen.

Um das Entfernen und Hinzufügen von Kanten zur Liste *openEdges* zu vereinfachen, wird die Adjazenzliste jedes Knotens in die Listen *startNode[i]* und *targetNode[i]* geteilt. *startNode[i]* enthält die Kanten, für die i Startknoten ist, *targetNode[i]* die Kanten, für die i Zielknoten ist.

Das beschriebene Vorgehen wird durch den Algorithmus *AllCrossings* implementiert.

Korrektheit

Der Algorithmus ist korrekt, er setzt die beschriebene Idee direkt um. Durch die Schleife 5 bis 11 werden alle Positionen besucht. Da zuerst alle Kanten, die an der aktuellen Position enden, aus *openEdges* gelöscht werden, und erst nach dem Kreuzungszählen die neuen Kanten hinzugefügt werden, ist die berechnete Anzahl in Zeile 9 korrekt.

Algorithm 4: AllCrossings

Eingabe: Ein Graph G mit Kreislayout π.
Ausgabe: Die Anzahl der Kreuzungen.

```
1.  list of edge openEdges ← ∅
2.  array of (list of edge) startNode ← Kanten nach den Startknoten
                                        gruppiert
3.  array of (list of edge) targetNode ← Kanten nach den Zielknoten
                                         gruppiert
4.  int numCross ← 0

5.  for (i ← 0, ..., n − 1) do
6.      forall (e in targetNode[i]) do
7.          Lösche e aus openEdges
8.      forall (e in targetNode[i]) do
9.          numCross ← numCross + Anzahl der Kanten in openEdges,
                       die nach s(e) starten
10.     forall (e in startNode[i]) do
11.         Füge e zu openEdges hinzu
12. return numCross
```

Laufzeit

Die Adjazenzlisten in den Zeilen 2 und 3 lassen sich mit einer Schleife über alle Kanten in $\mathcal{O}(m)$ erstellen. Das Einfügen und Entfernen von Listenelementen kann in konstanter Zeit ausgeführt werden. Jede Kante wird genau einmal in *openEdges* eingefügt und wieder entfernt, im Lauf des Algorithmus verursachen die Zeilen 7 und 11 also eine Laufzeit in $\mathcal{O}(m)$.

Bleibt noch die Laufzeit für die Berechnung der Kreuzungen. Die Kanten in *openEdges* wurden beim Besuch ihres Startknoten eingefügt und sind dadurch monoton wachsend nach den Positionen ihrer Startknoten sortiert. Durchwandert man *openEdges* zu jeder untersuchten Kante e von hinten, bis zum ersten Mal eine Kante einen Startknoten kleiner oder gleich $s(e)$ besitzt, hat man dabei bis auf die letzte Kante nur offene Kanten besucht. Über den ganzen Algorithmus betrachtet liegt die Laufzeit dafür in $\mathcal{O}(m + \chi(G))$, da außer den m Abbruchkanten nur Kanten besucht wurden, die sich kreuzen. Im (ungewöhnlichen) Fall $m < n$ müssen die n Schleifendurchläufe berücksichtigt werden, die Gesamtlaufzeit des Algorithmus beträgt also $\mathcal{O}(n + m + \chi(G))$.

Kreuzungsliste berechnen

Bei der Berechnung der Kreuzungsanzahl in Zeile 9 wird explizit jede von der aktuellen Kante e geschnittene Kante in *openEdges* besucht. Also kann hier, statt nur die Anzahl der Kreuzungen zu berechnen, auch die Liste der Kreuzungen von e und den entsprechenden Kanten aus *openEdges* erstellt werden. Die Variable *numCross* muss entsprechend in eine Liste von Kantenpaaren geändert werden. Die Laufzeit des Algorithmus verändert sich dadurch nicht.

4.2.3 Die Kreuzungen zählen

Der Algorithmus *AllCrossings* berechnet die Liste der Kreuzungen sehr effizient, in linearer Laufzeit im Verhältnis zur Ausgabe. Ist man aber nur an der Anzahl der Kreuzungen interessiert, liefert der Algorithmus zu viele Informationen und benötigt mehr Laufzeit als nötig.

In einem typischen Graph wird für das wiederholte Durchlaufen der Liste der offenen Kanten in den Zeilen 8 und 9 am meisten Zeit benötigt. Diese beiden Zeilen sind damit die ersten Kandidaten für eine Optimierung. Tatsächlich lässt sich jede der beiden Zeilen so modifizieren, dass sich die Laufzeit deutlich verringert.

Modifikation 1

Die erste Modifikation eliminiert die Schleife über alle Kanten aus *targetNode[i]* in Zeile 8. Für je zwei dieser Kanten werden teilweise die gleichen Elemente in *openEdges* besucht und gezählt, und zwar genau bis zum größeren Startknoten der beiden Kanten. Es liegt also nahe, bei der Berechnung der Kreuzungen nicht für jede Kante wieder neu am Ende von *openEdges* zu beginnen, sondern die bereits berechneten Ergebnisse zu benutzen und nur die weiteren Kanten hinzu zu zählen. Betrachtet man die Kanten aus *targetNode[i]* absteigend nach ihren Startknoten sortiert, sind so schrittweise alle nötigen Kreuzungsanzahlen mit einem Durchlauf durch *openEdges* zu berechnen.

Bei diesem Vorgehen werden für die gesamte Berechnung genau die Kanten aus *openEdges* betrachtet, die der Algorithmus *AllCrossings* nur für die Kante aus *targetNode[i]* mit kleinstem Startknoten betrachtet. Die Laufzeit für die Bearbeitung der anderen Kanten aus *targetNode[i]* wird also eingespart. Diese Überlegung zeigt erstens, dass die Modifikation die Laufzeit des Algorithmus auf keinen Fall verschlechtert, und zweitens, dass die Verbesserung um so größer ist, je mehr offene Kanten gleichzeitig vorhanden sind und je mehr Kanten am gleichen Knoten enden.

Die asymptotische Laufzeit verbessert sich noch nicht entscheidend. Für jede der n Knotenpositionen wird die Liste der offenen Kanten höchstens einmal durchlaufen. Im *worst-case* enthält sie fast alle Kanten. Die Listen *targetNode* lassen sich mit

Bucket Sort in $\mathcal{O}(m)$ sortieren. Die Laufzeit der anderen Teile des Algorithmus ändert sich nicht, die Gesamtlaufzeit liegt also in $\mathcal{O}(nm)$. Gleichzeitig gilt die frühere Abschätzung $\mathcal{O}(n + m + \chi(G))$ weiter. Nur wenn im Layout mehr als nm Kreuzungen existieren, verbessert sich die Laufzeit, sonst bleibt sie gleich.

Modifikation 2

Eine bessere Laufzeit wird aufgrund der Durchläufe durch die Liste *openEdges* verhindert, die zweite Modifikation verkürzt deshalb diese Liste. In *openEdges* sind die Kanten, wie bereits erwähnt, nach ihren Startknoten sortiert, insbesondere liegen Kanten mit gleichem Startknoten immer beieinander. Beim Abzählen in Zeile 9 des Algorithmus werden immer entweder alle oder keine der Kanten eines Startknotens gezählt. Es liegt also nahe, alle Kanten eines Startknotens zusammenzufassen und nur noch ihre Anzahl in *openEdges* zu speichern. Für jede Position existiert so höchstens ein Element in *openEdges*. Da die Liste so nicht länger als n werden kann, verbessert sich die Laufzeit zu $\mathcal{O}(n^2)$.

Damit *openEdges* keine unnötigen Einträge enthält, werden Listenelemente, deren Anzahl auf Null fällt, aus der Liste entfernt. Für Positionen, die keine startenden Kanten besitzen, werden keine Elemente in die Liste eingefügt. Dadurch enthält *openEdges* nie mehr Elemente als im ursprünglichen Algorithmus zur gleichen Zeit, da jedes Element für mindestens eine offene Kante steht. Die Laufzeit liegt also immer noch in $\mathcal{O}(n + m + \chi(G))$.

Vorgehensweise

Der Algorithmus *CountAllCrossings* realisiert die beiden Modifikationen und enthält noch einige kleinere Änderungen. Die Zeilennummern in der Beschreibung verweisen auf die entsprechenden Zeilen des ursprünglichen Algorithmus.

startNode bzw. *targetNode* enthält statt der inzidenten Kanten ihren Ziel- bzw. Startknoten. Der andere Endknoten ist bereits durch den Array-Index gegeben. Die Listen *openEdges* und *targetNode[i]* werden entgegen der Beschreibung absteigend sortiert, um sie (intuitiver) von vorne nach hinten durchlaufen zu können.

Die Schleife über alle Knotenpositionen reicht wieder von Zeile 5 bis 11. In den Zeilen 6 und 7a bis 7c werden für die Kanten in *targetNode[i]* die Einträge in *openEdges* verringert und nötigenfalls ganz entfernt. Dies ist mit geeigneten Datenstrukturen in konstanter Zeit möglich. Am Ende der Schleife in den Zeilen 10 und 11 wird die Anzahl der neuen offenen Kanten in *openEdges* eingefügt. Diese beiden Operationen werden vor bzw. nach dem Kreuzungszählen ausgeführt, damit die zur aktuellen Position inzidenten Kanten beim Zählen nicht extra berücksichtigt werden müssen.

In den Zeilen 8a bis 9d werden die Kreuzungen gezählt. Zuerst wird die Liste *tar-*

getNode[i] absteigend sortiert. *Listiter* in Zeile 8c ist ein *Listenelement* wie in LEDA [18] oder ein *Iterator* für die Liste *openEdges*. Mit ihm ist es möglich, direkt auf den Inhalt eines Listenelements zuzugreifen (Zeilen 9a, 9b) und schrittweise durch die Liste zu navigieren. In der *while*-Schleife werden die Elemente von *openEdges* in der Variablen *localNumCross* summiert. Die *forall*-Schleife unterbricht dies an jeder Startposition aus *targetNode[i]*, um das aktuelle Zwischenergebnis zur Gesamtkreuzungsanzahl zu addieren.

Algorithm 5: CountAllCrossings

Eingabe: Ein Graph G mit Kreislayout π.
Ausgabe: Die Anzahl der Kreuzungen.

```
1.  list of int openEdges ← ∅
2.  array of (list of edge) startNode ← Startknoten der Kanten
3.  array of (list of edge) targetNode ← Zielknoten der Kanten
4.  int numCross ← 0

5.  for (i ← 0, ..., n − 1) do
6.      forall (spos in targetNode[i]) do
7a.         Dekrementiere openEdges[spos]
7b.         if (openEdges[spos] = 0) then
7c.             Lösche das Listenelement der Position spos aus openEdges

8a.     Sortiere targetNode[i] absteigend.
8b.     int localNumCross ← 0
8c.     listiter it ← openEdges.first()
8d.     forall (spos in targetNode[i]) do
9a.         while ((it <= openEdges.last()) and (spos < openEdges[it])) do
9b.             localNumCross ← localNumCross + openEdges[it]
9c.             it ← openEdges.next(it)
9d.         numCross ← numCross + localNumCross

10.     if (startNode[i] ist nicht leer) then
11.         Füge startNode[i].length() vorne zu openEdges hinzu
12. return numCross
```

4.2.4 Laufzeitvergleich

Die asymptotische Laufzeit des Algorithmus *CountAllCrossings* ist höchstens so groß wie die Laufzeit des Algorithmus *AllCrossings*, da für beide die gleiche Abschätzung $\mathcal{O}(n + m + \chi(G))$ gilt. Für viele Layouts, besonders zu relativ dichten

Graphen, ist $\mathcal{O}(n^2)$ die deutlich schärfere Schranke, im Fall eines vollständigen Graphen beträgt der Unterschied zwei Größenordnungen, n^2 zu n^4.

Da uns besonders die reale Anwendung der Kreuzungsreduzierung interessiert, betrachten wir noch die tatsächliche Laufzeit der Algorithmen. Diese wird durch die Modifikation auf jeden Fall nicht erhöht, da die Komplexität der Implementierungen vergleichbar ist. Die Operationen auf der Liste *openEdges* benötigen fast gleichviel Zeit. Das Sortieren der Listen *targetNode[i]* wird durch das schnellere Einfügen der neuen Elemente in den Zeilen 10 und 11 kompensiert.

Ein signifikanter Zeitgewinn wird beim Zählen der Kreuzungen erreicht. Dies gilt nicht nur für Layouts mit vielen Kreuzungen, für die das aufgrund der asymptotischen Laufzeiten zu erwarten ist. Auch bei Layouts mit nur wenigen Kreuzungen liefert der modifizierte Algorithmus das Ergebnis deutlich schneller.

4.3 Kreuzungen eines Knotens

In diesem Abschnitt wird der Algorithmus *CountSingleNodeCrossings* vorgestellt, der die Anzahl der Kreuzungen, die von einem einzigen Knoten verursacht werden, berechnet. Er funktioniert ähnlich wie die vorherigen Algorithmen, hat aber eine geringere Laufzeit.

4.3.1 Vorgehensweise

Ausgehend von v werden alle Knotenpositionen besucht und eine Liste der offenen Kanten geführt. Immer wenn ein Zielknoten einer Kante $e \in E(v)$ erreicht wird, kreuzt e alle offenen Kanten, also wird die aktuelle Länge der Liste *openEdges* zur bisherigen Kreuzungsanzahl addiert. Wenn alle Nachbarn von v besucht wurden, ist der Algorithmus beendet. Die Liste offener Kanten wird für dieses Vorgehen nicht explizit benötigt, es genügt, jeweils ihre Länge zu kennen.

4.3.2 Laufzeit

Die Laufzeit des Algorithmus ist in $\mathcal{O}(n+m)$. Zur Berechnung der neuen offenen Kanten und der sich schließenden Kanten in den Zeilen 6 und 11 werden schlimmstenfalls alle m Kanten betrachtet, außerdem wird jede der n Positionen besucht.

Algorithm 6: CountSingleNodeCrossings

Eingabe: Ein Graph G, ein Knoten v und ein Layout π.
Ausgabe: Die Anzahl der Kreuzungen von v.
O.E. Sei v an Position 0.

```
1.  list of int targetNodes ← Nachbarknoten von v
2.  int openEdges ← 0
3.  int numCross ← 0

4.  Sortiere targetNodes aufsteigend
5.  for (i ← 1, ..., n − 1) do
6.      openEdges ← openEdges - Anzahl der in i endenden Kanten
7.      if (erstes Element von targetNodes ist i) then
8.          numCross ← numCross + openEdges
9.          Lösche das erste Element von targetNodes
10.         if (targetNodes ist leer) then return numCross
11.     openEdges ← openEdges + Anzahl der in i startenden Kanten
12. return numCross
```

Kapitel 5

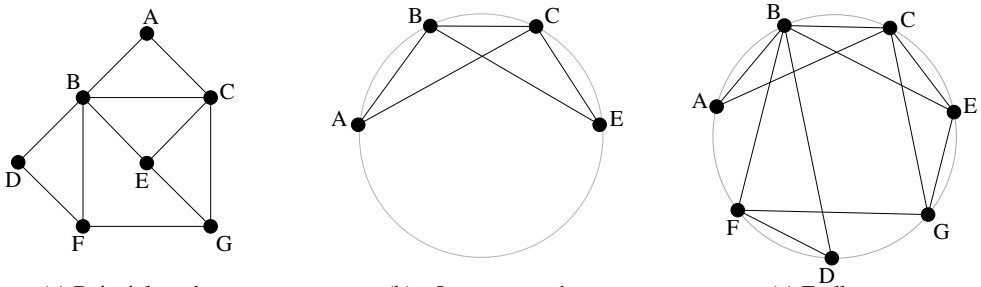
Bekannte Heuristiken zur Kreuzungsreduzierung

In diesem Kapitel betrachten wir verschiedene aus der Literatur bekannte Heuristiken zur Kreuzungsreduzierung. Im ersten Teil wird ein Algorithmus von E. Mäkinen [1] zur Reduzierung der Kantenlänge eines Layout besprochen, der auch die Kreuzungsanzahl verringert. Das zweite Verfahren wird im *Graph Layout Toolkit* von Tom Sawyer Software [16] zur Darstellung von Kreislayouts verwendet. Schließlich betrachten wir noch einen relativ neuen zweistufigen Algorithmus von J. Six und I. Tollis [12].

5.1 Algorithmus von E. Mäkinen

In Abschnitt 2.2.2 wurde die Summe der Kantenlängen als Optimierungskriterium für Layouts besprochen. Ein entsprechendes Layout enthält tendenziell kurze Kanten und, da eine kurze Kante nur wenige andere Kanten kreuzen kann, insgesamt wenig Kreuzungen. Eine Heuristik zur Reduzierung der Kantenlänge wird also benachbarte Knoten möglichst nahe beieinander platzieren und so auch die Anzahl der Kreuzungen verringern.

Erkki Mäkinen [1] hat eine solche Heuristik vorgestellt. Diese fügt nacheinander die Knoten in der Reihenfolge der Knotengrade zum Layout hinzu. Dabei sind in jedem Schritt die bereits bearbeiteten Knoten fortlaufend angeordnet und der neue Knoten wird an einem der beiden Enden des Teillayouts angefügt. Die bearbeiteten Knoten liegen also auf einem stetig wachsenden Kreisbogen.



(a) Beispielgraph (b) Layout nach Schritt 1 (c) Endlayout

Abbildung 5.1: Beispiel für den Algorithmus von Mäkinen.

5.1.1 Vorgehensweise

Zuerst werden die beiden Knoten mit größtem Grad oben ins Layout eingefügt, einer auf der linken Seite, der andere auf der rechten. Als nächstes werden die Knoten betrachtet, die zu beiden Startknoten adjazent sind. Diese werden abwechselnd links und rechts eingefügt, zuerst diejenigen mit größtem Grad.

Betrachten wir dazu als Beispiel den Graphen aus Abb. 5.1(a). Zuerst werden die Knoten B und C ausgewählt und an den obersten Positionen platziert. Die einzigen zu beiden adjazenten Knoten sind A und E. E wird zuerst platziert, da er den größeren Grad hat, beispielsweise auf der rechten Seite. Danach erhält A die Position auf der linken Seite. Das ergibt das Teillayout in Abb. 5.1(b).

Nun werden die restlichen Knoten in der Reihenfolge ihrer Knotengrade betrachtet, die größten zuerst. Jeder Knoten wird auf der Seite des Layouts platziert, auf der bereits mehr seiner Nachbarn liegen. Besitzt er auf beiden Seiten gleich viele Nachbarn, wird er beliebig zugeordnet. Ist noch keinem seiner Nachbarn eine Position zugeordnet, wird er ans Ende der Bearbeitungsliste verschoben. Da nur zweifach zusammenhängende Graphen bearbeitet werden, kann jeder zurückgestellte Knoten irgendwann platziert werden.

Um zu entscheiden, auf welcher Seite ein Knoten platziert werden soll, kann eine *linke* bzw. *rechte Adjazenzliste* benutzt werden, in der zu den Knoten, die mit der linken bzw. rechten Seite verbunden sind, die entsprechende Vielfachheit gespeichert wird.

In obigem Beispiel enthält die linke Adjazenzliste nach dem ersten Schritt die beiden Knoten D und F mit Vielfachheit 1, die rechte nur G mit Vielfachheit 1. Wird nun als nächstes der Knoten F betrachtet, wird er der linken Seite zugeordnet. G muss rechts eingefügt werden, und für D bleibt dann nur noch eine Position übrig. Am Ende erhält man das Layout in Abb. 5.1(c).

5.1.2 Laufzeit

Die Laufzeit des Algorithmus ist in $\mathcal{O}(m)$. Die Sortierung der Knotenliste nach dem Knotengrad ist mit *Bucket Sort* in $\mathcal{O}(n)$ möglich. Wird ein Knoten v einer Seite des Layouts hinzugefügt, muss die entsprechende Adjazenzliste aktualisiert werden. Dies ist in $\mathcal{O}(d(v))$ möglich. Wegen $\sum_{v \in V} d(v) = 2m$ liegt die Gesamtlaufzeit in $\mathcal{O}(m)$.

5.1.3 Qualität und Kritik

Die optimale Kantenlänge des Beispielgraphen beträgt 16. Das vom Algorithmus gefundene Layout liegt mit einer Länge von 18 nahe am Optimum. Mit einer anderen Knotenreihenfolge im zweiten Schritt, G-F-D, hätte sich sogar eine Länge von 17 ergeben. Mit einem verfeinerten Auswahlkriterium, das nicht nur den Knotengrad betrachtet, ist also vielleicht eine Verbesserung der Heuristik möglich.

Im Beispielgraph sieht man sofort eine Verbesserungsmöglichkeit. Dazu müssen die Knoten vom Grad zwei zwischen ihren Nachbarn platziert werden. Damit erhält man sogar die optimale Lösung. Dieses Vorgehen steht aber im Widerspruch zur Bearbeitungsreihenfolge, nach der diese Knoten immer ganz am Ende ins Layout eingefügt werden.

Betrachtet man die Kreuzungsanzahl, liefert der Algorithmus für den Beispielgraphen kein gutes Ergebnis. Das Problem sind auch hier die Knoten vom Grad zwei. Bei der Kreuzungsreduzierung sind sie besonders wichtig. Sind sie schlecht platziert, entdeckt ein Betrachter sehr schnell bessere Positionen und erhält so den – subjektiven – Eindruck eines schlechten Layouts.

Es ist nicht zu erwarten, dass durch die Bearbeitung der Knoten in der Reihenfolge der Knotengrade ein gutes Layout erzeugt wird. Benachbarte Knoten sollten nahe beieinander platziert werden. Dies wird durch diese Reihenfolge nicht erreicht, da zwischen Knotengrad und Adjazenzbeziehungen kein direkter Zusammenhang besteht. Diese Vermutung wird durch die Ergebnisse in Kapitel 7.2 gestützt. Durch die Wahlmöglichkeit zwischen linker und rechter Seite für die Position eines Knotens kann die schlechte Bearbeitungsreihenfolge nicht ausgeglichen werden. Nur an einer Stelle im Algorithmus wird für die Wahl der Knoten die Adjazenzbeziehung betrachtet: nach den beiden Startknoten werden als nächstes ihre gemeinsamen Nachbarn betrachtet.

5.1.4 Zusammenfassung

Der Algorithmus fügt die Knoten nacheinander an einem der beiden Enden des bereits erstellten fortlaufend angeordneten Teillayouts an. Da die Knoten im wesentlichen in der Reihenfolge der Knotengrade betrachtet werden, erzeugt der Algorithmus keine guten Layouts.

Mäkinen erwähnt als Kriterium für die Wahl zwischen linker und rechter Seite außer der Anzahl der Kanten noch die Zunahme der Kantenlänge. Die Experimente in 7.2 legen aber nahe, dass auch dies keine entscheidende Verbesserung bringt.

5.2 Kreislayouts mit dem Graph Layout Toolkit

Das *Graph Layout Toolkit (GLT)* ist eine Softwarebibliothek mit Layoutalgorithmen für Graphen von *Tom Sawyer Software* [16]. Sie enthält Algorithmen für vier verschiedene Arten von Layouts, darunter auch Kreislayouts. Wie diese erstellt werden, ist in [16] nur skizzenhaft beschrieben. Deshalb, und da die im nächsten Abschnitt besprochene Heuristik von J. Six und I. Tollis bessere Ergebnisse liefert, wird im Folgenden nur das grundlegende Vorgehen erklärt.

Mit dem GLT können erweiterte Kreislayouts erstellt werden. Diese Layouts geben die Aufteilung des Graphen in Cluster wieder. Ein Cluster ist eine Menge von Knoten. Normalerweise besitzen die Knoten eines Clusters eine gemeinsame Eigenschaft, sie liegen zum Beispiel in der gleichen (mehrfachen) Zusammenhangskomponente. In erweiterten Kreislayouts erhält jeder Cluster ein eigenes (einfaches) Kreislayout. Zusätzlich wird für den Cluster-Verbindungskanten-Graph ein Kreislayout berechnet, das die Position der einzelnen Clusterlayouts bestimmt.

Zur Berechnung des Layouts führt das GLT eine Kreuzungsreduzierung durch. Dabei werden die Kreuzungen innerhalb der Cluster verringert, ohne die Anzahl der Kreuzungen zwischen den Clustern stark zu erhöhen. Der Algorithmus unterscheidet deshalb zwischen *Out*-Knoten, die Kanten zu anderen Clustern besitzen, und *In*-Knoten, die nur mit Knoten des gleichen Clusters verbunden sind. Die Layouts jedes Clusters werden einzeln berechnet und die *In*- und *Out*-Knoten dabei unterschiedlich behandelt.

Um ein einfaches Kreislayout zu erhalten, nehmen wir an, dass alle Knoten zum gleichen Cluster gehören. Die Unterscheidung zwischen *In*- und *Out*-Knoten wird damit überflüssig und die Beschreibung des Algorithmus einfacher.

Zuerst wird ein initiales Layout ermittelt. Dieses basiert auf einem Tiefensuchbaum maximaler Höhe, der einen längsten Weg im Graphen, und damit eine Knotenordnung, induziert.

Im zweiten Schritt wird das Startlayout verbessert, indem adjazente Knoten vertauscht werden, wenn sich dadurch die Kreuzungsanzahl verringert. Die Veränderung der Kreuzungsanzahl kann aus der Anzahl der von den beiden Knoten im alten und neuen Layout verursachten Kreuzungen berechnet werden, also durch viermaliges Anwenden des Algorithmus *CountSingleNodeCrossings*. Das Vertauschen von Knoten wird solange wiederholt, wie noch eine Verbesserung erreicht wird.

Die Laufzeit für das Vertauschen zweier adjazenter Knoten liegt in $\Theta(m)$, der

Laufzeit von *CountSingleNodeCrossings*. Wenn der Algorithmus endet, wurden alle adjazenten Knotenpaare überprüft, die Laufzeit liegt also in $o(m^2)$.

Da sich das Layout durch jede Vertauschung ändert, ist es möglich, dass das gleiche Knotenpaar mehrmals vertauscht wird. Außerdem ist nicht klar, um wie viel sich das Layout jeweils verbessert. Deshalb ist es schwierig, die Laufzeit genauer anzugeben.

5.3 Circular von J. Six und I. Tollis

Das folgende Verfahren wurde 1999 von Janet Six und Ioannis Tollis vorgestellt [12]. Es besteht aus zwei grundlegend verschiedenen Heuristiken, die nacheinander ausgeführt werden.

Die erste Phase ist inspiriert vom Algorithmus zum Test auf Außenplanarität *MAP-Test*. Sie liefert in kurzer Zeit bereits gute Ergebnisse, insbesondere ein kreuzungsfreies Layout, wenn der Graph ein solches Layout besitzt. Aufgrund dieser Eigenschaften sind die Layouts dieses Algorithmus gut als Ausgangspunkt für weitere Verbesserungsverfahren geeignet.

In der zweiten Phase wird das Layout aus *Phase 1* schrittweise verbessert. Jeder Knoten wird neben jeden seiner Nachbarn geschoben und die daraus resultierende Kreuzungsanzahl berechnet. Wenn an einer dieser Positionen ein besseres Layout erzielt wird, erhält der Knoten diese Position. Nach einigen wenigen Durchläufen (ungefähr zehn) verringert sich die Kreuzungsanzahl nicht weiter. Damit ist das Endlayout erreicht.

5.3.1 Phase 1

Definitionen

Zuerst einige Bezeichnungen. Eine Kante $\{v, w\}$ heißt *Dreieckskante*, wenn die Knoten v und w mindestens einen gemeinsamen Nachbarn u haben. Damit existieren die Kanten $\{v, u\}$ und $\{w, u\}$ und bilden zusammen mit $\{v, w\}$ ein Dreieck. Wir sagen auch, die Dreieckskante $\{v, w\}$ *gehört* zum Knoten u. Eine *Triangulierungskante* ist eine Dreieckskante, die neu in den Graphen eingefügt wird.

Idee

Der Algorithmus überträgt den Algorithmus *MAP-Test* auf beliebige Graphen. Ist ein Graph außenplanar (und zweifach zusammenhängend), dann läuft der Algorithmus analog zu *MAP-Test* und liefert ein kreuzungsfreies Layout. Für nichtaußenplanare Graphen gibt es einige Erweiterungen.

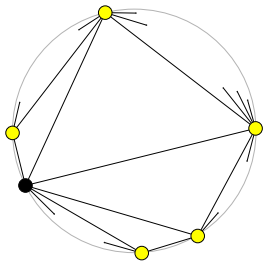


Abbildung 5.2: Ein Knoten mit vier Nachbarn und drei inzidenten Facetten.

In *MAP-Test* wird immer ein Knoten mit Grad zwei betrachtet. Dieser wird aus dem Graphen gelöscht, so dass die dreieckige Facette, die er mit seinen beiden Nachbarn bildete, verschwindet. Diese wird dann durch die (Dreiecks-) Kante zwischen den beiden Nachbarn repräsentiert. Löscht man alle im Laufe des Algorithmus gefundenen Dreieckskanten aus dem ursprünglichen Graphen, erhält man einen einfachen Kreis, der ein kreuzungsfreies Layout induziert.

In beliebigen Graphen gibt es meistens keinen Knoten mit Grad zwei. Der Algorithmus wählt stattdessen einen Knoten mit kleinstem Grad. Seine Dreieckskanten werden in einer Liste gespeichert, um sie, analog zu *MAP-Test*, am Ende zu löschen. Wenn der Graph nicht außenplanar war, erhält man dadurch keinen einfachen Weg, der direkt ein Layout induziert. Stattdessen wählt man einen beliebigen längsten Weg, der eine Knotenordnung für die Wegknoten ergibt und platziert Knoten, die nicht auf dem Weg liegen, nachträglich darin.

Die Anzahl und Lage der Dreieckskanten eines Knotens ist wichtig für den Ablauf des Algorithmus, da gelöschte Knoten durch ihre Dreieckskanten repräsentiert werden. In einem maximalen außenplanaren Graphen ist jeder Knoten mit Grad d inzident zu $(d-1)$ inneren Facetten und besitzt damit genau $(d-1)$ Dreieckskanten (siehe Abb. 5.2). Da der Algorithmus diesen Fall möglichst gut auf beliebige Graphen übertragen will, fügt er zusätzliche Dreieckskanten, die Triangulierungskanten, in den Graphen ein, wenn ein Knoten weniger als $(d-1)$ Dreieckskanten besitzt.

Vorgehensweise

Als erster Schritt erstellt der Algorithmus eine Liste der zu entfernenden Dreieckskanten. Dazu werden nacheinander alle Knoten betrachtet und aus dem Graphen entfernt. Der aktuelle Knoten wird nach den folgenden Kriterien ausgewählt:

- Es wird immer ein Knoten mit kleinstem Grad betrachtet.
- Gibt es mehrere Knoten mit kleinstem Grad, wird zuerst ein zum zuletzt betrachteten Knoten adjazenter Knoten gewählt, sonst ein Knoten, der zu einem schon früher betrachteten (und nun gelöschten) Knoten adjazent war.

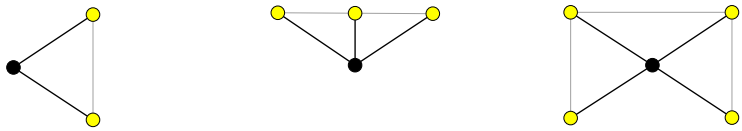


Abbildung 5.3: Verschiedene Fälle für das Einfügen von Triangulierungskanten (grau).

> Existieren keine solchen Knoten, wird ein beliebiger Knoten mit kleinstem Grad gewählt.

Für jeden aktuellen Knoten v werden die zugehörigen Dreieckskanten zur Liste der zu entfernenden Kanten hinzugefügt. Gibt es zu wenige Dreieckskanten, d.h. weniger als $d(v) - 1$, werden solange Triangulierungskanten erzeugt, bis die benötigte Anzahl erreicht ist. Diese werden zuerst zwischen Knoten eingefügt, die zu keiner Dreieckskante inzident sind, danach zwischen Knoten von kleinem Grad (bzgl. der Dreieckskanten). Die Kanten sind damit möglichst gleichmäßig verteilt (siehe Abb. 5.3). Schließlich wird v aus dem Graphen entfernt und der nächste Knoten betrachtet.

Im zweiten Schritt wird wieder der ursprüngliche Graph betrachtet. Aus ihm werden nun alle im ersten Schritt gefundenen Dreieckskanten gelöscht. Die Triangulierungskanten spielen hier keine Rolle mehr. Sie beeinflussen den Algorithmus nur im ersten Schritt, indem sie die Bearbeitungsreihenfolge der Knoten mitbestimmen und neue Adjazenzen herstellen.

Danach wird mittels Tiefensuche ein längster Weg im Restgraphen bestimmt. Dieser gibt das Teillayout der Wegknoten vor. Knoten, die nicht auf dem Weg liegen, werden bei ihren Nachbarn platziert. Existiert eine Position, so dass ein Knoten direkt zwischen zwei seiner Nachbarn liegt, erhält er diese, sonst eine beliebige neben einem seiner Nachbarn.

Qualität der Ergebnisse

Das Einfügen von Triangulierungskanten ist durch die Analogie zu außenplanaren Graphen motiviert. Trotzdem erscheint es sehr kompliziert. Es stellt sich die Frage, ob andere Vorgehensweisen zu besseren Ergebnissen führen oder durch ein einfacheres Verfahren das gleiche Ergebnis erreicht werden kann. Deshalb habe ich die folgenden drei Veränderungen des Algorithmus getestet:

- es werden keine Triangulierungskanten in den Graph eingefügt,

Algorithm 7: Six und Tollis, Phase 1

Eingabe: Ein Graph G.
Ausgabe: Ein Kreislayout mit wenigen Kreuzungen.

1. **list of edge** removalList $\leftarrow \emptyset$

2. **while** (G.numberOfNodes() ≤ 3) **do**
3. Sei v ein Knoten von kleinstem Grad, wie beschrieben gewählt
4. Füge alle Dreieckskanten von v zu *removalList* hinzu
5. Solange es weniger als $(d(v) - 1)$ Dreieckskanten gibt
6. Füge wie beschrieben eine Triangulierungskante zu G hinzu
7. Lösche v aus G

8. Stelle den Originalzustand von G wieder her
9. Lösche die Kanten in *removalList* aus G

10. Finde mit Tiefensuche einen längsten Weg in G und
platziere die Knoten entsprechend
11. Solange es noch einen unplatzierten Knoten v gibt
12. Platziere v entweder direkt zwischen zwei seiner Nachbarn
oder direkt neben einem seiner Nachbarn

- in Anlehnung an einen maximalen planaren Graphen werden Triangulierungskanten so eingefügt, dass es einen Weg über alle adjazenten Knoten gibt (siehe Abb. 5.2),
- alle Knoten, die zu keiner Dreieckskante inzident sind, erhalten eine Triangulierungskante.

Alle vier Verfahren wurden auf die Familien von Zufallsgraphen aus Kapitel 7.1 angewendet. Diese enthalten jeweils mehrere hundert Graphen verschiedener Größen zwischen 10 und 500 Knoten. Dabei zeigte sich, dass das Originalverfahren am Häufigsten die beste Lösung liefert, unabhängig von der betrachteten Graphenfamilie. Die dritte Modifikation ist praktisch gleich gut. Dies liegt daran, dass sie nur in wenigen Fällen Kanten anders einfügt als das Originalverfahren. Die beiden anderen Verfahren sind deutlich schlechter. Dabei ist der Verzicht auf Triangulierungskanten außer bei sehr dünnen Graphen die mit Abstand schlechteste Möglichkeit.

Das Einfügen der Triangulierungskanten ist also nicht nur durch die Analogie zu *MAP-Test* gerechtfertigt, sondern auch durch die Qualität der produzierten Layouts.

Laufzeit und Implementation

Die Schritte zwei und drei sind einfach zu implementieren. Das Löschen der Dreieckskanten aus dem ursprünglichen Graphen ist in Linearzeit $\mathcal{O}(n + m)$ möglich. Ebenso schnell lässt sich mit Tiefensuche ein längster Weg in G finden.

Für einen Nicht-Wegknoten müssen zuerst Positionen zwischen zwei seiner Nachbarn gefunden werden. Dazu genügt es, in seiner geordneten Adjazenzliste aufeinander folgende Knoten zu überprüfen. Gibt es keine solche Position, kann eine beliebige Stelle neben einem schon platzierten Nachbarn gewählt werden. Betrachtet man die Knoten entlang von Wegen im Tiefensuchbaum, ausgehend von Knoten auf dem gefundenen längsten Weg, gibt es für jeden Knoten mindestens einen schon platzierten Nachbarn. Insgesamt liegt die Laufzeit hierfür in $\mathcal{O}(n + m)$.

Eine asymptotisch gute Laufzeitanalyse des ersten Schrittes ist schwierig, da sich der Graph (genauer: einzelne Knotengrade) durch die Triangulierungskanten vergrößern kann. Insbesondere ist die oft hilfreiche Beziehung zwischen Kantenanzahl und Knotengraden $2m = \sum_{v \in V} d(v)$ nicht anwendbar. Sie gilt natürlich für jeden Zwischengraphen des Algorithmus, aber nicht, wenn über Knotengrade aus verschiedenen Graphen summiert wird. Natürlich gilt die offensichtliche Schranke $\sum_{v \in V} d(v) \leq n^2$. Jeder Zwischengraph enthält außerdem weniger als m Kanten, da mit dem aktuellen Knoten immer mehr Kanten gelöscht werden als neu hinzukommen.

Im ersten Schritt sind zwei Hauptaufgaben zu erledigen, das Auswählen eines Kno-

tens mit kleinstem Grad und das Finden und Einfügen von Dreieckskanten.

Um den Knoten mit kleinstem Grad entsprechend dem Auswahlkriterium zu bestimmen, verwenden wir eine *Array* von Knotenlisten. Die Liste im Feld i speichert dabei alle Knoten mit Grad i. Der aktuelle Knoten v wird vorne aus der nichtleeren Liste mit kleinstem Index entnommen. Da v aus G entfernt wird und Triangulierungskanten eingefügt werden können, ändern sich die Knotengrade der Nachbarn von v. Diese werden aus der Datenstruktur entfernt und vorne in ihre neuen Listen angefügt. Dadurch ist gewährleistet, dass die Sortierung jeder Liste der gewünschten Bedingung entspricht: zuerst enthält sie die Nachbarn des aktuellen Knoten, danach Knoten, die früher Nachbarn aktueller Knoten waren und schließlich Knoten, die noch nie Nachbarn aktueller Knoten waren. Das Erstellen der Datenstruktur ist in $\mathcal{O}(n)$ möglich, die Änderungen jeweils in $\mathcal{O}(d(v))$, insgesamt also in $\mathcal{O}(n^2)$.

Um die Dreieckskanten des Knotens v zu bestimmen, müssen die Adjazenzlisten der Nachbarknoten von v betrachtet werden. Dies ist in $\mathcal{O}(\sum_{w \in N(v)} d(w))$ möglich. Um eine Triangulierungskante einzufügen, müssen die Nachbarknoten von v aufsteigend nach ihrem Grad sortiert werden. Ein Endknoten der Kante wird der erste Knoten der Liste, der andere der kleinste Knoten in der Liste, zu dem nicht bereits eine Kante besteht. Eine solche Kante lässt sich immer erstellen, da der erste Knoten den kleinsten Grad hat und nicht zu allen anderen adjazent ist. Auf diese Weise werden neue Kanten wie gewünscht in den Graphen eingefügt. Insgesamt benötigt dieser Teil eine Laufzeit in $\mathcal{O}(nm)$, da jeder der Reduktionsgraphen höchstens m Kanten besitzt.

Insgesamt beträgt die Laufzeit $\mathcal{O}(nm)$. In echten Anwendungen ist die Laufzeit deutlich besser, als diese Abschätzung vermuten lässt. Das Layout von Graphen mit bis zu 200 Knoten und beliebig vielen Kanten kann auf modernen Rechnern (ab ca. 1 GHz) in deutlich weniger als 0.1 Sekunden berechnet werden und damit schnell genug für interaktive Anwendungen (siehe auch Kapitel 7).

Zusammenfassung

Der Algorithmus berechnet schnell ein Kreislayout mit wenig Kreuzungen. Darüber hinaus besitzt er die schöne Eigenschaft, dass er für (zweifach zusammenhängende) Graphen, die ein kreuzungsfreies Kreislayout besitzen, d.h. für außenplanare Graphen, ein solches Layout liefert.

Konzeptionell ist der Algorithmus eine Weiterentwicklung des Tests auf Außenplanarität. Der Nutzen der Triangulierungskanten ist dabei nicht gleich zu erkennen. Es zeigt sich aber, dass dieses Verfahren im Vergleich mit anderen ebenfalls plausiblen Möglichkeiten die besten Ergebnisse liefert.

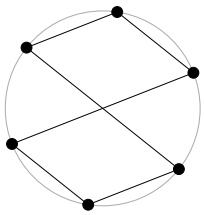


Abbildung 5.4: Ein außenplanarer Graph, der durch *Phase 2* kein kreuzungsfreies Layout erhält.

5.3.2 Phase 2

Um das Layout weiter zu verbessern, wird in der zweiten Phase des Algorithmus jeder Knoten probeweise an Positionen neben seinen Nachbarn platziert (ähnlich wie die Nicht-Wegknoten in *Phase 1*). Für jede dieser Positionen wird die Kreuzungsanzahl berechnet und der Knoten an die Stelle gesetzt, an der er am wenigsten Kreuzungen verursacht. Im Lauf des Algorithmus können für bereits betrachtete Knoten bessere Positionen entstehen. Deshalb wird das Verfahren mehrmals wiederholt.

Der Algorithmus hat die Eigenschaft, ein bestehendes Layout auf keinen Fall zu verschlechtern. Dafür ist er deutlich langsamer als *Phase 1* und von der Anfangsreihenfolge der Knoten abhängig. Ein weiterer wichtiger Nachteil ist, dass er für einen außenplanaren Graphen nicht immer ein kreuzungsfreies Layout liefert (siehe Abb. 5.4). Obwohl er auch einzeln mit einer zufälligen initialen Knotenordnung verwendet werden kann, ist er also eher zur Optimierung eines bereits ermittelten Startlayouts geeignet.

Da das Verschieben eines Knotens der wichtigste Schritt des Algorithmus ist, betrachten wir diesen Teil des Problems zuerst einzeln.

Einen Knoten verschieben

Sei π ein gegebenes Kreislayout, v der Knoten, der an eine neuen Position verschoben wird und $\bar{\pi}$ das dadurch entstehende neue Kreislayout.

Durch das Verschieben von v ändern sich nur Kreuzungen, die von Kanten aus $E(v)$ verursacht werden. Einige der Kreuzungen, die von diesen Kanten an der alten Position verursacht wurden, werden im neuen Layout verschwinden und dafür Kreuzungen mit neuen Kanten hinzukommen. Die neue Kreuzungsanzahl ergibt sich damit aus der alten Anzahl minus der Kreuzungen, die v an der alten Position verursacht hat, plus der Kreuzungen an der neuen Position:

$$\chi_{\bar{\pi}}(G) = \chi_{\pi}(G) - \chi_{\pi}(E(v)) + \chi_{\bar{\pi}}(E(v)) \ .$$

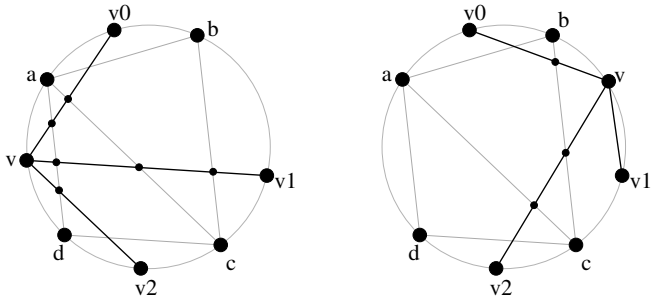


Abbildung 5.5: Veränderung der Kreuzungen durch das Verschieben von v.

Das Problem kann also durch zweimaliges Anwenden von *CountSingleNodeCrossings* gelöst werden. Um zu entscheiden, ob die neue Position eine Verbesserung bringt, ist es also nicht notwendig, die komplette Kreuzungsanzahl zu berechnen. Es genügt, die Differenz der von v an den beiden Positionen verursachten Kreuzungen zu betrachten.

Die Kreuzungen, die verschwinden oder neu entstehen, lassen sich noch genauer charakterisieren. Durch die alte und die neue Position von v wird das Layout in zwei Kreisbögen aufgeteilt. Wenn beide Endknoten einer Kante im gleichen Kreisbogen liegen und sie eine Kreuzung mit einer Kante aus $E(v)$ besitzt, existiert die Kreuzung zwischen diesen Kanten auch im neuen Layout. Liegen die Endknoten einer Kante e in verschiedenen Bögen, verändern sich ihre Kreuzungen. Im neuen Layout kreuzt sie genau die Kanten aus $E(v)$, die sie im alten Layout nicht gekreuzt hat. Anschaulich sind die Kanten mit Endknoten in verschiedenen Bögen genau diejenigen, die von einer Geraden durch die alte und neue Position geschnitten werden.

Abbildung 5.5 zeigt diese Beobachtung an einem Beispiel. Die grauen Kanten sind nicht inzident zu v, deshalb würden sich Kreuzungen, die sie miteinander haben, nicht ändern. Die Kreuzungen zwischen den Kanten $\{v, v_0\}$ und $\{a, b\}$ sowie $\{v, v_2\}$ und $\{c, d\}$ sind auch im neuen Layout vorhanden. Die Kreuzungen mit den Kanten $\{a, c\}$, $\{a, d\}$ und $\{b, c\}$ ändern sich. Sie sind durch Kreise markiert.

Vorgehensweise

Der Algorithmus *Phase 2* wählt nacheinander jeden Knoten aus und bestimmt die Liste seiner möglichen neuen Positionen. Besitzt er Nachbarn, die im Layout direkt nebeneinander liegen, werden die Positionen zwischen diesen gewählt, sonst die Positionen auf beiden Seiten aller seiner Nachbarn.

Danach wird er an jede dieser Positionen verschoben und die neue Kreuzungsanzahl bestimmt. Um zu entscheiden, ob eine Verschiebung eine Verbesserung bringt, kann der Algorithmus *CountSingleNodeCrossings* wie beschrieben verwendet werden. Am Ende wird der Knoten auf die Beste der neuen Positionen geschoben, oder, falls keine Verbesserung erreicht wurde, wieder an seine ursprüngliche Stelle.

Das Verfahren wird so lange wiederholt, bis sich keine Verbesserung mehr ergibt. Da sich in jeder Runde die Anzahl der Kreuzungen um mindestens eins verringert, terminiert der Algorithmus.

Algorithm 8: Six und Tollis, Phase 2

Eingabe: Ein Graph G mit Kreislayout π.
Ausgabe: Ein Kreislayout mit weniger oder gleich vielen Kreuzungen.

```
1.  while (Die Kreuzungsanzahl hat sich verringert) do
2.      forall (v in V) do
3.          deltaCross ← 0
4.          minCross ← 0
5.          positionList ← Positionen direkt zwischen zwei Nachbarn von v
                oder Positionen direkt neben allen Nachbarn von v

6.          forall (pos in positionList) do
7.              Setze v an Position pos und berechne die Kreuzungsänderung
8.              deltaCross ← deltaCross + Kreuzungsänderung
9.              if (deltaCross < minCross) then minCross ← deltaCross

10.         if (minCross < 0) then
11.             Setze v an die Position, an der minCross erzielt wurde
12.         else
13.             Setze v zurück auf seine Ausgangsposition
```

Laufzeit und Implementation

Die für die Verschiebung in Frage kommenden Positionen eines Knotens v lassen sich in $\mathcal{O}(d(v))$ bestimmen. Um direkt nebeneinander liegende Nachbarn von v zu finden, genügt es, die Adjazenzliste nach den Positionen zu sortieren und aufeinander folgende Knotenpaare zu überprüfen. Für eine Runde lassen sich die zu testenden Positionen damit in $\mathcal{O}(m)$ bestimmen.

Um die Kreuzungsänderung zu bestimmen, wird *CountSingleNodeCrossings* zweimal ausgeführt. Man erhält eine Laufzeit in $\mathcal{O}(m \cdot d(v))$ um alle möglichen Positionen des Knotens v zu testen, also $\mathcal{O}(m^2)$ für eine komplette Runde.

Die Laufzeit kann zu $\mathcal{O}(nm)$ verbessert werden. Die möglichen Positionen von v werden dazu in ihrer Reihenfolge auf dem Kreis betrachtet. Bei jeder Ausführung von *CountSingleNodeCrossings* müssen nur Kanten mit genau einem Endknoten zwischen der alten und der neuen Position von v betrachtet werden, da sich Kreu-

zungen mit anderen Kanten nicht ändern. Diese Kanten können im Voraus für alle Verschiebungen berechnet werden. Da die möglichen Positionen sortiert sind, ist jede Kante für höchstens zwei Verschiebungen relevant. Damit ist die Aufteilung der Kanten in $\mathcal{O}(m)$ möglich.

Bei jeder Verschiebung wird jede relevante Kante zweimal zu *openEdges* hinzugefügt und entfernt (einmal in jeder Ausführung von *CountSingleNodeCrossings*). Dies ist insgesamt ebenfalls in $\mathcal{O}(m)$. Es gibt höchstens $2 \cdot d(v)$ mögliche Positionen, für jede durchläuft der Algorithmus eine Schleife mit n Schritten. Also ergibt sich für einen Knoten v die Laufzeit $\mathcal{O}(n \cdot d(v) + m)$, für eine komplette Runde $\mathcal{O}(n \cdot m)$.

In jeder Runde verringert sich die Kreuzungsanzahl des Layouts, möglicherweise aber nur um eins. Im schlimmsten Fall benötigt er also Kreuzungsanzahl des Startlayouts viele Runden, um das Endlayout zu finden. Ist das Startlayout bereits relativ gut, zum Beispiel von *Phase 1* berechnet, erreicht man in den allermeisten Fällen spätestens nach zehn Runden das endgültige Ergebnis. Bereits nach zwei bis drei Runden verbessert sich das Layout in diesem Fall nur noch sehr wenig. Eine ausführlichere Analyse erfolgt in Kapitel 7.4 für das ähnliche *Sifting*-Verfahren.

Ist ein Benutzer mehr an Geschwindigkeit als an kleinen Qualitätsverbesserungen interessiert, kann die Berechnung bereits nach drei Runden gestoppt werden.

Zusammenfassung

Der Algorithmus versucht ein gegebenes Layout zu verbessern, indem er die Knoten nacheinander an neue Positionen verschiebt. Um die Laufzeit klein zu halten, werden dabei nur Positionen neben ihren Nachbarn betrachtet.

Er ist gut zur Verbesserung eines bereits optimierten Layouts geeignet, da er ein Ausgangslayout keinesfalls verschlechtert. Außerdem benötigt er in diesem Fall nur zwei bis drei Runden, um dem Endergebnis sehr nahe zu kommen und höchstens zehn Runden für die gesamte Berechnung.

Die Laufzeit einer Runde liegt in $\mathcal{O}(nm)$. Der Algorithmus ist in der Anwendung deutlich langsamer als Phase 1, aber schnell genug für interaktive Anwendungen. Das Layout von nicht sehr dichten Graphen mit bis zu 200 Knoten kann auf modernen Rechnern (ab ca. 1 GHz) in weniger als 0.25 Sekunden berechnet werden. Dabei wird eine Verringerung der Kreuzungsanzahl im Vergleich zu Phase eins von ungefähr 20% erzielt (siehe auch Kapitel 7).

5.4 Zusammenfassung

In diesem Kapitel wurden verschiedene Algorithmen zur Kreuzungsreduzierung vorgestellt. Diese lassen sich in zwei Gruppen aufteilen: die einen berechnen ihr

Endlayout unabhängig von einem gegebenen Anfangslayout, dazu gehören der Algorithmus von Mäkinen und *Phase 1*, die anderen verbessern ein gegebenes Startlayout, beispielsweise *Phase 2*. Auch der Algorithmus des GLT besteht eigentlich aus zwei Teilen, die sich diesen Gruppen zuordnen lassen.

Die Algorithmen der ersten Gruppe können besitzen außerdem eine geringe – fast lineare – Laufzeit. Dies gilt auch für *Phase 1*, zumindest für die reale Laufzeit, auch wenn die asymptotische Analyse zu einem anderen Ergebnis führt. Aufgrund dieser beiden Eigenschaften sind sie gut geeignet, um in einem zweistufigen Kreuzungsreduzierungsverfahren ein Startlayouts für den nächsten Schritt zu berechnen. Sie heißen deshalb *Initialisierungsalgorithmen*.

Da die Algorithmen der zweiten Gruppe jeden Knoten an verschiedene andere Positionen platzieren und die dadurch entstehenden Kreuzungen berechnen, ist ihre Laufzeit sowohl asymptotisch als auch real deutlich größer als der Initialisierungsalgorithmen. Obwohl sie auch alleine, mit einer zufälligen Startreihenfolge, zur Kreuzungsreduzierung verwendet werden können, erscheint es deshalb sinnvoller, sie als zweiten Schritt nach einem Initialisierungsalgorithmus zu verwenden.

In Kapitel 7 wird dieser Aspekt näher untersucht. Dabei werden für verschiedene Kombinationen der Algorithmen Kreuzungsanzahlen und Laufzeit betrachtet.

Kapitel 6

Neue Heuristiken zur Kreuzungsreduzierung

In diesem Abschnitt werden neue Heuristiken zur Kreuzungsminimierung vorgestellt. Zuerst betrachten wir *Sifting* für Kreislayouts. Dabei wird jeweils ein Knoten an jeder Position in der gleichbleibenden Ordnung der anderen Knoten platziert, die Anzahl der Kreuzungen jeder dieser Layouts berechnet und ihm schließlich die beste Position zugewiesen. Dieses Verfahren wurde schon erfolgreich zur Kreuzungsminimierung in Lagenlayouts [13] verwendet.

Im zweiten Teil werden einige Ideen des Algorithmus von Mäkinen aufgegriffen. Das grundlegende Vorgehen ist, das Layout schrittweise durch Hinzufügen von Knoten an einem der beiden Enden des bereits berechneten, fortlaufend angeordneten Teillayouts zu erstellen. Aus diesem Schema erhält man mehrere konkrete Algorithmen durch die Wahl der Bearbeitungsreihenfolge und des Kriteriums für die Entscheidung zwischen linker und rechter Seite.

Diese beiden Algorithmen passen in die beiden Kategorien des vorherigen Kapitels. Das zweite Verfahren ist ein Initialisierungsalgorithmus, *Sifting* gehört offensichtlich zur zweiten Gruppe.

6.1 Sifting

Zum ersten Mal wurde *Sifting* als Heuristik zur Minimierung der Knoten in geordneten *Binary Decision Diagrams (BDD)* [15] benutzt. Die Idee ist sehr einfach: Ein ausgewählter Knoten wird durch die gleichbleibende Ordnung der anderen Knoten bewegt und an jeder Position der Wert einer Zielfunktion berechnet. Am Ende erhält er die optimale Position zugewiesen. Diese ist nur *lokal optimal*, da sich die Knotenordnung durch das Sifting eines anderen Knoten ändern und so eine bessere Position für den Knoten entstehen kann.

Da ein Knoten nur dann eine neue Position erhält, wenn sich die Anordnung dadurch verbessert, ist das von Sifting erzeugte Layout auf keinen Fall schlechter als die Startordnung. Dafür hängt das Ergebnis natürlich von der Startordnung ab. Für eine Runde *Sifting* (d.h. jeder Knoten wird einmal ausgewählt und durch die Ordnung bewegt) müssen n^2 Positionen betrachtet und jeweils die Zielfunktion berechnet werden.

Für die Kreuzungsminimierung ist die Zielfunktion die aktuelle Anzahl der Kreuzungen. Für eine Runde *Sifting* müssen n^2 verschiedene Anordnungen betrachtet werden. Wird für jede dieser Ordnungen die Kreuzungsanzahl mit dem Algorithmus *CountAllCrossings* berechnet, liegt die Gesamtlaufzeit in $\mathcal{O}(n^4)$. Es ist also wichtig, ein schnelleres Verfahren zur Berechnung der Kreuzungen zu finden.

Sifting kann als Erweiterung der *Phase 2* von Six und Tollis aufgefasst werden, bei der für einen Knoten nicht nur Positionen neben seinen Nachbarn sondern alle Positionen betrachtet werden. Viele der Überlegungen für *Phase 2* gelten deshalb auch für *Sifting*, insbesondere was die schnellere Berechnung der Kreuzungsanzahl betrifft.

In Lagenlayouts ist *Sifting* schon länger als Algorithmus zur Kreuzungsminimierung bekannt [13]. In diesen Layouts ist die Knotenmenge in mehrere Partitionen aufgeteilt, die jeweils ein eigenes lineares Layout erhalten und zu einem Gesamtlayout in mehreren Lagen übereinander angeordnet werden (siehe Abb. 6.2). Kreuzungsreduzierung in Lagenlayouts ist gut untersucht, aus der Literatur sind einige Heuristiken bekannt. *Sifting* erzielt das beste Ergebnis dieser Heuristiken, ist aber auch deutlich langsamer als die meisten anderen. Mit zwei Beschleunigungsverfahren [14] lässt sich die Laufzeit immerhin noch um den Faktor 20 verkleinern.

Im ersten Teil dieses Abschnitts wird *Sifting* auf Kreislayouts besprochen, inklusive einer einfachen und schnellen Art, die Kreuzungen zu berechnen. Da der Algorithmus trotzdem relativ langsam ist, wäre es schön, die Beschleunigungsverfahren für Lagenlayouts zu übernehmen. Im zweiten Teil wird deshalb *Sifting* auf Lagenlayouts besprochen.

6.1.1 Circular Sifting

Ein ausgewählter Knoten wird in der festen Ordnung der anderen Knoten an jede Position verschoben und die Kreuzungsanzahl berechnet. Am Ende erhält er die Position, die zu den wenigsten Kreuzungen führt, sein momentanes Minimum. Dies ist ein Sifting-Schritt. Eine Sifting-Runde besteht aus einem Sifting-Schritt für jeden Knoten. Wenn sich nach einem Sifting-Schritt das Layout ändert, können neue minimale Positionen für bereits betrachtete Knoten entstehen. Deshalb sind mehrere Sifting-Runden nötig, um ein endgültiges Layout zu erreichen.

Das endgültige Layout kann nicht durch das Verschieben nur eines Knotens verbessert werden. Dies ist eine schöne Eigenschaft des Sifting-Verfahrens, da sie bei

einem Betrachter den Eindruck eines guten Layouts erzeugt. Trotzdem erhält man auch mit *Sifting* Layouts, die offensichtlich nicht schön sind. Ein Beispiel dafür ist der gekreuzte Kreis aus Abb. 5.4, der auch für *Phase 2* schlecht war.

Da *Circular Sifting* als Erweiterung von *Phase 2* gesehen werden kann, gelten die gleichen Überlegungen auch hier, besonders Abschnitt 5.3.2 über das Verschieben eines Knotens. Beim Verschieben des Knotens v durch die gleichbleibende Anordnung der anderen Knoten ändern sich nur Kreuzungen mit Kanten aus $E(v)$. Es ist also nicht nötig, jedesmal die komplette Kreuzungsanzahl zu berechnen, sondern es genügt, die Änderung der Kreuzungen von $E(v)$ zu betrachten. Für *Sifting* kann der gleiche Algorithmus wie für *Phase 2* verwendet werden, wenn die Berechnung der möglichen Positionen eines Knotens in Zeile 5 entsprechend angepasst wird. Die asymptotische Laufzeit verändert sich dadurch nicht und liegt in $\mathcal{O}(nm)$.

In *Phase 2* kann beim Verschieben eines Knotens die Menge der näher zu betrachtenden Kanten eingeschränkt werden auf diejenigen, deren Endknoten in verschiedenen Kreisbögen liegen. Da ein Knoten typischerweise nur wenige Nachbarn besitzt, werden dadurch einige Kanten aussortiert. Für einen *Sifting*-Schritt werden Kreuzungen mit allen Kanten berechnet, dadurch erhöht sich die (reale) Laufzeit. Gleichzeitig werden einige Überlegungen für *Sifting* einfacher, da immer nur direkt nebeneinander liegende Knoten vertauscht werden. Es ist deshalb sinnvoll, diese Situation genauer zu betrachten.

Nebeneinander liegende Knoten vertauschen

Seien π ein gegebenes Kreislayout, v und w zwei nebeneinander liegende Knoten, die vertauscht werden und $\bar{\pi}$ das dadurch entstehende neue Kreislayout.

Durch das Vertauschen zweier nebeneinander liegender Knoten ändern sich nur Kreuzungen zwischen den inzidenten Kanten dieser Knoten. Nach dem Vertauschen gibt es Kreuzungen genau zwischen den Kanten, die sich zuvor nicht gekreuzt haben (siehe Abb. 6.1). Dadurch lässt sich die Berechnung der Kreuzungsänderung vereinfachen zu:

$$\chi_{\bar{\pi}}(G) = \chi_{\pi}(G) - \chi_{\pi}(v, w) + \chi_{\bar{\pi}}(v, w) \ .$$

Mit dem Algorithmus *CountSingleNodeCrossings* lässt sich die Kreuzungsänderung in diesem Fall in $\mathcal{O}(n)$ berechnen, da nur die inzidenten Kanten der beiden Knoten betrachtet werden.

Vorgehensweise

Der Algorithmus *Sifting* orientiert sich an den Überlegungen, mit denen *Phase 2* eine Laufzeit in $\mathcal{O}(nm)$ erreicht.

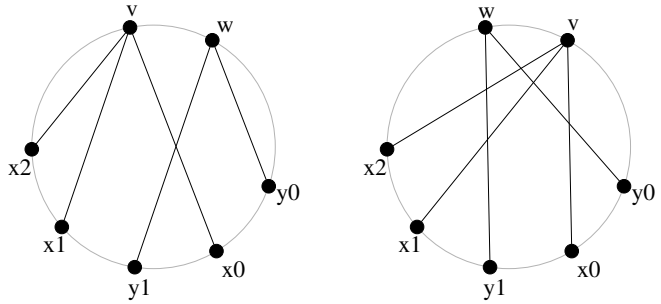


Abbildung 6.1: Veränderung der Kreuzungen durch das Vertauschen der Knoten v und w.

Für die Kreuzungsänderung beim Vertauschen zweier Knoten sind nur die inzidenten Kanten, genauer: die Positionen der adjazenten Knoten, interessant. Während des *Sifting* eines festen Knotens v ändert sich nur die Position dieses Knotens, die Ordnung der anderen Knoten untereinander bleibt gleich. Bei jedem Vertauschen von v mit einem Knoten v_k verursacht eine möglicherweise zwischen diesen Knoten existierende Kante $\{v, v_k\}$ keine Kreuzungen, die Vorkommen von v in den Adjazenzlisten der anderen Knoten können also gelöscht werden. Also können alle Adjazenzlisten bereits vor dem eigentlichen *Sifting* angeordnet werden.

Seien nun v und v_k die Knoten, die vertauscht werden sollen, $x_0 \prec_{v_k} x_1 \prec_{v_k} \ldots \prec_{v_k} x_{r-1}$ und $y_0 \prec_{v_k} y_1 \prec_{v_k} \ldots \prec_{v_k} y_{s-1}$ ihre geordneten Adjazenzlisten und π und $\bar{\pi}$ die Layouts vor und nach dem Vertauschen. In π kreuzt jede zu v inzidente Kante $\{v, x_i\}$ alle zu v_k inzidenten Kanten $\{v_k, y_j\}$, die eine größere Position als x_i besitzen (siehe Abb. 6.1 mit $w = v_k$). Also gilt:

$$\chi_\pi(v, v_k) = \sum_{i<r} |\{y \in N(v_k) : x_i \prec_{v_k} y\}| \ .$$

Analog kreuzt nach dem Vertauschen jede zu v_k inzidenten Kante $\{v_k, y_j\}$ alle zu v inzidente Kanten $\{v, x_i\}$, die eine größere Position als y_j besitzen:

$$\chi_{\bar{\pi}}(v_k, v) = \sum_{j<s} |\{x \in N(v) : y_j \prec_{v_k} x\}| \ .$$

Sind die Adjazenzlisten angeordnet bezüglich v_k, entspricht die Mächtigkeit einer der betrachteten Knotenmengen der Suffixlänge der entsprechenden geordneten Adjazenzliste. Mit dieser Beziehung sind die Summen effizient zu berechnen. Man benötigt die Sortierung der Adjazenzlisten bezüglich v_k. Für v genügt dazu eine zyklische Liste mit einem Zeiger auf das jeweilige Startelement, die anderen Adjazenzlisten werden nur einmal benötigt und können schon zu Beginn entsprechend geordnet werden.

Der Algorithmus 9 zeigt eine Sifting-Runde. Zuerst werden die Adjazenzlisten geordnet. δ speichert die Differenz zur ursprünglichen Kreuzungsanzahl, $\bar{v}$ die bisher

gefundene optimale Position und $\bar{\delta}$ die zugehörige Differenz. Die Kreuzungsänderungen werden, wie beschrieben, mit den Suffixlängen der Adjazenzlisten berechnet. Diese entsprechen den Differenzen $(s - j)$ bzw. $(r - i)$. Schließlich wird der Knoten an die beste gefundene Position platziert.

Algorithm 9: Circular Sifting

Eingabe: Ein Graph G mit Kreislayout π.
Ausgabe: Ein Kreislayout mit weniger oder gleichvielen Kreuzungen.

1. **forall** (v **in** V) **do**
2. Sei $v_0 = v \prec_v v_1 \prec_v \ldots \prec_v v_{n-1}$ das aktuelle Layout
3. **forall** (w **in** V) **do**
4. Ordne die Adjazenzliste von w nach den aktuellen Positionen
5. $\delta \leftarrow 0$
6. $\bar{\delta} \leftarrow 0$; $\bar{v} \leftarrow v_{n-1}$
7. **for** $(k \leftarrow 1, \ldots, n-1)$ **do**
8. Sei $x_0 \prec_{v_k} \ldots \prec_{v_k} x_{r-1}$ die sortierte Adjazenzliste von v ohne v_k
9. Sei $y_0 \prec_{v_k} \ldots \prec_{v_k} y_{s-1}$ die sortierte Adjazenzliste von v_k ohne v
10. $c \leftarrow 0$; $i \leftarrow 0$; $j \leftarrow 0$
11. **while** $(i < r$ **and** $j < s)$ **do**
12. **if** $(x_i \prec_{v_k} y_j)$ **then**
$c \leftarrow c - (s - j)$; $i \leftarrow i + 1$
13. **elseif** $(y_j \prec_{v_k} x_i)$ **then**
$c \leftarrow c + (r - i)$; $j \leftarrow j + 1$
14. **else**
$c \leftarrow c - (s - j) + (r - i)$; $i \leftarrow i + 1$; $j \leftarrow j + 1$
15. $\delta \leftarrow \delta + c$
16. **if** $(\delta < \bar{\delta})$ **then** $\bar{\delta} \leftarrow \delta$; $\bar{v} \leftarrow v_k$
17. Platziere v hinter $\bar{v}$

Laufzeit

Für einen festen Knoten können die Adjazenzlisten in $\mathcal{O}(m)$ sortiert werden, indem man die Knoten entsprechend ihrer Anordnung betrachtet und jeden Knoten zur Adjazenzliste seiner Nachbarn hinzufügt. Das Platzieren an die beste Position ist in konstanter Zeit möglich.

Die Berechnung der Kreuzungsänderung beim Vertauschen von v und v_k benötigt

$\mathcal{O}(d(v) + d(v_k))$ und wird für alle $v_k \in V$ durchgeführt. Es gilt:

$$\sum_{v \in V} \sum_{w \in V} (d(v) + d(w)) = \sum_{v \in V} \sum_{w \in V} d(v) + \sum_{v \in V} \sum_{w \in V} d(w) = 2(n \cdot 2m) \ .$$

Für eine komplette Sifting-Runde liegt die Laufzeit damit in $\mathcal{O}(nm)$. In Kapitel 7.4 werden wir sehen, dass wie bei *Phase 2* nur wenige Runden nötig sind, um ein endgültiges Layout zu finden.

Offene Fragen

Die Reihenfolge, in der die Knoten in der äußeren Schleife ausgewählt werden, wurde noch nicht besprochen. Sie hat möglicherweise einen großen Einfluaa auf die Qualität der erzeugten Layouts, da sie mit für die im Lauf des Algorithmus erzeugten Zwischenlayouts verantwortlich ist.

Dies führt gleich zur nächsten offenen Frage: Wie wirkt sich das Startlayout auf das Ergebnis aus? Da sich ein endgültiges Layout nicht durch das Umordnen eines Knotens verbessern lässt, ist es denkbar, dass der Algorithmus für einen Graph unabhängig vom Startlayout immer ein ähnliches Ergebnis liefert. Andererseits ist das Endlayout eine Verbesserung des Startlayout und deshalb die Qualität für verschiedene initiale Knotenordnungen möglicherweise sehr unterschiedlich.

Die Laufzeit des Algorithmus hängt direkt von der Anzahl der benötigten Runden ab. Deshalb ist es interessant, die Abnahme der Kreuzungen nach jeder Runde zu betrachten. Wenn sich die Anzahl der Kreuzungen in den ersten Runden stark verringert und dann nur noch wenig, kann in zeit-kritischen Anwendungen auf einige Runden verzichtet werden. Nehmen sie hingegen gleichmäßig oder unregelmäßig ab, sind auf jeden Fall alle Runden nötig.

Interessant ist auch die Auswirkung verschiedener Startlayouts auf diesen Prozeß. Falls unabhängig vom Startlayout ähnliche Ergebnisse erreicht werden, werden sie dann bei gutem Startlayout schneller gefunden?

In Kapitel 7.4 werden diese Fragen beantwortet.

Zusammenfassung

Sifting ist ein konzeptionell einfaches Verfahren. Jeder Knoten wird bei fester Anordnung der anderen Knoten an jede Position platziert und die Kreuzungsanzahl berechnet. Dann erhält er die beste dieser Positionen. Mit dem Algorithmus *CountAllCrossings* ist dieses Verfahren einfach zu implementieren, allerdings mit sehr schlechter Laufzeit. Da sich durch jedes Vertauschen zweier Knoten nur wenige Kreuzungen ändern, kann das Verfahren auch deutlich schneller implementiert werden.

Das Anfangslayout wird durch *Sifting* nicht verschlechtert, deshalb kann es verwendet werden, um bereits optimierte Layouts weiter zu verbessern. Die erzeugten Layouts lassen sich nicht durch umplatzieren nur eines Knotens verbessern. Dadurch wirken sie auf einen Betrachter schöner als andere Layouts mit vergleichbarer Kreuzungsanzahl.

Bei dünnen Graphen erreicht man schönere Layouts, wenn *Sifting* mit einem guten Layout startet, zum Beispiel einem von *Phase 1* erstellten. Bei dichten Graphen macht es keinen Unterschied, ob das Startlayout gut oder schlecht war. Da *Sifting* alle möglichen Positionen betrachtet, sind die Layouts von *Phase 1* und *Sifting* auf jeden Fall besser, als die von *Phase 1* und*Phase 2*.

Sifting ist nicht nur asymptotisch langsamer als einige der anderen besprochenen Heuristiken, sondern benötigt auch die meiste reale Laufzeit. Es ist deshalb wünschenswert, die Beschleunigungsverfahren der Lagenlayouts auf Kreislayouts zu übertragen.

6.1.2 Sifting in Lagenlayouts

In diesem Abschnitt wird zuerst *Sifting* in Lagenlayouts besprochen, wie es in [13] vorgestellt wird. Im Vergleich mit den bisher bekannten Heuristiken liefert es deutlich bessere Ergebnisse, ist aber auch einiges langsamer. Deshalb werden zwei Beschleunigungsmethoden untersucht, die in [14] vorgestellt werden. Am Ende werden wir sehen, dass sich diese leider nicht auf Kreislayouts übertragen lassen.

Lagenlayouts

Bei Lagenlayouts wird die Knotenmenge eines Graphen $G = (V, E)$ in Partitionen (Ebenen) $V = V_0 \cup \ldots \cup V_{k-1}$ aufgeteilt, die jeweils ein eigenes lineares Layout erhalten und übereinander angeordnet werden. Oft fordert man zusätzlich, dass keine Kanten zwischen den Knoten der gleichen Partition verlaufen. Abbildung 6.2 zeigt zwei solche Lagenlayouts mit drei Ebenen.

Sollen die Kreuzungen im Layout minimiert werden, wird zusätzlich verlangt, dass Kanten nur zwischen benachbarten Ebenen verlaufen. Dies kann durch das Einfügen neuer Knoten an jedem Schnitt zwischen einer Kante und einer Ebene erreicht werden. Das rechte Layout in Abb. 6.2 entsteht auf diese Weise aus dem linken. In einem solchen Layout werden die Kreuzungen eines Knotens nur von den Ordnungen seiner eigenen und der beiden benachbarten Ebenen bestimmt.

Um die Kreuzungen zu reduzieren, werden die Ebenen nacheinander betrachtet. Durch die Umordnung der Knoten einer Ebene ändern sich auch Kreuzungen für Knoten benachbarter Ebenen, deshalb wird die gleiche Ebene mehrmals bearbeitet.

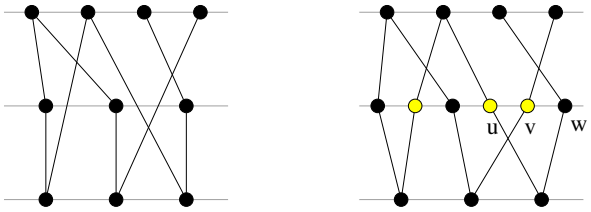


Abbildung 6.2: Beispiele für Lagenlayouts mit 3 Ebenen.

Vorgehensweise

Beim *Sifting* in Lagenlayouts wird ein ausgewählter Knoten an jede Position in seiner Ebene verschoben, ohne dabei die Ordnung der anderen Knoten zueinander zu verändern. Dazu wird der Knoten wiederholt mit seinen linken bzw. rechten Nachbarn vertauscht. An jeder Stelle wird die Kreuzungsanzahl bestimmt und der Knoten dann an die beste Position platziert.

Wie in Kreislayouts ändern sich durch das Vertauschen zweier nebeneinander liegender Knoten nur Kreuzungen zwischen Kanten dieser beiden Knoten. Um die optimale Position zu finden genügt es also, jeweils die Veränderung der Kreuzungsanzahl zu berechnen und diese zu summieren.

Beschleunigung

Für die folgenden Überlegungen betrachten wir eine Ebene V_i, deren Ordnung verbessert werden soll, während die Layouts der anderen Ebenen gleich bleiben. Es können sich also nur Kreuzungen von Kanten zwischen den Ebenen V_{i-1} und V_i und zwischen V_i und V_{i+1} ändern. Ist π_i die Knotenordnung der Ebene V_i, dann sei $\chi(\pi_i)$ die Anzahl dieser Kreuzungen.

Für zwei Knoten $u, v \in V_i$ ist die Kreuzungsanzahl c_{uv} die Anzahl der Kreuzungen zwischen den Kanten aus $E(u)$ und $E(v)$ in einem Layout π_i mit $\pi_i(u) < \pi_i(v)$. Die Kreuzungsanzahl zweier Knoten hängt von der Ordnung der benachbarten Ebenen und der Reihenfolge von u und v ab, nicht aber von den anderen Knoten in V_i. In Abb. 6.2 gilt: $c_{uv} = c_{vu} = 1$, $c_{vw} = c_{wv} = 1$, aber $0 = c_{uw} \neq c_{wu} = 1$.

Kreuzungsmatrix Für das Startlayout π werden für alle Knotenpaare $u, v \in V_i$ aller Ebenen V_i die Kreuzungsanzahlen c_{uv} und c_{vu} berechnet. Diese werden in einem dreidimensionalen *Array*, der Kreuzungsmatrix c, gespeichert. Dabei gilt $c[i, \pi(u), \pi(v)] := c_{uv}$ und $c[i, \pi(v), \pi(u)] := c_{vu}$. Die für das *Sifting* notwendigen Operationen können nun vollständig auf der Knotenmatrix durchgeführt werden, ohne den Graphen zu betrachten.

Werden die Knoten an den Positionen p_1 und p_2 in Ebene V_i vertauscht, müssen in

der Matrix $c[i]$ die Zeilen und Spalten mit Index p_1 und p_2 miteinander vertauscht werden.

Wird ein Knoten v endgültig an einer neuen Position platziert, müssen zusätzlich zur Verschiebung auch Änderungen an Werten benachbarter Ebenen durchgeführt werden. Davon sind die adjazenten Knoten von v betroffen und alle Knoten, die zu Knoten zwischen der alten und neuen Position von v adjazent sind.

Durch die Knotenmatrix kann die Laufzeit auf 10% bis 12% verringert werden.

Untere Schranke Wenn die Kreuzungsanzahlen für alle Knotenpaare einer Ebene V_i bekannt sind, kann eine untere Schranke für die Anzahl der Kreuzungen für alle möglichen Knotenordnungen dieser Ebene angeben werden (wenn die anderen Ordnungen fest sind). Für jedes Knotenpaar $u, v \in V_i$ gilt in jeder Ordnung π_i entweder $\pi_i(u) < \pi_i(v)$ oder $\pi_i(v) < \pi_i(u)$ und damit trägt entweder c_{uv} oder c_{vu} zu $\chi(\pi_i)$ bei.

$$LB := \sum_{u=1}^{|V_i|-1} \sum_{v=u+1}^{|V_i|} \min\{c_{uv}, c_{vu}\}$$

ist also eine untere Schranke für alle möglichen Ordnungen von V_i.

Beim *Sifting* wird immer nur ein Knoten v durch die feste Ordnung der anderen Knoten bewegt. In diesem Fall kann die Schranke noch verschärft werden, indem man nur für die Knotenpaare, an denen v beteiligt ist, das Minimum $\min\{c_{uv}, c_{vu}\}$ betrachtet und von den anderen, für die die Anordnung feststeht, die richtige Kreuzungsanzahl nimmt. Weiter verschärft werden kann die Schranke, wenn das Vertauschen nach links und nach rechts getrennt betrachtet wird.

$$\begin{aligned} LB^{\leftarrow}(v) &= \sum_{u_1=1}^{|V_i|-1} \sum_{u_2=u+1}^{|V_i|} c_{u_1u_2} - \sum_{u=1}^{\pi(v)-1} \max\{0, c_{vu} - c_{uv}\} \\ LB^{\rightarrow}(v) &= \sum_{u_1=1}^{|V_i|-1} \sum_{u_2=u+1}^{|V_i|} c_{u_1u_2} - \sum_{u=\pi(v)+1}^{|V_i|} \max\{0, c_{uv} - c_{vu}\} \end{aligned}$$

sind untere Schranken für die Kreuzungsanzahl, die durch Sifting des Knotens v nach links bzw. rechts erreicht werden kann.

Mit diesen unteren Schranken kann das *Sifting* der Knoten beschleunigt werden. Vor dem Vertauschen werden die beiden Schranken berechnet. Dies ist in $\mathcal{O}(|V_i|)$ möglich. Bei jedem Vertauschen wird die entsprechende Schranke aktualisiert (in konstanter Zeit) und mit der berechneten Kreuzungsanzahl verglichen. Sind beide Werte gleich, kann das Vertauschen in diese Richtung beendet werden, da an den restlichen Positionen keine Verbesserung erreicht werden kann.

Durch die unteren Schranken verringert sich die Laufzeit nochmals auf die Hälfte bis ein Drittel. Insgesamt verringert sich die Laufzeit auf ungefähr 5%.

Beschleunigung für Kreislayouts

Die Beschleunigungsverfahren beruhen auf den beiden Kreuzungsanzahlen c_{uv} und c_{vu}. Diese sind in Lagenlayouts dank der eindeutigen linearen Ordnung jeder Ebene wohldefiniert. In Kreislayouts gibt es keine eindeutige lineare Ordnung der Knoten, zwei Knoten u und v sind nicht in einer eindeutigen Links-Rechts-Beziehung. Die Kreuzungsanzahlen c_{uv} und c_{vu} sind damit nicht für alle möglichen Knotenpaare wohldefiniert.

6.2 Circular Insert

In Abschnitt 5.1 haben wir einen Algorithmus von Erkki Mäkinen gesehen, der eigentlich zur Reduzierung der Kantenlänge gedacht ist, aber auch die Kreuzungsanzahl verringert. Von diesem wird die Idee übernommen, die Knoten nacheinander an einer Seite des bereits berechneten Teillayouts anzufügen, um einen einfachen und schnellen Algorithmus zur Kreuzungsminimierung zu entwerfen.

Wir betrachten zuerst einen Basisalgorithmus, der nacheinander Knoten an einem der beiden Enden eines fortlaufend angeordneten Teillayouts anhängt. In diesem ist sowohl die Reihenfolge, in der die Knoten betrachtet werden (*Bearbeitungsreihenfolge*), als auch das Kriterium, nach dem entschieden wird, auf welcher Seite der Knoten hinzugefügt wird (*Positionswahl*), noch nicht definiert.

Im zweiten und dritten Teil werden verschiedene Möglichkeiten für die Bearbeitungsreihenfolge und die Positionswahl besprochen und schließlich die beste Kombination für die Kreuzungsreduzierung ausführlicher vorgestellt.

Da die Idee zu diesem Abschnitt von einer Heuristik zur Reduzierung der Kantenlänge stammt, werden für die Algorithmen nicht nur die Kreuzungsanzahlen, sondern auch die Kantenlängen der produzierten Layouts betrachtet.

6.2.1 Basisalgorithmus

Die Idee des Basisalgorithmus ist sehr einfach. Das Layout wird schrittweise durch Hinzufügen von Knoten erstellt. Zuerst erhält ein Startknoten eine beliebige Position im noch leeren Layout. Danach werden nacheinander alle anderen Knoten in der Bearbeitungsreihenfolge betrachtet und mit dem Positionswahlkriterium entschieden, auf welcher Seite sie zum bereits bestehenden Layout hinzugefügt werden. Dadurch entsteht in jedem Schritt aus einem fortlaufenden Teillayout wieder ein fortlaufendes Teillayout.

Meist wird die Bearbeitungsreihenfolge vom bereits erstellten Teillayout abhängen. Der Algorithmus sieht deshalb nach dem Einfügen eines Knotens die Möglichkeit vor, diese zu ändern. Sollen beispielsweise nur Knoten bearbeitet werden, die mit dem Teillayout verbunden sind, enthält die Bearbeitungsreihenfolge zu Beginn nur die adjazenten Knoten des Startknoten und bekommt nach jedem Einfügen die Nachbarn des aktuellen Knoten hinzu.

Damit ist die Grundstruktur bereits fertig. Der Algorithmus *Base Circular Insert* zeigt den entsprechenden Pseudocode.

Algorithm 10: Base Circular Insert

Eingabe: Ein Graph G.
Ausgabe: Ein Kreislayout mit kleiner Kantenlänge.

1. *nodeOrder* $\leftarrow$ Die Bearbeitungsreihenfolge der Knoten
2. Platziere den ersten Knoten beliebig

3. **while** (nodeOrder.length() > 0) **do**
4. Sei v der nächste Knoten in *nodeOrder*
5. Füge v am besseren Ende zum Teillayout hinzu
6. Aktualisiere die Reihenfolge in *nodeOrder*

Der Algorithmus von Mäkinen verwaltet die bereits platzierten Knoten in zwei Layouts, eines für die linke Seite, eines für die rechte. Ein Knoten wird dann in das Layout eingefügt, in dem er mehr Nachbarn besitzt. Da der Basisalgorithmus ein zusammenhängendes Layout verwaltet, kann dieses Verfahren mit ihm nicht umgesetzt werden.

6.2.2 Bearbeitungsreihenfolge

In diesem Abschnitt werden drei Bearbeitungsreihenfolgen vorgestellt, von denen zu erwarten ist, dass sie bezüglich Kreuzungsanzahl und Kantenlänge zu guten Ergebnissen führen. Zum Vergleich mit dem Algorithmus von Mäkinen wird außerdem die Bearbeitung in der Reihenfolge der Knotengrade betrachtet.

Bei der Motivation der Reihenfolgen werden *offene* Kanten betrachtet. Dies sind in diesem Zusammenhang Kanten zwischen Knoten des Teillayout und noch unplatzierten Knoten. Die Idee ist, die Anzahl der offenen Kanten möglichst klein zu halten, da sie im Laufe des Algorithmus immer länger werden bzw. immer mehr Kreuzungen erzeugen.

Nun die vier Bearbeitungsreihenfolgen:

1. Knotengrad. Die Knoten werden absteigend nach den Knotengraden sortiert. Dieses Vorgehen entspricht in etwa der Heuristik von Mäkinen. In diesem Fall ist in Zeile 5 des Algorithmus keine Aktualisierung der Knotenreihenfolge nötig.

2. Verbundenheit. Es wird immer der Knoten bearbeitet, der am stärksten mit dem schon erstellten Teillayout verbunden ist. Dadurch werden viele *offene* Kanten beendet. Diese Reihenfolge kann mit einer *Priority Queue* realisiert werden, in der die unplatzierten Knoten mit der Anzahl ihrer platzierten Nachbarn gespeichert sind. Es wird immer der Knoten v mit höchster Priorität bearbeitet. Nach dem Einfügen von v müssen die Prioritäten seiner Nachbarn aktualisiert werden (Zeile 5).

3. Anzahl unplatzierter Nachbarn. Bei diesem Vorgehen wird die Anzahl der nicht platzierten Nachbarn der Knoten betrachtet und der Knoten mit minimalem Wert gewählt. Die Idee ist, dass dieser Knoten wenig neue Kanten *öffnet*. Da die betrachtete Anzahl die Differenz aus Knotengrad und Verbundenheit ist, ist das Verfahren eng mit Version 2 verwandt. Es kann ebenfalls mit einer *Priority Queue* realisiert werden, in der nach dem platzieren eines Knotens seine Nachbarn aktualisiert werden.

4. Kombination. Dieses Verfahren kombiniert die vorherigen beiden Kriterien. Die Knoten werden zuerst nach ihrer Verbundenheit ausgewählt. Gibt es mehrere Knoten mit gleichem Wert, wird derjenige mit den wenigsten unplatzierten Nachbarn gewählt.

Für diese vier Verfahren und den Algorithmus von Mäkinen wurden die Kantenlänge und die Kreuzungsanzahl für die Graphen aus Kapitel 7.1 berechnet. Zur Positionswahl wurde dazu die Kantenlänge verwendet, da sie schnell und einfach zu berechnen ist. Im nächsten Abschnitt wird dies genauer beschrieben.

Für alle untersuchten Graphenfamilien und beide Optimierungskriterien ergeben sich ähnliche Ergebnisse. Ein Algorithmus liefert entweder durchgehend gute oder schlechte Ergebnisse.

Die Wahl nach den Knotengraden ist wie erwartet nicht gut. Version 1 und der Algorithmus von Mäkinen liegen ungefähr gleich und sind die schlechtesten Verfahren. Die Auswahl nach der Verbundenheit eines Knotens (Version 2) und der Anzahl unplatzierter Knoten (Version 3) sind deutlich besser. Die Kombination der beiden Auswahlverfahren (Version 4) liefert durchgehend die besten Ergebnisse. In Kapitel 7.2 sind die Ergebnisse ausführlicher dargestellt.

6.2.3 Positionswahl

Betrachten wir nun verschiedene Möglichkeiten für die Wahl der Position des einzufügenden Knotens. Einfache Methoden sind die zufällige Wahl der Seite und das

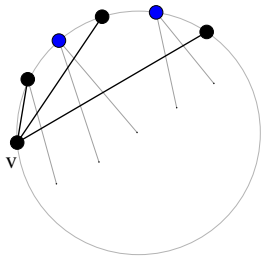


Abbildung 6.3: Die inzidenten Kanten von v kreuzen offene Kanten.

Einfügen auf immer der gleichen Seite. Letzteres führt zu einer Knotenordnung, die der Bearbeitungsreihenfolge entspricht. Etwas kompliziertere Kriterien betrachten die Kantenlänge und die Anzahl der Kreuzungen, die für die beiden Positionen entstehen.

Kantenlängen

Dieses Verfahren berechnet für einen einzufügenden Knoten v für beide möglichen Positionen die Summe der Kantenlängen zu seinen schon platzierten Nachbarn und wählt die Position, die zum kleineren Wert führt.

Da die Länge einer Kante in konstanter Zeit berechnet werden kann, kann diese Wahl in $\mathcal{O}(d(v))$ getroffen werden. Für den gesamten Algorithmus liegt die Laufzeit für die Positionswahl damit in $\mathcal{O}(m)$.

Kantenkreuzungen

Da das interessantere Optimierungskriterium die Anzahl der Kreuzungen ist, ist es naheliegend, nicht die Kantenlänge, sondern die Anzahl erzeugter Kreuzungen zu betrachten. Dabei können nicht die unmittelbar durch das Einfügen eines Knoten entstehenden Kreuzungen im Teillayout betrachtet werden, da sich die beide Einfügepositionen in der Teilknotenordnung nicht unterscheiden und deshalb zu den gleichen Kreuzungen führen.

Stattdessen werden Kreuzungen gezählt, die im aktuellen Teillayout noch gar nicht vorhanden sind. Dazu betrachten wir für jeden platzierten Knoten seine offenen Kanten, d.h. Kanten, deren anderer Endknoten noch unplatziert ist. Wir nehmen an, der aktuelle Knoten wird auf der linken Seite ins Teillayout eingefügt und wählen eine seiner inzidenten Kanten zu einem schon platzierten Knoten. Diese Kante wird im Endlayout alle offenen Kanten derjenigen Knoten kreuzen, die im Teillayout zwischen ihren beiden Endknoten liegen (siehe Abb. 6.3). Entsprechendes gilt für das Einfügen auf der rechten Seiten. Durch die Berechnung der Kreuzungen mit offenen Kanten werden für die beiden Positionen offensichtlich unterschiedliche Kreuzungen betrachtet.

Vergleich

Für diese vier Kriterien wurden wieder die Kantenlänge und die Kreuzungsanzahl für die Graphen aus Kapitel 7.1 berechnet. Da sich Version 4 schon als beste Bearbeitungsreihenfolge herausgestellt hat, wird diese für die Tests verwendet.

Wenig überraschend ist, dass die Zufallsplatzierung sehr schlechte Layouts erzeugt. Für die drei anderen Verfahren liegen die Kreuzungsanzahlen erstaunlich nahe beieinander. Bei dünnen Graphen ist die Wahl nach den Kreuzungen relativ deutlich besser als nach den Kantenlängen. Bei dichteren Graphen schwindet der Vorsprung schnell. Das Einfügen auf nur einer Seite ist nur wenig schlechter als die Wahl nach den Kantenlängen.

Die Kantenlänge ist bei allen drei Verfahren praktisch gleich. Genauere Ergebnisse liefert wieder Kapitel 7.2.

6.2.4 Circular Insert

Wir betrachten nun den endgültigen Algorithmus. Dazu wird der Basisalgorithmus mit der besten Bearbeitungsreihenfolge (Version 4) und der besten Positionswahl (Kantenkreuzungen) kombiniert.

Vorgehensweise

Der Startknoten wird beliebig gewählt. Verschiedene Experimente haben gezeigt, dass es für die Qualität der Ergebnisse im Mittel egal ist, ob der Startknoten mit minimalem oder maximalem Grad oder zufällig gewählt wird.

Mit zwei Datenstrukturen *Layout* und *PriorityTable* lässt sich der Algorithmus einfach und übersichtlich implementieren. *Layout* repräsentiert natürlich das bisher erzeugte Teillayout. Dies kann realisiert werden durch ein *Array* und zwei Positionen *links* und *rechts*, die die beiden Enden des Teillayouts definieren.

Um die Kreuzungsanzahl schnell zu berechnen, wird für jeden Knoten in *Layout* die Anzahl seiner offenen Kanten gespeichert. Immer wenn ein Knoten ins Layout eingefügt wird, muss diese für seine Nachbarn um eins verringert werden. Nun genügt ein Durchlauf durch das Layout, beispielsweise von links nach rechts, um die Kreuzungsanzahl zu berechnen. Für jeden Knoten wird die Präfixsumme der offenen Kanten seiner Vorgänger berechnet. Ist ein Knoten adjazent zum eingefügten Knoten, kreuzt die entsprechende Kante alle bisher gezählten offenen Kanten, also wird seine Präfixsumme zur Gesamtzahl addiert. Auf diese Weise werden die Kreuzungen für die linke Position gezählt.

Für die rechte Position kann das gleiche Verfahren mit umgekehrter Knotenordnung wiederholt werden. Schöner ist aber die gleichzeitige Berechnung der Summe der Suffixsummen im ersten Durchlauf. Diese ergeben die Kreuzungen für die

rechte Seite.

Die *PriorityTable* repräsentiert die Bearbeitungsreihenfolge. Um die Knoten nach dem Wert ihrer Verbundenheit zu wählen, wird eine *Priority Queue* verwendet. Da Knoten mit gleicher Verbundenheit nochmals geordnet sind, enthält sie nicht direkt die Knoten, sondern *Priority Queues*, die jeweils Knoten gleicher Verbundenheit enthalten und diese mit der Anzahl ihrer unplatzierten Nachbarn speichern. Die äußere Queue liefert immer ihr größtes Element zurück, die inneren ihr kleinstes.

Die *PriorityTable* muss zwei Operationen unterstützen, Finden und Entfernen des nächsten Elements und Aktualisieren der Priorität von Knoten. Das Aktualisieren wird für die Nachbarn des aktuell zum Layout hinzugefügten Knotens benötigt. Dabei nimmt die Anzahl der nicht-platzierten Nachbarn um eins ab und die Verbundenheit um eins zu, also müssen die neuen Prioritäten nicht extra angegeben werden.

Algorithm 11: Circular Insert

Eingabe: Ein Graph G.
Ausgabe: Ein Kreislayout mit kleiner Kantenlänge.

1. **PriorityTable** nodeOrder $\leftarrow$ Startknoten mit Priorität $(1, 1)$,
   alle anderen Knoten v mit Priorität $(0, d(v))$
2. **Layout** layout $\leftarrow \emptyset$

3. **while** (nodeOrder.length() > 0) **do**
4. $v \leftarrow$ nodeOrder.delNext()
5. layout $\leftarrow v$
6. **forall** (w **in** $N(v)$ **and** layout.notPlaced(w)) **do**
7. nodeOrder.update(w)

Laufzeit

Mit der üblichen Implementation der *Priority Queues* mit Fibonacci-Heaps ist das Finden und Entfernen des nächsten Knotens in $\mathcal{O}(\log(n))$ möglich. Wird die Priorität eines Knotens aktualisiert, muss er aus einer Queue entfernt werden und in eine andere eingefügt werden, dies ist in $\mathcal{O}(\log(n))$ möglich.

Zu Beginn des Algorithmus werden alle Knoten außer dem Startknoten mit Verbundenheit 0 und dem Knotengrad zur *PriorityTable* hinzugefügt. Dies benötigt $\mathcal{O}(n \log(n))$ Laufzeit. Das Entfernen des jeweils nächsten Knotens benötigt insgesamt ebenfalls $\mathcal{O}(n \log(n))$, das Aktualisieren der *PriorityTable* benötigt $\mathcal{O}(m \log(n))$.

Die Berechnung der besseren Position eines Knotens benötigt $\mathcal{O}(n)$. Damit liegt die Gesamtlaufzeit in $\mathcal{O}(n^2 + m \log(n))$.

6.2.5 Zusammenfassung

Der Algorithmus *Circular Insert* ist ein einfaches und schnelles Verfahren, um sowohl die Kantenlänge als auch die Kreuzungsanzahl eines Layouts zu reduzieren. Außerdem liefert er sehr gute Ergebnisse. Deshalb ist er gut geeignet, um Startlayouts für weitere Optimierungsschritte, beispielsweise mit *Sifting*, zu produzieren.

Durch die Bearbeitungsreihenfolge *Version 4* werden bereits so gute Ergebnisse erzielt, dass die Positionswahl nur noch kleine Verbesserungen bringt. Spielt der Implementationsaufwand eine Rolle, kann diese Reihenfolge als Layout gewählt werden, ohne dass sich das Ergebnis stark verschlechtert.

Kapitel 7

Experimente

In diesem Kapitel werden die an vielen Stellen zuvor angekündigten Experimente besprochen. Zuerst werden die für die Tests verwendeten Graphen beschrieben. Danach folgt ein Vergleich der verschiedenen Verfahren aus Abschnitt 6.2, die zum Algorithmus *Circular Insert* geführt haben, und des Algorithmus von Mäkinen. Der dritte Abschnitt vergleicht die Ergebnisse der wichtigsten besprochenen Heuristiken. Im letzten Teil werden einige Aspekte der *Sifting*-Heuristik genauer untersucht.

Die Experimente wurden auf verschiedenen modernen PCs (ca. 1.5 GHz, 512 MB Ram) unter Suse Linux 8.1 durchgeführt. Die Algorithmen wurden in C++ mit der Graphenbibliothek LEDA [18] implementiert.

Alle Algorithmen lassen noch Wahlmöglichkeiten offen, bei *Phase 1* ist das beispielsweise die Wahl des nächsten Knoten, die trotz der drei Bedingungen nicht immer eindeutig ist. Dadurch können für einen Graphen (bzw. ein Layout) unterschiedliche Endlayouts berechnet werden. Um die dadurch auftretenden Verzerrung der Resultate zu verkleinern, wurden die Berechnungen für jeden Algorithmus auf jedem Graphen fünfmal wiederholt, jeweils mit unterschiedlicher interner Knotenordnung der Graphen, und die Ergebnisse gemittelt.

7.1 Untersuchte Graphen

Die Ergebnisse der Algorithmen werden für drei verschiedene Gruppen von Graphen verglichen:

- Rom-Graphen. Diese Gruppe wird von Giuseppe Di Battista bereitgestellt [17] und oft verwendet, um Algorithmen zu testen. Sie enthält 11481 Graphen mit 10 bis 100 Knoten und geringer Dichte. Der durchschnittliche Knotengrad liegt unter drei. Die Experimente wurden nicht mit den Originalgraphen

durchgeführt, sondern mit ihren zweifachen Zusammenhangskomponenten, die aus mindestens 10 Knoten bestehen. Dies sind 10541 Graphen.

- Zufallsgraphen mit festem durchschnittlichen Knotengrad. Diese Gruppe enthält vier Familien von Graphen mit den durchschnittlichen Knotengraden 3, 4, 5 und 10, bezeichnet mit *D03*, *D04*, *D05* und *D10*. Damit steigt die Kantenanzahl in diesen Familien linear mit der Knotenanzahl. Jede Familie besteht aus zweifach zusammenhängenden Zufallsgraphen mit 10 bis 500 Knoten. Sie enthält zu jeder durch fünf teilbaren Knotenzahl 10 verschiedene Graphen.

- Zufallsgraphen mit fester durchschnittlicher Dichte. Die letzte Gruppe besteht aus drei Graphenfamilien mit jeweils fester durchschnittlicher Dichte von 2%, 5% und 10%. Sie werden mit *P02*, *P05* und *P10* bezeichnet. Die Kantenanzahl dieser Graphen ist proportional zu n^2. Jede Familie besteht aus zweifach zusammenhängenden Zufallsgraphen mit 20 bis 200 Knoten. Sie enthält zu jeder durch fünf teilbaren Knotenzahl 10 verschiedene Graphen.

Für die Visualisierung sind vor allem Graphen mit bis zu 100 Knoten und geringer Dichte interessant. Kreislayouts größerer Graphen sind nur auf einer großen Fläche gut darstellbar. In Graphen hoher Dichte enthält schon ein optimales Layout viele Kreuzungen, der Einfluss der Reduzierung ist also geringer. Bei anderen Anwendungen werden auch größere und dichtere Graphen betrachtet.

Als Vertreter für dünne Graphen werden meistens die Ergebnisse für die Rom-Graphen und D03 gezeigt. Für kleine Graphen sind die Resultate beider Familien sehr nahe beieinander, da beide eine ähnliche Dichte aufweisen. Die Rom-Graphen zeigen den Bereich bis 80 Knoten sehr detailliert, D03 den Verlauf für größere Graphen.

Für die dichteren Graphen werden D10 und P05 betrachtet. Durch den Vergleich zwischen D03 und D10 kann auf die Skalierbarkeit eines Algorithmus bei höherer Dichte geschlossen werden. Um nicht nur Graphen mit linear zur Knotenanzahl steigender Kantenanzahl zu untersuchen, wird P05 exemplarisch für P02 und P10 betrachtet. Da diese Graphen schnell sehr viele Kanten und damit Kreuzungen enthalten, werden nur Graphen mit höchstens 200 Knoten betrachtet.

7.2 Circular Insert und Mäkinen

In diesem Abschnitt werden die verschiedenen Verfahren, die schließlich zum Algorithmus *Circular Insert* führen, verglichen. Das für alle Verfahren gleiche grundlegende Vorgehen – das Layout wird schrittweise erstellt, indem Knoten an einem Ende des bereits bestehenden Teillayout hinzugefügt werden, je nachdem, welche

Position die bessere ist – wird dabei durch die Wahl der Bearbeitungsreihenfolge und die Art der Positionswahl variiert.

Zuerst werden die verschiedenen Bearbeitungsreihenfolgen und der Algorithmus von Mäkinen betrachtet. Danach werden die vier Möglichkeiten, um die Position eines Knoten zu wählen, verglichen.

In diesem Abschnitt werden auch die Kantenlängen der berechneten Layouts betrachtet, da die ursprüngliche Idee für diese Algorithmen von diesem Optimierungskriterium stammt. Das Hauptaugenmerk liegt aber auf der Anzahl der Kreuzungen. Um die Algorithmen dieses Abschnitts mit den anderen Heuristiken vergleichen zu können, sind in den Diagrammen immer auch die Ergebnisse von *Phase 1* aufgeführt, ohne das sie im Text besonders erwähnt werden.

7.2.1 Bearbeitungsreihenfolgen

In Abschnitt 6.2 sind die folgenden vier Bearbeitungsreihenfolgen beschrieben:

1. Knotengrad,
2. Verbundenheit (Anzahl platzierter Nachbarn),
3. Anzahl unplatzierter Nachbarn,
4. Kombination von Version 2 und 3.

Diese Verfahren wurden mit der Positionswahl nach der Kantenlänge kombiniert, da sie schnell und einfach durchzuführen ist und die Reduktion der Kantenlänge das ursprüngliche Ziel der Algorithmen ist. Zusätzlich wurde der Algorithmus von Mäkinen getestet.

Mit diesen fünf Algorithmen wurden für die vier Graphenfamilien Layouts erstellt und die Kreuzungsanzahl und die Kantenlänge berechnet. Die Ergebnisse der Graphen gleicher Knotenanzahl wurden gemittelt. Die Abbildungen 7.1 und 7.2 zeigen die Resultate.

Zuerst fällt auf, dass die Ergebnisse für die Kreuzungsanzahl und die Kantenlänge sehr ähnlich sind. Liefert ein Verfahren ein Layout mit wenigen Kreuzungen, dann hat es auch eine geringe Kantenlänge, hat das Layout eine große Kantenlänge, dann enthält es auch viele Kreuzungen. Dies bestätigt die Intuition, dass eine Reduzierung der Kantenlänge auch zu einer kleineren Kreuzungsanzahl führt.

Für die verschiedenen Graphenfamilien sind die Ergebnisse ebenfalls sehr ähnlich. Auch hier gilt, dass ein Verfahren entweder für alle Familien gut oder für alle schlecht ist.

Eindeutig die schlechtesten Ergebnisse liefert die Wahl nach dem Knotengrad. Das ist nicht überraschend, da dabei die Adjazenzbeziehung der Knoten nicht beachtet

wird, die die Kreuzungsanzahl und die Kantenlänge bestimmt. Etwas besser ist der Algorithmus von Mäkinen, der zwar auch die Knoten in der Reihenfolge der Knotengrade betrachtet, aber ein paar zusätzliche Kriterien beachtet.

Das beste Verfahren ist die kombinierte Auswahl nach Version 4. Es liefert in allen Experimenten die besten Layouts. Dies macht sich besonders bei dünnen Graphen bemerkbar, bei denen es deutlich bessere Ergebnisse produziert als die alleinige Wahl nach der Verbundenheit. Bei dichteren Graphen nimmt der Vorteil ab. Die Wahl nach den unplatzierten Nachbarn ist meistens schlechter als Version 2, aber auch deutlich besser als die Wahl nach den Knotengraden.

Die Laufzeit aller Verfahren ist so gering, dass die Bilder nicht aufgeführt sind. Sie liegt auch für große Graphen unter 0.1 Sekunden. Abbildung 7.5 zeigt die Laufzeit für den interessantesten Algorithmus, Version 4.

7.2.2 Positionswahl

Betrachten wir nun den Einfluss verschiedener Kriterien für die Wahl zwischen linker und rechter Position. Als Bearbeitungsreihenfolge der Knoten wird dazu Version 4 gewählt, da sie die besten Ergebnisse liefert. Verglichen wird die Wahl der Position nach der Kantenlänge (*Length*) und der Kreuzungsanzahl (*Cross*), das Hinzufügen auf nur einer Seite (*One Side*) und eine zufällige Verteilung auf links und rechts.

Wie oben wurden für die vier Graphenfamilien mit diesen vier Verfahren Layouts erstellt, die Kreuzungsanzahl und die Kantenlänge berechnet und die Ergebnisse über alle gleichgroßen Graphen gemittelt. Die Abbildungen 7.3, 7.4 und 7.5 zeigen die Resultate.

Wie erwartet ist das Ergebnis bei zufälliger Wahl der Seite am schlechtesten. Betrachtet man die Kantenlängen, sind fast keine Unterschiede zwischen den drei anderen Verfahren festzustellen.

Für die Kreuzungsanzahl sind bei den sehr dünnen Rom-Graphen noch deutliche Unterschiede zu sehen, die aber schon bei D03 fast verschwinden. Die Auswahl nach den Kreuzungen ist am besten, gefolgt von der Kantenlänge und dem Einfügen auf nur einer Seite.

Die Laufzeit von *Circular Insert* ist deutlich höher als die der anderen Verfahren, aber trotzdem sehr gering.

7.2.3 Zusammenfassung

Wir haben im ersten Teil gesehen, dass die Wahl der Bearbeitungsreihenfolge einen großen Einfluss auf die Qualität der berechneten Layouts hat. Die Ergebnisse für verschiedene Verfahren unterscheiden sich sehr stark. Die kombinierte Auswahl nach der Verbundenheit und der Anzahl unplatzierter Nachbarn ist eine sehr gute

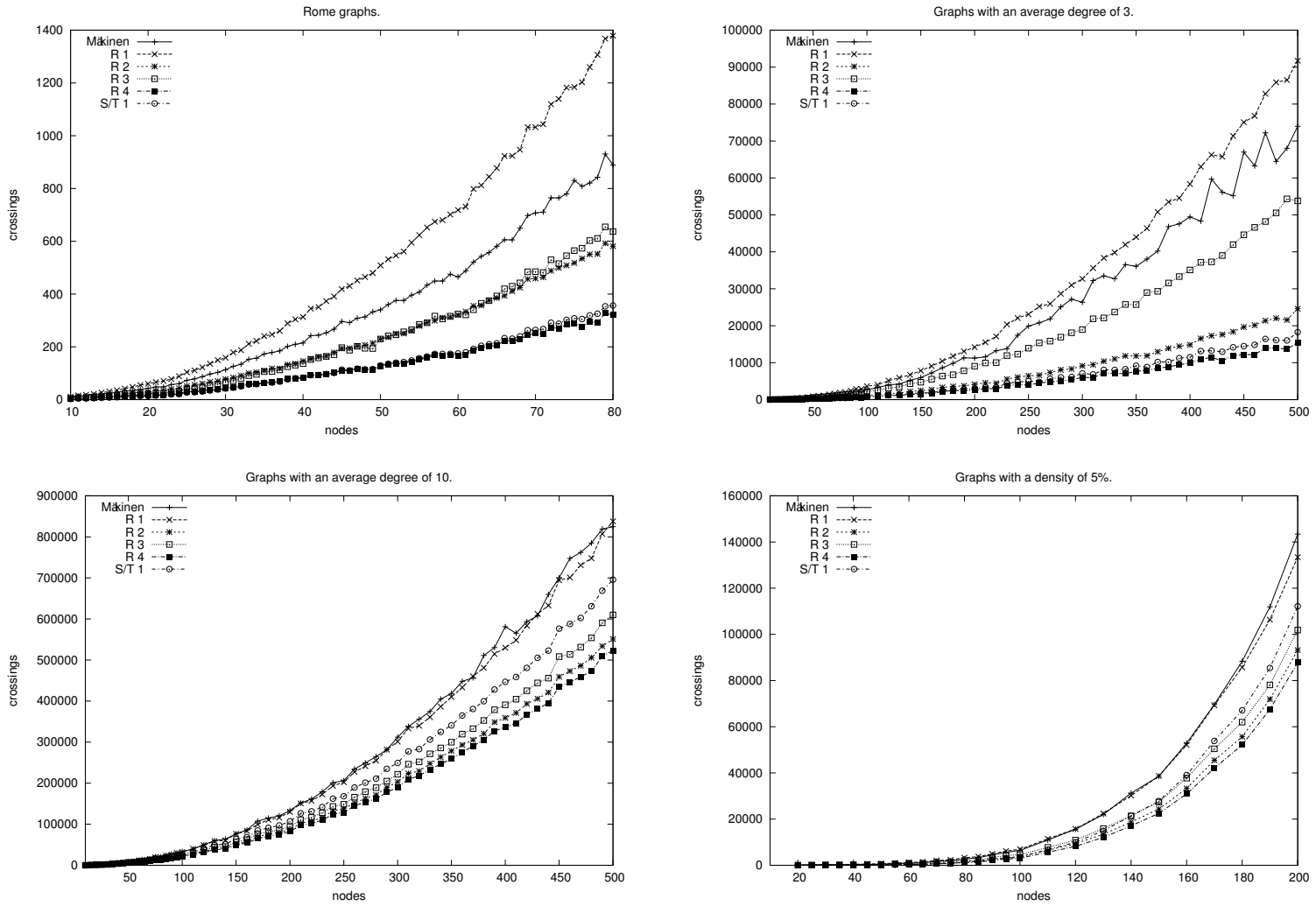


Abbildung 7.1: Kreuzungsanzahl für verschiedene Bearbeitungsreihenfolgen.

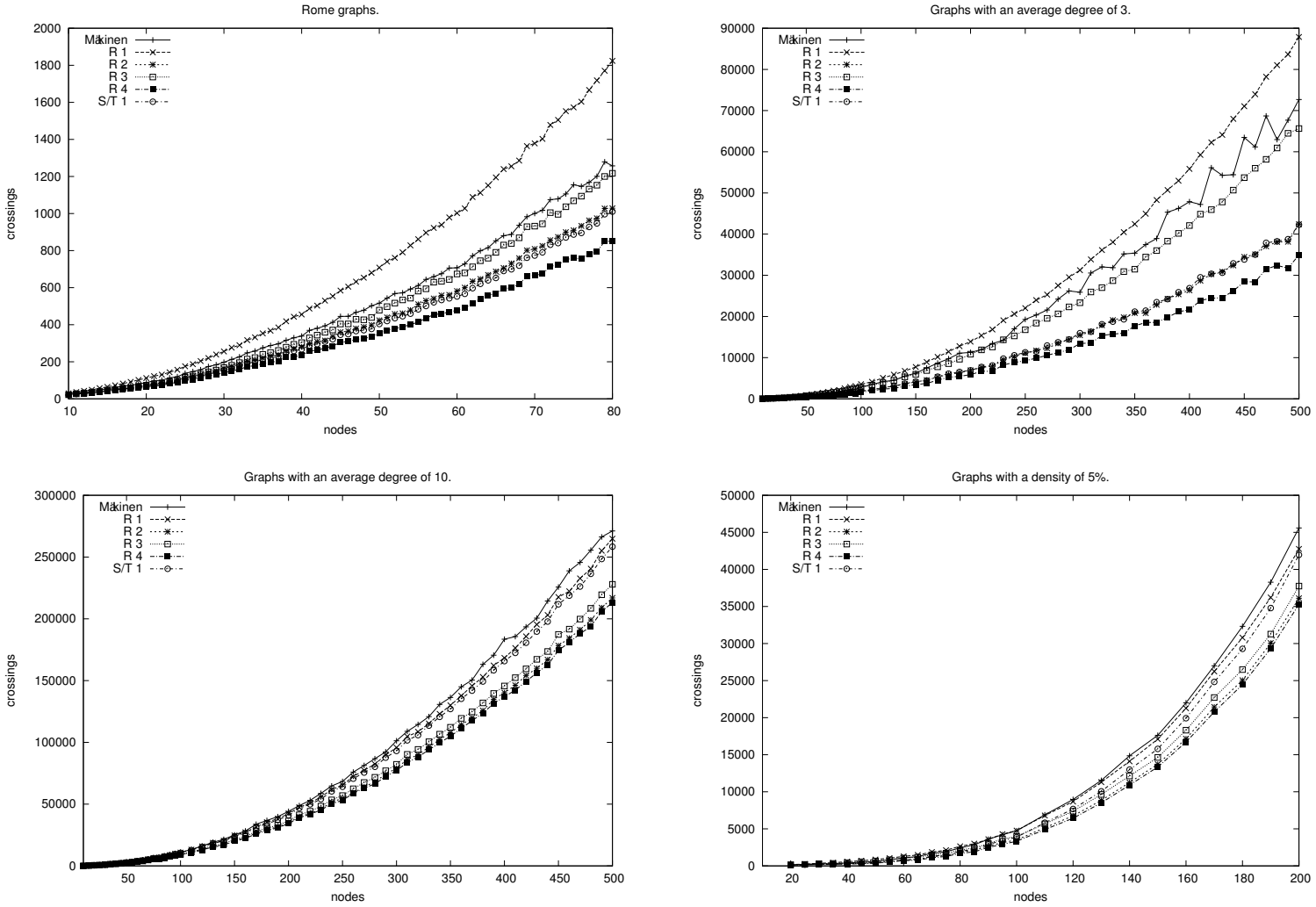


Abbildung 7.2: Kantenlänge für verschiedene Bearbeitungsreihenfolgen.

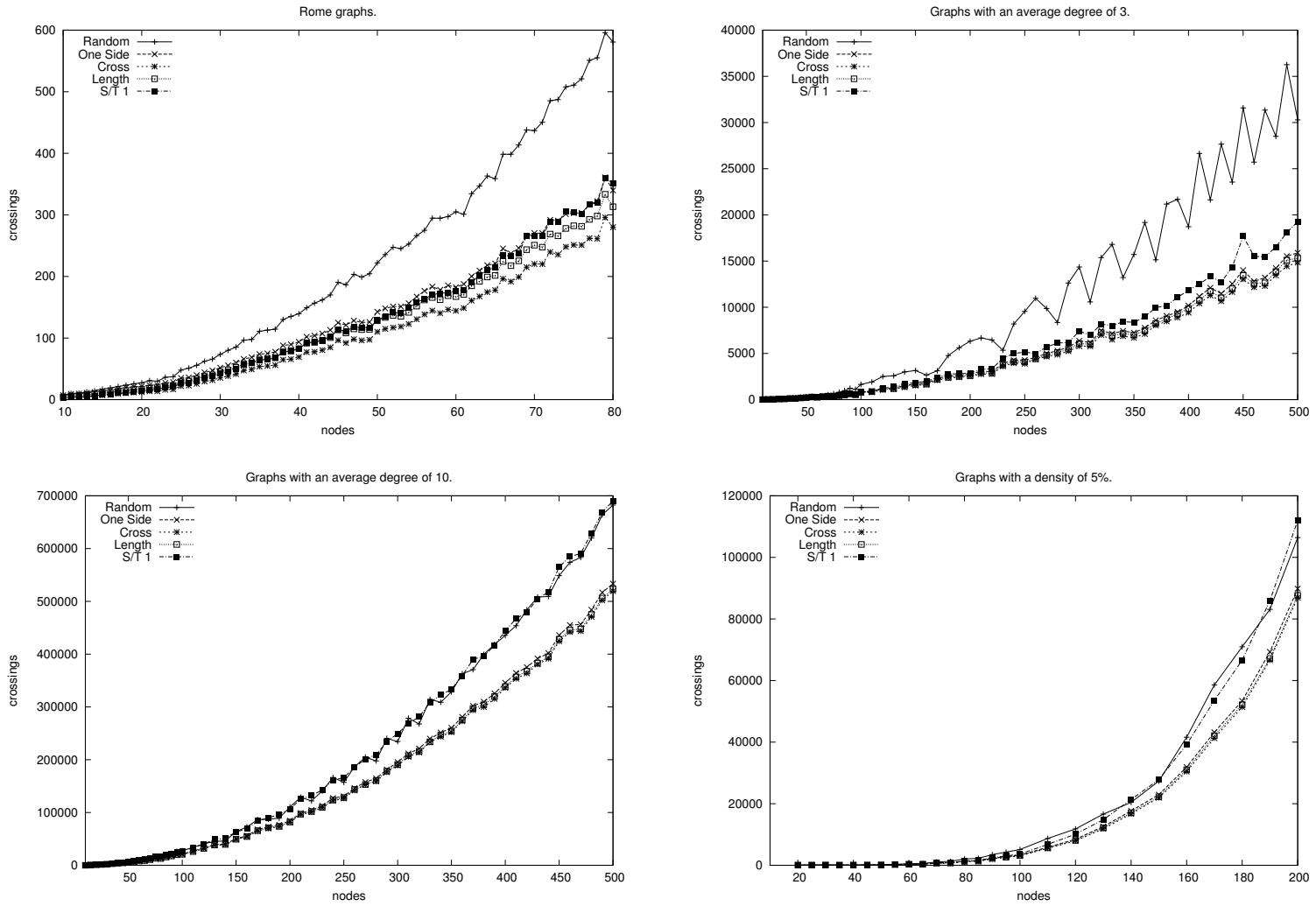


Abbildung 7.3: Kreuzungsanzahl für verschiedene Platzierungsstrategien.

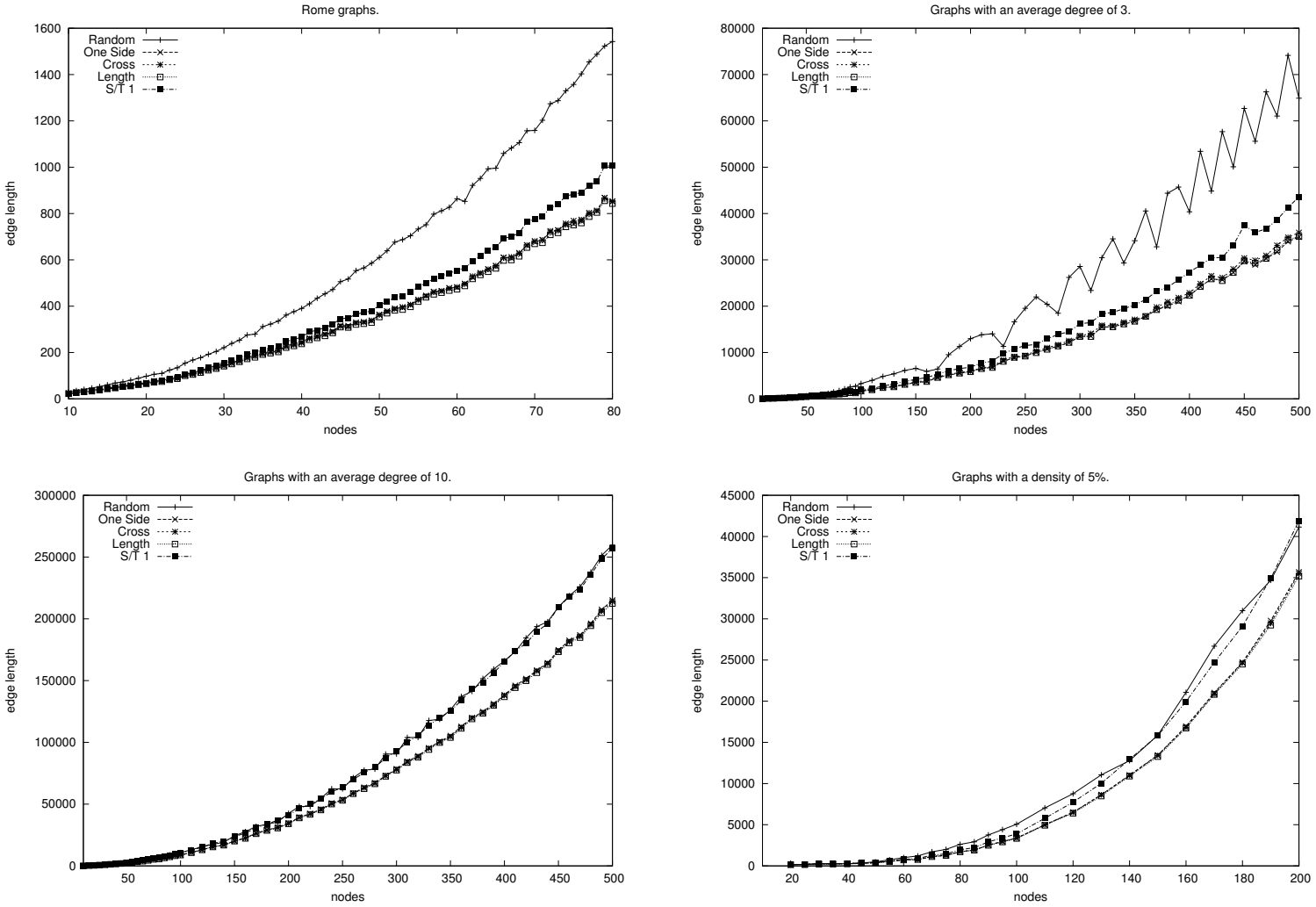


Abbildung 7.4: Kantenlänge für verschiedene Platzierungsstrategien.

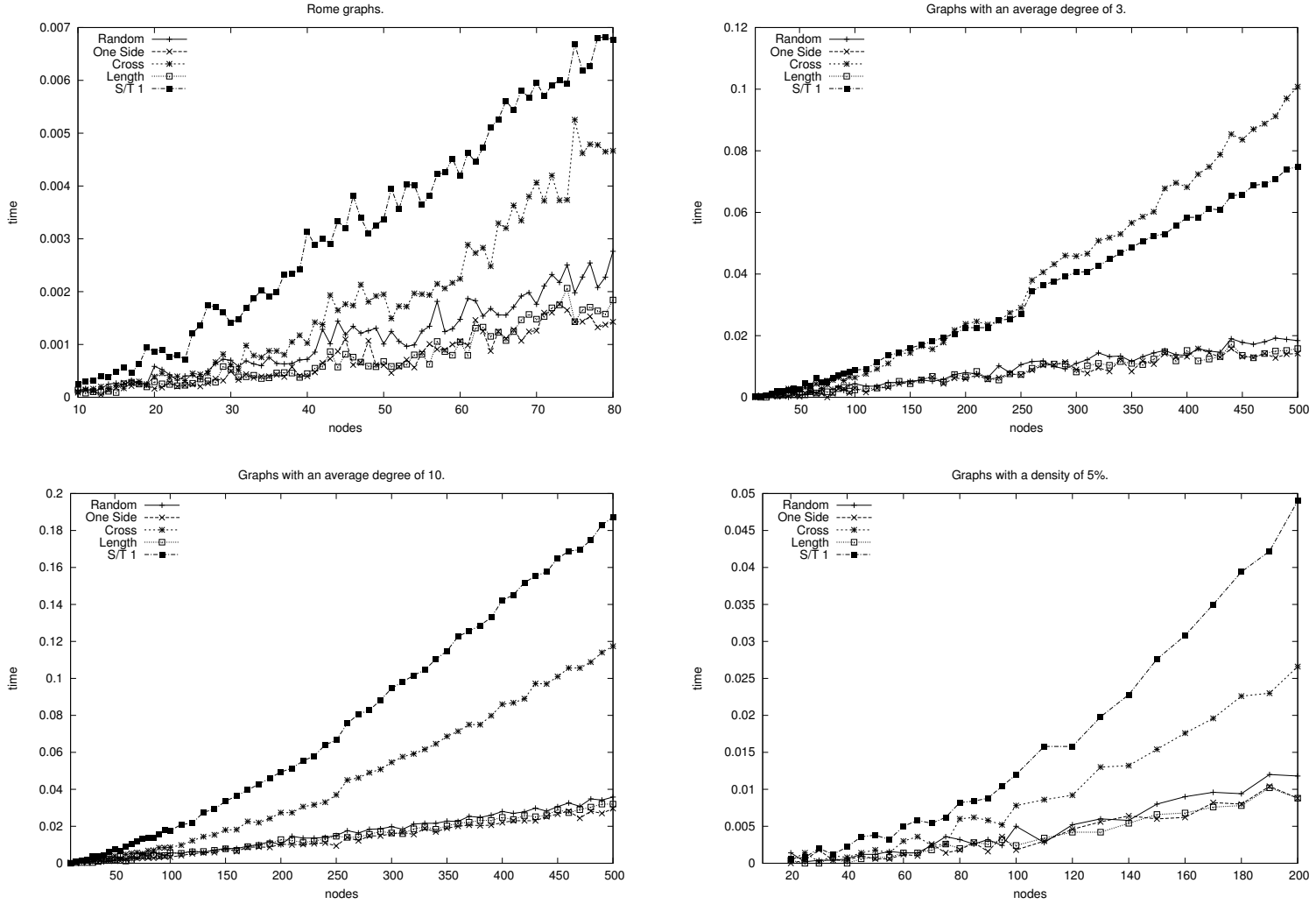


Abbildung 7.5: Laufzeit der verschiedenen Platzierungsstrategien.

Reihenfolge.

Welches Kriterium benutzt wird, um für einen Knoten die Wahl zwischen den beiden Positionen zu treffen, ist mit einer guten Bearbeitungsreihenfolge nicht entscheidend. Schon durch simples Anordnen der Knoten in dieser Reihenfolge wird ein gutes Layout erzielt. Betrachtet man die höhere Laufzeit und den größeren Implementationsaufwand, stellt sich die Frage, ob sich der Algorithmus *Circular Insert* lohnt. Da aber gerade bei den dünnen Graphen, die für die Visualisierung am wichtigsten sind, ein Vorteil erzielt wird, ist es sinnvoll, ihn zu benutzen.

7.3 Vergleich der Heuristiken

Wir haben zwei Arten von Heuristiken kennen gelernt: die einen berechnen ihr Endlayout unabhängig von einem gegebenen Anfangslayout, die anderen verbessern ein gegebenes Startlayout. Wir könnten alle möglichen Kombinationen dieser Algorithmen betrachten. Da dies zu sehr vielen Kombinationen führt, betrachten wir nur die wichtigsten davon.

Im vorherigen Kapitel hat sich gezeigt, dass *Circular Insert* deutlich besser als der Algorithmus von Mäkinen und die anderen Variationen des Basisalgorithmus ist. Deshalb werden nur er und *Phase 1* als Initialisierungsalgorithmus verwendet. Da *Phase 2* eine Einschränkung von *Sifting* ist, betrachten wir ihn nur in der ursprünglichen Kombination zusammen mit *Phase 1*.

Das GLT stand leider nicht zur Verfügung und konnte deshalb nicht verwendet werden. Im ersten Teil des Algorithmus wird, wie in *Phase 1*, ein längster Weg im Graphen gesucht, der die Anordnung ergibt, das Vertauschen im zweiten Teil ist ähnlich wie in *Phase 2* oder *Sifting*, deshalb können diese Algorithmen als Anhaltspunkt für die Ergebnisse dienen. Six und Tollis vergleichen ihre Heuristik mit dem GLT und erzielen mit *Phase 1* bereits um über 20% bessere Ergebnisse [12].

Damit ergeben sich insgesamt sechs untersuchte Verfahren:

- *Phase 1*,
- *Phase 1* und *Phase 2*,
- *Circular Insert*,
- *Sifting*,
- *Phase 1* und *Sifting*,
- *Circular Insert* und *Sifting*.

Mit diesen sechs Algorithmen wurden Layouts für die vier oben genannten Graphenfamilien und zusätzlich für D05 erstellt und die Kreuzungsanzahlen ermittelt.

Zuerst werden sie anhand dieser Ergebnisse verglichen, danach betrachten wir die von den Algorithmen benötigte Laufzeit.

7.3.1 Kreuzungsanzahl

Die Abbildungen 7.6 und 7.7 zeigen die gemittelten Kreuzungsanzahlen für die oben beschriebenen Graphen und Algorithmen.

Von den betrachteten Algorithmen produziert *Phase 1* die schlechtesten Layouts, unabhängig von der Graphenfamilie und der Knotenanzahl. Sie sind aber besser als die des Algorithmus von Mäkinen, wie der vorherige Abschnitt gezeigt hat, und der GLT, wie Six und Tollis in ihrer Arbeit ermittelt haben [12].

Die Layouts von *Circular Insert* sind deutlich besser und fast so gut wie die durch *Phase 2* verbesserte Layouts von *Phase 1*. Der Vorsprung von *Phase 2* nimmt mit zunehmender Dichte der Graphen ab, so dass beide Verfahren auf Graphen aus D10 gleich gut sind.

Die besten Layouts werden von der Kombination von *Sifting* mit *Circular Insert* produziert. An zweiter Stelle folgt die Kombination von *Sifting* mit *Phase 1*. Bei dünnen Graphen ist der Abstand zwischen beiden noch relativ groß, bei dichteren sind sie praktisch gleich gut.

Sifting ohne Initialisierung zeigt ein bemerkenswertes Verhalten. Bei den dünnen Rom-Graphen sind die Layouts schlechter als die von *Circular Insert* produzierten und damit die zweitschlechtesten. Bei dichteren Graphen werden die Layouts im Vergleich mit denen anderer Heuristiken immer besser, bei Graphen aus D05 sind sie bereits so gut wie die der Kombination von *Sifting* mit *Phase 1* und bei D10 schließlich mit die besten.

Interessant ist auch der direkte Vergleich der Kombinationen *Phase 2* nach *Phase 1* und *Sifting* nach *Phase 1*. Bei dünnen Graphen ist der Unterschied zwischen den Ergebnissen nur gering, wird aber mit zunehmender Dichte größer. Das ist erstaunlich, da ein Knoten in dichteren Graphen mehr Nachbarn besitzt und *Phase 2* damit mehr mögliche Positionen testet und eher wie *Sifting* funktioniert. In dünnen Graphen scheinen also öfters die besten Positionen eines Knotens neben einem seiner Nachbarn zu liegen, als in dichten Graphen.

7.3.2 Laufzeit

Die Abbildung 7.8 zeigt die gemittelten Laufzeiten für die Algorithmen. Die beiden Initialisierungsalgorithmen sind so schnell, dass sie am unteren Rand des Schaubilds kaum zu erkennen sind. Ihre Laufzeit ist in Abb. 7.5 besser zu erkennen.

Die Kombination von *Phase 1* und *Phase 2* ist deutlich schneller, als das von der asymptotischen Laufzeit gleiche Sifting. Offenbar lohnt es sich (bezüglich der Laufzeit), die Anzahl der möglicher Positionen eines Knotens einzuschränken.

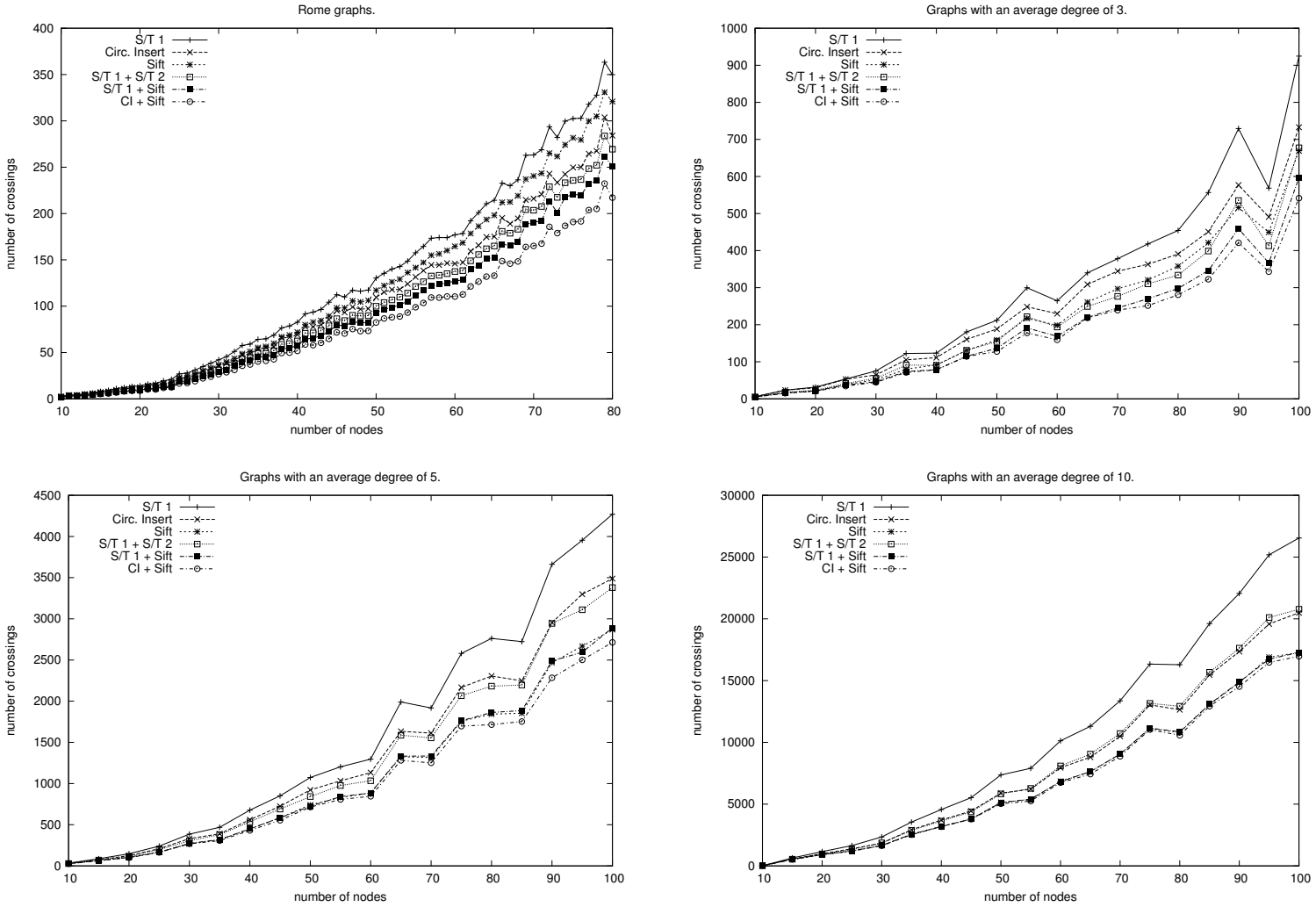


Abbildung 7.6: Kreuzungsanzahl der wichtigsten Algorithmen (Teil 1).

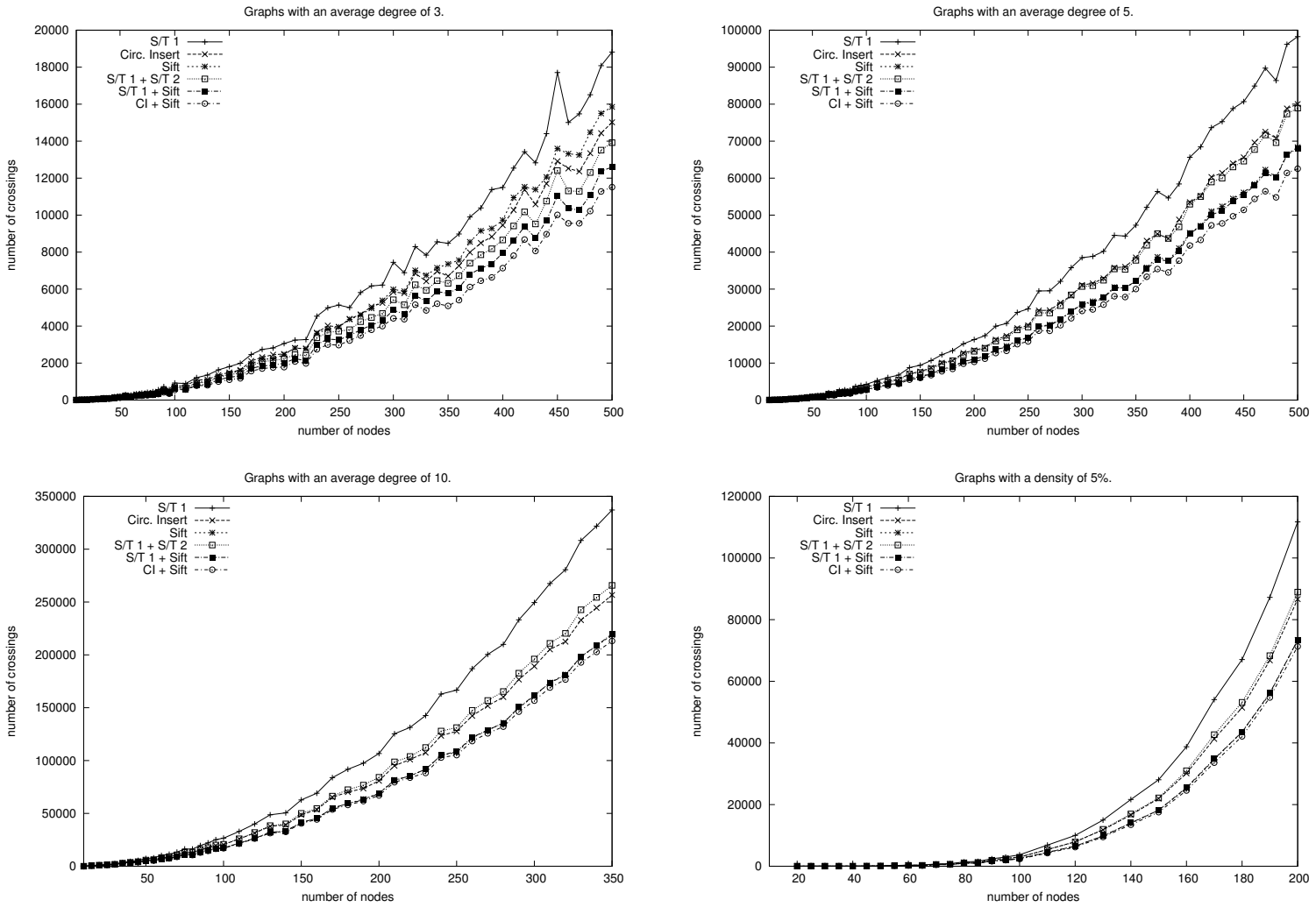


Abbildung 7.7: Kreuzungsanzahl der wichtigsten Algorithmen (Teil 2).

Die Laufzeit der drei Verfahren mit *Sifting* spiegelt direkt die Anzahl der benötigten Runden wieder, da die beiden Initialisierungsalgorithmen praktisch keine Laufzeit benötigen. Bei dünnen Graphen benötigt *Sifting* durch die Initialisierung nur knapp halb so viele Runden wie ohne Initialisierung, um ein Endlayout zu finden. Dieses enthält zusätzlich noch weniger Kreuzungen. Bei dichten Graphen verringert sich die Laufzeit durch eine Initialisierung nicht so stark. Mit *Circular Insert* kann aber teilweise noch eine Verringerung um ein Drittel erreicht werden.

7.3.3 Zusammenfassung

Die Kombination aus *Circular Insert* und *Sifting* erzeugt Layouts mit deutlich weniger Kreuzungen als der beste bisher bekannte Algorithmus *Circular*, der aus der Kombination von *Phase 1* und *Phase 2* besteht.

Die Laufzeit des neuen Verfahren ist deutlich größer, allerdings klein genug, um bei den bei der Visualisierung üblichen Knotenzahlen nicht negativ aufzufallen. Bei größeren und vor allem bei etwas dichteren Graphen kann mit *Circular Insert* fast die Kreuzungsanzahl von *Circular* erreicht werden, und das bei einem Bruchteil der Laufzeit.

7.4 Verhalten von Sifting

In diesem Abschnitt werden die für *Sifting* offen gebliebenen Fragen beantwortet. Zuerst betrachten wir die Auswirkungen verschiedener Startlayouts. Diese waren bereits im letzten Abschnitt zu sehen, werden hier aber der Vollständigkeit wegen wiederholt. Danach untersuchen wir den Einfluss der Bearbeitungsreihenfolge der Knoten und die Abnahme der Kreuzungsanzahl nach jeder Sifting-Runde.

7.4.1 Startlayout

Der Einfluss des Startlayout ist bei dünnen und dichten Graphen sehr unterschiedlich. In den dichten Graphen ist die Kreuzungsanzahl der von *Sifting* erzeugten Layouts immer gleich groß, egal ob das Startlayout mit *Circular Insert* oder *Phase 1* erzeugt wurde oder zufällig gewählt ist. Immerhin verringert sich die Laufzeit durch die Verwendung guter Startlayouts, d.h. es sind weniger Runden nötig, um ein endgültiges Layout zu ermitteln. Wird *Circular Insert* verwendet, verringert sie sich immerhin um ein Drittel.

In dünnen Graphen bietet sich ein ganz anderes Bild. Uninitialisiertes *Sifting* funktioniert hier nur schlecht und produziert Layouts mit vielen Kreuzungen. Mit von *Phase 1* berechneten Startlayouts werden schon viel weniger Kreuzungen erzeugt, mit *Circular Insert* noch etwas weniger. Da davon auszugehen ist, dass das Zufallslayout mehr Kreuzungen als das von *Phase 1* erzeugt Layout enthält, führt ein

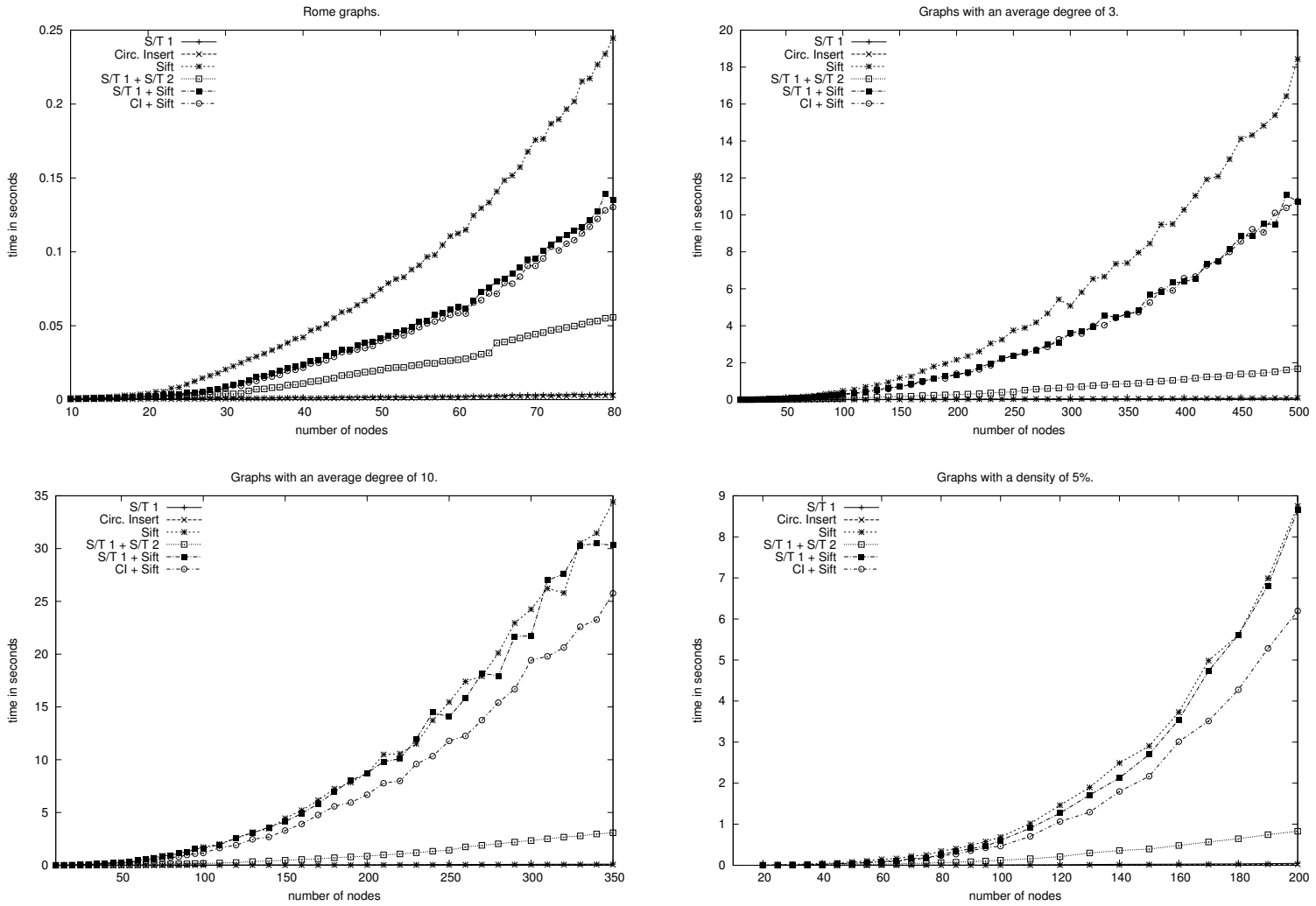


Abbildung 7.8: Laufzeit der wichtigsten Algorithmen.

besseres Startlayout hier direkt zu einem besseren Endlayout. Die Laufzeit verringert sich durch eine gute Initialisierung teilweise bis auf die Hälfte.

7.4.2 Bearbeitungsreihenfolge

Für Sifting in Lagenlayouts werden in [13] die Knoten in der Reihenfolge ihrer Knotengrade bearbeitet. Dabei wird die Reihenfolge nach jeder Runde umgekehrt, die Knoten werden also abwechselnd absteigend und aufsteigend sortiert betrachtet. Dies bringt einen kleinen Vorteil gegenüber der Verwendung einer festen Reihenfolge.

Um zu überprüfen, welchen Einfluss die Bearbeitungsreihenfolge auf *Circular Sifting* hat, wurden die durch die folgenden Reihenfolgen entstehenden Layouts verglichen:

- Random: Eine zufällige Reihenfolge.
- Mindeg: Sortierung nach den Knotengraden mit dem kleinsten zuerst.
- Maxdeg: Sortierung nach den Knotengraden mit dem größten zuerst.
- Reverse: Sortierung nach den Knotengraden, nach jeder Runde wird die Sortierung umgekehrt.
- Maxcross: Für jeden Knoten wird die Anzahl seiner Kreuzungen ermittelt, und die Knoten ansteigend in dieser Reihenfolge gewählt.

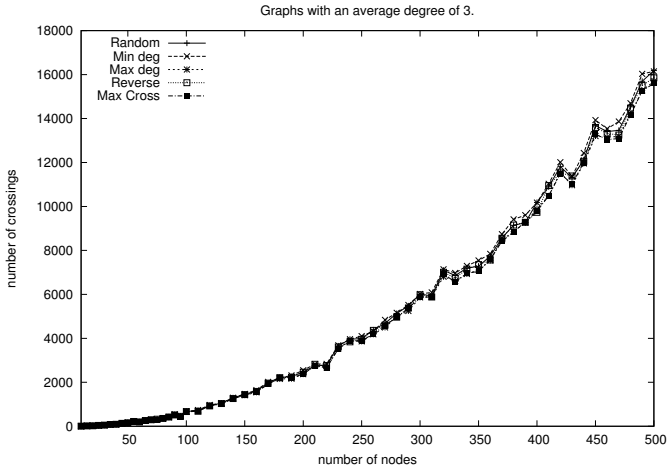
Die Tests wurden für *Sifting* ohne Initialisierung und für die Kombination mit *Circular Insert* durchgeführt. Abb. 7.9 zeigt die Ergebnisse nur für die Graphen aus D03, da alle getesteten Verfahren gleich gut sind. Sogar die zufällige Knotenwahl führt zu den gleich guten Ergebnissen wie die anderen Verfahren. Ob *Sifting* mit oder ohne Initialisierung benutzt wird, macht ebenfalls keinen Unterschied.

7.4.3 Kreuzungsabnahme

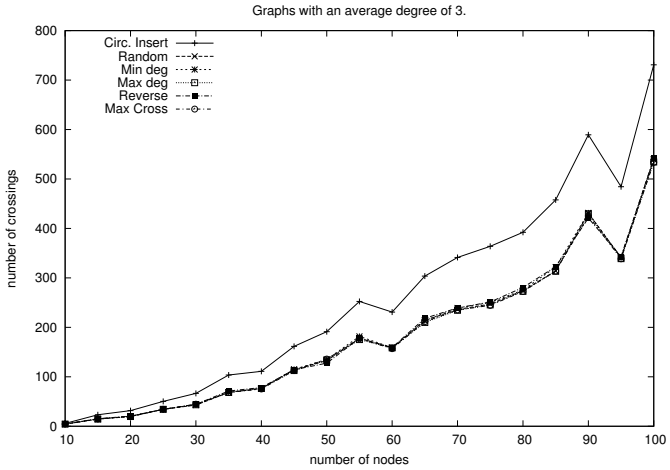
Es wurde an mehreren Stellen behauptet, dass *Sifting* bereits nach wenigen Runden ein Layout erzeugt, das nur noch wenig mehr Kreuzungen enthält, als das Endlayout. In diesem Abschnitt wird untersucht, ob diese Behauptung stimmt.

Dazu wurde der *Sifting*-Algorithmus nach jeder Runde unterbrochen und die Anzahl der Kreuzungen des Layouts berechnet. Abb. 7.9 zeigt die Ergebnisse für die Graphen aus D03 für *Sifting* ohne Initialisierung und in Kombination mit *Circular Insert*. Die Ergebnisse für die Kombination von *Sifting* und *Phase 1* entsprechen denen von *Circular Insert*.

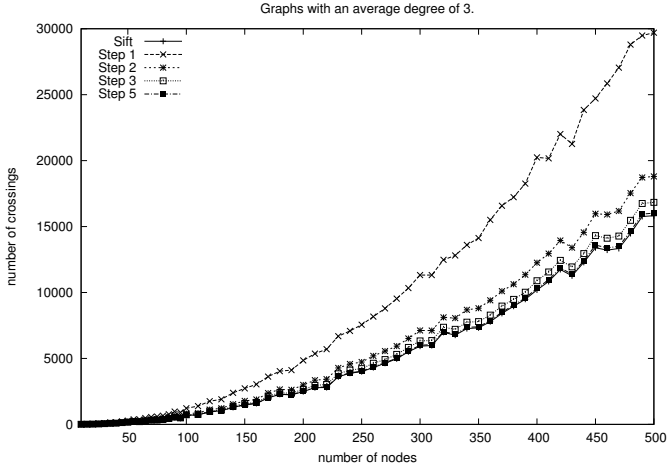
Für nicht-initialisiertes *Sifting* sind die Ergebnisse nach fünf Runden nicht mehr vom Endergebnis zu unterscheiden. Mit guten Startlayouts werden bereits nach zwei Runden nur noch geringe Verbesserungen erreicht.



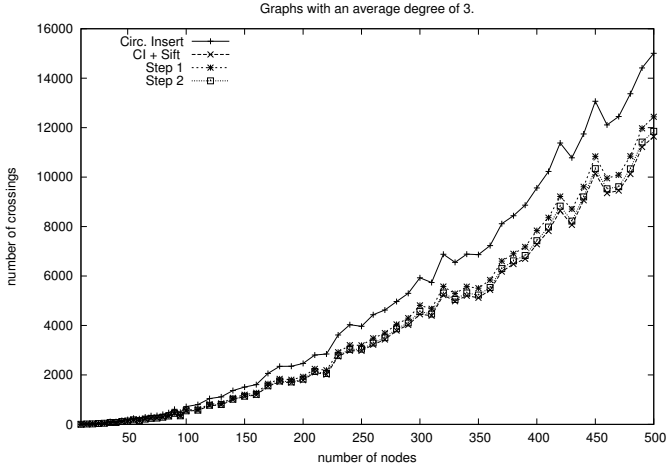
(a) Verschiedene Bearbeitungsreihenfolgen (ohne Initialisierung).



(b) Verschiedene Bearbeitungsreihenfolgen (mit *Circular Insert*).



(c) Sifting-Runden (ohne Initialisierung).



(d) Sifting-Runden (mit *Circular Insert*).

Abbildung 7.9: Verschiedene Varianten von Sifting.

Kapitel 8

Zusammenfassung

Diese Arbeit bietet einen Überblick über die bei der Kreuzungsreduzierung in Kreislayouts auftretenden Probleme. Im ersten Teil wurden die theoretischen Aspekte besprochen, unter anderem die $\mathcal{NP}$-Vollständigkeit der Kreuzungsminimierung. Der zweite Teil beschäftigte sich mit der praktischen Durchführung der Kreuzungsreduzierung.

Die wichtigsten Ergebnisse sind die beiden Heuristiken zur Kreuzungsreduzierung *Circular Insert* und *Sifting*. Werden sie zusammen benutzt, erzeugen sie Kreislayouts mit deutlich weniger Kreuzungen als die beste bisher bekannte Heuristik *Circular* von J. Six und I. Tollis [12]. *Circular Insert* alleine erzeugt Layouts, die fast so gut wie die von *Circular* erzeugten sind, und benötigt dazu nur einen Bruchteil der Laufzeit. Kapitel 7 zeigt einen ausführlichen Vergleich aller bekannter Verfahren mit diesen beiden neuen Algorithmen.

Ein anderer interessanter Algorithmus ist *CountAllCrossings* zur Berechnung der Kreuzungsanzahl. Er ist vor allem für Graphen mit vielen Kreuzungen deutlich schneller als der bisher bekannte Algorithmus.

Ausblick

Beide vorgestellten Algorithmen können möglicherweise noch verbessert werden. Für *Sifting* ist vor allem eine Verringerung der Laufzeit interessant. Dies könnte beispielsweise analog zur zweiten Phase von *Circular* durch ein Einschränkung auf bestimmte Positionen geschehen. Diese Einschränkung sollte allerdings so gut sein, dass sie das Ergebnis nicht so stark wie *Phase 2* verschlechtert.

Ein interessanter Ansatz kombiniert die beiden Verfahren, indem die Knoten durch *Circular Insert* nicht an einem der beiden Enden des Teillayout angefügt werden, sondern durch das Teillayout gesiftet werden und damit die Beste aller Positionen erhalten.

Urhebervermerk

Ich versichere, dass ich die vorliegende Arbeit selbständig angefertigt habe und nur die angegebenen Hilfsmittel und Quellen verwendet wurden.

Konstanz, im Juli 2003

Michael Baur

Literaturverzeichnis

[1] Erkki Mäkinen. On circular layouts.
International Journal of Computer Mathematics, volume 24 (1988), pp. 29–37.

[2] Joseph Y-T. Leung, Oliver Vornberger and James D. Witthoff. On some variants of the bandwidth minimization problem.
SIAM Journal of Comput., volume 13 no. 3 (August 1984), pp. 650–666.

[3] C. H. Papadimitriou. The NP-completeness of the Bandwidth Minimisation Problem.
Computing, Volume 16 (1976), pp. 177–192.

[4] B. Monien and I. H. Sudborough. Graph layout problems.
Surveys in Computer Science, 1984, pp. 145–192.

[5] M. R. Garey and D. S. Johnson. Computers and Intractability.
W. H. Freeman, San Francisco, 1979.

[6] M. R. Garey, D. S. Johnson and L. Stockmeyer. Some simplified NP-complete graph problems.
Theoretical Computer Science, volume 1 (1976), pp. 237–267.

[7] F. Harary. Graph theory.
Addison-Wesley (1969), Reading, MA, USA.

[8] J. Hopcroft and R. Tarjan. Efficient planarity testing.
Journal ACM 21 (4), 1974, pp. 549–568.

[9] Sandra L. Mitchell. Linear algorithms to recognize outerplanar and maximal outerplanar graphs.
Information Processing Letters, volume 9 no. 5 (1979), pp. 229–232.

[10] Sumio Masuda, Toshinobu Kashiwabara, Kazuo Nakajima and Toshio Fujisawa. On the NP-completeness of a computer network layout problem.
1987.

[11] Sumio Masuda, Toshinobu Kashiwabara, Kazuo Nakajima and Toshio Fujisawa.

An NP-hard crossing minimization problem for computer netword layout. *Technical report TR-86-80*, Systems research center, University of Maryland, College Park, MD, USA (1986).

[12] Janet M. Six and Ioannis G. Tollis. Circular drawings of biconnected graphs. *Proceedings of the 1st Workshop on Algorithm Engineering and Experimentation (ALENEX '99)*, volume 1619 of *Lecture Notes in Computer Science*, pp. 57–73. Springer, 1999.

[13] Christian Matuszewski, Robby Schönfeld and Paul Molitor. Using sifting for k-layer straightline crossing minimization.
Proceedings of the 7th International Symposium on Graph Drawing (GD '99), volume 1731 of *Lecture Notes in Computer Science*, pp. 217–224. Springer, 1999.

[14] Wolfgang Günther, Christian Matuszewski, Bernd Becker and Paul Molitor. k-layer straightline crossing minimization by speeding up sifting.

[15] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. *Proceedings of the IEEE International Conference on Computer Aided Design (ICCAD '93)*, pp. 42–47, 1993.

[16] U. Dogrusöz, B. Madden and P. Madden. Circular layout in the Graph Layout Toolkit.
Proceedings of the 4th International Symposium on Graph Drawing (GD '96), volume 1190 of *Lecture Notes in Computer Science*, pp. 92–100. Springer, 1996.

[17] Giuseppe Di Battista. Test suite undirected 1.
http://www.inf.uniroma3.it/people/gdb/wp12/LOG.html

[18] K. Mehlhorn and S. Näher. The LEDA Platform of Combinatorial and Geometric Computing.
Cambridge University Press, 1999.