Layouts ungerichteter Graphen

Diplomarbeit von Andreas Leiser

Betreuer:
Prof. Dr. F.Locher
Lehrgebiet Numerik
Mathematisches Institut
FernUniversität Hagen
D-58084 Hagen

Andreas Leiser
Rennweg 43
CH-2504 Biel/Bienne
andreas.leiser@fernuni-hagen.de

25. Juni 2007

Vorwort und Danksagung [1]

Die vorliegende Diplomarbeit habe ich im Frühjahr 2007 zum Abschluss meines Mathematik II Diplomstudiengangs an der FernUniversität Hagen erstellt. Mit dem Themenbereich kam ich erstmals anlässlich eines mathematischen Praktikums am Lehrgebiet Numerik in Kontakt. Den Vorschlag von Prof. Dr. F. Locher das Thema zu einer Diplomarbeit auszudehnen, nahm ich gerne an, da ich mich damals schon dafür interessierte, an seinem Lehrgebiet die Arbeit zu schreiben.

Mein Dank richtet sich also insbesondere an Prof. Dr. F. Locher für das interessante Thema und die Betreuung der Diplomarbeit. Ebenfalls möchte ich mich bei Dr. Jens Schubert bedanken, welcher damals das mathematische Praktikum mitgeleitet hatte. Die Diplomarbeit basiert auf einer Publikation von Harel und Koren ([HK02]). Deshalb danke ich auch ihnen, insbesondere Dr. Yehuda Koren, welcher mir bei zwei Fragen bereitswillig Klärung verschaffte.

Ein wichtiger Dank geht auch an meine Frau Rébecca, für die grosse Unterstützung während des ganzen Studiums und für ihr Verständnis.

Andreas Leiser
Biel, Juni 2007

[1] Ich werde mich in der ganzen Diplomarbeit an die Schweizer Angewohnheit halten, das „scharfe s“ (ß) vom doppelten s nicht zu unterscheiden und in beiden Fällen „ss“ zu schreiben.

Inhaltsverzeichnis

Kapitel 1

Problemstellung und Lösungsansätze

1.1 Layouts von Graphen

Um einen ungerichteten Graphen $G(V, E)$ computergestützt zu layouten, muss eine Abbildung

$$L : V \to \mathbb{R}^2 \tag{1.1}$$

gefunden werden, die einem zugrundegelegten Kriterium für ein schönes Layout genügt.

Eine häufig angewandte Methode besteht darin, dass das Layout $L(V)$ eines Graphen mit einem bestimmten Energiemodell verknüpft wird und dann als schön erachtet wird, wenn die Energie minimal ist. Diesem Ansatz folgen auch die hier betrachteten Methoden.

Da für unzusammenhängende Graphen das Problem für jede Zusammenhangskomponente getrennt betrachtet werden kann, setzen wir im Folgenden voraus, dass der Graph zusammenhängend ist. Im weiteren betrachten wir auch nur Graphen mit geradlinigen Kanten, so dass das Layout der Kanten nicht behandelt werden muss. Die Kanten werden einzig bei der Definition eines schönen Layouts bzw. dessen algorithmischer Berechnung (mittels des Energieminimierungsverfahrens) eine Rolle spielen.

Ein Problem, welches sich generell bei der Erzeugung von Graphlayouts stellt, ist die der Effizienz des Verfahrens, d.h., seine Laufzeit und sein Speicherbedarf. In der Praxis können unter Umständen Graphen auftreten, die eine sehr grosse Anzahl Knoten besitzen (mehrere Tausend), so dass vor allem die Laufzeit des Algorithmus von zentraler Bedeutung ist. In dieser Diplomarbeit wollen wir deshalb mit der „Schnellen Multi-Skalen Methode“ von Harel und Koren insbesondere ein Verfahren untersuchen, welches auf grosse Graphen ausgerichtet ist ([HK02]).

1.2 Federmodell eines Graphlayouts

Zur computergestützten Generierung eines Layouts haben Kamada und Kawai im Jahre 1989 folgendes, aus der Physik entlehntes, Modell vorgeschlagen ([KK89]):

Ein Graph wird aufgefasst als ein Federsystem, welches sich gemäss dem Hookeschen Gesetz (Lineares Kraftgesetz) verhält:

$$F = -c\Delta l = -c(l - l_0) \tag{1.2}$$

Hierin ist F die Rückstellkraft, Δl die Auslenkung, l_0 die Ruhelänge, l die aktuelle Länge und c die Federkonstante. Die Steifheit der Feder, d.h., die Federkonstante c ist invers proportional zum Quadrat seiner Ruhelänge l_0:

$$c = p \cdot \frac{1}{l_0^2} \tag{1.3}$$

Die Kanten des Graphen werden nun als Federn interpretiert, die jeweils an ihren Enden fest miteinander verbunden sein können. Die Enden der Federn entsprechen damit den Knoten im Graphen. Weiter nehmen wir an, dass jede Feder dieselbe Federkonstante besitzt.

In diesem Modell eines Graphlayouts $L : V \to \mathbb{R}^2$ entspricht die aktuelle Länge einer Feder gerade der euklidischen Distanz zweier Knoten u, v im Layout:

$$l := l(u, v) = \|L(u) - L(v)\| \tag{1.4}$$

Weiter nehmen wir an, dass die Ruhelänge l_0 jeder Feder proportional zur graphentheoretischen Distanz d_{uv} zweier, durch eine Kante verbundener Knoten sei (die Proportionalitätskonstante bezeichnen wir mit L):

$$l_0 = Ld_{uv} \tag{1.5}$$

1.3 Energiemodell nach Kamada und Kamai

Wie schon erwähnt, kann ein schönes Layout definiert werden als ein Layout, in welchem die Energie minimal ist. Kamada und Kamai leiten nun ein Energiemodell im Federsystem und damit für das Graphlayout her.

Die Spannungsenergie E_F einer Feder ist gegeben durch

$$E_F = \frac{1}{2}c(\Delta l)^2 \tag{1.6}$$

Da wir eine Feder mit einer Kante zwischen zwei Knoten u, v des Graphen identifiziert haben, gilt für die (Spannungs-) Energie $E(u, v)$ des „Systems“ zweier, durch eine Kante verbundenen Graphenknoten u, v:

$$\begin{aligned} E(u,v) &:= E_F = \frac{1}{2}c(\Delta l)^2 = \frac{1}{2}c(l - l_0)^2 \\ &= \frac{p}{2L}\frac{1}{{d_{uv}}^2}\left(\|L(u) - L(v)\| - Ld_{uv}\right)^2 \end{aligned} \tag{1.7}$$

Hieraus erhalten wir für das System eines Graphen $G(V, E)$ eine

1.3.1 Definition: Gesamtenergie eines Layouts

$$E := \sum_{u,v \in V} E(u, v)$$

$$
\begin{aligned}
&:= \sum_{u,v \in V} \frac{p}{2L} \frac{1}{{d_{uv}}^2} \left(\|L(u) - L(v)\| - L d_{uv} \right)^2 \\
&= \frac{p}{2L} \sum_{u,v \in V} \frac{1}{{d_{uv}}^2} \left(\|L(u) - L(v)\| - L d_{uv} \right)^2 \qquad (1.8)
\end{aligned}
$$

Das Energiemodell (1.8) für unser Graphlayout kann nun weiter vereinfacht werden. Als erstes spielt der konstante Faktor für den Prozess der Energieminimierung keine Rolle und wir lassen ihn weg. Wir bezeichnen die Kantenlänge, d.h., die Proportionalitätskonstante L, jetzt neu mit l um eine konsistente Schreibweise mit [HK02] zu erhalten, deren Methode näher untersucht werden soll. Ebenso benutzen [HK02] meist allgemeiner eine Gewichtskonstante k_{uv}, welche entweder $k_{uv} := \frac{1}{d_{uv}}$ oder $k_{uv} := \frac{1}{{d_{uv}}^2}$ beträgt. Die zu minimierende Energiefunktion für ein Graphlayout wird also zu:

1.3.2 Definition: Zu minimierende Energiefunktion

$$
\begin{aligned}
E &= \sum_{u,v \in V} k_{uv} \left(\|L(u) - L(v)\| - l d_{uv} \right)^2 \\
&= \sum_{u,v \in V} k_{uv} \left(\left((x_u - x_v)^2 + (y_u - y_v)^2 \right)^{\frac{1}{2}} - l d_{uv} \right)^2 \qquad (1.9)
\end{aligned}
$$

Im letzten Ausdruck wurde die euklidische Norm angewandt und wir haben Koordinaten $L(u) := (x_u, y_u)^t$, $L(v) := (x_v, y_v)^t$ für die Position der Graphenknoten im Layout eingeführt.

Zur Energieminimierung wird üblicherweise ein iteratives Gradientenabstiegsverfahren gewählt und man begnügt sich schon mit einem lokalen Minimum von E. In den nächsten beiden Abschnitten werden nun zwei Verbesserungsansätze eingeführt, welche hauptsächlich darauf abzielen, die Laufzeitordnung zu vermindern und damit auch grosse Graphen einer Layout-Berechnung zugänglich zu machen.

1.4 Lokale Verschönerung mittels Nachbarschaften

Ein Verbesserungsansatz besteht darin, dass man - ausgehend von einer zufälligen Anfangsanordung der Knoten im $\mathbb{R}^2$ - anstatt in einem Schritt den ganzen Graphen schön zu layouten, dies zuerst nur in Teilbereichen tut („lokal layoutet“) und dann schrittweise den Bereich vergrössert. Die Energiefunktion E wird dabei in dem Sinne vereinfacht, dass man eine aufsteigende Folge von Radien r wählt und für jeden Knoten $v \in V$ jeweils nur die r-Nachbarschaftsknoten in die Energieberechnung einfliessen lässt:

$$
E_r := \sum_{v \in V} \sum_{u \in N^r(v)} k_{uv} \left(\left((x_u - x_v)^2 + (y_u - y_v)^2 \right)^{\frac{1}{2}} - l d_{uv} \right)^2 \qquad (1.10)
$$

Hier sind zusätzlich

r der Radius, der eine Nachbarschaft zwischen Knoten definiert,

$$
N^r(v) := \{ u \in V | \; 0 < d_{uv} \leq r \}, \qquad (1.11)
$$

die r-Nachbarschaft eines Knotens v, d.h., die Menge seiner benachbarten Knoten

Minimiert man nun die Energie dieser Folge von Energiefunktionen, so heisst das also: Man beginnt zuerst nur lokal schön zu layouten und betrachtet noch nicht die ganzen Verhältnisse. Erst mit steigendem r wird immer stärker der ganze Graph in die Energieminimierung und damit in das Layouten miteinbezogen.

1.5 Multi-Scale-Ansatz

[HH99] und [HK02] beschreiben mit dem Multi-Skalen-Ansatz eine weitere Methode, um vor allem im Hinblick auf sehr grosse Graphen das Laufzeitverhalten für die Layout-Generierung beträchtlich zu verbessern.

Ausgehend von der Idee, dass die Schönheit eines Layouts sich auf verschiedene Skalen beziehen muss, m.a.W., dass ein Layout nur dann auch schön ist, wenn dies sowohl für seine Feinstruktur (lokal) als auch für seine Grobstruktur (global) gilt, entwickelten [HH99] und [HK02] den „Schnellen Multi-Skalen-Algorithmus“.

Die Motivierung für die folgenden Definitionen ist die, dass wir Repräsentationen eines Ausgangsgraphen $G(V, E)$ suchen, die sich auf verschiedene Skalen beziehen. Das Layout soll dann effizient berechnet werden, indem man von einer groben Repräsentation, die nur die globalen Verhältnisse berücksichtigt, startet, diese schön layoutet und dann schrittweise zu verfeinerten Repräsentationen übergeht, die immer mehr auch die lokalen, feinen Strukturen des Graphen berücksichtigen. Eine gröbere Repräsentation erhält man dabei jeweils durch das Zusammenfassen von (graphentheoretisch) „nahen“ Knoten zu einem einzigen Knoten.

Um Missverständnissen vorzubeugen, sei hier erwähnt, dass es vorerst alleine um eine solche Abstrahierung des Ausgangsgraphen geht. Der Multi-Skalen-Ansatz sagt an sich noch nichts darüber aus, wie man das schöne Layout konkret berechnet bzw. was man als Kriterium für ein schönes Layout anwendet. [HK02] benutzen dazu ein übliches Gradientenabstiegsverfahren basierend auf dem Federsystem-Energiemodell des Graphen, wobei sie auch noch die lokale Verschönerung in einer r-Nachbarschaft benutzen.

Im nächsten Abschnitt soll nun vorerst die theoretische Grundlage des Multi-Skalen-Ansatzes vorgestellt werden und dann seine Kombination mit dem Energieminimierungsverfahren.

1.6 Kombination beider Ideen nach Harel und Koren

Als erstes folgt eine Diskussion der grundlegenden Definitionen von Harel und Koren [HK02]. Die Begriffe wurden möglichst naheliegend übersetzt.

1.6.1 Bezeichnung

Das (endgültige, in allen Skalen) *schöne Layout eines Graphen G* wird mit L^*_G oder einfach mit L^* bezeichnet.

1.6.2 Bemerkung

Schön heisst das Layout eines Graphen immer im Bezug auf ein zugrundegelegtes Kriterium. Wir betrachten hier immer das Kriterium eines Minimums in unserem Energiemodell.

1.6.3 Beispiel

Als Beispiel betrachten wir einen sehr einfachen Graphen mit $|V| = 5$ und der Adjazenzmatrix

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Das schöne Layout L^* kann hier unmittelbar gezeichnet werden:

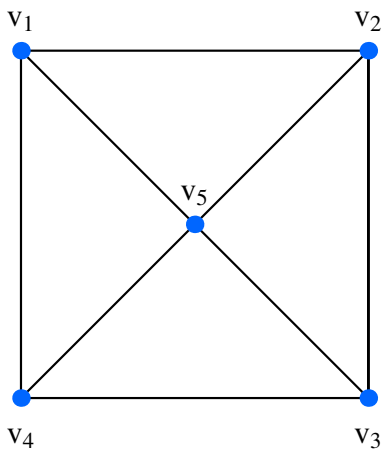


Abbildung 1.1: Das schöne Layout L^* des Beispielsgraphen G.

1.6.4 Definition: Bzgl. r lokal schönes Layout

Ein Layout L eines Graphen $G(V, E)$ heisst *bzgl. r lokal schön*

:⇔ Der Durchschnitt von $L(V)$ mit jedem Kreis vom Radius r erzeugt ein schönes Layout des betreffenden Subgraphen von G

1.6.5 Beispiel

In den Abbildungen 1.2 und 1.3 sehen wir ein lokal schönes Layout und ein lokal nicht schönes Layout.

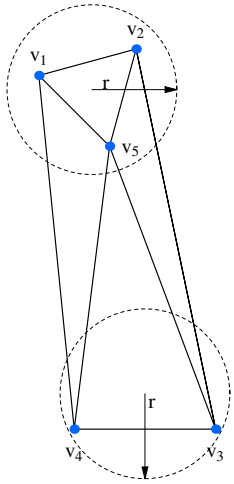


Abbildung 1.2: Ein lokal schönes Layout von G bzgl. r.

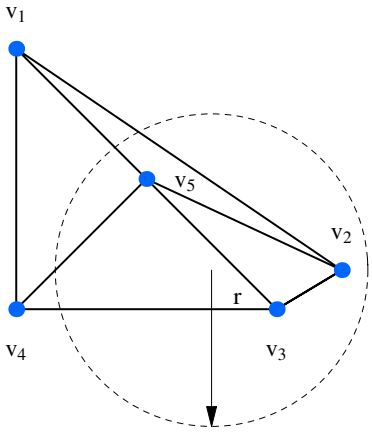


Abbildung 1.3: Ein lokal *nicht* schönes Layout von G bzgl. r.

Man sieht schon an diesem simplen Beispiel, dass der Radius nicht zu klein gewählt werden darf, sonst wird der Fall trivial. Ist der Radius kleiner dem kleinsten Abstand zweier Knoten im Layout, so ist jedes Layout lokal schön, da jeder Kreis nur noch seinen Zentrumsknoten umfasst. Dieser Umstand ist für die Methode von Harel und Koren aber nicht etwa unwichtig, sondern sehr zentral: Radius Null erlaubt uns dort nämlich gerade den Übergang von den nachfolgend definierten Graphabstraktionen zurück zum Ausgangsgraphen.

1.6.6 Definition: Bzgl. r global schönes Layout

Ein Layout L eines Graphen $G(V, E)$ heisst *global schön bzgl. r*

$$:\Leftrightarrow \max_{v \in V}\{|L(v) - L^*(v)|\} < r \tag{1.12}$$

1.6.7 Beispiel

Abb. 1.4 bzw. Abb. 1.5 zeigen ein global schönes bzw. ein global nicht schönes Layout.

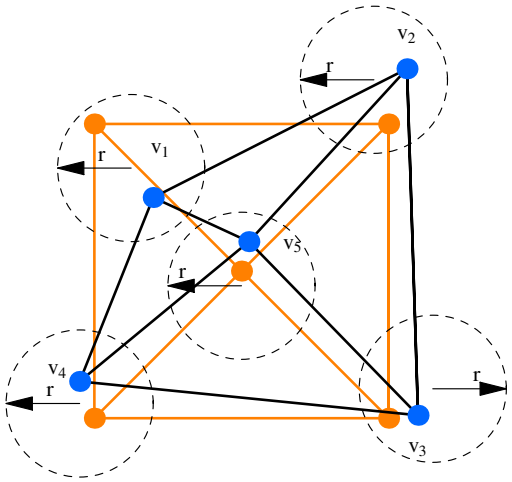


Abbildung 1.4: Ein global schönes Layout von G bzgl. r, unterlegt mit dem schönen Layout L^*.

Global schönes Layout bzgl. dem Radius r heisst also, dass kein Knoten mehr weiter als r von seiner Position im schönen Layout L* entfernt ist.

1.6.8 Bemerkung

Mit folgenden zwei einfachen Überlegungen können wir die beiden Begriffe abgrenzen:

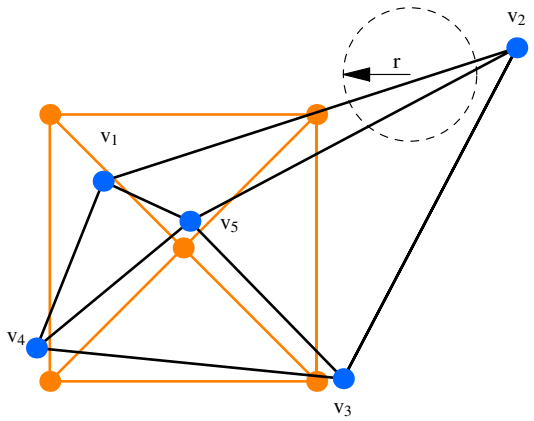


Abbildung 1.5: Ein global *nicht* schönes Layout von G bzgl. r, unterlegt mit dem schönen Layout L^*.

Dass ein global schönes Layout nicht automatisch auch lokal schön zu sein braucht, sehen wir an obigem Beispiel des global schönem Layouts: Wir könnnen dort r natürlich vergrössern und das Layout bleibt global schön. Jedoch werden so für gewisse r z.B. die Knoten v_1, v_2, v_5 in einem Kreis zu liegen kommen. Dieser Teilgraph ist aber offensichtlich nicht schön, also ist für dieses r das Layout nicht lokal schön.

Die Umkehrung gilt schon fast trivialerweise ebenfalls nicht. Am Beispiel des lokal schönen Layouts erkennt man (wie bereits erwähnt), dass die Verkleinerung des Kreisradius' r das Layout lokal schön belässt. Damit wird aber z.B. Knoten v_1 irgendwann weiter als r von seiner Position in L^* entfernt sein und das Layout ist dann sicher nicht mehr global schön.

1.6.9 Definition: Bzgl. r die Lokalität erhaltender k-Cluster (k-lpc)

Sei $G(V, E)$ ein Graph und d_{vu} sei die graphentheoretische Distanz zwischen den Knoten $v, u \in V$.

Ein *bzgl. r die Lokalität erhaltender k-Cluster (kurz: k-lpc) des Graphen* $G(V, E)$ ist definiert als ein gewichteter Graph $G(\{V_1, V_2, ..., V_k\}, E', w)$ mit:

$$V = V_1 \cup V_2 \cup ... \cup V_k \text{ und } \forall i \neq j : V_i \cap V_j = \emptyset \tag{1.13}$$

$$E' = \{(V_i, V_j) \mid \exists (v_i, v_j) \in E \wedge v_i \in V_i \wedge v_j \in V_j\} \tag{1.14}$$

$$w(V_i, V_j) = \frac{1}{|V_i||V_j|} \sum_{v \in V_i, u \in V_j} d_{vu}, \qquad \forall (V_i, V_j) \in E' \tag{1.15}$$

$$\forall i \in \mathbb{N}_k : \max_{v,u \in V_i} \{|L^*(v) - L^*(u)|\} < r \tag{1.16}$$

1.6.10 Bezeichnungen

Die Bezeichnung „k-lpc“ kommt vom englischen Ausdruck *locality preserving k-clustering* und wird hier unübersetzt übernommen.

Die Knoten V_i eines k-lpc's - also Knoten-*Teilmengen* des Ausgangsgraphen - nennen wir auch *Clusters*. Einen k-lpc eines Graphen G bezeichnen wir mit G^k.

1.6.11 Bemerkungen

Gemäss der Definition ist ein k-lpc G^k also ein Graph, dessen Knoten eine V überdeckende Menge disjunkter Teilmengen von Knoten von G sind. G^k hat weiter genau dann eine Kante $(V_i.V_j)$, wenn es je einen Knoten in V_i bzw. V_j gibt, welche *in* G mit einer Kante verbunden sind. Eine solche Kante wird durch eine Gewichtung mit den Distanzen der beteiligten Knoten des Ausgangsgraphen in Beziehung gesetzt.

Die letzte Bedingung (1.16) in der Definition hat gerade die Aufgabe, die Lokalität der Knoten von G im Hinblick auf das schöne Layout L^* im k-lpc G^k zu erhalten.

1.6.12 Beispiel

Um ein aussagekräftiges Beispiel eines k-lpc's bzgl. r zu bekommen, müssen wir einen etwas grösseren Graphen betrachten. Wir wählen einen Kreisgraphen mit $|V| = 16$. Dieser hat dennoch eine einfache Struktur, so dass das Wesentliche gut ersichtlich ist. Selbstverständlich sind aber die Verhältnisse bei (grösseren) Graphen mit komplexerer Struktur dann auch im Hinblick auf den k-lpc nicht mehr derart einfach.

Die Adjazenzmatrix des Kreisgraphen Circle16 hat die einfache Struktur, dass die (erste) obere und die (erste) untere Nebendiagonale sowie $(1, 16)$ und $(16, 1)$ aus Einsen bestehen und alle anderen Stelle sonst Null sind (bzw. die Sequenz $1, 0, 1$ verschiebt sich zyklisch mit steigender Zeilenzahl immer um eine Stelle nach rechts):

$$A_{\text{Circle16}} = \begin{pmatrix} 0 & 1 & 0 & ... & ... & 0 & 1 \\ 1 & 0 & 1 & 0 & ... & ... & 0 \\ 0 & 1 & 0 & 1 & 0 & ... & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & ... & ... & 0 & 1 & 0 & 1 \\ 1 & 0 & ... & ... & 0 & 1 & 0 \end{pmatrix}$$

Abb. 1.6 zeigt einen 4-lpc G^4 bzgl. des eingezeichneten Radius r. Im unterlegten schönen Layout L^* von Circle16 seien die Knoten wie angedeutet im Uhrzeigersinn mit $v_1, ..., v_{16}$ nummeriert. Dann gilt für den eingezeichneten 4-lpc G^4 offensichtlich:

$$V_{G^4} = \{V_1, V_2, V_3, V_4\} = \{\{v_1, ..., v_4\}, \{v_5, ..., v_8\}, \{v_9, ..., v_{12}\}, \{v_{13}, ..., v_{16}\}\}$$
$$E_{G^4} = \{(V_1, V_2), (V_2, V_3), (V_3, V_4), (V_4, V_1)\}$$
$$w(V_1, V_2) = w(V_2, V_3) = w(V_3, V_4) = w(V_4, V_1) = 4 \text{ (jeweils } 1/16 \cdot 64)$$

und die Bedingung der Erhaltung der Lokalität ist erfüllt.

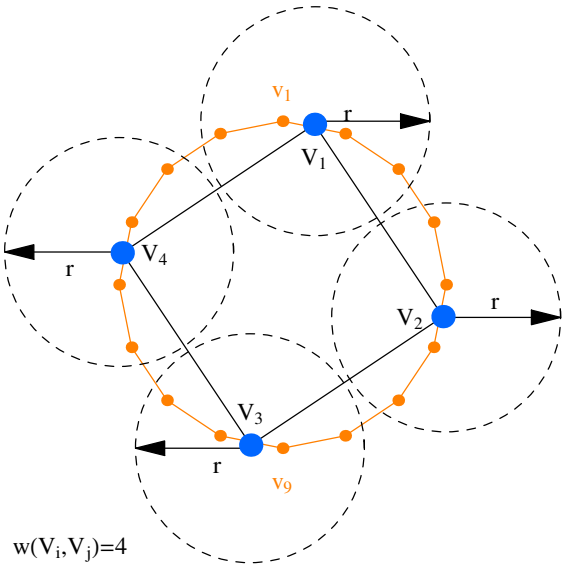


Abbildung 1.6: Das (schöne) Layout des 4-lpc G^4 bzgl. r des 16-knotigen Kreisgraphen, unterlegt mit dessem schönen Layout L^*.

1.6.13 Definition: Multi-Skalen-Repräsentation eines Graphen

Eine *Multi-Skalen-Repräsentation eines Graphen* $G(V, E)$ ist definiert als eine Folge von Graphen $G^{k_i}, i \in \mathbb{N}_l$ mit

$$k_i < k_{i+1}, \forall i \in \mathbb{N}_{l-1} \tag{1.17}$$

$$k_l = |V| \tag{1.18}$$

$$\forall i \in \mathbb{N}_l : G^{k_i} \text{ ist ein k-lpc von } G(V, E) \text{ bzgl. } r_i \tag{1.19}$$

$$\text{und } r_1 > r_2 > ... > r_l = 0 \tag{1.20}$$

1.6.14 Beispiel

Die Abbildung 1.7 zeigt für den Kreisgraphen $G_{Circle16}$ die k-lpc's G^k einer Multiskalen-Repräsentation mit $k \in \{2, 4, 8, 16\}$. Zu beachten ist, dass es bei den gezeichneten Radien mit Ausnahme des 2-lpc's noch nicht um solche handelt, die für die Energieminimierung geeignet sind! Dort müssen sie grösser sein, so dass sie auch mehrere Clusters umfassen. Dafür verzichtet man dann darauf, dass die Clusters disjunkte Mengen von Knoten sind. Dieser Umstand wurde hier der Übersichtlichkeit halber und weil dies eine - von Harel und Koren nicht erwähnte - Änderung der definierenden Bedingungen ist, noch nicht berücksichtigt.

Folgende Definition ist bei [HK02] nicht explizit erwähnt. Es scheint mir aber sinnvoll zu sein, sie hier anzuführen:

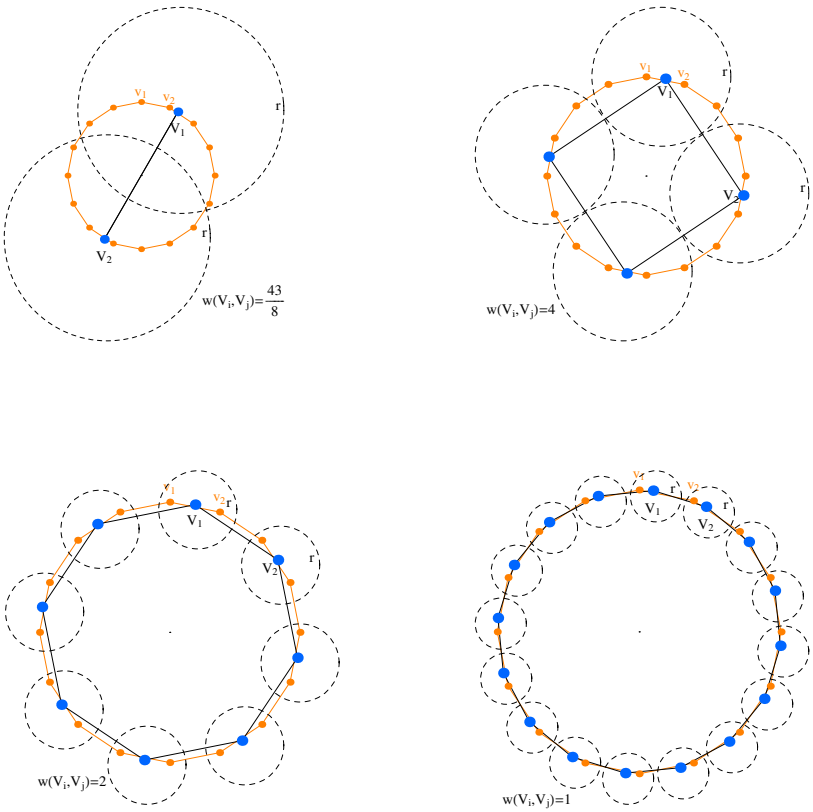


Abbildung 1.7: Eine Multiskalen-Repräsentation (G^2, G^4, G^8, G^{16}) des 16-knotigen Kreisgraphen $G_{Circle16}$, jeweils wieder unterlegt mit dessem schönen Layout L^*. (Radien noch nicht für die Energieminimierung angepasst.)

1.6.15 Definition: G_r

Ist die Folge $(G^{k_i})_{i=1,\ldots,l}$ eine Multiskalen-Repräsentation eines Graphen G zur zugehörigen Folge von Radien $(r_i)_{i=1,\ldots,l}$, so bezeichnen wir für jedes $k := k_i$ und zugehöriges $r := r_i$ mit G_r einen $k := k_i$-lpc bzgl. $r := r_i$, wenn wir den Radius in den Vordergrund stellen wollen. (Also $G_{r_i} := G^{k_i}$)

1.6.16 Grundannahme 1 von Harel und Koren

Sei G_r ein k-lpc eines Graphen G bzgl. r und sei $\hat{r} \geq r$. Setzen wir in G $\forall v \in V_i$: $L(v) := L(V_i)$, so gilt:

L ist ein global schönes Layout von G_r bzgl. $\hat{r}$

$\Leftrightarrow$ L ist ein global schönes Layout von G bzgl. $\hat{r}$

1.6.17 Korollar

L ist ein schönes Layout eines k-lpc's G^k von G bzgl. r

$\Rightarrow$ L ist ein global schönes Layout von G bzgl. r

1.6.18 Bemerkungen

Diese Grundannahme besagt also folgendes: Für jeden Radius grösser als r entspricht dem global schönen Layout des k-lpc G_r gerade das global schöne Layout des Ausgangsgraphen G. Insbesondere ändert die gewählte Positionierung also nichts an der Eigenschaft, ein global schönes Layout zu sein.

Harel und Koren formulieren diesen Sachverhalt prägnant auch so, dass die globale Ästhetik eines Graphen unabhängig von seiner Feinstruktur ist. Und dass deshalb der Unterschied zwischen den Layouts von G_r und von G durch r beschränkt sind.

1.6.19 Grundannahme 2 von Harel und Koren

L ist sowohl ein lokal als auch ein global schönes Layout eines Graphen G bzgl. r

$\Rightarrow L$ ist ein schönes Layout (d.h., $L = L^*$)

1.6.20 Bemerkungen

Eng mit dieser Grundannahme verknüpft, ist die Grundidee der Multiskalen-Methode von Harel und Koren: Wir versuchen (iterativ) ein sowohl lokal, wie auch global schönes Layout von G bzgl. r zu finden. Wenn wir das nun für eine abnehmende Folge von Radien r tun, so werden diese im letzten Schritt ($r_l = 0$) nur noch die einzelnen Knoten des Ursprungsgraphen G umfassen. Also ist dann - aufgrund von Annahme 2 - das lokale und das globale Layout von G_{r_l} das schöne Layout L^* von G_{r_l} und damit - wegen $r_l = 0$ - das schöne Layout L^* von G.

Damit haben wir schliesslich das gesuchte schöne Layout des Ursprungsgraphen G gefunden.

1.6.21 Harel und Koren's Schema der Multi-Skalen-Methode

Harel und Koren nennen in ihrer Publikation [HK02] folgendes Schema:

1. Positioniere die Knoten von G zufällig in der Zeichenfläche.
2. Wähle eine passende abnehmende Folge von Radien $\infty = r_0 > r_1 > ... > r_l = 0$.
3. for $i = 1$ to l do // d.h., für jeden Radius
   - 3.1 Wähle einen geeigneten Wert für k_i und konstruiere G^{k_i}, d.h., einen k_i-lpc von G bzgl. r_i.
   - 3.2 Positioniere jeden Knoten von G^{k_i} an die (gewichtete) Stelle der Knoten von G, welche ihn definieren.
   - 3.3 Verschönere lokal die lokalen Nachbarschaften von G^{k_i}.
   - 3.4 Positioniere jeden Knoten von G an die Stelle seines Clusters.
4. end.

1.6.22 Bemerkungen

Das Schema muss sowohl von der Methode her, wie auch implementierungstechnisch in einigen Punkten näher spezifiziert werden. Diese Aspekte werden bei der Implementierung und bei der Analyse diskutiert. Zudem vereinfachen Harel und Koren später ihr Schema an mehreren Stellen.

1.7 Korrektheit des Schemas

Abschliessend wollen wir noch die Frage der Korrektheit des Schemas betrachten, orientiert an der Argumentation von Harel und Koren in [HK02].

Sei das Layout eines Graphen G also im ersten Schritt zufällig im $\mathbb{R}^2$ positioniert und in einem zweiten Schritt eine abnehmende Folge von Radien $\infty = r_0 > r_1 > r_2 > ... > r_l = 0$ gegeben.

Im ersten Durchgang der for-Schleife ($i := 1$) wird der k_1-lpc G^{k_1} von G bzgl. dem Radius r_1 konstruiert und die r_1-Nachbarschaften von G^{k_1} lokal verschönert. r_1 muss dabei so im Zusammenspiel mit k_1 gewählt werden, dass die Clusters auch alle Knoten in einer r-Nachbarschaft einschliessen und dass der k_1-lpc auf einfache Weise schön gezeichnet werden kann (mit unserem noch zu behandelnden Energieminimierungsverfahren).
Wenn wir so nach Schritt 3.3 ein lokal schönes Layout von G^{k_1} bzgl. r_0 erhalten haben, dann heisst das aufgrund von Annahme 1.6.16, dass wir nach dem Positionieren gemäss 3.4 für G^{k_1} ein global schönes Layout bzgl. dem Radius r_0 haben. Das Layout von G^{k_1} ist also lokal und global schön, d.h., mit 1.6.19 ist das Layout von G^{k_1} schön.

Im i-ten Durchlauf starten wir also die for-Schleife mit letztgenanntem Layout für $i-1$. Insbesondere haben wir ein global schönes Layout von G^{k_i} bzgl. r_{i-1} (wir verkleinern ja den Radius). Analog zum Durchlauf $i := 1$ haben wir nach dem Schritt 3.4 ein schönes Layout, nun von G^{k_i}. Mit Grundannahme 1.6.16 ist dies erneut ein global schönes Layout von $G_{k_{i+1}}$ bzgl. dem Radius r_i und wir können mit derselben Vorbedingung den Schleifendurchlauf $i+1$ beginnnen.

Wir sehen also, dass wir mit diesem Schema eine Folge von schönen Layouts der k-lpc's G^k generieren. Haben wir nun im letzten Durchlauf $k_l := |V|$ des Ausgangsgraphen $G(V, E)$ und $r_l := 0$ (oder zumindest kleiner dem kleinsten Abstand zweier Knoten im schönen Layout L^*_G) und ein schönes Layout von $G^k := G^{|V|}$, so haben wir unser Ziel, ein schönes Layout von G erreicht, denn wir können ja $G^{|V|}$ mit G identifizieren.

1.7.1 Bemerkung

Das Schema bzw. die eben geführte Argumentation zeigt, dass es im Wesentlichen nicht darauf ankommt, ob man die Multiskalen-Repräsentation anhand der abnehmenden Folge von Radien oder anhand der steigenden Folge der k's definiert. Wichtig ist einzig, dass dabei Radien und k's aufeinander abgestimmt sind (z.B. jeweils alle Knoten des Ausgangsgraphen durch Clusters berücksichtigt werden) und - was Harel und Koren in ihrem Algorithmus (siehe Abschnitt 3.1) nicht explizit formuliert haben - im letzten Durchgang tatsächlich $G^{|V|}$ berechnet wird.

1.8 Vereinfachungen des Schemas

Im Hinblick auf die Implementierung des Schemas haben Harel und Koren folgende Vereinfachungen gemacht:

1. Die Knoten V_i eines k-lpc's müssen nicht disjunkt sein (jeder Knoten von G ist aber mind. in einem Cluster enthalten).

2. Die k-lpc's G^k werden nicht vollständig berechnet, sondern statt die Position eines Clusters im Layout von G^k anhand all seiner Knoten zu berechnen, wird einer der Knoten (ein „centers“-Knoten), der Lösung des k-Center-Problems ist (vgl. Kap. 3), mit dem Cluster identifiziert. Deshalb müssen keine separaten Layout-Strukturen für die G^k's verwaltet werden.

3. Die G^k werden auch in der Weise vereinfacht, dass keine Kantengewichtung berechnet wird. Damit ist auch keine separate APSP-Matrix für einen G^k notwendig. Die Distanz zwischen zwei Clusters ist einfach gegeben durch die Distanz zwischen den mit ihnen identifizierten Knoten im Ausgangsgraphen G, wird also durch dessen APSP-Matrix mitverwaltet.

Diese Vereinfachungen wurden zur Untersuchung auch in unsere Implementierung übernommen. Wir werden sehen, dass sie nicht unproblematisch sind.

Kapitel 2

Berechnung der Energieminimierung

Wie wir im Schema 1.6.21 der Multiskalen-Methode gesehen haben, verschönern wir dort im Schritt 3.3 das Layout lokal bzgl. dem jeweiligen Radius r und dem jeweiligen G^k. Das ist also der Punkt, in dem wir unser Energiemodell eines Graphenlayouts zu Hilfe nehmen und das Layout mit minimaler Energie berechnen müssen. In diesem Kapitel soll die von [HK02] benutzte Methode vorgestellt und ihren Algorithmus „LocalLayout“ besprochen werden.

2.1 Bedingungen für lokales Minimum der Energiefunktion

Ein entscheidender Punkt bei der Methode von Harel und Koren ist, dass sie *zwei*, an und für sich getrennte Verbesserungsansätze *kombinieren*: Der Multiskalen-Ansatz (s.a. Abschnitt 1.5) und die lokale Verschönerung mittels Nachbarschaften (s.a. Abschnitt 1.4). Bei der Berechnung der Verschönerung eines Layouts kommt nun der zweite Ansatz zum Tragen. Anders als in der Publikation von Harel und Koren ([HK02]), werde ich Radien von Nachbarschaften auch weiterhin mit r bezeichnen und den Buchstaben k nur für die Graphabstraktionen G^k (d.h., die $k-lpc$) verwenden. Immer ist dabei natürlich in Erinnerung zu behalten, dass für eine Multiskalen-Repräsentation eines Graphen k und r im Allgemeinen voneinander abhängen.

Nach 1.4 betrachten wir also einen Graphen $G(V,E)$, Layouts $L : V \to \mathbb{R}^2$ mit $(x_v, y_v) := L(v),\ \forall v \in V$ dieses Graphen, und die Energiefunktion bei r-Nachbarschaften ((1.10) und (1.11)). Die Funktion E_r fassen wir nun als eine Funktion in den Koordinaten ihrer Graphenknoten auf, d.h.,

$$\begin{aligned} E_r : \mathbb{R}^{|V|} \times \mathbb{R}^{|V|} &\to \mathbb{R} \\ (\mathfrak{v},\mathfrak{y}) := \left(\begin{pmatrix} x_1 \\ \vdots \\ x_{|V|} \end{pmatrix}, \begin{pmatrix} y_1 \\ \vdots \\ y_{|V|} \end{pmatrix} \right) &\mapsto E_r\left((\mathfrak{v},\mathfrak{y})\right) \end{aligned} \tag{2.1}$$

Dies ist also eine reellwertige Funktion auf dem $\mathbb{R}^{2|V|}$. Ein lokales Minimum zu finden heisst notwendig, dass

$$\forall v \in V : \qquad \frac{\partial E_r}{\partial x_v} = \frac{\partial E_r}{\partial y_v} = 0 \tag{2.2}$$

gelten muss. (Wie bereits früher erwähnt, begnügen wir uns damit ein lokales Energieminimum zu finden.)

Um ein Layout zu finden, welches dieser notwendigen Bedingung für ein lokales Minimum genügt, benutzen wir ein übliches Gradientenabstiegsverfahren:

2.2 Das Gradientenabstiegsverfahren

Die Idee beim Gradientenabstiegsverfahren ist bekanntlich die, irgendwo auf dem „Energiegebirge“ $E_r(\mathfrak{x}, \mathfrak{y})$ über dem $\mathbb{R}^{2|V|}$ zu starten und iterativ jeweils in Richtung des grössten Gradienten abzusteigen, um schliesslich in einem lokalen Minimum zu landen. Da bei uns die Energie vom Layout und damit von der Position der Knoten abhängt, wollen wir den Gradientenabstieg jeweils durch die Verschiebung einzelner Knoten durchführen. Mit anderen Worten, wir führen iterativ folgende Schritte aus:

1. Als erstes suchen wir den Knoten $v \in V$ mit dem betragsmässig grössten Gradienten $\nabla E_r := \left(\frac{\partial E_r}{\partial x_v}, \frac{\partial E_r}{\partial y_v}\right)^t$, d.h., wir suchen $v' \in V$ mit

$$\Delta^r_{v'} = \max_{v \in V}\{\Delta^r_v\} := \max_{v \in V}\{\sqrt{\left(\frac{\partial E_r}{\partial x_v}\right)^2 + \left(\frac{\partial E_r}{\partial y_v}\right)^2}\} \tag{2.3}$$

2. Im zweiten Schritt stellt sich nun die Frage, wohin wir v' im Layout verschieben sollen, so dass die Energie E_r minimiert wird. Betrachten wir E_r für einen Augenblick als Funktion von $(x_{v'}, y_{v'})$ alleine: Dann sehen wir, dass wir E_r bzgl. v' genau dann minimieren, wenn wir diesen Knoten im Layout so verschieben, dass die Energie lokal minimiert wird. Das heisst aber gerade, dass wir die Nullstelle der Ableitung Funktion $E_r(v')$ suchen. Numerisch lässt sich diese Nullstelle mit dem Newton-Raphson-Verfahren in zwei Dimensionen lösen. Man erhält so (vgl. auch anschliessende Bemerkung) für die Verschiebung $\delta^r_{v'} := (\delta^r_{v'}(x), \delta^r_{v'}(y))$ des Knotens v' das Lineare Gleichungssystem (LGS):

$$\begin{aligned} \frac{\partial^2 E_r}{\partial x^2_{v'}}\delta^r_{v'}(x) + \frac{\partial^2 E_r}{\partial x_{v'}\partial y_{v'}}\delta^r_{v'}(y) &= -\frac{\partial E_r}{\partial x_{v'}} \\ \frac{\partial^2 E_r}{\partial y_{v'}\partial x_{v'}}\delta^r_{v'}(x) + \frac{\partial^2 E_r}{\partial y^2_{v'}}\delta^r_{v'}(y) &= -\frac{\partial E_r}{\partial y_{v'}} \end{aligned} \tag{2.4}$$

2.2.1 Bemerkung

Wiederholen wir nun diese zwei Schritte, so werden wir mit unserem Verfahren zu einem Layout gelangen, für welches die Energiefunktion ein lokales Minimum hat. Die Anzahl Iterationen sollte dabei so gewählt werden, dass sicher jeder Knoten berücksichtigt wird. Da sich mit jedem Verschieben eines Knotens im Layout die Funktion E_r wieder ändert, ist es sinnvoll, mehrmals die Knotenanzahl zu iterieren, d.h. man wählt als Schleifengrenze

$$iterations \cdot |V| \tag{2.5}$$

mit einer natürlichen Zahl *iterations* (s.a. Abschnitt 2.3).

2.2.2 Bemerkung

Das LGS (2.4) erhält man folgendermassen aufgrund der Newton-Raphson-Methode:

Bekanntlich besteht das Newton-Raphson-Verfahren zur iterativen Bestimmung einer (existierenden) Lösung $\xi \in [a, b]$ einer Gleichung

$$f(x) = 0, \qquad f \in C_1 [a, b] \tag{2.6}$$

daraus, iterativ

$$x_{n+1} := x_n - \left(f' (x_n)\right)^{-1} f (x^n) \tag{2.7}$$

zu berechnen. Dieses Verfahren konvergiert in einer Umgebung von ξ superlinear und lässt sich geometrisch deuten als schrittweise Annäherung mittels der Nullstellen der Tangenten im jeweils vorhergehenden Punkt $(x_{n-1}, f(x_{n-1})$ bzw. als schrittweise Annäherung durch jeweils den linearen Teil der Taylor-Entwicklung von f. (Die Lösung der nichtlinearen Gleichung wird also durch eine Folge von Lösungen von linearen Gleichungen approximiert.)

Das Newton-Raphson-Verfahren ist allgemeiner auch für höherdimensionale nichtlineare Funktionen $F : \mathbb{R}^n \to \mathbb{R}^m$ anwendbar, d.h., auf

$$F(\mathfrak{x}) := \begin{pmatrix} f_1 (x_1, ..., x_n) \\ \vdots \\ f_m (x_1, ..., x_n) \end{pmatrix} = \mathfrak{o} := \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix} \tag{2.8}$$

Mit der Jacobi-(Funktional-) Matrix von f

$$J_f(\mathfrak{x}) := \begin{pmatrix} \frac{\partial f_1(\mathfrak{x})}{\partial x_1} & \cdots & \frac{\partial f_1(\mathfrak{x})}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_m(\mathfrak{x})}{\partial x_1} & \cdots & \frac{\partial f_m(\mathfrak{x})}{\partial x_n} \end{pmatrix} \tag{2.9}$$

lautet ein Iterationsschritt jetzt bekanntlich

$$\mathfrak{x}_{n+1} := \mathfrak{x}_n - J_f^{-1} (\mathfrak{x}_n) (f (\mathfrak{x}_n)) \tag{2.10}$$

oder anders geschrieben

$$J_f (\mathfrak{x}_n) (\mathfrak{x}_{n+1} - \mathfrak{x}_n) = -f (\mathfrak{x}_n) \tag{2.11}$$

Gleichung (2.11) ist jetzt aber gerade unser LGS (2.4), wenn wir beachten, dass unser f die Funktion der ersten Ableitungen $\partial E_r(x_v)$ und $\partial E_r(y_v)$ ist, also

$$\begin{aligned} f : \mathbb{R}^2 &\to \mathbb{R}^2 \\ \begin{pmatrix} x_v \\ y_v \end{pmatrix} &\mapsto f \begin{pmatrix} x_v \\ y_v \end{pmatrix} := \begin{pmatrix} \partial E_r(x_v) \\ \partial E_r(y_v) \end{pmatrix}, \end{aligned} \tag{2.12}$$

und dem gesuchten Verschiebungsvektor δ_v^r, die Verschiebung $\mathfrak{x}_{n+1} - \mathfrak{x}_n$ entspricht.

2.3 Algorithmus für das lokal schöne Layout

[HK02] beschreiben den Teil-Algorithmus für das lokal schöne Layouten von r-Nachbarschaften eines Graphen äusserst knapp und teilweise auch etwas unpräzis. Deshalb soll dieser Algorithmus „LocalLayout“ hier in einer (hoffentlich) etwas präziseren Form wiedergegeben werden, so dass er unmittelbar implementiert werden kann.

2.3.1 Algorithmus LocalLayout

ALGORITHMUS LocalLayout($d_{V\times V}, L, r, iterations$**)**
% Findet ein lokal schönes Layout L durch Verschönerung
% der r-Nachbarschaften.
% Aufrufparameter:
% $d_{V\times V}$: APSP-Matrix des Graphen
% L: Layout des Graphen
% r: Radius, welcher die r-Nachbarschaften definiert
% iterations: Faktor für die Anzahl Durchgänge
% Rückgabewert:
% Bzgl. r lokal schönes Layout L des durch
% $d_{V\times V}$ def. Graphen
[1] Berechne die r-Nachbarschaften des Graphen
[2] Berechne die ersten Ableitungen
[3] Berechne die Deltas, d.h., die Beträge der Gradienten
[4] **FOR** i=1 **TO** iterations · |V| **DO**
[5] Finde den Knoten v' mit maximalen Gradientenbetrag
[6] Berechne die zweiten Ableitungen von v'
[7] Berechne den Verschiebungsvektor $\delta^r_{v'}$
[8] Verschiebe in L v' um $\delta^r_{v'}$
[9] Aktualisiere die ersten Ableitungen und die Deltas
[10] **RETURN** L
END ALGORITHMUS.

Für die Laufzeit-Analyse und für implementierungstechnische Aspekte des Algorithmus' sei hier noch auf die Kapitel 3 bzw. 3.3 verwiesen.

2.3.2 Diskussion und Bemerkungen

Wir sollten uns hier nochmals erinnern, dass der Algorithmus im Kontext des Multiskalen-Verfahrens mit den k-lpc's G^k aufgerufen wird und auf diesen wird der Algorithmus ausgeführt. Auch das Layout L und der Radius r beziehen sich auf die Knoten der Repräsentanten G^k. Weiter sei auch nochmals auf die Vereinfachungen (siehe 1.8) des Schemas hingewiesen.

In Zeile [1] werden die r-Nachbarschaften berechnet. Diese Notwendigkeit ist in [HK02] nicht explizit erwähnt.

In den Zeilen [2] und [3] werden die Strukturen der ersten Ableitungen und der Deltas initialisiert.

In Zeile [4] beginnt die Schleife in der das Layout so (lokal) verschönert wird, dass die Energie minimiert wird. Die Anzahl der Durchläufe wurde - wie schon erwähnt - so ausgelegt, dass sicher mindestens so oft durchlaufen wird wie es Anzahl Knoten hat, mal einen kleinen Faktor *iterations* (meist etwa zwischen 4 und 12).

In den Zeilen [5] und [6] werden die notwendigen Grössen für die anschliessende Ermittlung des Verschiebungsvektors (Zeile [7]) berechnet. Dies wurde bei [HK02] sehr verkürzt in ihrem ersten Schritt erwähnt.

In [9] werden gewisse Werte bzgl. dem neuen Layout aktualisiert. Dies ist sicher jeweils nur für die betreffenden Nachbarschaften notwendig, was aus Effizienzgründen so implementiert werden sollte. Empirisch hat sich sogar gezeigt, dass die alleinige Aktualisierung der ersten Ableitung des gerade verschobenen Knotens ausreicht (siehe anschliessender Absatz).

Eine kurze Notiz auf S.11 von [HK02] lautet: „After the movement of each vertex, the first derivatives of vertices in $N^k(v)$ are updated."
Ich empfand diese Formulierung etwas mehrdeutig, insbesondere weil sie das Updaten des verschobenen Knotens nicht erwähnt. Dieser muss aber sicher aktualisiert werden, hingegen zeigte eine anders implementierte Variante, dass auf das Updaten der ersten Ableitungen seiner Nachbarschaftsknoten unter Umständen sogar verzichtet werden kann. Der Algorithmus scheint dann nur leicht langsamer zu konvergieren, was bei einer bedeutend schnelleren Laufzeit eines Schleifendurchlaufs gut durch eine minimale Erhöhung von *iterations* kompensiert werden kann. Ein Nachteil ist, dass dies die Instabilität des Verfahrens erhöht.

2.4 Die expliziten Lösungen des LGS

In diesem Abschnitt wollen wir noch die Lösungen des LGS (2.4) explizit herleiten, so dass diese schliesslich auch zur effektiven Berechnung des Verschiebungsvektors $\delta^r_{v'}$ in der Implementierung verwendet werden können. Als erstes leiten wir die Ausdrücke für die Ableitungen her:

2.4.1 Herleitung 1.Ableitungen

Für die ersten Ableitungen von E_r nach den Koordinaten $(\delta^r_{v'}(x), \delta^r_{v'}(y))$ eines Knotens $v' \in V$ erhalten wir:

$$
\begin{aligned}
\frac{\partial E_r}{\partial x_{v'}} &= \frac{\partial}{\partial x_{v'}} \left(\sum_{v \in V} \sum_{u \in N^r(v)} k_{uv} \left(\sqrt{(x_u - x_v)^2 + (y_u - y_v)^2} - ld_{uv} \right)^2 \right) \\
&= \frac{\partial}{\partial x_{v'}} \left(\sum_{\substack{v \in V \\ v \neq v'}} \sum_{u \in N^r(v)} k_{uv} \left(\sqrt{(x_u - x_v)^2 + (y_u - y_v)^2} - ld_{uv} \right)^2 \right. \\
&\qquad \left. + \sum_{u \in N^r(v')} k_{uv'} \left(\sqrt{(x_u - x_{v'})^2 + (y_u - y_{v'})^2} - ld_{uv'} \right)^2 \right) \\
&= \frac{\partial}{\partial x_{v'}} \left(\sum_{\substack{v \in V \\ v \neq v'}} \sum_{u \in N^r(v)} k_{uv} \left(\sqrt{(x_u - x_v)^2 + (y_u - y_v)^2} - ld_{uv} \right)^2 \right) \\
&\quad + \frac{\partial}{\partial x_{v'}} \left(\sum_{u \in N^r(v')} k_{uv'} \left(\sqrt{(x_u - x_{v'})^2 + (y_u - y_{v'})^2} - ld_{uv'} \right)^2 \right) \\
&= \underbrace{\frac{\partial}{\partial x_{v'}} \left(\sum_{\substack{v \in V \\ v \neq v' \\ v' \in N^r(v)}} \sum_{u \in N^r(v)} k_{uv} \left(\sqrt{(x_u - x_v)^2 + (y_u - y_v)^2} - ld_{uv} \right)^2 \right)}_{\text{alle anderen Ableitungen sind 0, da die Summanden } x_{v'} \text{ nicht enthalten}}
\end{aligned}
$$

$$
\begin{aligned}
& + \frac{\partial}{\partial x_{v'}} \left(\sum_{u \in N^r(v')} k_{uv'} \left(\sqrt{(x_u - x_{v'})^2 + (y_u - y_{v'})^2} - ld_{uv'} \right)^2 \right) \\
= \quad & \underbrace{\frac{\partial}{\partial x_{v'}} \left(\sum_{\substack{v \in V \\ v \neq v' \\ v' \in N^r(v)}} k_{v'v} \left(\sqrt{(x'_v - x_v)^2 + (y'_v - y_v)^2} - ld_{v'v} \right)^2 \right)}_{\substack{\text{es muss nur derjenige Summand berücksichtigt} \\ \text{werden, der dem r-Nachbarschaftsknoten v' entspricht,} \\ \text{d.h., } u := v' \in N^r(v)}} \\
& + \frac{\partial}{\partial x_{v'}} \left(\sum_{u \in N^r(v')} k_{uv'} \left(\sqrt{(x_u - x_{v'})^2 + (y_u - y_{v'})^2} - ld_{uv'} \right)^2 \right) \\
= \quad & \sum_{\substack{v \in V \\ v \neq v' \\ v' \in N^r(v)}} \frac{\partial}{\partial x_{v'}} \left(k_{v'v} \left(\sqrt{(x'_v - x_v)^2 + (y'_v - y_v)^2} - ld_{v'v} \right)^2 \right) \\
& + \sum_{u \in N^r(v')} \frac{\partial}{\partial x_{v'}} \left(k_{uv'} \left(\sqrt{(x_u - x_{v'})^2 + (y_u - y_{v'})^2} - ld_{uv'} \right)^2 \right) \quad (2.13)
\end{aligned}
$$

Das sieht jetzt alles ein bisschen komplizierter aus, als es ist. Alle Schritte, auch die folgenden, sind eigentlich elementar. Wir setzen

$$w := \sqrt{(x_u - x_{v'})^2 + (y_u - y_{v'})^2} \quad (2.14)$$

$$u := \sqrt{(x_{v'} - x_v)^2 + (y_{v'} - y_{v'})^2} \quad (2.15)$$

und erhalten durch Ableitung der Summanden

$$
\begin{aligned}
\frac{\partial E_r}{\partial x_{v'}} \quad = \quad & \sum_{\substack{v \in V \\ v \neq v' \\ v' \in N^r(v)}} \frac{2k_{v'v} (x_{v'} - x_v) (u - ld_{v'v})}{u} \\
& + \sum_{u \in N^r(v')} \frac{-2k_{uv'} (x_u - x_{v'}) (w - ld_{uv'})}{w} \quad (2.16)
\end{aligned}
$$

Nun ist nur noch zu beachten, dass

$$v \in V, v \neq v', v' \in N^r(v) \Leftrightarrow v \in N^r(v') \quad (2.17)$$

und

$$u = w \quad (2.18)$$

ist. Dies führt zu

$$
\begin{aligned}
\frac{\partial E_r}{\partial x_{v'}} \quad = \quad & \sum_{v \in N^r(v')} \frac{2k_{v'v} (x_{v'} - x_v) (u - ld_{v'v})}{u} \\
& + \sum_{v \in N^r(v')} \frac{-2k_{vv'} (x_v - x_{v'}) (u - ld_{vv'})}{u} \\
= \quad & \sum_{v \in N^r(v')} \frac{2k_{v'v} (u - ld_{v'v}) [(x_{v'} - x_v) - (x_v - x_{v'})]}{u}
\end{aligned}
$$

$$= \sum_{v \in N^r(v')} \frac{4k_{v'v}\left(u - ld_{v'v}\right)\left(x_{v'} - x_v\right)}{u} \tag{2.19}$$

Völlig analog erhält man die erste Ableitung bzgl. der Koordinate $y_{v'}$. Insgesamt lauten also die

2.4.2 1. Ableitungen

$$\frac{\partial E_r}{\partial x_{v'}} = 4 \sum_{v \in N^r(v')} \frac{k_{v'v}\left(u - ld_{v'v}\right)\left(x_{v'} - x_v\right)}{u} \tag{2.20}$$

$$\frac{\partial E_r}{\partial y_{v'}} = 4 \sum_{v \in N^r(v')} \frac{k_{v'v}\left(u - ld_{v'v}\right)\left(y_{v'} - y_v\right)}{u} \tag{2.21}$$

mit

$$u := \sqrt{\left(x_{v'} - x_v\right)^2 + \left(y_{v'} - y_{v'}\right)^2} \tag{2.22}$$

2.4.3 Herleitung 2. Ableitungen

Die zweiten Ableitungen machen keine Schwierigkeiten und können auf (2.20) und (2.21) angewendet werden. Wir verzichten auf die Details der Anwendung elementarer Ableitungsregeln.

$$\begin{aligned}
\frac{\partial^2 E_r}{\partial x_{v'}^2} &= \frac{\partial}{\partial x_{v'}} \left[4 \sum_{v \in N^r(v')} \frac{k_{v'v}\left(u - ld_{v'v}\right)\left(x_{v'} - x_v\right)}{u} \right] \\
&= 4 \left[\sum_{v \in N^r(v')} \frac{\partial}{\partial x_{v'}} \left(\frac{k_{v'v}\left(u - ld_{v'v}\right)\left(x_{v'} - x_v\right)}{u} \right) \right] \\
&= 4 \left[\sum_{v \in N^r(v')} \frac{k_{v'v}\left(u^3 - ld_{v'v}\left(y_{v'} - y_v\right)^2\right)}{u^3} \right]
\end{aligned} \tag{2.23}$$

$$\begin{aligned}
\frac{\partial^2 E_r}{\partial y_{v'}^2} &= \frac{\partial}{\partial y_{v'}} \left[4 \sum_{v \in N^r(v')} \frac{k_{v'v}\left(u - ld_{v'v}\right)\left(y_{v'} - y_v\right)}{u} \right] \\
&= 4 \left[\sum_{v \in N^r(v')} \frac{\partial}{\partial y_{v'}} \left(\frac{k_{v'v}\left(u - ld_{v'v}\right)\left(y_{v'} - y_v\right)}{u} \right) \right] \\
&= 4 \left[\sum_{v \in N^r(v')} \frac{k_{v'v}\left(u^3 - ld_{v'v}\left(x_{v'} - x_v\right)^2\right)}{u^3} \right]
\end{aligned} \tag{2.24}$$

$$\begin{aligned}
\frac{\partial^2 E_r}{\partial y_{v'} \partial x_{v'}} &= \frac{\partial}{\partial x_{v'}} \left[4 \sum_{v \in N^r(v')} \frac{k_{v'v}\left(u - ld_{v'v}\right)\left(y_{v'} - y_v\right)}{u} \right] \\
&= 4 \left[\sum_{v \in N^r(v')} \frac{\partial}{\partial x_{v'}} \left(\frac{k_{v'v}\left(u - ld_{v'v}\right)\left(y_{v'} - y_v\right)}{u} \right) \right] \\
&= 4 \left[\sum_{v \in N^r(v')} \frac{k_{v'v} ld_{v'v}\left(x_{v'} - x_v\right)\left(y_{v'} - y_v\right)}{u^3} \right]
\end{aligned}$$

$$= \frac{\partial^2 E_r}{\partial x_{v'} \partial y_{v'}} \tag{2.25}$$

In der Übersicht also:

2.4.4 2. Ableitungen

$$\frac{\partial^2 E_r}{\partial x_{v'}^2} = 4 \left[\sum_{v \in N^r(v')} \frac{k_{v'v} \left(u^3 - l d_{v'v} \left(y_{v'} - y_v \right)^2 \right)}{u^3} \right] \tag{2.26}$$

$$\frac{\partial^2 E_r}{\partial y_{v'}^2} = 4 \left[\sum_{v \in N^r(v')} \frac{k_{v'v} \left(u^3 - l d_{v'v} \left(x_{v'} - x_v \right)^2 \right)}{u^3} \right] \tag{2.27}$$

$$\frac{\partial^2 E_r}{\partial y_{v'} \partial x_{v'}} = \frac{\partial^2 E_r}{\partial x_{v'} \partial y_{v'}} = 4 \left[\sum_{v \in N^r(v')} \frac{k_{v'v} l d_{v'v} \left(x_{v'} - x_v \right) \left(y_{v'} - y_v \right)}{u^3} \right] \tag{2.28}$$

wobei u wie in 2.4.2 ist.

2.4.5 Der Verschiebungsvektor

Was wir für die effektive Implementierung benötigen, ist der Verschiebungsvektor

$$\delta_{v'}^r := \left(\delta_{v'}^r (x), \delta_{v'}^r (y) \right)$$

eines Knotens $v' \in V$, welchen wir als denjenigen mit maximalem Gradientenbetrag ermittelt haben. Diesen Vektor erhalten wir, wie gesehen, als Lösung des LGS (2.4). Zur expliziten Angabe haben wir mit den Ausdrücken für die Ableitungen alles beisammen. Zur übersichtlicheren Schreibweise, setzen wir

$$\begin{aligned} a &:= \frac{\partial^2 E_r}{\partial x_{v'}^2} & b &:= \frac{\partial^2 E_r}{\partial x_{v'} \partial y_{v'}} & c &:= -\frac{\partial E_r}{\partial x_{v'}} \\ d &:= \frac{\partial^2 E_r}{\partial y_{v'} \partial x_{v'}} & e &:= \frac{\partial^2 E_r}{\partial y_{v'}^2} & f &:= -\frac{\partial E_r}{\partial y_{v'}} \end{aligned} \tag{2.29}$$

Dann lässt sich unser LGS (2.4), dessen Lösung und damit die Komponenten unseres Verschiebungsvektors berechnen durch:

$$\delta_{v'}^r (x) = \frac{ce - bf}{ae - bd}, \qquad ae \neq bd \tag{2.30}$$

$$\delta_{v'}^r (y) = \frac{af - cd}{ae - bd}, \qquad ae \neq bd \tag{2.31}$$

wobei unsere Ableitungen in $a, b, ..., f$ durch die Ausdrücke 2.4.2 und 2.4.4 bestimmt sind.

Kapitel 3

Implementierung

In diesem Kapitel soll nun der ganze Algorithmus von Harel und Koren nach ihrer Publikation [HK02], bzw. unsere Implementierung im Java-Programm „GraphLayout“ vorgestellt und diskutiert werden. Ebenso soll eine Anzahl erzeugter Layouts typischer Graphen vorgestellt werden (Kapitel 5).

3.1 Algorithmus nach Harel und Koren

Der vollständige Multiskalen-Algorithmus von Harel und Koren lautet (vgl. [HK02]):

ALGORITHMUS Layout$G(V, E)$
% Findet ein schönes Layout L des Graphen G
% Die Definition der Schönheit ist dabei definiert
% als ein lokales Energieminimum im Federsystem-Modell
% Parameter (mit Defaultwerten):
% rad(=7): bestimmt den Radius lokaler Nachbarschaften
% $iterations$(=4): Anzahl lokaler Verschönerungsschritte
% $ratio$(=3): Quotient der k's aufeinanderfolgender k-lpc's
% $minSize$(=10): Anfangsgrösse der Repräsentation (k_0)
% Notwendige Bedingungen an die Parameter:
% $\exists e \in \mathbb{N} : ratio^e \cdot minSize = |V|$
% Rückgabewert:
% Schönes Layout des Graphen G
[1] Berechne die APSP-(all-pair-shortest-path-)Matrix von G
[2] Generiere ein zufälliges Layout L von G
[3] k:=minSize
[4] **WHILE** $k \leq |V|$ **DO**
[5] Berechne die centers-Knoten bzgl. k (vgl. 1.8 2.)
[6] Berechne den Nachbarschaftsradius r
[7] LocalLayout($d_{centers \times centers}, L(centers), r, iterations$)
[8] **IF** ist nicht letzter while-Durchlauf **THEN**
[9] Positioniere jeden Knoten von G an die leicht
. verschobene Stelle seines centers-Knotens
[10] k:=k · ratio
[11] **RETURN** L
END ALGORITHMUS.

3.1.1 Bemerkungen

Der hier formulierte Algorithmus ist der, welcher in Java implementiert wurde. Er weicht in ein paar Punkten von demjenigen von Harel und Koren in [HK02] ab. Die Abweichungen bzw. Präzisierungen in der Unterprozedur LocalLayout (siehe 2.3) wurden bereits dort besprochen.

Als Vorbedingung an die Parameter habe ich

$$\exists e \in \mathbb{N} : ratio^e \cdot minSize = |V| \tag{3.1}$$

hinzugefügt. Dies ist meiner Meinung nach wesentlich, denn (erst) dies garantiert, dass im letzten Durchgang $k = |V|$ ist und damit dort $G^{|V|}$ berechnet wird, womit dann der Ausgangsgraph identifiziert werden kann (siehe auch Abschnitt 1.7 zur Korrektheit des Schemas). Andernfalls wird nämlich der Graph nicht vollständig schön gelayoutet.

Weiter ist im Hauptalgorithmus oben die zusätzliche Zeile [8] eingefügt, d.h., dass die Neupositionierung nur bis zum vorletzten Durchlauf der while-Schleife durchgeführt wird. Warum? Im letzten Durchlauf wird eben gerade das Layout von $G^{|V|}$ berechnet. Da wir dort jeden Knoten des Ausgangsgraphen mit seinem Cluster (der nur noch aus ihm selbst besteht) identifizieren

$$\begin{array}{l} \forall i \in \mathbb{N}_{|V|} : \\ v_i \in V_G, V_i = \{v_i\} \in V_{G^{|V|}} \implies v_i \equiv V_i \end{array} \tag{3.2}$$

Deshalb darf im letzten Durchgang das Layout nicht mehr verrauscht werden, denn das würde das schöne Layout wieder zerstören! (Dass im letzten while-Durchlauf überhaupt nichts mehr getan werden muss, kommt dann aus implementierungstechnischen Gründen dazu: Wie bereits hingewiesen, verwalten wir die Layouts von G und der G^k in derselben Struktur.)

In [5] werden die centers-Knoten als Lösung des sogenannten „k-Centers-Problem“ ermittelt, bei dem wir in einem Graphen k Knoten so suchen, dass die längste (graphentheoretische) Distanz von V zu diesen k Centers minimal ist. Dieser Algorithmus wurde unverändert aus [HK02] übernommen.

In Zeile [6] wird der Radius r für jeden spezifischen G^k berechnet, und zwar nach der in [HK02] genannten Formel:

$$r := \max_{v \in centers} \min_{u \in centers} \{d_{uv}\} \cdot rad \tag{3.3}$$

3.1.2 Parameter-Einstellungen für Spezialfälle

Da die Parameter $minSize$, $ratio$ den Multiskalen-Ansatz steuern und rad, $iterations$ den Ansatz der lokalen Verschönerung, können wir zur Veranschaulichung deren Werte für einige Spezialfälle betrachten.

Spezialfall 1: Wir können im Algorithmus $minSize = |V|$ setzen und eliminieren so den Multiskalenansatz. Die while-Schleife wird genau einmal ausgeführt. ($ratio$ muss wie immer > 1 sein, sonst terminiert die Schleife nicht). Mit dieser Einstellung hat unsere Folge von k-lpc's G^k genau ein Folgenglied $G^{|V|}$, nämlich unseren Ausgangsgraph.

Spezialfall 2: Setzen wir $rad > |V|$, so erreichen wir damit, dass als lokale Verschönerung jeweils der ganze aufrufende Graph betrachtet wird. D.h., wir eliminieren den Ansatz der Verschönerung von r-Nachbarschaften. Die Energieminimierung erfolgt dann immer bzgl. aller Knoten.

Spezialfall 3: Eliminieren wir wie im Spezialfall 1 den Multiskalen-Ansatz und wie im Spezialfall 2 den Ansatz der lokalen Verschönerung, so haben wir einen Layout-Algorithmus, der alleine auf dem Ausgangsgraphen die Energieminimierung mittels iterativem Gradientenabstiegsverfahren durchführt.

Als Beispiel haben wir den 144-Knoten-SquareGrid-Graphen „SquareGrid_144“ einmal mit der Multiskalen-Methode (siehe Abb. 3.1) und dann noch je mit den Einstellungen für die drei Spezialfälle generiert (siehe Abb. 3.2 - 3.4).

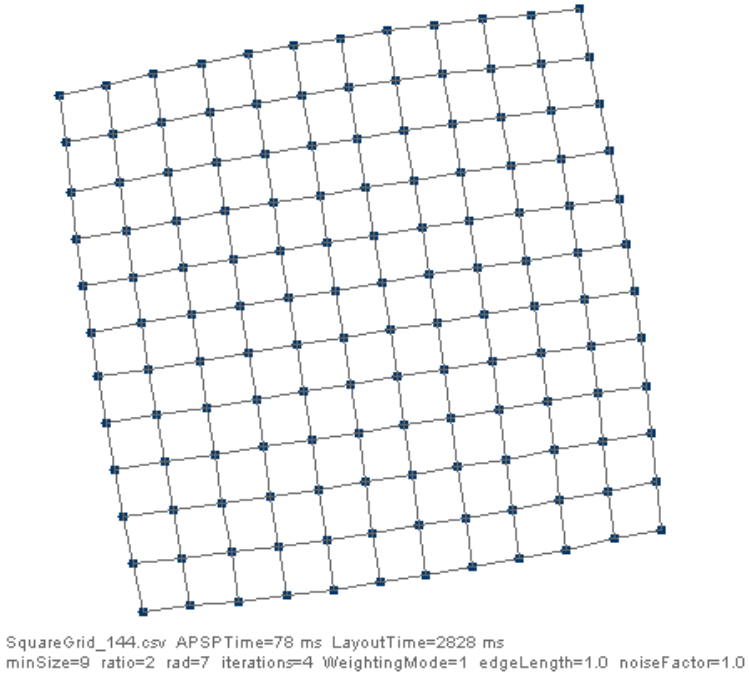


Abbildung 3.1: Das Layout von SquareGrid_144 mit der Multiskalen-Methode (nicht als Spezialfall) erzeugt.

3.2 „GraphLayout“: Eine Implementation in Java

Die Hauptziele der Implementierung in Java waren:

1. Test der Implementierbarkeit des Algorithmus von Harel und Koren und des Algorithmus' selbst
2. (Bequeme) Möglichkeit der Erzeugung und Untersuchung von Layouts mit verschiedenen Parametern
3. Möglichkeit der permanenten Speicherung von erzeugten Layouts inkl. der Parameter-Konfiguration

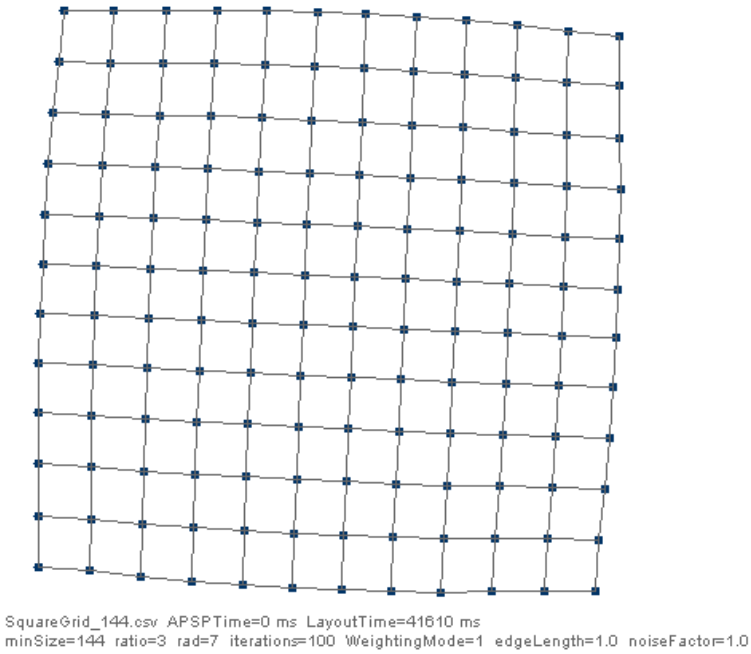


Abbildung 3.2: Das Layout von SquareGrid_144 nach Spezialfall 1 erzeugt.

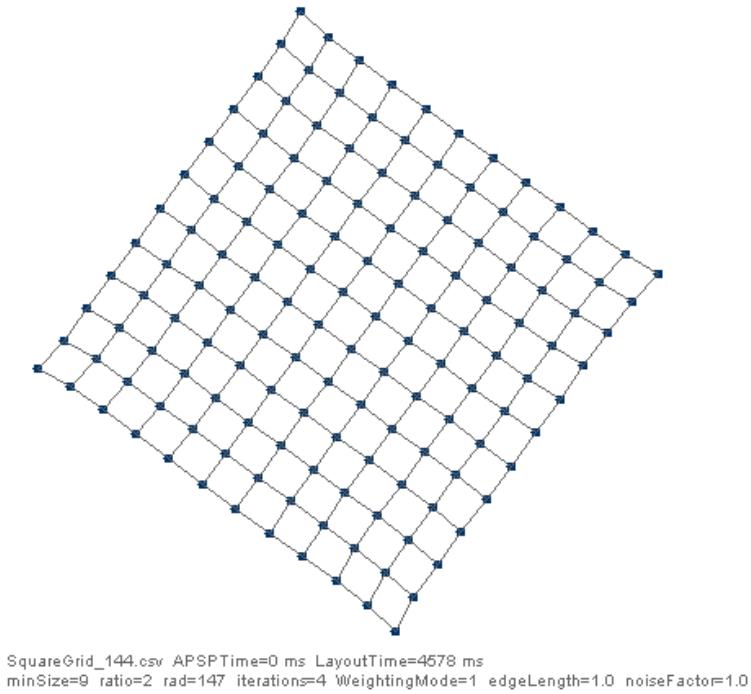


Abbildung 3.3: Das Layout von SquareGrid_144 nach Spezialfall 2 erzeugt.

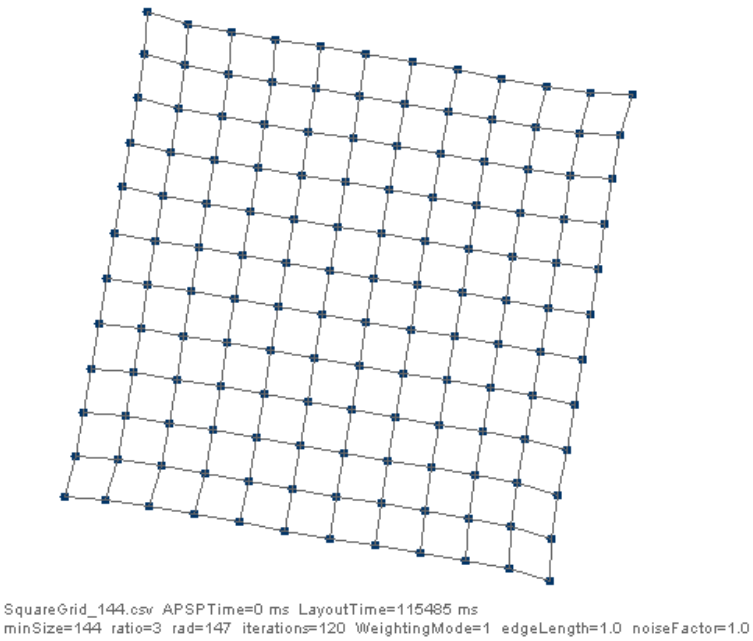


Abbildung 3.4: Das Layout von SquareGrid_144 nach Spezialfall 3 erzeugt.

4. Möglichkeit der Untersuchung des Laufzeit-Verhaltens der Methode in einer realen Implementierung

Dies führte zum Programm „GraphLayout Rel.1.0“. (1.0 ist der für die Diplomarbeit massgebliche Release. Ich kann mir vorstellen, das Programm später noch zu erweitern, deshalb die Versionierung.)

Ein implizites Ziel war es auch, mit einer zweckmässigen GUI (Benutzeroberfläche) die Benutzung effizient und intuitiv zu machen. Im folgenden sollen einige wichtige Implementierungsteile von „GraphLayout Rel.1.0“ vorgestellt werden. (Man verzeihe mir dort die Angewohnheit, Code in Englisch zu erstellen und zu kommentieren.) Für weitere Details wird auf den Sourcecode verwiesen. Ein Kurzüberblick über das Programm aus Usersicht folgt in Kap. 4.

3.2.1 Berechnung der APSP-Matrix

Die Berechnung der APSP-Matrix ist entscheidend für die Laufzeit-Komplexität des Gesamtalgorithmus'. Wie [HK02] habe ich ein Vorgehen gewählt, welches für jeden Knoten eine SSSP-(single-source-shortest-path-)Suche mittels Breitensuche (BFS, breadt-first-search) durchführt. Dafür wurde die Klasse `ApspMatrix` implementiert. Das Prinzip ist aus dem Code des Konstruktors ersichtlich (weitere Details im Sourcecode der Klasse):

```
public ApspMatrix(AdjacencyMatrix in_adjMatrix) {
  // initialize
  N = in_adjMatrix.getNumberOfVertices();
  apspMatrix = new int[N][N];
  noedge = N; // Denotes "infty" in other implementations;
              // N works too, because a path is always
              // smaller than the number of vertices!
```

```
  for(int i=0;i<N;i++) {
    for(int j=0;j<N;j++) {
      apspMatrix[i][j] = noedge;
    }
  }
  adjList = makeAdjList(in_adjMatrix);  // this allows us to
                                        // do the BFS
  for(int i=0;i<N;i++) {  // for all vertices
    queue = new IntQueue(N);
    apspMatrix[i][i]=0;
    queue.enqueue(i);

    // SSSP calculation with BFS follows:
    while(!queue.isEmpty()) {
      v=queue.front();

      queue.dequeue();
      a=0;   // denotes current location in the adj.list of v
      while(adjList[v][a]!=noedge) {
        w=adjList[v][a];   // index of next adjacent vertex of v

        if(apspMatrix[i][w]==noedge) {
          apspMatrix[i][w]=apspMatrix[i][v]+1;
          queue.enqueue(w);  // because we do breath-first
        } else {
        }
        a=a+1;
      }
    }
  }
}
```

Ein zentraler Punkt ist dabei, dass wir durch

```
private int[][] makeAdjList(AdjacencyMatrix in_adjMatrix) {
  // The adjList is a representation of the graph's adjacency
  // relation that allows us the efficient BFS
  // (breath-first-search)
  // For each vertex i adjList[i] represents a (unordered) list
  // of its adjacent vertices.
  // Hence, visiting all the vertices in a vertex i's list,
  // equals the BFS starting with i
  int[][] adjList;

  adjList = new int[N][N];
  for(int i=0;i<N;i++) {
  int a=0;
  for(int j=0;j<N;j++) {
    if(in_adjMatrix.getElement(i,j)==true) {
      adjList[i][a]=j; // says that i is adjacent to j (but in
                       // an unordered list adjList for each i)
      a=a+1;
    } else {
    }
```

```
  }
  adjList[i][a]=noedge;
  }
return adjList;
}
```

eine Adjazenzliste mitverwalten, womit die Breitensuche (BFS) effizient durchgeführt werden kann. Damit erreichen wir für die Berechnung der APSP-Matrix eine Laufzeit-Komplexität von $\Theta(|V||E|)$. Der Speicherbedarf ist $\Theta\left(|V|^2\right)$.

3.2.2 Berechnung der k-Centers Knoten

Die k Centers-Knoten werden in jedem Durchgang dadurch ermittelt, dass das „k-Centers-Problem“ gelöst wird. Der implementierte Code basiert auf dem in [HK02] angegebenen Algorithmus. Da die Methode als „private“ deklariert ist, werden die Attribute der Klasse direkt manipuliert und nicht über die Schnittstelle der Methode. Deshalb seien diese hier zur besseren Leserlichkeit zuerst noch erwähnt:

```
private int max;
private int numberOfVertices;
private int[] currentDistances;
private int startVertex;
private int dist,u;
```

Die Methode lautet dann:

```
private int[] kCenters() {
    // Finds a subset S  with k elements of V of a graph, such that
    // S is a solution of the k centers problem, ie, the following
    // holds:
    // max_{v \in V} min_{s \in S} {d_{sv}} is minimal

    max = numberOfVertices;
    currentDistances = new int[max];
    startVertex = (int)(Math.round(Math.pow(10,9)
                    * Math.random()) % max);

    int[] cntrs = new int[k];
    cntrs[0] = startVertex;
    currentDistances = apsp.copyAPSPMatrixRow(startVertex);
    for(int i=1;i<k;i++) {
        dist = 0;
        u = 0;
        // search the vertex farthest from the current ctrs vertices:
        for(int j=0;j<max;j++) {
                if(currentDistances[j] > dist) {
                dist = currentDistances[j];
                u = j;
            } else {
            }
        }
        // update the currentDistances table for the vertex u
        // just found:
        for(int j=0;j<max;j++) {
```

```
            if(apsp.getElement(j,u) < currentDistances[j]) {
                currentDistances[j] = apsp.getElement(j,u);
            } else {
            }
        }
        cntrs[i] = u;
    }
    return cntrs;
} // kCenters
```

3.2.3 LocalLayout

Auch hier zuerst die zugehörigen Variablen-Deklarationen:

```
private int[][] neighbourhood;
private Parameters params; // class that manages all param.
private int numberOfCenters;
private float[][] firstDerivs;// first derivs of all centers
// float[v'][0]: 1st deriv, wrt x_v'
// float[v'][1]: 1st deriv. wrt y_v'
private float[] secondDerivs = new float[3]; // second derivs
                            // of a specific vertex
// float[0]: 2nd deriv. wrt (x_v')^2
// float[1]: 2nd deriv. wrt (y_v')^2
// float[2]: 2nd deriv. wrt x_v', y_v'
// and y_v', x_v
private int maxDeltaCenter; // index of the "a" in the centers
                                // array ("a" denotes the "v'" in
                                // the text)
private BinaryHeap deltas;  // A standard implementation of a
                                // binary heap (see text)
private float[] translation = new float[2]; // [0]: x-Coord.
                                                // [1]:y-Coord.
private float[][] layoutFloat; // integer values for pixels are
                                   // computed at a later stage when
                                   // generating the graphics
private int[] centers;
```

Die Methode localLayout lautet nun:

```
private void localLayout() {
    /** Finds a locally nice layout L by local beautification
     *  of the r-neighbouhoods.
     *  Parameters:
     *    d_{VxV}: APSP-matrix of the graph
     *    L: Layout of the graph
     *    r: Radius, which defines the r-neighbourhoods
     *    iterations: Factor for the number of iterations
     *  Return value:
     *    Locally nice layout L wrt r of the graph defined by
     *    d_{VxV}
     */

    neighbourhood = kNeighboursOfCenters();
```

```
    firstDerivs = firstDerivsOfCenters();
    deltas = computeDeltasOfCenters();

    for(int i=1;i<=params.getIterations()*numberOfCenters;i++) {

        // [HK02] step1: Choose vertex v with maximal delta:
        maxDeltaCenter = computeMaxDeltaCenter(deltas);

        secondDerivs = computeSecondDerivsOfCenter(maxDeltaCenter);

        // [HK02] step2:
        translation = computeTranslationVectorForCenter(maxDeltaCenter);

        // [HK02] step3: Move vertex v
        layoutFloat[centers[maxDeltaCenter]][0]
                = layoutFloat[centers[maxDeltaCenter]][0] + translation[0];
        layoutFloat[centers[maxDeltaCenter]][1]
                = layoutFloat[centers[maxDeltaCenter]][1] + translation[1];

        // [HK02] see p.188-189 (about updating); less updating may work too
        // but may increase instability
        updateFirstDerivs(maxDeltaCenter, firstDerivs);
        updateDeltasOfCenters(deltas, maxDeltaCenter);
    }
} // localLayout
```

3.3 Analyse

In diesem Abschnitt soll der ganze Algorithmus unter einigen Standardaspekten weiter analysiert werden. Um keine relevante Stelle zu übersehen, müssen wir schrittweise die einzelnen Teile des Algorithmus durchgehen und den Algorithmus falls nötig noch präzisieren bzw. ausformulieren. Dabei wollen wir auf der abstrakteren Ebene der Algorithmik bleiben und uns nicht um Implementierungsdetails kümmern. Grössenordnungsangaben gefolgt von einem Stern * bedeuten im Folgenden, dass die Begründung an separater Stelle anschliessend erfolgt.

Zur Erinnerung sei noch einmal erwähnt, dass wir uns auf *zusammenhängende, endliche, schlichte (=keine Schlingen, keine Parallelitäten), ungerichtete Graphen* beschränken. (D.h., wir bewegen uns in einer Teilmenge der Kategorie $\mathfrak{Graph}$ (vgl. z.B. [Se02])). Als Konsequenz gilt für die Kantenanzahl:

$$|V| - 1 \leq |E| \leq |V|(|V| - 1)/2 \tag{3.4}$$

LayoutG(V,E)

Den Algorithmus der obersten Ebene haben wir schon in 3.1 kennengelernt. Das Wesentliche ist bei folgender Analyse der Laufzeit-Komplexität noch einmal verkürzt aufgestellt:

3.3.1 Laufzeit-Komplexität: Einzelne Teile

ALGORITHMUS Layout$G(V, E)$

[1]	Berechne die APSP-(all-pair-shortest-path-)Matrix von G	$\Theta(\|V\|(\|V\|+\|E\|))$*
[2]	Generiere ein zufälliges Layout L von G	$\Theta\left(\|V\|^2\right)$
[3]	k:=minSize	$\Theta(1)$
[4]	**WHILE** $k \leq \|V\|$ **DO**	$\Theta(\|V\|log(\|V\|)$
[5]	Berechne die centers-Knoten bzgl. k (vgl. 1.8 2.)	$\Theta(k\|V\|)$*
[6]	Berechne den Nachbarschaftsradius r	$\Theta(1)$
[7]	LocalLayout($d_{centers \times centers}, L(centers), r, iterations$)	$\Theta(klog(k))$*
[8]	**IF** ist nicht letzter while-Durchlauf **THEN**	$\Theta(1)$
[9]	Positioniere jeden Knoten von G an die leicht verschobene Stelle seines centers-Knotens	$\Theta(\|V\|)$
[10]	k:=k · ratio	$\Theta(1)$
[11]	**RETURN** L	

END ALGORITHMUS.

Die einzelnen Zeilen sehen aus algorithmischer Sicht folgendermassen aus:

Zeile [1]: Berechne APSP-Matrix

ALGORITHMUS ComputeAPSPMatrix

% Zeitkomplexität gesamt: $\Theta(|V|(|V|+|E|))$

[1.1]	Initialisiere APSP-Matrix (α_{ij}) mit „noedge“ (infinity)	$\Theta(\|V\|^2)$
[1.2]	Berechne eine Adjazenzliste	$\Theta(\|V\|^2)$*
[1.3]	**FOR** i=0 **TO** $\|V\|$ **DO**	$\Theta(\|V\|)$
[1.4]	Setze das i-te Diagonalelement der APSP-Matrix auf 0	$\Theta(1)$
[1.5]	Füge i in eine Warteschlange	$\Theta(1)$
[1.6]	**WHILE** Warteschlange ist nicht leer **DO**	$\Theta(\|V\|+\|E\|)$*
[1.7]	Entnimm ein Knoten v der Warteschlange	.
[1.8]	**WHILE** Adjazenzliste von v ist nichtleer **DO**	.
[1.9]	Nimm einen Knoten w der Adj.-liste von v	.
[1.10]	**IF** i und w sind nicht adjazent **THEN**	.
[1.11]	Setze $\alpha_{iw} := \alpha_{iv} + 1$	.
[1.12]	Füge w wieder in die Warteschlange	.
[1.13]	Gehe zum nächsten adjazenten Knoten von v	.
[1.14]	**RETURN** APSPMatrix	

END ALGORITHMUS.

Zeile [1.2]: Berechne Adjazenzliste

ALGORITHMUS ComputeAdjList

% Zeitkomplexität gesamt: $\Theta\left(|V|^2\right)$

[1.2.1]	**FOR** i=0 **TO** $\|V\|$ **DO**
[1.2.2]	**FOR** j=0 **TO** $\|V\|$ **DO**
[1.2.3]	**IF** i und j sind adjazent **THEN**
[1.2.4]	Füge Knoten j in die Liste von i
[1.2.5]	Markiere für i das Listenende (z.B. mit „noedge“ oder „infinity“)
[1.2.6]	**RETURN** adjList

END ALGORITHMUS.

Zeilen [1.3],[1.6]-[1.13]: $|V|$-mal SSSP-Suche mittels BFS

Die Zeilen [1.6] bis [1.13] entsprechen der „Single-source-shortest-path“- (SSSP-) Suche mittels BFS („Breatdh-first-search“; Breitensuche). Dieser Standard-Algorithmus hat bekanntlich eine Zeitkomplexität von $\Theta(|V|+|E|)$ (vgl. z.B. [Gu96]). Zur Erzeugung der APSP-Matrix führen wir die SSSP-Suche für jeden Knoten durch (Zeile [1.3]) und erhalten somit als Ordnung von ComputeAPSPMatrix $\Theta\left(|V|(|V|+|E|)\right)$.

Zeile [5]: K-Centers (nach [HK02] S.187)

ALGORITHMUS KCenters

% Zeitkomplexität gesamt: $\Theta(k|V|)$
% S sei die Menge der k Centers

[5.1]	S := v für einen beliebigen Knoten v	
[5.2]	**FOR** i=2 **TO** k **DO**	$\Theta(k\|V\|)$
[5.3]	Finde den Knoten u, der am weitesten von S entfernt ist (d.h., $\min_{s\in S}\{d_{us}\} \geq \min_{s\in S}\{d_{ws}\}, \forall w \in V$)	$\Theta(\|V\|)$*
[5.4]	S := S ∪ u	

END ALGORITHMUS.

Zeile [5.3]:

Diese Zeile lässt sich schneller implementieren als standardmässig mit BFS (wofür wir $\Theta(|E|)$ bräuchten), denn wir haben ja in einem früheren Schritt die APSP-Matrix berechnen müssen. Somit reicht der einmalige Durchlauf einer APSP-Matrix-Zeile in $\Theta(|V|)$ Zeit.

Zeile [7]: LocalLayout

Das Wesentliche in diesem Teilalgorithmus (vgl. auch 2.3) lautet:

ALGORITHMUS LocalLayout($d_{V\times V}, L, r, iterations$**)**

% Zeitkomplexität gesamt: $\Theta(klog(k))$
% (Anm.: Dies bedeutet für den übergeordneten Algorithmus
% $\Theta(|V|log(|V|))$, da die Schleife bis $k := |V|$ aufruft.

[7.1]	Berechne die r-Nachbarschaften des Graphen	$\Theta(k^2)$*
[7.2]	Berechne die ersten Ableitungen	$O(k^2)$*
[7.3]	Berechne die Deltas, d.h., die Beträge der Gradienten	$\Theta(klog(k))$*
[7.4]	**FOR** i=1 **TO** iterations · |V| **DO**	$\Theta(k))$*
[7.5]	Finde den Knoten v' mit maximalen Gradientenbetrag	$\Theta(log(k)))$*
[7.6]	Berechne die zweiten Ableitungen von v'	$O(k)$*
[7.7]	Berechne den Verschiebungsvektor $\delta^r_{v'}$	$\Theta(1)$*
[7.8]	Verschiebe in L v' um $\delta^r_{v'}$	$\Theta(1)$
[7.9]	Aktualisiere die ersten Ableitungen	$O(k)$*
[7.10]	Aktualisiere die Deltas	$\Theta(log(k))$*
[7.11]	**RETURN** L	

END ALGORITHMUS.

Zeile [7.1]:

Für jedes Paar von Centers-Knoten muss geprüft werden, ob sie r-Nachbarn sind: $\Theta(k^2)$ Laufzeit.

Zeile [7.2]:

Zu jedem Centers-Knoten v muss zur Berechnung seiner ersten Ableitungen jeder seiner Nachbarn aufgerufen werden: $\Theta(k|N^r(v)|)$. Wegen $|N^r(v)| \leq k$ haben wir $O(k^2)$.

Zeile [7.3]:

Die Deltas können in einem „MaxHeap“, d.h., einem links-vollständigen, partiell geordneten Max-Binärbaum (ein Max Priority Queue) verwaltet werden. Zu seinem Aufbau benötigen wir $\Theta(klog(k))$ Laufzeit: Denn für jeden Centersknoten wird ein insert mit $\Theta(log(k))$ Laufzeit durchgeführt.

Zeile [7.4]:

LocalLayout wir ja mit der Repräsentation G^k aufgerufen. D.h., es ist jeweils $|V| := k$.

Zeile [7.5]:

Sehr kostengünstige Operation, aufgrund unseres MaxHeaps. Nach der Entnahme der Wurzel wird ein „downheap“ zu $\Theta(log(k))$ durchgeführt (die Standardoperation „deletemax“ eines MaxHeaps).

Zeile [7.6]:

Die Berechnung nach der Formel für die zweiten Ableitungen kostet uns $\Theta(|N^r(v)|$ für den Centers-Knoten v mit maximalem Delta. D.h., $O(k)$.

Zeile [7.7]:

Die Berechnung des Verschiebungsvektor kostet uns hier „nichts“ mehr, da wir vorgängig alle Grössen ermittelt haben und jetzt nur noch die Formeln 2.30 und 2.31 anzuwenden brauchen.

Zeile [7.9]:

Im Prinzip benötigen wir für die Updates der ersten Ableitungen des verschobenen Knotens und seiner Nachbarn eine Laufzeit von $O(k^2)$. Es hat sich jedoch gezeigt, dass oft alleine die Aktualisierung der ersten Ableitungen des verschobenen Knotens ausreicht, d.h., wir brauchen dann sogar nur $\Theta(|N^r(v)|$ bzw. $O(k)$.

Zeile [7.10]:

Wegen dem „insert“ in einen MaxBinärbaum brauchen wir hier nochmals $\Theta(log(k))$.

3.3.2 Laufzeit-Komplexität: Gesamter Algorithmus

Wenn wir nun die einzelnen Teile zusammensetzen, dann kommen wir auf eine Gesamtlaufzeit-Komplexität von

$$\Theta(|V||E|) \tag{3.5}$$

denn der Algorithmus wird offensichtlich von Zeile [1], der Berechnung der APSP-Matrix dominiert. Dies natürlich nur unter der Bedingung, dass an den anderen Stellen die erwähnten, effizienten Implementierungen gewählt werden.

3.3.3 Speicher-Komplexität

Alle Strukturen sind kleiner oder gleich der APSP-Matrix bzw. der Adjazenzmatrix. Somit ist die Speicher-Komplexität $\Theta(|V|)$.

3.3.4 Terminierung

1. Der Algorithmus terminiert, wenn die while-Schleife und die Teil-Algorithmen ihres Rumpfes terminieren.

2. Die while-Schleife terminiert, weil k mit $minSize > 1$ initialisiert wird und in jedem Durchlauf durch $k := k \cdot ratio$ für ein $ratio > 1$ erhöht wird.

3. Im Schleifenrumpf terminiert der Teil-Algorithmus K-Centers (siehe [HK02], S.187) nach k Schritten und lässt k unverändert.

4. Der Teil-Algorithmus LocalLayout terminiert nach $iterations \cdot |V|$-Schritten bzgl. des aufrufenden Graphen.

3.3.5 Korrektheit, Stabilität, Konvergenz

Die Korrektheit und Konvergenz des *Schemas* wurde bereits in 1.7 gezeigt. Für die genannte Klasse von Graphen (zusammenhängende, endliche, schlichte (=keine Schlingen, keine Parallelitäten), ungerichtete Graphen) wird die Stabilität des Schemas davon abhängen, wie gut das gefundene lokale Energieminimum des Layouts ist. Dies hängt offensichtlich vom jeweiligen Graphen ab, so dass hier für gewisse Graphen alleine schon wegen der Energieminimierung gewisse Komplikationen zu erwarten sind. Dies haben alle Verfahren, die auf einem Energiemodell des Layouts basieren, gemeinsam. (Die zugrundegelegte Klasse ist aber sonst gross genug, um für viele in Anwendungen vorkommende Graphen interessant zu sein.)

Etwas anders sieht die Situation aus, wenn man den *Algorithmus* mit den Vereinfachungen betrachtet. Es mag sein, dass dadurch eine effizientere Implementierung erst möglich wird. Allerdings wird dies meiner Meinung nach relativ teuer erkauft: Es ist kaum mehr möglich, die Korrektheit des Algorithmus auf diejenige des mathematisch fundierten Schemas zu beziehen. Das gleiche gilt auch für die Konvergenz und die Stabilität des Verfahrens und es kann nur teilweise bestätigt werden, dass der Algorithmus auch mit diesen Vereinfachungen funktioniert. (Teilweise ist der Algorithmus auch zu wenig detailliert spezifiziert, so dass relativ viel Interpretationsspielraum bleibt.) In Kapitel 5 werde ich einige Beispiele dazu geben.

Weiter ist das Verfahren noch weit davon entfernt, die Berechnung eines schönen Layouts für einen beliebigen Graphen der betrachteten Klasse völlig „automatisch“ zu übernehmen. Dafür fehlt eine Theorie, welche es erlauben würde, geeignete Algorithmen-Parameter aus Eigenschaften des zu layoutenden Graphen ableiten zu lassen.

Es zeigt sich sogar, wie auch [HK02] ausführen, dass bei gewissen Graphtypen wie etwa Binärbäumen, der Algorithmus eine falsche Annahme macht: Die Verknüpfung der Nähe zweier Knoten im euklidischen Sinn im Layout mit der graphentheoretischen Nähe ist hier nämlich nicht korrekt. Bei einem Binärbaum sollten gewisse graphentheoretisch weit auseinanderliegende Blätter im Layout nahe beieinander gezeichnet werden! Ein weiterer Beleg, dass das Verfahren nur begrenzt stabil ist und durchaus nicht für beliebige Eingaben gute Ergebnisse liefert.

Schliesslich soll aber nicht unerwähnt bleiben, dass für gewisse Typen von Graphen - z.B. SquareGrids, Kreisgraphen, FoldedGrids - gute Ergebnisse erzielt werden können und dies in einer eindrücklich schnellen Laufzeit.

Kapitel 4

GUI des Programms „GraphLayout“

In diesem Kapitel folgt ein kurzer Überblick über die GUI des Programms.

4.1 Systemvoraussetzungen

Als Systemvoraussetzung muss Java auf dem Rechner installiert sein, und zwar die JRE 1.6.0_01 (Sun nennt diese Version auch Java Version 6, Update 1). Es ist durchaus möglich, dass das Programm auch auf älteren Java-Plattformen läuft, ich empfehle aber diese, um Probleme zu vermeiden.

Bzgl. Hardware-Ausstattung sind sicher alle P4 akzeptabel, für die mitgelieferten Graphen, sollte auch „wenig“ RAM und ein PIII Prozessor reichen. Letzlich wirkt sich dies einfach auf die Laufzeiten aus. Das JAR-File „GraphLayout.jar“ ist klein (weit unter 1MB) und die mitgelieferten Adjazenzmatrix-Files sind etwa 22MB, also alles i.A. praktikabel, sowohl im Hinblick auf Diskspace wie auch Download.

Deklaration der beiden Maschinen, auf denen ich getestet habe:

1. WinXP PC
   - CPU: Intel Pentium IV, 3 GHz
   - RAM: 512 MB DDR
   - Grafikkarte: Radeon 9250, AGP 8x, 128 MB DDR, 400MHz DAC
   - OS: Window XP Pro. 2002, SP2
   - Java-Plattform: JRE 1.6.0_01 („Version 6, Update 1“)
2. LINUX Notebook
   - CPU: Intel Pentium M735, 1.7 GHz
   - RAM: 512 MB DDR
   - Grafikkarte: Intel 82852/855GM
   - Scientific Linux 4.0 (RHEL-basierte Linux Distro.)
   - Java-Plattform: JRE 1.6.0_01 („Version 6, Update 1“)

Alle Laufzeit-Angaben in dieser Arbeit beziehen sich, wenn nichts anderes gesagt wird, auf den WinXP PC.

4.2 Installation und Start

4.2.1 Dateien

Folgende Dateien sind von Bedeutung:

1. „GraphLayout_Rel_X_Y.jar“: Das Programm der Release-Version X.Y als Java-Archivdatei.
2. „AdjacencyMatrices.zip“: Eine Kollektion von comma-separated (csv) Textdateien mit Adjazenzmatrizen verschiedener ungerichteter Graphen.
3. „GraphLayout_Rel_X_Y_src.zip“: Der Sourcecode des Programms in Release-Version X.Y (optional).

4.2.2 Programm installieren und starten

Bei funktionierender Java-Plattform ist die Installation bzw. Ausführung des Programms denkbar einfach. Je nach Konfiguration des Systems genügt schon ein Doppelklicken auf das .jar-File. Falls nicht, sollten folgende Hinweise genügen:

1. Speichern Sie die .jar-Datei an einer Stelle Ihrer Wahl.
2. Erstellen Sie im Verzeichnis der .jar-Datei ein Unterverzeichnis „Adjacency-Matrices“. Dies ist optional, wird aber dazu führen, dass das Programm bei der Auswahl des Graphen von dort aus startet, so dass nicht durch den Dateibaum geklickt werden muss.
3. Die Datei mit dem Sourcecode kann an einem beliebigen Ort gespeichert werden (oder überhaupt nicht).
4. Je nach dem kann das JAR-Archiv nun durch Doppelklick gestartet werden. Wenn nicht, dann kann dies über die Kommandozeile getan werden:
   - Windows:
     - Konsole öffnen: Menü >Start >Ausführen und
       ```
       cmd.exe
       ```
       eintippen, <enter>.
     - In der Textkonsole:
       ```
       java -jar "<MeinPfad>\<jarFileName>.jar"
       ```
       eingeben (inkl.Anführungszeichen) und <enter>.
     - Das Programm wird gestartet.
     - Kann das System die JRE nicht finden, so ist noch zusätzlich der Pfad vor Run Time Engine anzugeben
       ```
       C:\<MeinPfadZurJRE>\java -jar "<MeinPfad>\<jarFileName>.jar"
       ```
       oder in das betreffende Verzeichnis mit der JRE zu wechseln.
   - Linux/Unices:
     - In der Kommandozeile (bash, sh, usw.)
       ```
       java -jar "<MeinPfad>/<jarFileName>.jar"
       ```
       eingeben (inkl. Anführungszeichen) und <enter>
     - Findet das System die Java Run Time Engine nicht („Datei java nicht gefunden“), so muss der Pfad mitangegeben werden:
       ```
       /<MeinPfadZurJRE>/java -jar "<MeinPfad>/<jarFileName>.jar"
       ```
       (kann je nach Systemkonfiguration verschieden sein).
     - Das Programm wird gestartet.

4.3 Ein Graph-Layout berechnen

Unter der Voraussetzung, dass man die Adjazenzmatrix eines berechnenden Graphen als csv-File parat hat, ist die Erzeugung eines Layouts denkbar einfach:

Gestartet wird der Prozess im betreffenden Menüpunkt (vgl. Abb.4.1). Es erscheint ein Datei-Dialog in welchem das File mit den Adjazenzmatrix-Daten selektiert wird (Abb. 4.2). Entspricht das File dem gültigen Format, dann öffnet sich ein Fenster zur Eingabe der Algorithmen- (erstes Tab; Abb. 4.3) und der Bild- und Ausgabe-Parameter (zweites Tab; Abb. 4.4). Es werden dabei gewisse Defaultwerte angezeigt, die aber für den konkret zu berechnenden Graphen *nicht* sinnvoll zu sein brauchen. Auch wurde bislang noch keine überprüfung implementiert (wie z.B. die Bedingung 3.1).

Nach dem Klicken des Buttons „Generieren!“ wird die Berechnung gestartet. Man beachte, dass dies für sehr grosse Graphen eine gewisse Zeit in Anspruch nehmen kann. Konnte die Erzeugung ordnungsgemäss abgeschlossen werden, dann erscheint eine kurze Meldung und die erzeugten Layouts sind als (noch geschlossene) Fenster am unteren Rand der GUI aufgereiht (Abb. 4.5). Klicken auf das gewünschte Layout öffnet das Panel (Abb. 4.6).

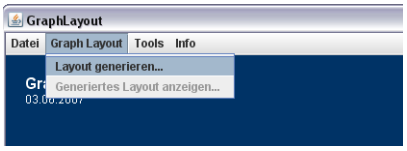


Abbildung 4.1: Starten der Layout-Berechnung via Menü.

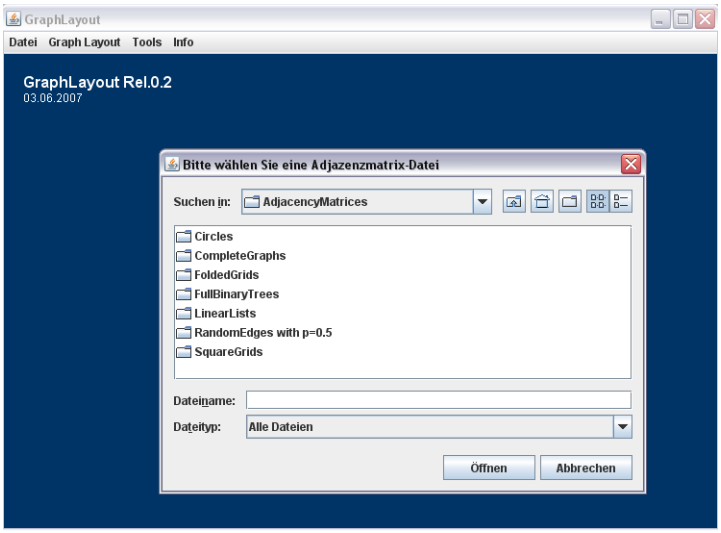


Abbildung 4.2: Laden einer csv-Datei mit den Adjazanzmatrix-Daten.

4.4 Erzeugte Layouts inkl. Parameter speichern

Zur Zeit lässt sich ein erzegutes Layout als .png- und .jpg-Bilddatei speichern (siehe den entsprechenden Button im Layout-Panel, Abb. 4.6). (Es ist noch kein

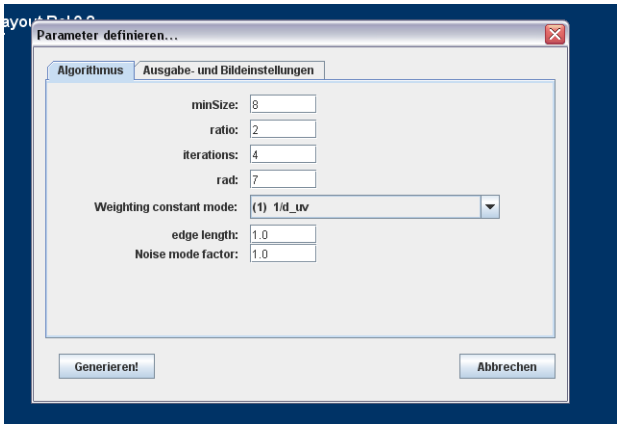


Abbildung 4.3: Dialog zum Eingeben der algorithmusspezifischen Parameter.

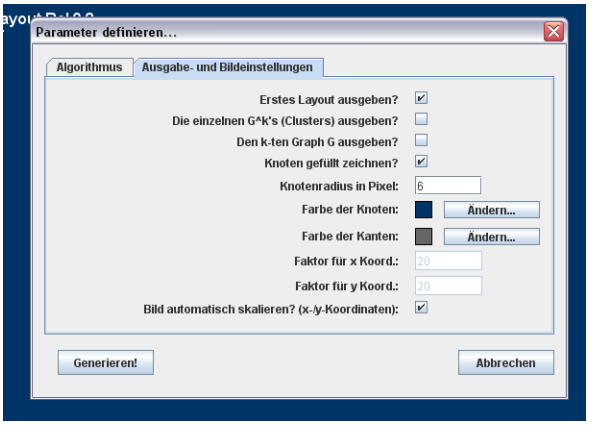


Abbildung 4.4: Dialog zum Eingeben von ausgabe- und bildspezifischen Einstellungen.

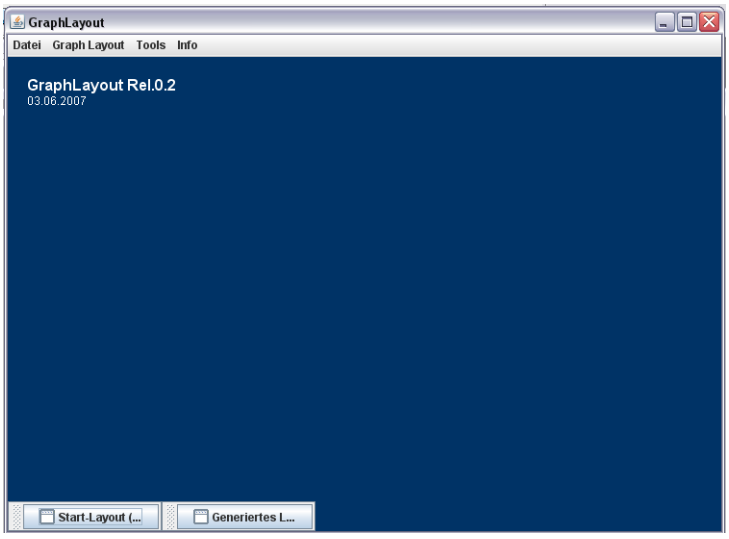


Abbildung 4.5: Frames mit erzeugten Layouts werden nach Erzeugung unten vorerst geschlossen angeordnet. Durch Klicken lässt sich ein gewünschtes Layout öffnen.

Dialog implementiert, der die Auswahl erlaubt, es werden einfach beide Formate gespeichert.) Die Speicherung erfolgt wieder bequem über einen Filedialog, wobei nur der Name ohne die Dateiendung eingegeben werden muss.

Es sei hier noch angemerkt, dass sich die Anzeigegrösse des Layouts durch Verändern des Panels bzw. der ganzen GUI beinflussen lässt. Bei der Speicherung wird dabei die momentan angezeigte Grösse genommen, so dass man dies in einem gewissen Bereich steuern kann.

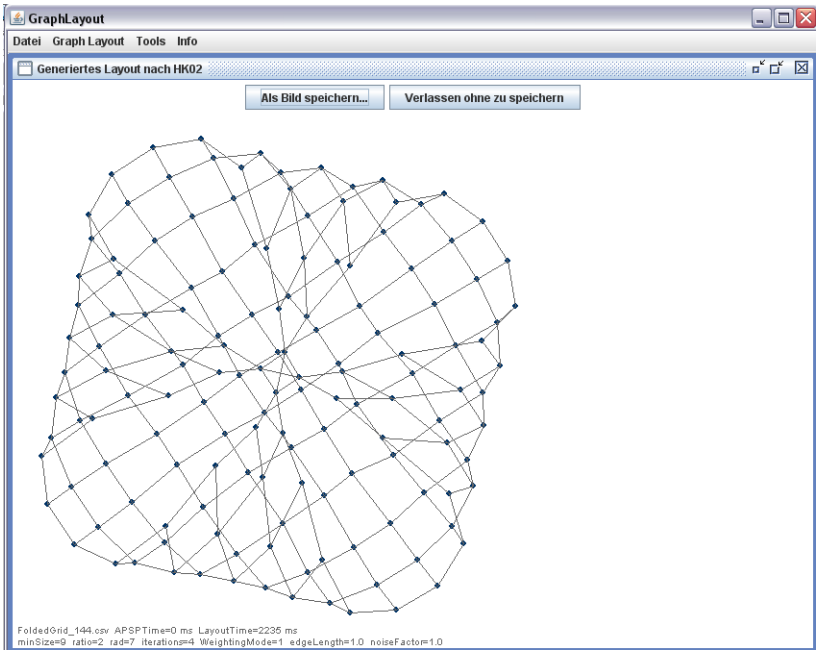


Abbildung 4.6: Layout Panel mit der Möglichkeit das Layout als Bilddatei zu speichern (.png und .jpg).

Schliesslich sind weitere Funktionen im Programm geplant, aber noch nicht implementiert. Deshalb sind gewisse Menüpunkte vorerst deaktiviert.

Kapitel 5

Beispiele von erzeugten Layouts

Nun sollen einige der erzeugten Layouts vorgestellt und diskutiert werden.

5.1 Layouts von SquareGrids, FoldedGrids und Kreisgraphen

Für diese drei Typen von Graphen lassen sich sehr gute Ergebnisse erzielen. Betrachten wir als erstes den SquareGrid1024 Graphen, d.h, ein 32x32-Knoten Gitter. Das erste Layout wurde in 28s erzeugt (Abb. 5.1). Zur Effizienzsteigerung habe ich eine Variante implementiert (die in der GUI optional gewählt werden kann), welche nur die ersten Ableitungen des jeweils verschobenen Knotens aktualisiert, d.h., *ohne* seine Nachbarn. Damit kann die Laufzeit ziemlich stark nach unten gedrückt werden (ca. 3-5s; siehe Abb. 5.2. Meines Wissens haben [HK02] aber diesen Ansatz nicht verwendet). Das Ergebnis bleibt zufriedenstellend.

Auch FoldedGrids (d.h., gefaltete quadratische Gitter) lassen sich nach ein paar Parameteranpassungen einigermassen schön layouten. Ein erstes Layout ist noch unbefriedigend (Abb. 5.3). Aber durch Ausprobieren der Parametereinstellungen lässt es sich relativ gut verbessern (Abb. 5.4). Die von mir implementierte Variante der Vernachlässigung der Nachbarknoten beim Aktualisieren, führt dagegen zu einem unbefriedigenden Ergebnis. Dies als Demonstration dafür, dass „spontane“ Ansätze zur Laufzeitverbesserung mit Vorsicht zu geniessen sind. Auch wenn die Laufzeit von unter 3s eindrücklich sein mag (Abb. 5.5).

Kreisgraphen schliesslich bieten keine prinzipiellen Probleme und ihre Layouts lassen sich relativ mühelos erzeugen, wenn man akzeptiert, dass das Layout nicht gerade ein geometrisch perfekter Kreis sein muss. Zwei Beispiele seien hierzu angeführt (Abb. 5.6).

5.2 Einfluss der Vereinfachungen im Algorithmus

Als ein (weiteres) Beispiel dafür, wie sich die Vereinfachungen im Algorithmus im Vergleich zum Multiskalen-Schema auswirken, können wir uns bei der Erzeugung die Layouts der Multiskalen-Repräsentation ausgeben lassen (die Laufzeiten sind dann aber nicht mehr aussagekräftig). Für den Fall des 16-Knoten-Kreisgraphen

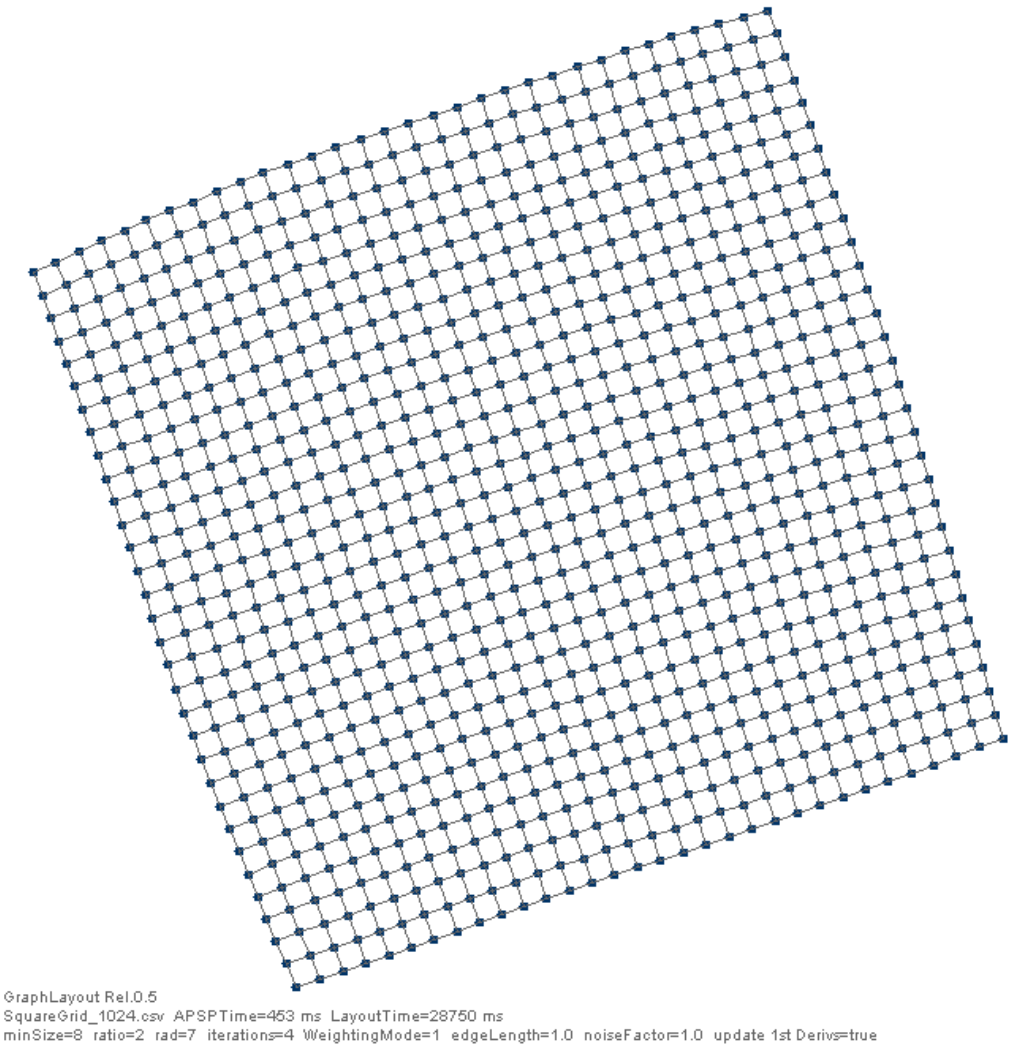


Abbildung 5.1: SquareGrid_1024 Graph mit dem [HK02]-Algorithmus.

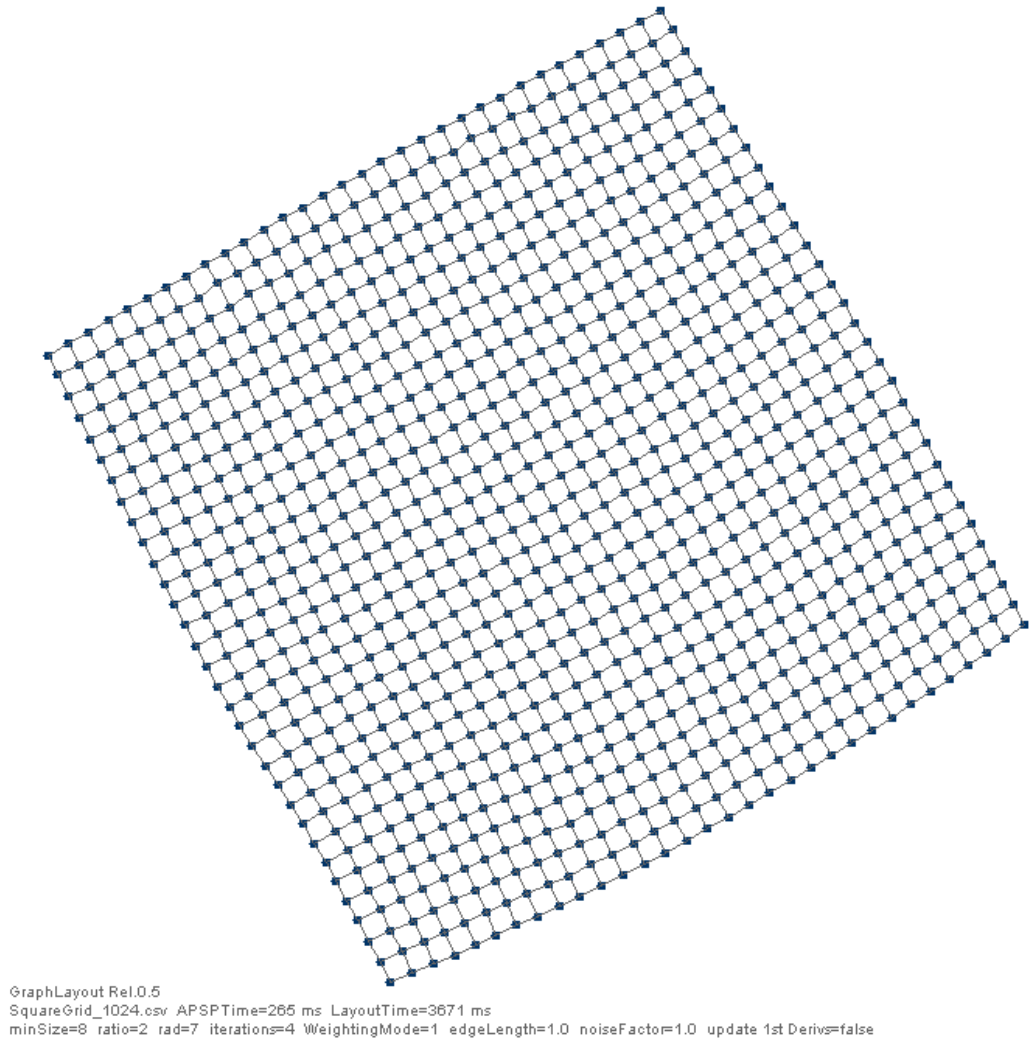


Abbildung 5.2: Layout von SquareGrid_1024 mit einer weiteren Effizienzsteigerung: Die ersten Ableitungen der Nachbarn des maxCenters-Knoten (Knoten mit grösstem Gradientenbetrag) werden nicht aktualisiert.

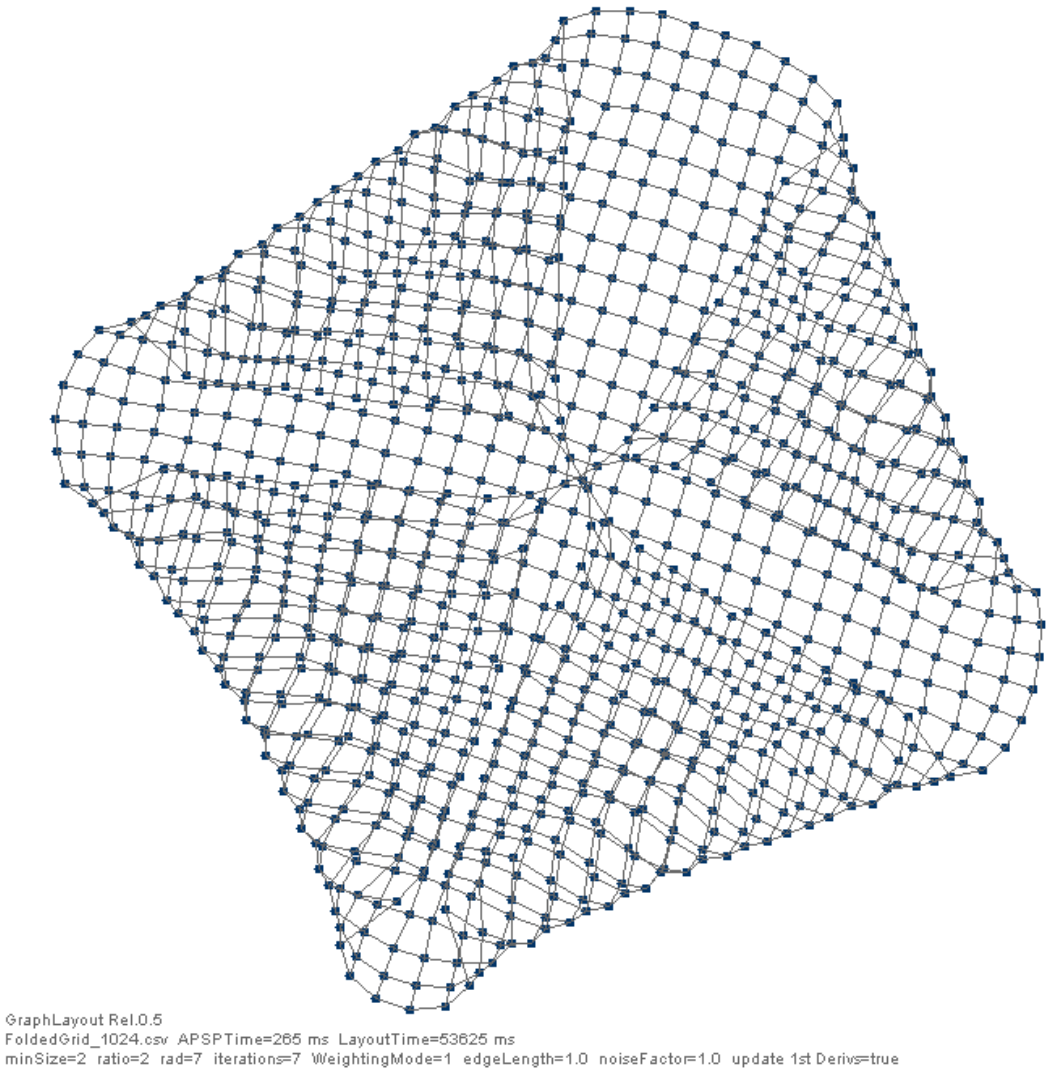


Abbildung 5.3: Layout des FoldedGrid_1024, vor allem in den Details noch unbefriedigend.

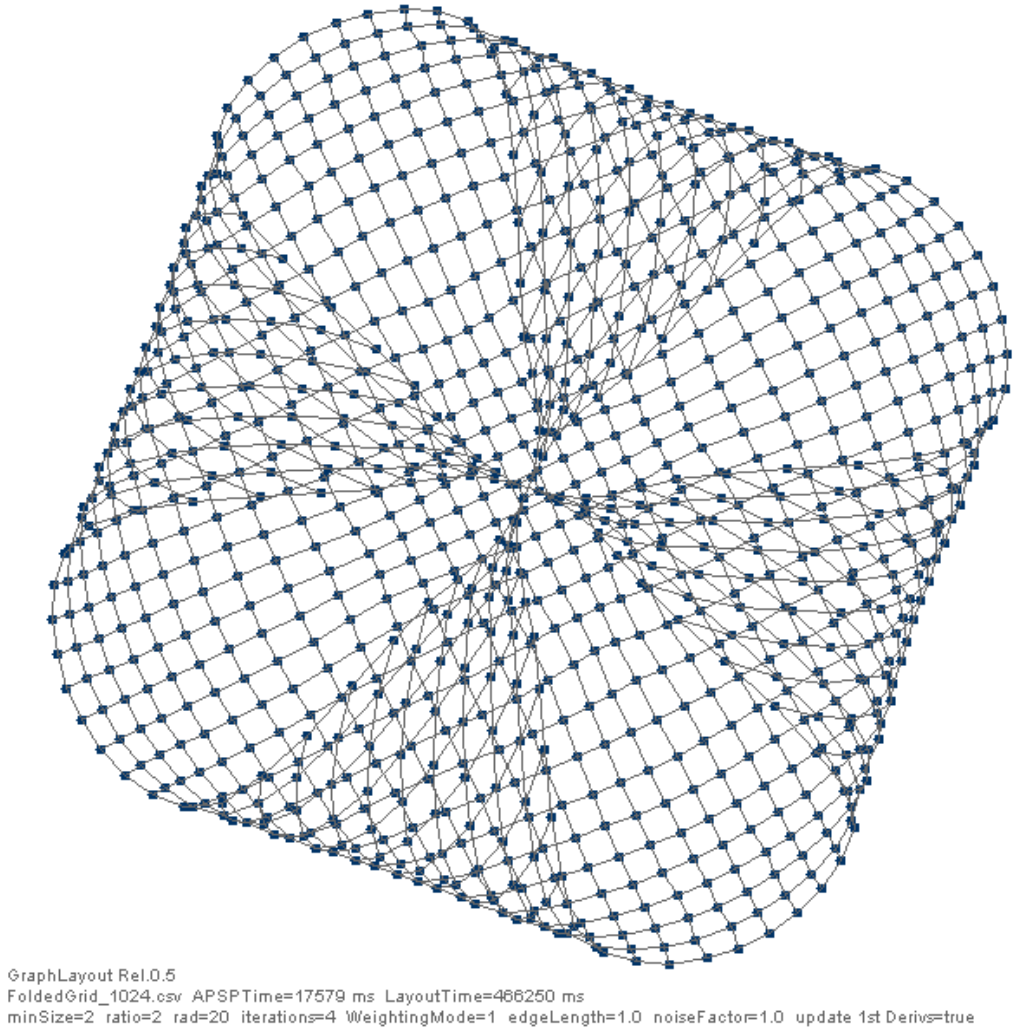


Abbildung 5.4: Anpassung der Parameter verbessert das Layout auch in den Details.

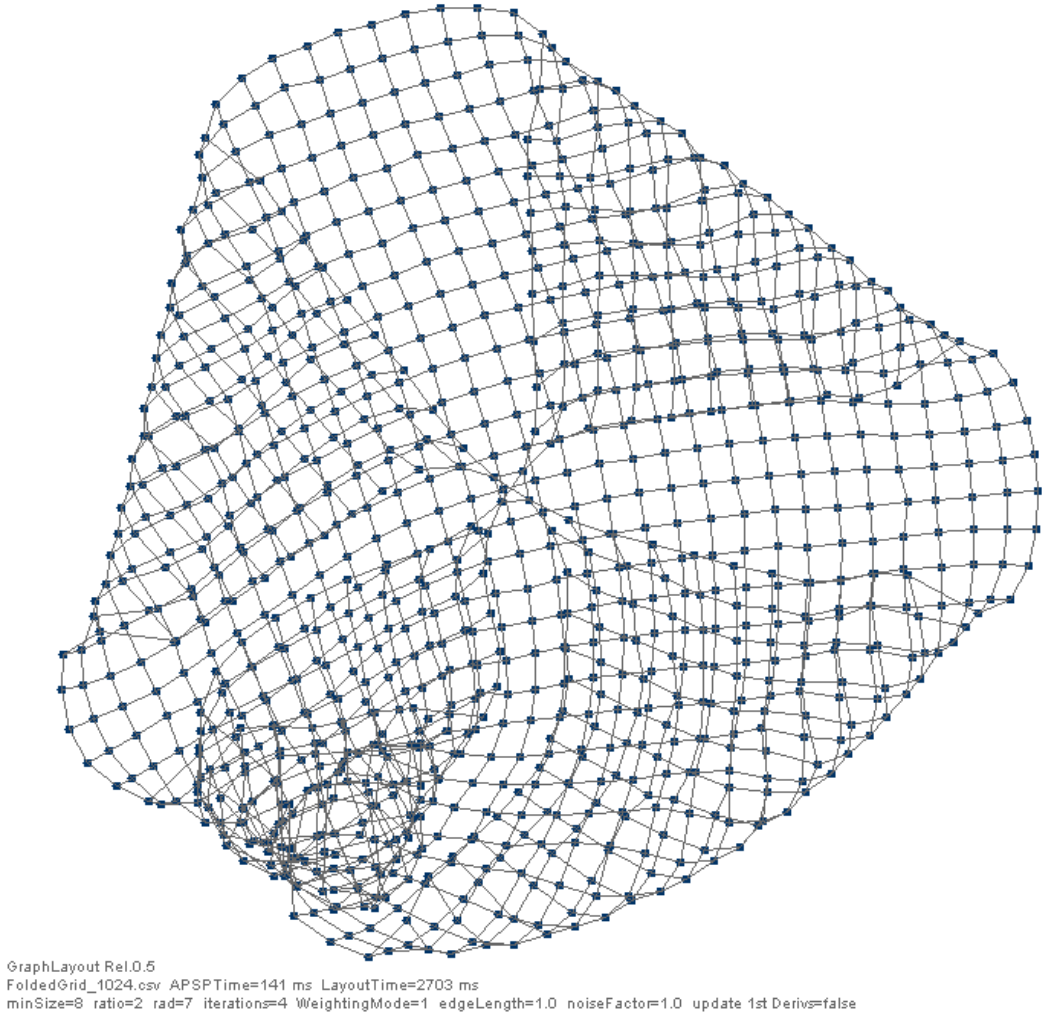


Abbildung 5.5: FoldedGrid_1024 generiert ohne Aktualisierung der maxCenters-Nachbarschaft.

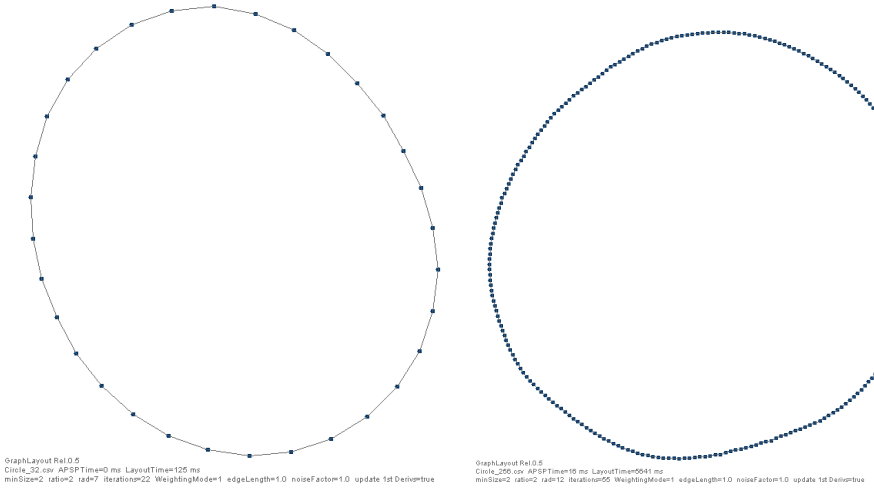


Abbildung 5.6: Das Layout eines 32-Knoten-Kreisgraphen (links) und eines 256-Knoten-Kreisgraphen (rechts).

$Circle_{16}$ sehen diese theoretisch wie in Abb. 1.7 gezeigt aus. Der Algorithmus (mit den Vereinfachungen) erzeugt aber Layouts wie in Abb. 5.7 dargestellt. Insbesondere ist zu beachten, dass die G^k für kleine k's nicht lokal schöne Layouts sind, wie es theoretisch der Fall sein muss. Offensichtlich kann das der Algorithmus in vielen Fällen bei zunehmendem k korrigieren (im Beispiel ab $k := 16$). Dennoch zeigt dies, dass sich der Algorithmus klar in seinem Verhalten vom Schema unterscheidet.

5.3 Binäre Bäume

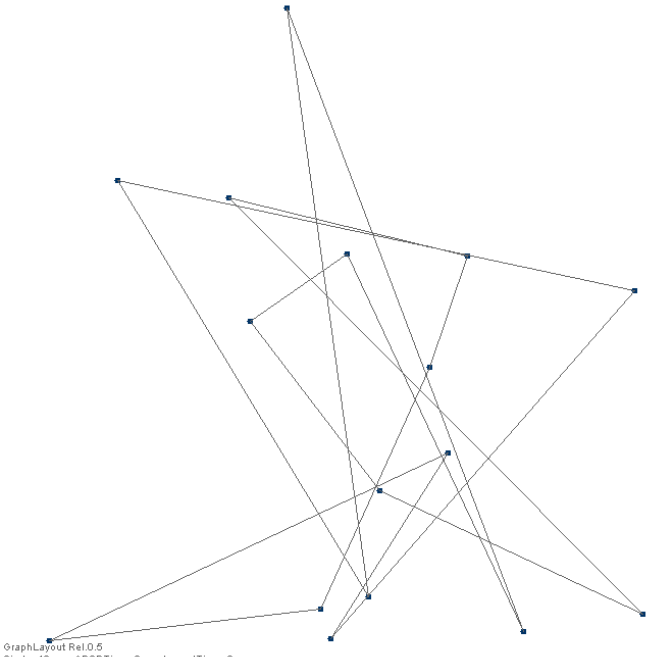
Auf die Schwierigkeiten bei der Layouterzeugung von Binären Bäumen wurde schon hingewiesen (vgl. 3.3.5 und [HK02] S. 190). Dies ist aber nicht primär auf die Vereinfachungen im Algorithmus zurückzuführen, sondern generell auf die Verschönerung mittels eines Energiemodells. Als Beispiel sehen wir genau die in 3.3.5 genannte Schwierigkeit bei graphentheoretisch weit auseinanderliegenden Blätter, für die aber eine kleine euklidische Distanz erwartet wird (Abb. 5.8). Nur mit sehr viel Aufwand konnte das fast befriedigende Ergebnis erzeugt werden. Schon eine zweite Berechnung mit den *gleichen* Paramatern, zeigt ein deutlich schlechteres Layout (Abb. 5.9). Auch die Stabilität bzgl. des zufällig erzeugten Start-Layouts ist also nicht sehr gross.

Bei *vollständigen* Binärbäumen können diese Probleme sowieso nicht auf den Algorithmus und auch nicht auf den Multiskalen-Ansatz bezogen werden, denn hier ist $|V|$ eine Primzahl, so dass im Algorithmus $minSize := |V|$ gewählt werden muss. D.h., - wie in 3.1.2 gesehen - dass gar keine Multiskalen-Repräsentation benutzt wird.

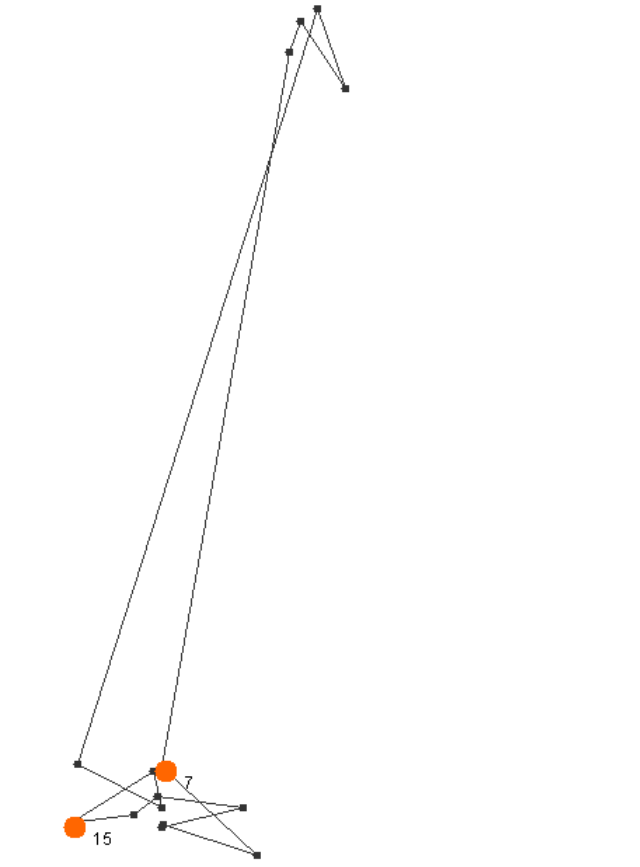
5.4 Graphen mit unregelmässiger Struktur

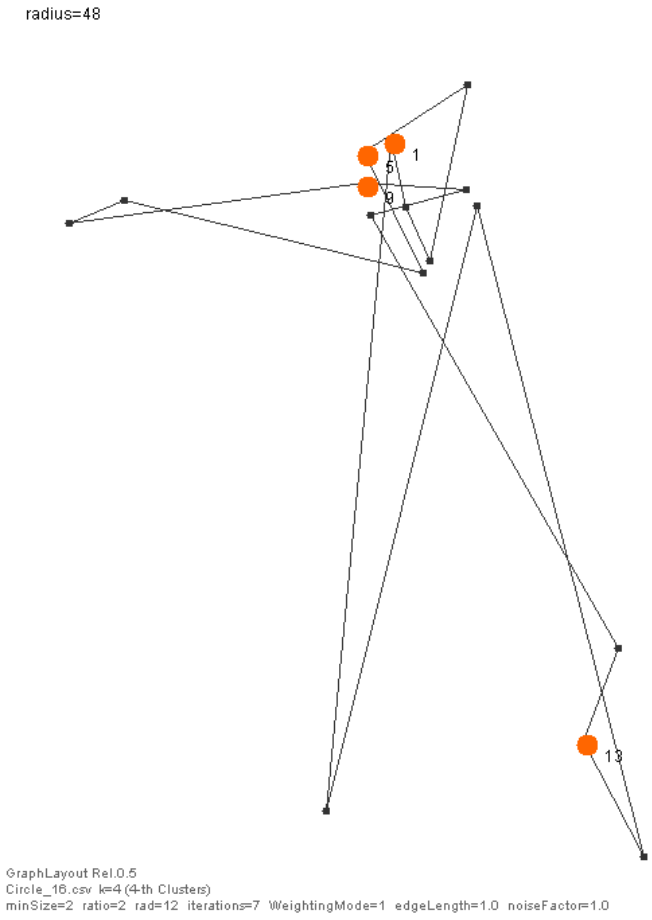
Eine interessante Frage ist schliesslich auch, wie sich die Methode bzw. der Algorithmus bei weniger regelmässigen Graphen verhält. Als Beispiel sei hier ein SquareGrid vorgestellt, in welchem die Kanten mit einer bestimmten Wahrscheinlichkeit weggelassen wurden (Abb. 5.10).

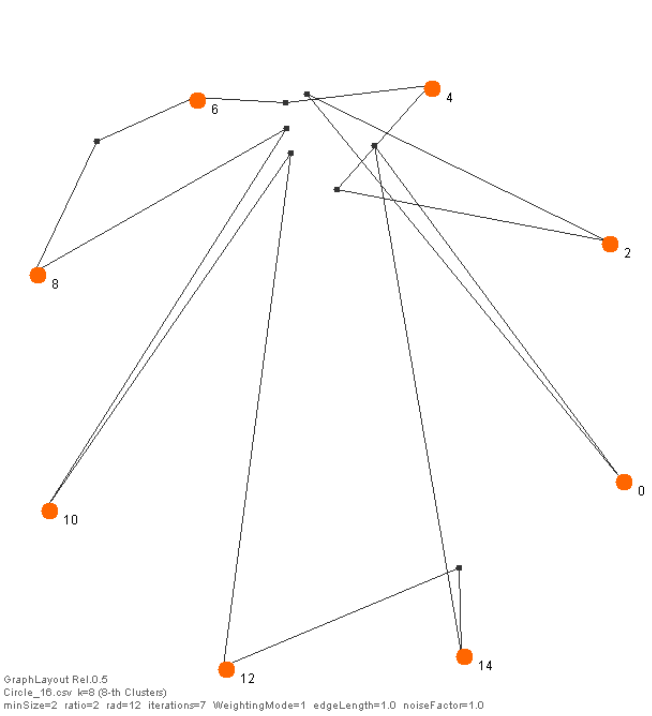
Ein Problem bei der Betrachtung solcher Graphen ist, dass unter Umständen der Graph nicht mehr zusammenhängend ist. Dann „driftet“ das Layout auseinander und die Zusammenhangskomponenten werden klein gezeichnet. Dies liesse sich umgehen, indem zuerst die Zusammenhangskomponenten des Graphen berechnet würden und danach jede Komponente separat gelayoutet würde. Zusammenhangskomponenten lassen sich effizient in $O(|V|+|E|)$ (siehe z.B. [Ot96]) implementieren.


GraphLayout Rel.0.5
Circle_16.csv APSPTime=0 ms LayoutTime=0 ms
minSize=2 ratio=2 rad=12 iterations=7 WeightingMode=1 edgeLength=1.0 noiseFactor=1.0 update 1st Derivs=true

radius=96


7
15
GraphLayout Rel.0.5
Circle_16.csv k=2 (2-th Clusters)
minSize=2 ratio=2 rad=12 iterations=7 WeightingMode=1 edgeLength=1.0 noiseFactor=1.0


radius=48
1
5
9
13
GraphLayout Rel.0.5
Circle_16.csv k=4 (4-th Clusters)
minSize=2 ratio=2 rad=12 iterations=7 WeightingMode=1 edgeLength=1.0 noiseFactor=1.0


radius=24
6
4
2
8
0
10
12
14
GraphLayout Rel.0.5
Circle_16.csv k=8 (8-th Clusters)
minSize=2 ratio=2 rad=12 iterations=7 WeightingMode=1 edgeLength=1.0 noiseFactor=1.0

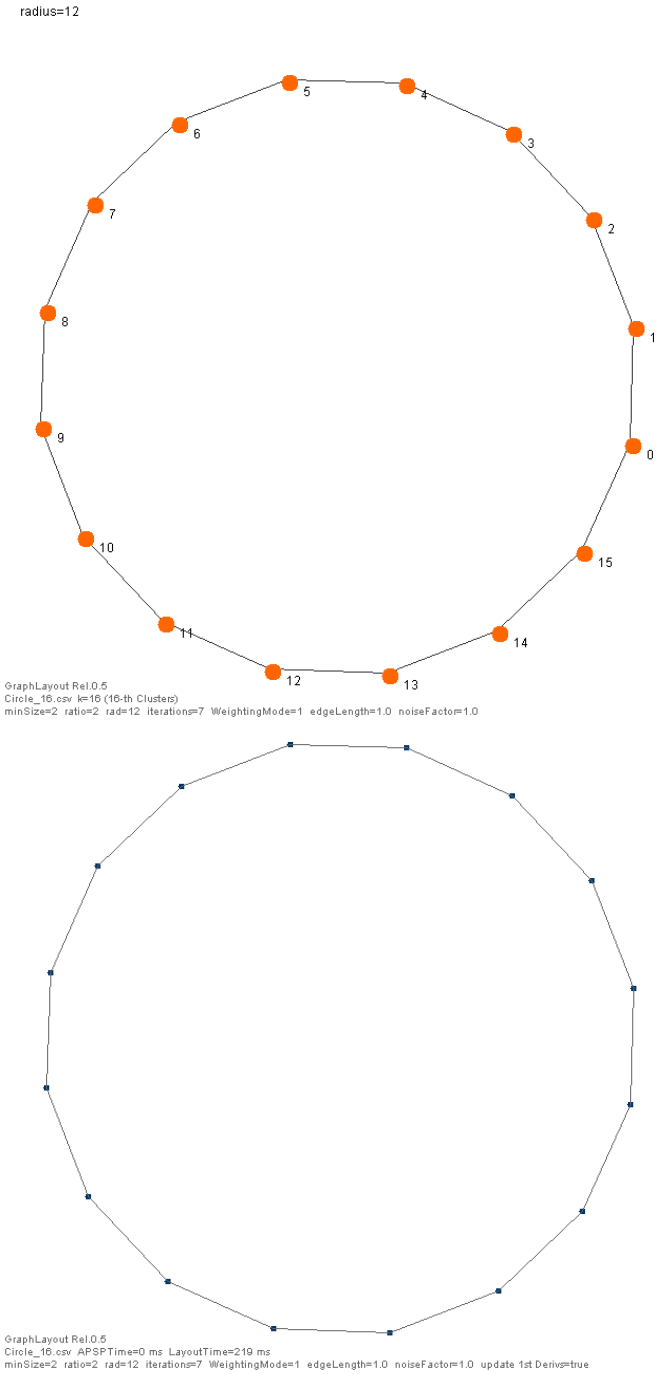


Abbildung 5.7: Das zufällige Start-Layout, die Layouts der Cluster G^2, G^4, G^8, G^{16} und das End-Layout des Circle_16-Graphen, berechnet mit dem Algorithmus von [HK02].

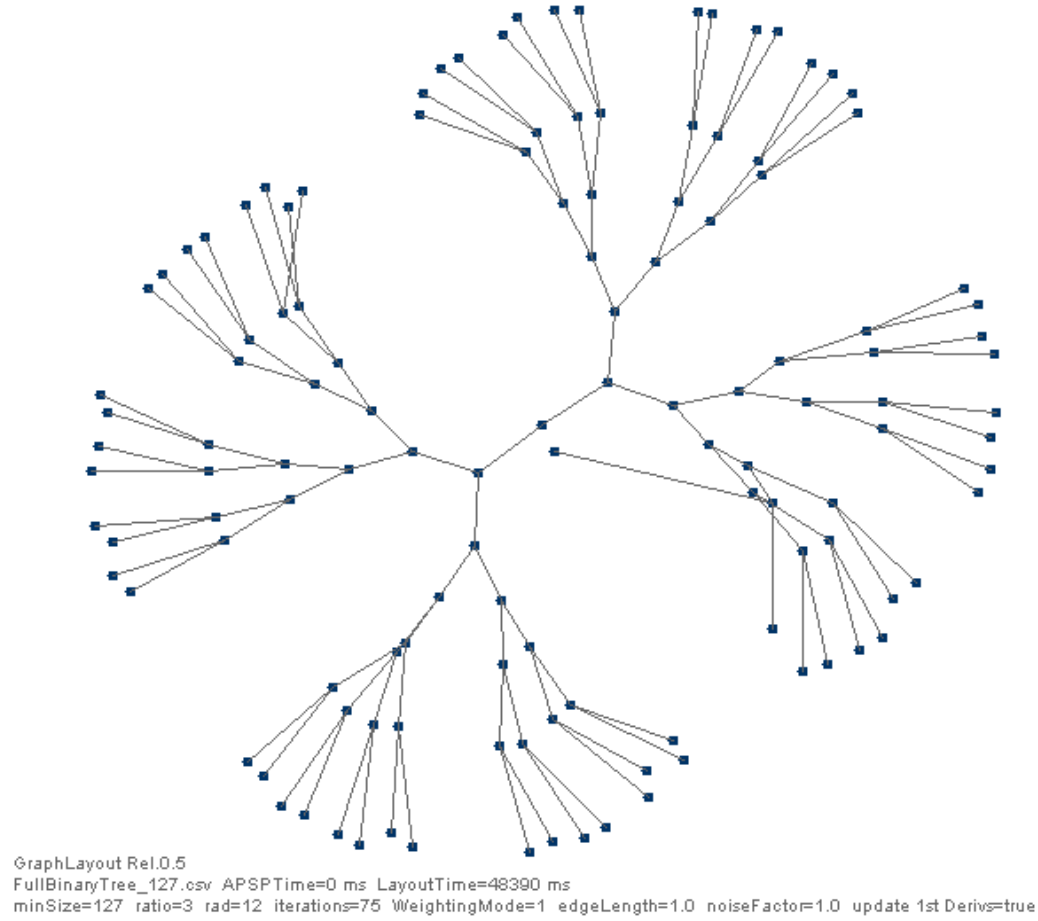


Abbildung 5.8: Mit grossem Aufwand erzeugtes, fast perfektes Layout eines FullBinaryTree_127.

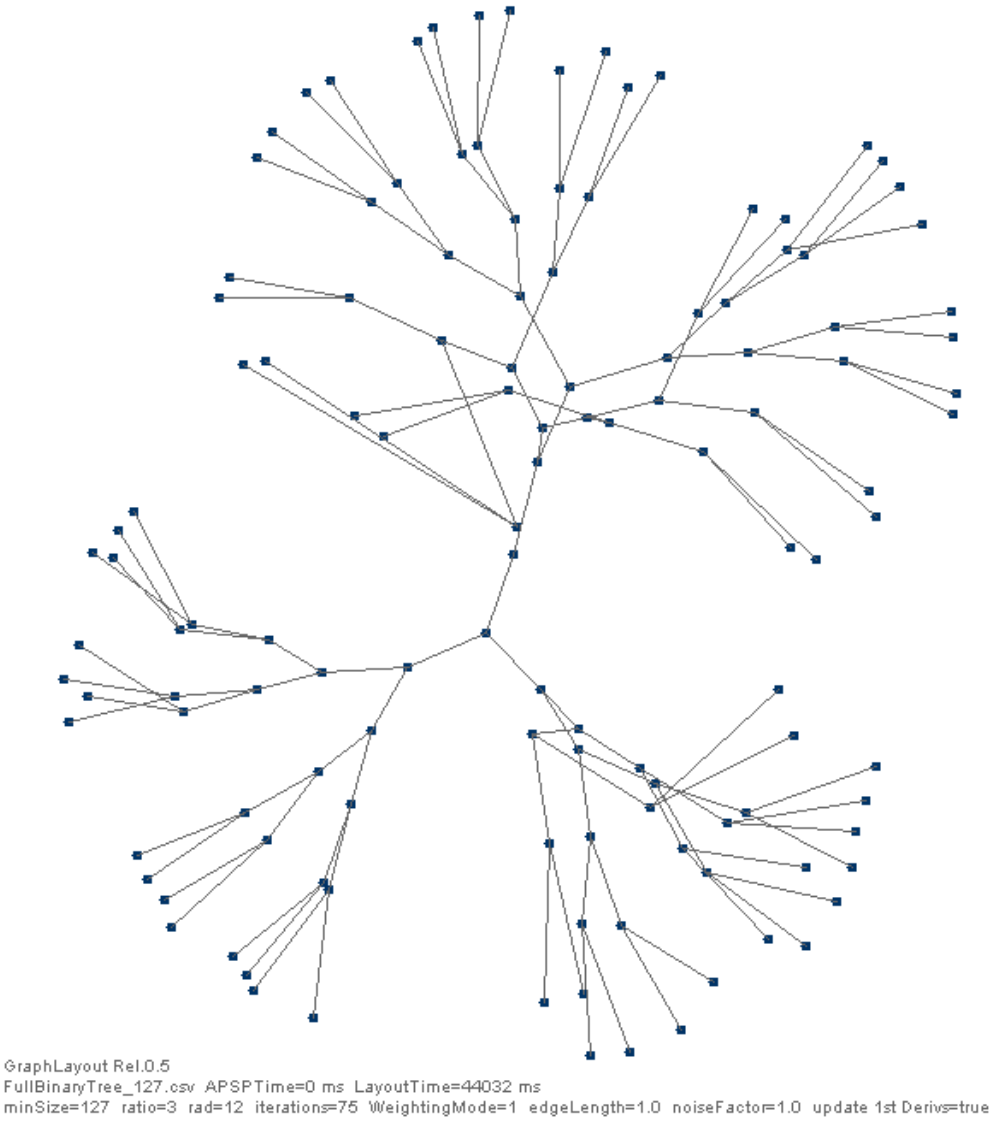


Abbildung 5.9: Mit identischen Parametern wie in Abb. 5.8 erzeugtes Layout. Auch die Instabilität bzgl. des zufällig erzeugten Start-Layouts ist nicht unproblematisch.

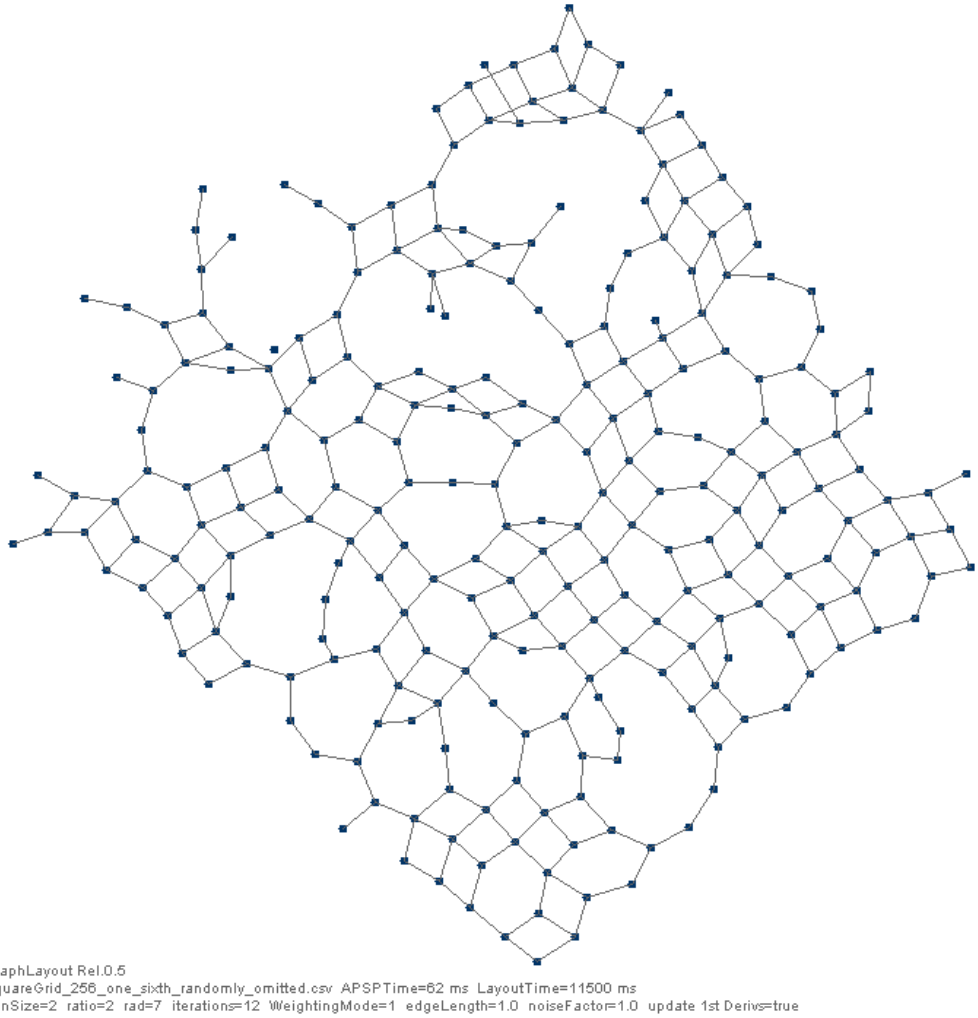


Abbildung 5.10: 256-Knoten-SquareGrid, bei dem ca. 1/6 der Kanten zufällig entfernt wurden.

Kapitel 6

Schlussfolgerungen

Abschliessend folgt ein Überblick über meine wichtigsten Schlussfolgerungen zum Multiskalen-Ansatz von [HK02]:

1. Das Multiskalen-*Schema* ist recht schlüssig, wird mathematisch streng hergeleitet und ist deshalb gut begründbar.

2. Die anschliessenden Vereinfachungen des Schemas finde ich ziemlich problematisch: Es ist nicht ganz nachvollziehbar, wieso Harel und Koren im *Algorithmus* die solide mathematische Basis aufgeben und das Schema unkontrolliert abändern. Die Feststellung, dass „der Algorithmus auch so funktioniert“ finde ich unbefriedigend. „Wieviel“ Multiskalen-*Schema* enthält der Algorithmus noch? Dass diese Frage berechtigt ist, hoffe ich durch die vorgehende Untersuchung gezeigt zu haben. Es wäre interessant, einen „strikten“ Multiskalen-Algorithmus zu implementieren und zu vergleichen.

3. Der Algorithmus ist sicher eindrücklich schnell. Dies wurde aber möglicherweise durch weniger Stabilität und mehr Fälle, in denen die Korrektheit nicht mehr gegeben ist, erkauft.

4. Wie ebenfalls gezeigt, gibt es gewisse Präzisierungen im Algorithmus, die notwendig sind.

5. Die Methode ist ziemlich weit davon entfernt, „automatisch“ ein schönes Layout zu generieren. In vielen Fällen muss zuerst ausgiebig probiert werden, welche Parameter-Einstellungen ein gutes Resultat liefern. Interessant könnte hier sein zu untersuchen, ob aus graphentheoretischen Eigenschaften „gute“ Parameterwerte berechnet werden können.

6. Weiter sei auch nochmals erwähnt, dass bei der Energieminimierung durch das Gradientenabstiegsverfahren nie garantiert werden kann, dass der Algorithmus nicht in einem relativ grossen, unvorteilhaften lokalen Minimum steckenbleibt. Dies hängt gerade auch von der Struktur des betrachteten Graphen ab. (Je dichter der Graph, desto „gebirgiger“ die Energiefunktion und desto grösser diese Gefahr?).

7. Das Verfahren ist auch relativ instabil bzgl. des zufällig erzeugten Start-Layouts.

8. Ein etwas anderer, rein technischer Aspekt, der trotzdem interessant ist, wäre ein direkter Vergleich der Laufzeit der Java-Implementierung mit derjenigen in C/C++. Gemäss [HK02] ist ihre C-Implementierung immer noch schneller

als meine in Java, obwohl mein Rechner schneller ist. Dafür müsste natürlich sichergestellt werden, dass die Details beider Implementierungen möglichst identisch sind.

Literaturverzeichnis

[Gu96] Güting R.-H.: Datenstrukturen. FernUniversität Hagen, Kurs 1663 (1996)

[HH99] Hadany R. & Harel D.: A Multi-Scale Algorithm for Drawing Graphs Nicely. Proc. 25th Inter. Workshop on Graph-Theoretic Concepts in Computer Science (WG'99), 1999.

[HK02] Harel D. & Koren Y.: A Fast Multi-Scale Method for Drawing Large Graphs. Journal of Graph Algorithms and Applications, vol.6 no.3 (2002), pp. 179-202

[KK89] Kamada T. & Kawai S.: An Algorithm for Drawing General Undirected Graphs. Information Processing Letters 31 (1989), 7-15

[Kn98] Knuth D.: The Art of Computer Programming. Addison-Wesley, Vol. 1-3 (1998)

[Lo98] Locher F.: Numerische Mathematik I. FernUniversität Hagen, Kurs 1271 (1998)

[Lo00] Locher F.: Numerische Mathematik II. FernUniversität Hagen, Kurs 1372 (2000)

[Ot96] Ottmann T. & Widmayer P.: Algorithmen und Datenstrukturen. Spektrum Akad. Verlag (1996)

[Se92] Sedgewick R.: Algorithmen. Addison-Wesley 1992

[Se02] Seip U.: Graphentheorie. FernUniversität Hagen, Kurs 1306 (2002)