Implementierung eines Spring-Embedder-Verfahrens in JSXGraph

Universität Bayreuth

Lehrstuhl für Mathematik und ihre Didaktik

Bachelorarbeit

Themensteller: Prof. Dr. A. Wassermann

Lena Freudenberger

26. August 2010

Inhaltsverzeichnis

1 Einleitung **3**

2 Ästhetikkriterien **6**

3 Theoretische Grundlagen **7**

4 Spring-Embedder-Verfahren **10**
4.1 Eades (1984) . . . 10
4.2 Kamda & Kawai (1989) . . . 10
4.3 Davidson & Harel (1996) . . . 11
4.4 Fruchterman & Reingold (1991) . . . 11
4.5 Frick et al. (1995) . . . 11
4.6 Tunkelang (1994) . . . 12

5 Algorithmische Details **13**

6 JSXGraph **17**

7 Erweiterungen **19**
7.1 Zufallsgraph . . . 19
7.2 Prüfung auf Zusammenhang . . . 20
7.3 Erweiterung auf nicht-zusammenhängende Graphen . . . 21
7.4 Scramble-Funktion . . . 21

8 Implementierung in JSXGraph **23**
8.1 HTML-Hauptdatei . . . 23
8.2 Initialisierung des Zufallsgraphen . . . 25
8.3 Scramble . . . 28
8.4 Floyd-Algorithmus . . . 28
8.5 Prüfung auf Zusammenhang . . . 29
8.6 Weitere Funktionen . . . 29
8.7 Algorithmusfunktion . . . 35
8.8 Hauptfunktion . . . 37

9 Bewertung **39**
9.1 Erfüllung der Ästhetikkriterien . . . 39
9.2 Anwendung . . . 40
9.2.1 Mit Zufallsgraphen . . . 40
9.2.2 Mit vorgegebenen Adjazenzmatrizen . . . 41
9.3 Laufzeit . . . 42
9.4 Verwendung der Konstanten . . . 44

10 Ausblick **46**

11 Zusammenfassung **47**

12 Literatur **48**

A Danksagung **50**

B Eidesstattliche Erklärung **51**

C Anhang: CD-Rom **52**

1 Einleitung

"Ein Bild sagt mehr als tausend Worte", so lautet ein chinesisches Sprichwort (Forster 1999: 1). Das ist der Grund dafür, dass in vielen wissenschaftlichen Bereichen Graphiken gezeichnet werden, um Strukturen und abstrakte Informationen vereinfacht und anschaulich darzustellen. Oft sind diese Bilder Graphen, Gebilde aus Knoten und Kanten, die komplexe Sachverhalte auf das Wesentliche reduzieren, sodass sie mit einem Blick erfassbar werden. Bekannte Beispiele aus dem Alltag sind Molekülstrukturen (Abb. 1), Streckennetze öffentlicher Verkehrsmittel, soziale Netzwerke wie Stammbäume etc.

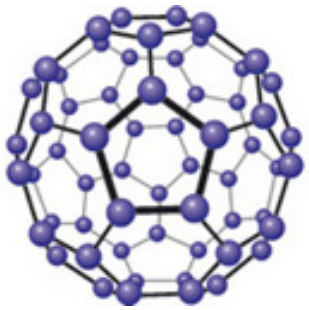

Abb. 1: Schematische Darstellung der Kristallstruktur von C60-Fulleren (Vohrer 2007)

Der Vorgang der Graphenplatzierung wurde im Laufe der Zeit automatisiert. Es etablierte sich in der Theoretischen Informatik der Bereich "Graph Drawing", um Wege zu finden, Graphen aus gegebenen Daten ästhetisch ansprechend und automatisch zeichnen zu können. Dafür werden Algorithmen entwickelt, analysiert, implementiert und evaluiert (Brandenburg et al. 1997). Die einzelnen Layouts unterscheiden sich im Wesentlichen in der Kantenführung, Knotenverteilung und Symmetrie. Welche für den Graph die richtige ist, hängt von der Aussage des Modells ab, das dahinter steht. Entsprechend dazu wird ein Algorithmus genutzt oder neu entwickelt. Abbildung 2 zeigt 5 verschiedene Darstellungen des vollständigen Graphen K_4, bestehend aus 4 Knoten und 6 Kanten, so dass jeder Knoten mit jedem anderen verbunden ist (Jungnickel 1987: 14). Falls jeder Knoten ein Teil einer polygonalen Kette aus Verbindungslinien ist, spricht man von "Polyline Drawing" (Abb. 2a). Sind die Kanten zusätzlich geradling, so handelt es sich um "Straight-line Drawing" (Abb. 2b). "Orthogonal Drawing" (Abb. 2c) ist ein Polyline Drawing, das ausschließlich aus horizontalen und vertikalen Kanten besteht. Enthält der Graph keine Kantenkreuzungen, so ist er planar (Abb. 2d) (Di Battista 1994: 2f.). Zeichnungen gerichteter Graphen mit richtungsweisenden Pfeilen als Kanten, die in vertikaler Richtung ansteigen, wie in Abbildung 2e, heißen "Upward Drawing" (Tunkelang 1999: 12).

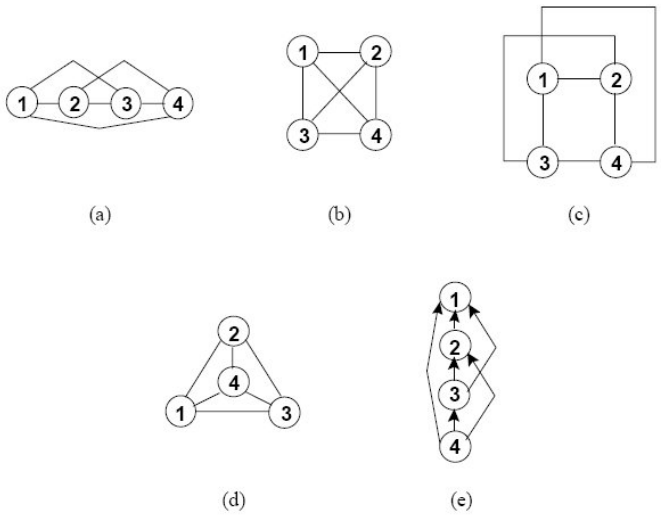

Abb. 2: (a) Polyline Drawing, (b) Straight-line Drawing, (c) Orthogonal Drawing, (d) Planar Drawing, (e) Upward Drawing (Tunkelang 1999: 13)

Es existieren bereits zahlreiche Graph-Drawing-Algorithmen, die die Platzierung von Knoten und Kanten regeln. Viele dieser Verfahren beschränken sich auf Graphen mit speziellen Eigenschaften und erzeugen Layouts, die ausgewählte Ästhetikkriterien erfüllen. Eine Reihe von Algorithmen sind in Di Battista et al. (1998) beschrieben. Beispielsweise ist der Algorithmus von De Fraysseix et al. (1990) nur für planare Graphen. Das Resultat (Abb. 3, Mitte) ist ein planarer Graph, dessen Knoten auf einem Zeichengitter liegen, also die Knotenpositionen ganzzahlige Koordinaten haben (De Fraysseix et al. 1990). Hierbei werden die Knoten zunächst kanonisch geordnet (Abb. 3, links oben) und der Graph gemäß dieser Ordnung inkrementell gezeichnet (Wagner & Brandes 2009).

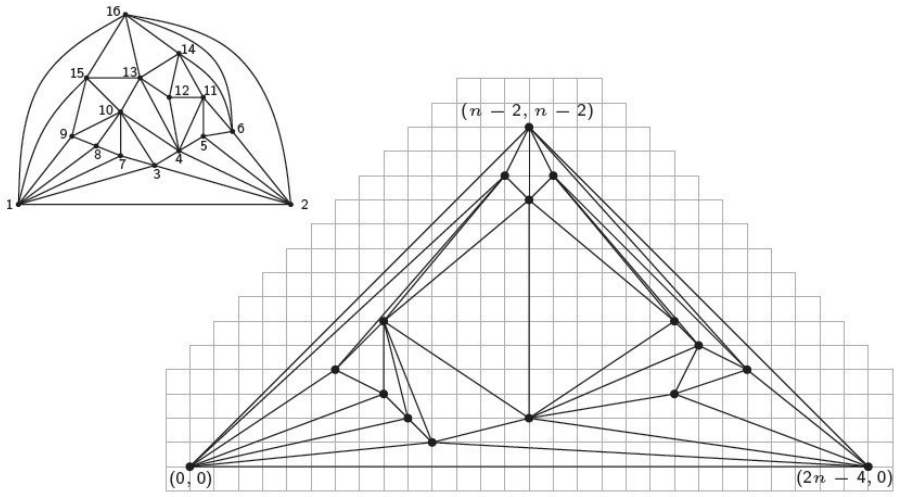

Abb. 3: Links oben: Kanonische Ordnung. Mitte: Planare Darstellung auf Zeichengitter (Nöllenburg 2009)

Die "Sugiyama"-Methode setzt gerichtete azyklische Graphen so, dass die hierarchische Struktur deutlich wird (Abb. 4). Die Kantenknicke und Knoten liegen auf Schichten. Die Kanten zeigen in eine Richtung. Die Platzierung erfolgt in drei großen Schritten: Die Verteilung der Knoten auf Schichten, die Kreuzungsreduktion der Kanten und die Positionierung der Knoten (Mutzel 2005).

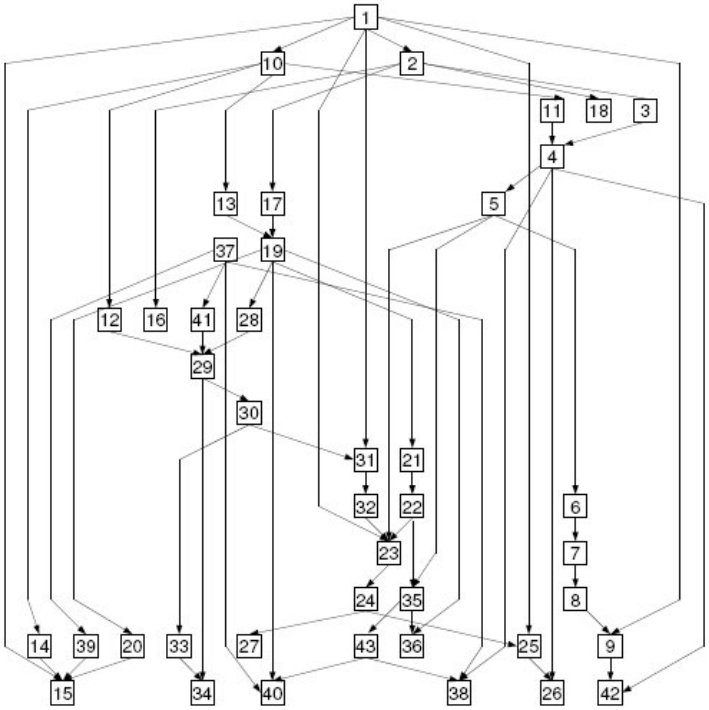

Abb. 4: Eine hierarchische Zeichnung eines gerichteten Graphen aus Mutzel (2005)

In dieser Arbeit wird das Spring-Embedder-Verfahren nach Kamada & Kawai (1989) zur Implementierung gewählt, das einen aus graphentheoretischer Sicht ungewöhnlichen Ansatz verfolgt. Der zusammenhängende Graph wird als physikalisches System aus sich abstoßenden Knoten und Federn als Verbindungslinien modelliert. Die auftretenden Kräfte verschieben die Knoten so, dass die Gesamtenergie minimiert wird. Auf diese Weise erhält man ein ansprechendes Layout.

Die Implementierung erfolgt in JSXGraph, einer auf JavaScript basierenden Cross-Browser-Bibliothek. JSXGraph wurde zur Visualisierung dynamischer Mathematik entwickelt und dient unter anderem als Funktionsplotter und zur Erzeugung von Diagrammen (Wassermann et al. 2010). Aufgrund der einfachen Programmierbefehle, setzt JSXGraph keine Programmierkenntnisse voraus. Ziel dieser Arbeit ist es, die Anwendungsmöglichkeiten dieser Software für dynamische Mathematik auf den Bereich Graph Drawing zu erweitern.

2 Ästhetikkriterien

Bevor auf die Art und Weise der Platzierung von Knoten und Kanten eingegangen wird, muss der Begriff "Ästhetik" geklärt werden, der bei Graph Drawing eine wesentliche Rolle spielt. Die Geschmäcker der Menschen sind verschieden, deswegen wird man keine allgemeingültige Definition für schöne Layouts finden können, die dieses Kriterium exakt festlegen. Man kann hier jedoch versuchen eine Richtung anzugeben, die dies charakterisiert. Im Folgenden werden dazu einige wichtige Bestandteile betrachtet (Forster 1999: 3):

- Knotenverteilung:
  Die Knoten sollten möglichst gleichmäßig über die Zeichenfläche verteilt werden, so dass die Fläche optimal ausgenutzt wird und die Abbildung nicht gedrängt wirkt. Knoten, die durch eine Kante verbunden sind, sollten näher beieinander liegen als Knoten die nicht adjazent sind, um ihre Verknüpfung zu unterstreichen. Trotzdem müssen alle Knoten voneinander trennbar sein und dürfen in keinem Fall übereinanderliegen.
- Kantenführung:
  Besonders übersichtlich wirken planare Graphen, die keine Kantenkreuzungen enthalten, oder solche, die möglichst wenig aufweisen. Die Anzahl der Kantenknicke ist zu minimieren. Außerdem muss auf die Kantenlänge geachtet werden. Diese sollte möglichst einheitlich sein.
- Symmetrie:
  Die Darstellung des Graphen sollte, wenn möglich, symmetrisch sein.
- Flächenbedarf:
  Das Layout darf nicht zu klein sein, aber gerade nur so viel Platz auf der Zeichenfäche einnehmen, damit es noch mit einem Blick erfassbar ist.

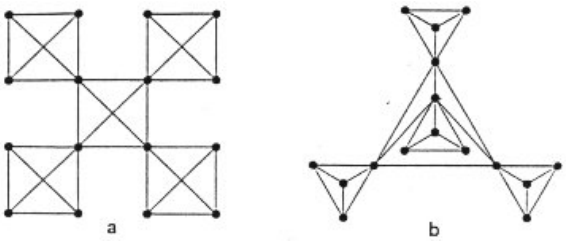

Abb. 5: Symmetrische u. planare Zeichnung eines Graphen (Kamada & Kawai 1989: 8)

Alle Kriterien können nicht gleichzeitig erfüllt werden, da sie sich sogar teilweise widersprechen (Abb. 5). Deswegen muss für jeden Graphen und dessen Anwendung überlegt werden, welche Ästhetikkriterien stärker gewichtet werden müssen als andere, um die beste Graphenplatzierung zu finden (Forster 1999: 3).

3 Theoretische Grundlagen

Um die einzelnen Bestandteile des Spring-Embedder-Verfahrens zu begreifen, bedarf es der Kenntnis einiger graphentheoretischer Begriffe, die an dieser Stelle nach Jungnickel (1987) definiert werden.

Definition: Graph

Ein Graph G ist ein Paar $G = (V, E)$ aus einer endlichen Menge $V \neq \emptyset$ und einer Menge E von zweielementigen Teilmengen von V. Die Elemente von V heißen Punkte (auch Knoten oder Ecken), die Elemente von E Kanten. Für eine Kante $e = \{a, b\}$ sind a und b die Endpunkte von e; man sagt, dass a und b mit e inzident und dass a und b adjazent sind (Jungnickel 1987: 14).

Definition: Kantenzug

Es sei $(e_1, \ldots, e_n)$ eine Folge von Kanten eines Graphen G. Wenn es Punkte $v_0, \ldots, v_n$ mit $e_i = v_{i-1}v_i$ für $i = 1, \ldots, n$ gibt, wird diese Folge Kantenzug genannt. Wenn e_i paarweise verschieden sind, handelt es sich um einen Weg (Jungnickel 1987: 17).

Definition: Zusammenhang

Zwei Punkte a und b eines Graphen G heißen verbindbar, wenn es einen Kantenzug mit Anfangspunkt a und Endpunkt b gibt. Wenn je zwei Punkte von G verbindbar sind, spricht man von einem zusammenhängenden Graphen G. Für jeden Knoten a existiert (a) ein trivialer Kantenzug der Länge 0. Jeder Punkt ist also mit sich selbst verbindbar. Der Graph in Abbildung 6 ist ein nicht-zusammenhängender Graph, da Knoten 5 und 6 nicht mit Knoten 1-4 verbunden sind. Die Verbindbarkeit ist eine Äquivalenzrelation auf der Punktmenge V von G. Die Äquivalenzklassen dieser Relation heißen die Zusammenhangskomponenten von G. Somit ist G genau dann zusammenhängend, wenn ganz V eine Zusammenhangskomponente ist. Einpunktige Komponenten werden auch isolierte Punkte genannt (Jungnickel 1987: 18).

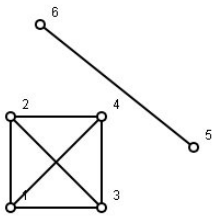

Abb. 6: Nicht-zusammenhängender Graph

Definition: Adjazenzmatrix

Ein Graph auf $1, \ldots, n$ (Abb. 7, rechts) wird durch eine $(n \times n)$-Matrix $A = (a_{ij})$ (Abb. 7, links) mit Einträgen 0 und 1 spezifiziert: (i, j) ist genau eine Kante, wenn $a_{ij} = 1$ ist. A ist Adjazenzmatrix von G (Jungnickel 1987: 38f.).

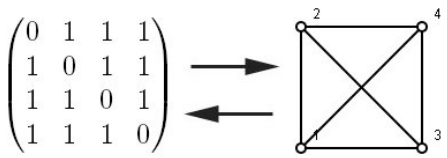

Abb. 7: Adjazenzmatrix mit zugehörigem Graphen

Definition: Kürzeste Wege

Gegeben sei ein Graph $G = (V, E)$ mit einer Abbildung $w : E \to \mathbb{R}$. Die Zahl $w(e)$ ist die Länge der Kante e. Für jeden Kantenzug $(e_1, \ldots, e_n)$ in G ist die Länge die Summe der einzelnen Kantenlängen, also $w(e_1) + \cdots + w(e_n)$. Die Länge des kürzesten Weges zweier Punkte a und b in G wird mit $d(a, b)$ bezeichnet (Jungnickel 1987: 51f.).

Definition: Endliche metrische Räume

Ein metrischer Raum ist ein Paar (X, d) aus einer Menge X und einer Abbildung $d : X^2 \to \mathbb{R}^+$, das den folgenen Bedingungen genügt (für alle x, y, z aus X):

1. $d(x, y) \geqslant 0$ mit $d(x, y) = 0$ genau für $x = y$
2. $d(x, y) = d(y, x)$
3. $d(x, z) \leqslant d(x, y) + d(y, z)$ (Dreiecksungleichung)

Die Matrix $D = (d(x, y))_{x,y \in X}$ heißt die Abstandsmatrix von (X, d) (Jungnickel 1987: 54).

Lemma

"$G = (V, E)$ sei ein Graph mit einer positiven Längenfunktion w. Dann ist (V, d) ein (endlicher) metrischer Raum, wobei der Abstand zweier Punkte wie [oben] definiert sei."(Jungnickel 1987: 54) .

Algorithmen

"Algorithmen sind Verfahren zur Lösung von Problemen"(Jungnickel 1987: 34). Nach Bauer & Wössner (1982) in Jungnickel (1987: 14) sollte ein Algorithmus die folgenden Eigenschaften haben:

1. Endliche Beschreibbarkeit: Verfahren kann mit einem endlichen Text beschrieben werden.
2. Effektivität: Jeder einzelne Schritt des Verfahrens muss mechanisch durchführbar sein.
3. Endlichkeit: Das Verfahren muss für jeden Einzelfall nach endlich vielen Schritten abbrechen.
4. Determinismus: Für jeden Einzelfall muss die Reihenfolge der Schritte eindeutig festliegen.

Desweiteren sollte ein Algorithmus für jeden Einzelfall das Problem korrekt lösen (Jungnickel 1987: 34).

4 Existierende Spring-Embedder-Verfahren

4.1 Grundidee von Eades (1984)

Die grundlegende Idee stammt von Eades (1984). Er modelliert aus zusammenhängenden ungerichteten Graphen ein physikalisches System. Die Knoten betrachtet er als Stahlringe und die Kanten als eingebettete (engl. embedded) Federn (engl. spring), die dem Verfahren den Namen "Spring-Embedder" verleihen. Die Federn ziehen adjazente Knoten zueinander und die Stahlringe (oder auch elektrisch geladene Teilchen) stoßen nichtadjazente Knoten von einander ab (Abb. 8) (Eades 1984, Forster 1999: 9f.). Zu Beginn werden den Knoten zufällige Startkoordinaten zugewiesen und die Kräfte berechnet, die auf die einzelnen Knoten in der momentanen Position wirken. Die Bewegung der Knoten erfolgt in Richtung der berechneten Gesamtkraft. Die Weite der Verschiebung hängt von der Größe der auf die einzelnen Knoten wirkenden Kräfte ab. Dieses Vorgehen wird für eine feste Anzahl von Iterationen wiederholt. Eades (1984) weicht vom HOOK'schen Gesetz absichtlich ab und verwendet für die Federhärte statt linearer Funktionen logarithmische, da sonst auf zu weit von einander entfernte Knoten zu starke Kräfte wirken würden (Eades 1984).

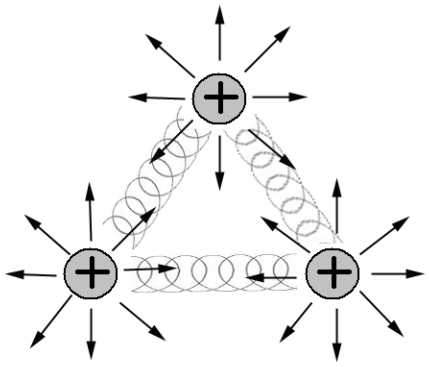

Abb. 8: Schematische Darstellung eines Spring-Embedder-Verfahrens

4.2 Verfahren nach Kamada & Kawai (1989)

Kamada & Kawai (1989) variieren das Grundprinzip von Eades (1984). Sie verwenden ausschließlich Federn und das zwischen allen Knoten des Graphen. Die Federhärten der einzelnen Federn nehmen "mit wachsendem graphentheoretischen Abstand der Knoten quadratisch ab" (Forster 1999: 10). Das Ziel ist das Erreichen eines Energieminimums. In jedem Iterationsschritt des Verfahrens wird genau ein Knoten ausgewählt, der den größten Fortschritt verspricht und wird solange verschoben, bis dieser Knoten seine momentane energieminimale Lage erreicht. Der Algorithmus stoppt, falls die Größe des Fortschrittswertes unter einen vorgegebenen Stopp-Wert fällt (Kamada & Kawai 1989).

4.3 Algorithmus nach Davidson & Harel (1996)

Das Spring-Embedder-Verfahren nach Davidson & Harel (1996) basiert auf Simulated Annealing, einer flexiblen Optimierungsmethode, die erfolgreich auf kombinatorische Optimierungsprobleme angewendet wird. Aus einem gegebenem Suchraum von Konfigurationen wird ein bezüglich einer Kostenfunktion minimales Element gesucht (Forster 1999: 11, Davidson & Harel 1996: 303). Der Suchraum von Konfigurationen sind die möglichen Darstellungen des Graphen, und die Kostenfunktion beinhaltet die gewünschten Merkmale des gesuchten Layouts. Es werden kleine zufällige Änderungen an der Startkonfiguration des Ausgangsgraphen vorgenommen. Ist die Kostenfunktion nach der Änderung günstiger, so wird der Zustand übernommen, ist das nicht der Fall, entscheidet der Zufall, ob die Änderung übernommen wird oder nicht. Die Wahrscheinlichkeit, Verschlechterungen zu akzeptieren (Temperatur) wird im Laufe des Verfahren kontinuierlich gesenkt (Forster 1999: 11). "Simulated Annealing ist strenggenommen kein Springembedder-Verfahren" (Forster 1999: 11).

4.4 Methode nach Fruchterman & Reingold (1991)

Fruchterman & Reingold (1991) kombinieren Simualted Annealing mit dem Spring-Embedder-Verfahren. Die Temperaturen aus Simulated Annealing werden zur Verbesserung des Konvergenzverhaltens verwendet, in dem dadurch geregelt wird, wie weit die Knoten bewegt werden. Die abstoßenden Kräfte wirken zwischen allen Knoten, anziehenden Kräfte nur auf adjazente Knotenpaare. Die Knoten werden in Richtung der auf sie wirkenden Kraft verschoben (Forster 1999: 11f.).

4.5 Graph EMbedder nach Frick et al. (1995)

Graph EMbedder entwickelten Frick et al. (1995) aus dem Verfahren von Fruchterman & Reingold (1991). Die anziehenden Kräfte wurden so verändert, dass sich Knoten mit großem Knotengrad, sozusagen schwerere Knoten, langsamer bewegen als leichtere Knoten, also Knoten mit kleinerem Knotengrad. Außerdem wurde eine weitere Kraft eingeführt, eine Art Gravitationskraft, die alle Knoten in Richtung des Schwerpunktes des Systems zieht. Diese bringt einen Vorteil für unzusammenhängende Graphen, da die einzelnen Teilgraphen dann nicht zu weit auseinander driften. Desweiteren wurde die Anzahl der Iterationen reduziert, in dem die Knoten entsprechend ihrer vorherigen Verschiebung bewertet werden und dementsprechend verschoben werden. Dies verhindert mögliche Rotationen und Oszillationen einzelner Knoten des Systems. In jedem Iterationsschritt werden alle Knoten in zufälliger Reihenfolge gemäß der vorher berechneten Verschiebungskonstante verschoben (Wagner & Brandes 2009: 9ff.).

4.6 Algorithmus nach Tunkelang (1994)

Tunkelang (1994) verwendet eine andere Herangehensweise. Für das Anfangslayout werden die Knoten des Graphen auf eine bestimmte Art und Weise ("minimal height spanning tree") geordnet und nacheinander gesetzt. Für jeden Knoten und deren Nachbarknoten werden lokal optimale Positionen gesucht. Das "Fine-Tuning" bringt zum Schluss alle Knoten in ihre lokal optimale Lage. Zur Platzierung und Bewegung der Knoten wird eine Kostenfunktion verwendet, der die Kräfte und die gewünschten Ästhetikkriterien zugrunde liegen (Tunkelang 1994).

In dieser Bachelorarbeit wird auf den Algorithmus von Kamada & Kawai (1989) eingegangen. Dieser liefert akzeptable Layouts (Brandenburg et al. 1996: 81) und kommt dem ursprünglichen, anschaulichen Spring-Embedder-Gedanken am nächsten.

5 Algorithmische Details zu Kamada & Kawai (1989)

Das Spring-Embedder-Verfahren von Kamada & Kawai (1989) beschäftig sich mit ungerichteten und gewichteten Graphen. Der Graph wird als physikalisches System modelliert, das ausschließlich aus Federn zwischen allen Knoten v_i und v_j besteht. Ziel ist es die Gesamtenergie E des Systems zu minimieren und dadurch ein ästhetisch schönes Layout zu erhalten. Die Federhärten k_{ij} der einzelnen Federn hängen vom graphentheoretischen Abstand $d_{ij} \neq 0$ der einzelnen Knoten zueinander ab.

$$k_{ij} = \frac{K}{d_{ij}^2} \qquad \text{mit der Konstanten} \quad K$$

Adjazente Knotenpaare ziehen sich somit wesentlich stärker an als Knotenpaare, die graphentheoretisch gesehen, weiter von einander entfernt liegen. Die graphentheoretisch kürzesten Abstände zwischen allen Knoten werden mit Hilfe des Floyd-Algorithmus (Algorithm 1) (Floyd 1962) berechnet. Dafür wird, wie in Abbildung 9 (unten links), die Matrix verwendet, die sich aus der Summe der Einheitsmatrix und der Adjazenzmatrix ergibt. Als Resultat erhält man eine Distanzmatrix (Abb. 9, unten rechts), die die kürzesten Wege zwischen allen Knoten widergibt.

Algorithm 1 Algorithmus nach Floyd (1962) in Sedgewick (1992: 541)

```
1:  for y = 1 to n do
2:      for x = 1 to n do
3:          if a[x, y] > 0 then
4:              for j := 1 to n do
5:                  if a[y, j] > 0 then
6:                      if (a[x, j] = 0) or (a[x, y] + a[y, j] < a[x, j]) then
7:                          a[x, j] := a[x, y] + a[y, j];
8:                      end if
9:                  end if
10:             end for
11:         end if
12:     end for
13: end for
```

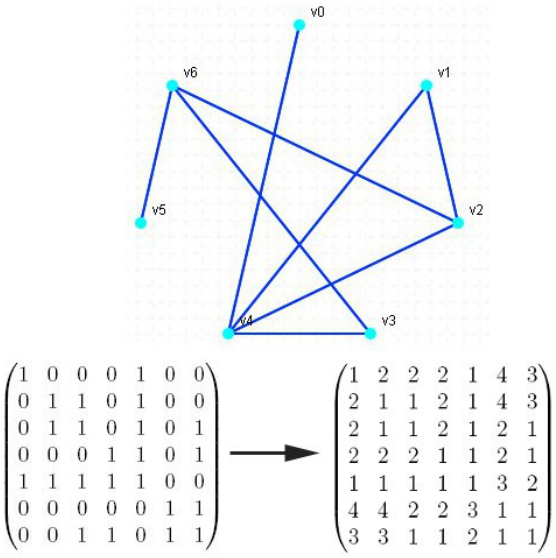

Abb. 9: Schematische Darstellung des Floyd-Algorithmus

Die Ausgangslängen der Federn l_{ij} werden so gesetzt, dass sowohl die Seitenlänge des quadratischen Displays L_0 als auch der längste graphentheoretische Weg $max_{ij}d_{ij}$ des Graphen berücksichtigt werden.

$$l_{ij} = L \cdot d_{ij}$$

mit

$$L = L_0 / max_{ij} d_{ij}$$

Daraus ergibt sich die Gesamtenergie des Systems in kartesischen Koordinaten wie folgt:

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{ij} \left\{ (x_i - x_j)^2 + (y_i - y_j)^2 + l_{ij}^2 - 2 l_{ij} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \right\}$$

Die globale Energierminimierung erweist sich bei diesem dynamischen System als "quite difficult" in der Implementierung und zu aufwendig im Rechenaufwand (Kamada & Kawai 1989: 9). Deswegen benutzen Kamada & Kawai (1989) die lokale Energieminimierung, die an folgende Bedingungen geknüpft ist:

$$\frac{\partial E}{\partial x_m} = \frac{\partial E}{\partial y_m} = 0 \qquad \text{für} \quad 1 \leqslant m \leqslant n$$

$$\frac{\partial E}{\partial x_m} = \sum_{i \neq m} k_{mi} \left\{ (x_m - x_i) - \frac{l_{mi}(x_m - x_i)}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{\frac{1}{2}}} \right\}$$

$$\frac{\partial E}{\partial y_m} = \sum_{i \neq m} k_{mi} \left\{ (y_m - y_i) - \frac{l_{mi}(y_m - y_i)}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{\frac{1}{2}}} \right\}$$

Es handelt sich hier um ein $2n$ nicht-lineares Gleichungssystem (mit Knotenanzahl n), das aufgrund der Abhängigkeit der Gleichungen nicht direkt mit dem $2n$-dimensionalen Newton-Raphson-Verfahren gelöst werden kann. Aufgrund dessen wird dieses Verfahren variiert und immer nur ein Knoten in Richtung Energieminimum verschoben, nämlich derjenige, der den größten Fortschritt verspricht bzw. den größten Δ_m-Wert hat.

$$\Delta_m = \sqrt{\left\{\frac{\partial E}{\partial x_m}\right\}^2 + \left\{\frac{\partial E}{\partial y_m}\right\}^2}$$

Der ausgewählte Knoten wird in die momentan optimale Lage verschoben. Die Verschiebung in x-Richtung δx und in y-Richtung δy ist durch die Lösung des folgenden linearen Gleichungssystems gegeben:

$$\frac{\partial^2 E}{\partial x_m^2}\delta x + \frac{\partial^2 E}{\partial x_m \partial y_m}\delta y = -\frac{\partial E}{\partial x_m}$$
$$\frac{\partial^2 E}{\partial y_m \partial x_m}\delta x + \frac{\partial^2 E}{\partial y_m^2}\delta y = -\frac{\partial E}{\partial y_m}$$

Mit den doppelten partiellen Ableitungen der Energie nach x- und y-Koordinaten:

$$\frac{\partial^2 E}{\partial x_m^2} = \sum_{i \neq m} k_{mi}\left\{1 - \frac{l_{mi}(y_m - y_i)^2}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{\frac{3}{2}}}\right\}$$
$$\frac{\partial^2 E}{\partial y_m^2} = \sum_{i \neq m} k_{mi}\left\{1 - \frac{l_{mi}(x_m - x_i)^2}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{\frac{3}{2}}}\right\}$$

$$\frac{\partial^2 E}{\partial x_m \partial y_m} = \frac{\partial^2 E}{\partial y_m \partial x_m} = \sum_{i \neq m} k_{mi}\frac{l_{mi}(x_m - x_i)(y_m - y_i)}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{\frac{3}{2}}}$$

Dies ergibt nach Auflösung des Gleichungssystems die Werte für δ_x und δ_y:

$$E_x = \frac{\partial E}{\partial x}, \quad E_{xx} = \frac{\partial^2 E}{\partial x^2}, \quad E_{xy} = \frac{\partial^2 E}{\partial x \partial y} = \frac{\partial^2 E}{\partial y \partial x}, \quad E_y = \frac{\partial E}{\partial y}, \quad E_{yy} = \frac{\partial^2 E}{\partial y^2}$$

$$\delta_x = -\frac{E_x}{E_{xx}} - \frac{E_{xy}}{E_{xx}}\left(\frac{-E_y + \dfrac{E_x \cdot E_{xy}}{E_{xx}}}{-\dfrac{E_{xy}^2}{E_{xx}} + E_{yy}}\right)$$

$$\delta_y = \frac{-E_y + \dfrac{E_x \cdot E_{xy}}{E_{xx}}}{-\dfrac{E_{xy}^2}{E_{xx}} + E_{yy}}$$

Dieser Knoten m wird solange verschoben bis der Wert von Δ_m unter einem vordefinierten Stopp-Wert liegt und somit bezüglich dieses Wertes optimal positioniert ist. Es wird ein neuer Punkt ausgewählt. Das Iterationsverfahren stoppt nach einer gewissen Anzahl von Iterationsschritten, z. B. $10 \cdot n$ (n sei Anzahl der Knoten), oder falls der Wert von Δ_m einen definierten Stopp-Wert unterschreitet (Kamada & Kawai 1989).

Algorithm 2 NACH KAMADA & KAWAI (1989)

1: Berechnung von d_{ij} für $1 \leqslant i \neq j \leqslant n$;
2: Berechnung von l_{ij} für $1 \leqslant i \neq j \leqslant n$;
3: Berechnung von k_{ij} für $1 \leqslant i \neq j \leqslant n$;
4: Initialisierung der Knoten $p_1, p_2, \ldots, p_n$;
5: **while** $\max_i \Delta_i > stopp$ **do**
6: Sei p_m der Knoten mit $\Delta_m = \max_i \Delta_i$;
7: **while** $\Delta_m > stopp$ **do**
8: Berechnung von δx und δy
9: $x_m := x_m + \delta x$;
10: $y_m := y_m + \delta y$
11: **end while**
12: **end while**

Die Erfahrung zeigt, dass die Wahl des Anfangslayouts in der Regel keinen großen Einfluss auf das Resultat ausübt. Es gilt jedoch den Spezialfall, dass viele Knoten auf einer Linie liegen, zu vermeiden. Für die Startposition der n Punkte (Algorithm 2, Z. 4) werden die Ecken eines regulären n-Ecks (Abb. 10-11) empfohlen.

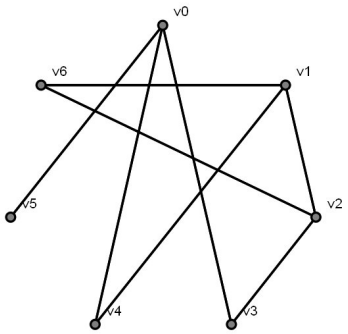

Abb. 10: Beispiel eines Anfangslayouts eines Graphen mit 7 Knoten

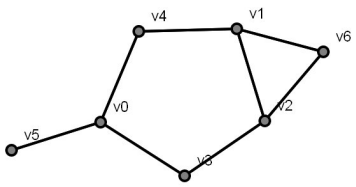

Abb. 11: Layout des Graphen aus (Abb. 10) nach Verwendung des Algorithmus

6 JSXGraph

JSXGraph ist eine Software zur Visualisierung dynamischer Mathematik. Sie dient unter anderem als Funktionsplotter sowie zur Erzeugung von Diagrammen im Webbrowser (Wassermann et al. 2010). Wie bereits die dynamische Geometriesoftware GEONExT (Baptist et al. 2007) wurde JSXGraph an der Universität Bayreuth entwickelt. JSXGraph gehört zu den Cross-Browser-Bibliotheken, d. h. die Ausgabe der Dateien ist unahängig von der Wahl des Webbrowsers. Im Moment werden Mozilla Firefox (seit Version 2.0), Opera, Safari (seit Version 3), Google Chrome (alle Versionen) sowie Microsoft Internet Explorer (seit Version 6) unterstützt. JSXGraph basiert auf der Programmiersprache JavaScript und nutzt zur Darstellung die Vektorzeichenformate SVG (Scalable Vector Graphics) sowie VML (Vector Markup Language) (Koch 2009: 282f.). Die Bibliothek ist in eine HTML-Datei einfach einzubinden und weitere Plug-Ins werden nicht benötigt (Gerhäuser 2010). Bei der Anwendung wird, wie im folgenden Quellcode gezeigt, eine HTML-Datei erstellt, die die Bibliothek einbindet (Quellcode, Zeile 2-10) und die gewünschten JSXGraph-Befehle enthält. Diese Datei wird dann in einem der unterstützten Browser aufgerufen (Wassermann et al. 2010).

Vorlage eines JSXGraph-Quellcodes

```
1  <html>
2  <head>
3   <title>JSXGraph Konstruktionstemplate</title>
4   <link rel="stylesheet" type="text/css"
5    href="http://jsxgraph.uni-bayreuth.de/distrib/jsxgraph.css" />
6
7   <script type="text/javascript"
8    src="http://jsxgraph.uni-bayreuth.de/distrib/jsxgraphcore.js">
9   </script>
10 </head>
11 <body>
12  <div id="jxgbox" class="jxgbox"
13   style="width:500px;height:500px;"></div>
14  <script type="text/javascript">
15   /* <![CDATA[ */
16
17  board = JXG.JSXGraph.initBoard('jxgbox',
18   boundingbox: [-2, 16, 16, -2], axis: true, grid: false,
19   keepaspectratio: true, showcopyright: false});
20
21 /* Hier werden Programmierbefehle von JSXGraph verwendet. */
22
23  /* ]]> */
24  </script>
25 </body>
26 </html>
```

Die großen Vorteile hiervon sind zu einem die geringe Größe von weniger als 100 KByte bei einer Einbettung in eine Website, zum anderen die Unterstützung von GEONExT und Intergeo-Dateien (Wassermann et al. 2010). Außerdem ist es bereits möglich, JSXGraph mit einem Apple iPhone oder iPod touch zu nutzen und bleibt somit nicht nur für den Nutzer zu Hause zugänglich, sondern kann unterwegs genutzt werden. Das Erlernen des Umgangs mit JSXGraph ist aufgrund der einfach gestalteten Befehle extrem leicht und setzt keinerlei Programmiererfahrung voraus. Anhand der vielen Beispiele auf jsxgraph.uni-bayreuth.de, ist es für den Laien durchaus möglich, schnell zu erlernen und eigene Ideen umzusetzen. Vor allem für Schulen wird JSXGraph große Vorteile bringen. Sowohl für die Lehrkraft, der es ein einfaches und umfangreiches Hilfmittel ist, um Mathematik zu visualisieren, als auch für den Schüler, dem die Möglichkeit gegeben wird, Mathematik virtuell zu entdecken. JSXGraph bietet somit einen spielerischen Einstieg in die Programmierung. Um die große Vielfalt von JSXGraph aufzuzeigen und um weitere Beispiele zu geben, wurde das Spring-Embedder-Verfahren ausgesucht und für diese Arbeit implementiert.

7 Erweiterungen

7.1 Zufallsgraph

Der Algorithmus nach Kamada & Kawai (1989) benötigt als Anfangsdaten die Anzahl der Knoten und Informationen über die Kanten, die durch eine Adjazenzmatrix gegeben werden. Um das Verfahren an möglichst vielen unterschiedlichen Graphen zu testen, wurde eine Funktion implementiert, die Zufallsgraphen mit gewünschter Knotenanzahl und Wahrscheinlichkeit für die Adjazenz erstellt. Die Daten werden vom Benutzer in die Eingabefelder "number of vertices" und "probabilty of adjacency" eingegeben und daraus eine Adjazenzmatrix erstellt. Die Anzahl der Knoten bestimmt die Größe der quadratischen Matrix, mit der Funktion *Math.random* wird anschließend jedem Matrixeintrag einen zufälligen Wert zwischen 0 und 1 zu geordnet. Liegt dieser Zufallswert unter dem eingegebenen Wahrscheinlichkeitswert, wird dieser Position der Wert "0" zu gewiesen, andernfalls der Wert "1". Aus dieser Information wird ein Anfangslayout erstellt. Natürlich erhält man auf diese Weise nicht zwingend zusammhänge Graphen (Abb. 12-13).

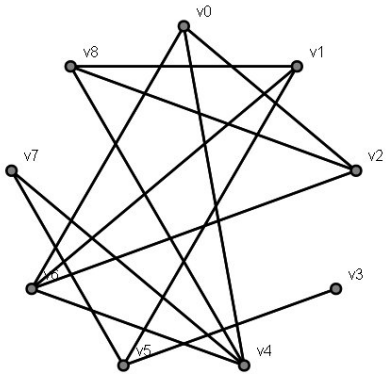

Abb. 12: Zusammenhängender Zufallsgraph

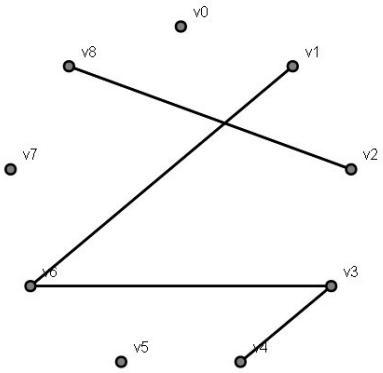

Abb. 13: Nicht-zusammenhängender Zufallsgraph

7.2 Prüfung auf Zusammenhang

Da das Verfahren nach Kamada & Kawai (1989) zusammenhängende Graphen als Input benötigt, muss bei den wie oben erstellten Zufallsgraphen geprüft werden, ob diese zusammenhängend sind oder nicht. Falls ein Graph nicht-zusammenhängend ist, wie in Abbildung 14 (oben, Mitte), wird man nach Anwendung des Algorithmus nach Floyd (1962) (Abb. 14, unten, rechts) eine Distanzmatrix erhalten, die unter anderem "0" enthält. An dieser Stelle wurde also kein Weg zwischen zwei Knoten gefunden. Für diesen Fall, kann man entweder einen neuen Zufallsgraph erzeugen, in der Hoffnung, dass dieser zusammenhängend ist oder versuchen das Verfahren nach Kamada & Kawai (1989) auf nicht-zusammenhängende Graphen zu erweitern.

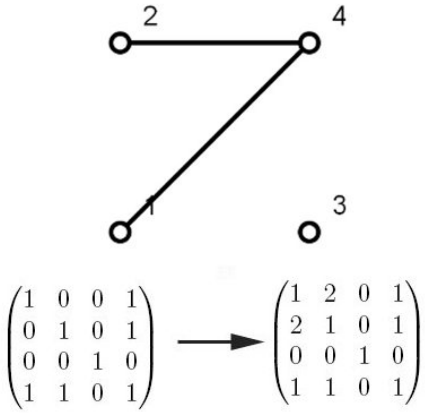

Abb. 14: Nicht-zusammenhängender Graph (oben, Mitte) mit Adjazenzmatrix (unten, links) und Distanzmatrix (unten, rechts)

7.3 Erweiterung auf nicht-zusammenhängende Graphen

Auf Anregung von Herrn Prof. Dr. A. Wassermann, Universität Bayreuth, wurde bei dieser Implementierung versucht, das genannte Verfahren auf nicht-zusammenhängende Graphen zu erweitern. Prinzipiell gibt es dafür zwei Möglichkeiten: Zum einen kann versucht werden aus der berechneten Distanzmatrix die zusammenhängenden Komponenten der Teilgraphen auszulesen, den Algorithmus auf jeden Teilgraphen anzuwenden und nebeneinander zu platzieren. Zum anderen kann man gemäß der Spring-Embedder-Idee zusätzliche Federn zwischen den einzelnen Teilgraphen einfügen. In dieser Arbeit wurde die zweite Variante umgesetzt. Nach der Prüfung auf Zusammenhang des Graphen werden die Nullen durch eine vom Graphen abhängige Konstante ersetzt (Abb. 15), die sich wie folgt aus der Länge des maximalen Weges berechnet:

$$\frac{max_{ij}d_{ij}}{1.5}$$

$$\begin{pmatrix} 1 & 2 & 0 & 1 \\ 2 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 2 & 1.3 & 1 \\ 2 & 1 & 1.3 & 1 \\ 1.3 & 1.3 & 1 & 1.3 \\ 1 & 1 & 1.3 & 1 \end{pmatrix}$$

Abb. 15: Links: Distanzmatrix, rechts: Distanzmatrix nach Erweiterung

7.4 Scramble-Funktion

Um dem Anwender dieses Verfahrens mittels JSXGraph Möglichkeit zu geben anhand verschiedener Ausgangslayouts die Funktionsweise auszuprobieren (Abb. 16-19), wurde hier die Funktion "Scramble" implementiert. Diese kann über einen Button aufgerufen werden. Die Knoten des bestehenden Graphen werden dann zu zufälligen neuen Positionen verschoben. Die neuen Koordinaten ergeben sich aus der Funktion $Math.random$ und der Konstanten L_0, die in etwa die Seitenlänge der Zeichenfläche ist (Universität Konstanz 2004).

$$x_{neu} = 10 + (L_0 - 20) \cdot Math.random()$$

$$y_{neu} = 10 + (L_0 - 20) \cdot Math.random()$$

Die Knoten können auch per Mausklick vom Benutzer verschoben werden. Auf dieses neue Anfangslayout kann das Spring-Embedder-Verfahren neu angewendet werden.

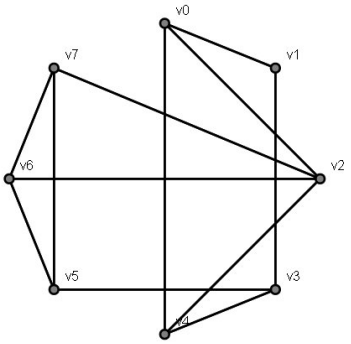

Abb. 16: Anfangslayout

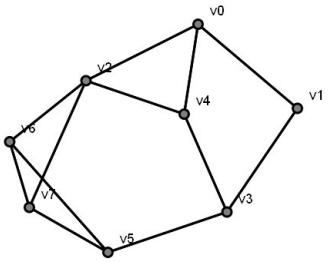

Abb. 17: Layout nach Anwendung des Spring-Embedder-Verfahrens

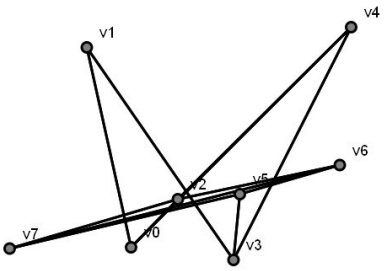

Abb. 18: Verschiebung der Knoten mittels Scramble-Button

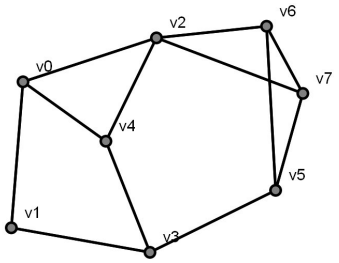

Abb. 19: Layout nach wiederholter Anwendung des Spring-Embedder-Verfahrens

8 Implementierung in JSXGraph

8.1 HTML-Hauptdatei

An dieser Stelle wird auf den Quellcode des implementierten Spring-Embedder-Verfahrens eingegangen. Die Hauptdatei ist die hier dargestellte HTML-Datei, die die Cross-Browser-Bibliothek für JSXGraph (Wassermann et al. 2010) und die Algorithmusfunktion aus der JavaScript-Datei "Algorithm_KK.js" aufruft (Z. 2-11). Hier sind außerdem die Eingabefelder "number of nodes" und "probability of adjacency" sowie die Buttons "reload", "set graph", "do algorithm" und "scramble" definiert (Z. 17-29). Durch die Betätigung des Buttons "set graph" wird ein Zufallsgraph mit der eingegebenen Knotenanzahl und Wahrscheinlichkeit für Adjazenz gezeichnet. Mit "do algorithm" wird das Verfahren auf diesen Graphen angewendet und das neue Layout dargestellt. Durch die *Scramble*-Funktion werden den Knoten neue zufällige Positionen zugeordnet. "reload" lädt die HTML-Datei neu und leert somit die JSXGraph-Zeichenfläche (Abb. 20), die in Zeile 30-38 und 14-15 generiert wird.

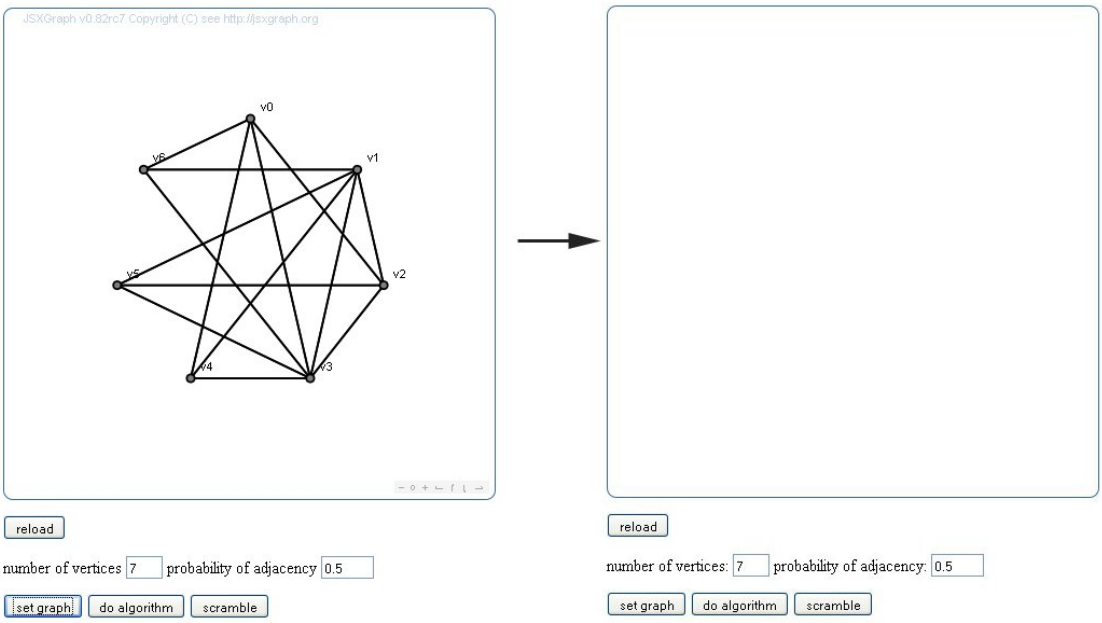

Abb. 20: Leerung der JSXGraph-Zeichenfläche mit dem "reload"- Button

HTML-Datei

```
<html>
<head>
 <title>JSXGraph Konstruktionstemplate</title>
 <link rel="stylesheet" type="text/css"
  href="http://jsxgraph.uni-bayreuth.de/distrib/jsxgraph.css"/>
 <script type="text/javascript"
  src="http://jsxgraph.uni-bayreuth.de/distrib/jsxgraphcore.js">
 </script>
 <script type="text/javascript" src="Algorithm_KK.js">
 </script>
</head>

<body>
 <div id="jxgbox" class="jxgbox" style="width:500px;height:500px;">
 </div>

<form action="input_text.htm">
 <p><input type="button" name="Reload" value= "reload"
  onclick="javascript:location.reload()"></p>
 <p>number of vertices: <input type="text" id="VerticesNumber"
  size="2" maxlength="2" value="7">
  probability of adjacency: <input type="text" id="Probability"
  size="5" maxlength="5" value="0.5"></p>
 <p><input type="button" value="set␣graph"
   onClick="graph=SetGraph();">
  <input type="button" value="do␣algorithm" onClick="Main(graph);">
  <input type="button" value="scramble" onClick="Scramble(graph);">
 </p>
</form>
<script type="text/javascript">
  /* <![CDATA[ */

board = JXG.JSXGraph.initBoard('jxgbox',
 boundingbox: [-2, 16,16, -2], axis: false, grid: false,
 keepaspectratio: true , showcopyright: false});

  /* ]]> */
</script>
</body>
</html>
```

8.2 Initialisierung des Zufallsgraphen

Die Berechnung des Zufallsgraphen erfolgt in zwei Schritten. Zunächst wird im nachfolgenden Quellcode (Z. 3) die Knotenanzahl n aus dem Eingabefeld "number of vertices" ausgelesen und eine $n \times 2$-Matrix v definiert (Z. 4-6). v werden dann die x- und y-Koordinaten der n Knoten zugewiesen (Z. 7-10), dass die Knoten auf einem Kreis in Mitten der Zeichenfläche (Radius $L_0/2$, Mittelpunkt mit Koordinaten (7,7)) gleichmäßig verteilt werden. Bei n Knoten berechnen sich x- und y-Koordinaten des j-ten Punktes folgendermaßen:

$$v_{[j][0]} = \sin(j/n) \cdot 2\pi \cdot (L_0/2) + 7$$

$$v_{[j][1]} = \cos(j/n) \cdot 2\pi \cdot (L_0/2) + 7$$

Berechnung der Knotenkoordinaten

```
1  function PositionVertices(){
2   var L0 = 10;
3   var n = eval(document.getElementById("VerticesNumber").value);
4   var v = [];
5   for(var i=0; i<n; i++) { v[i] = []; };
6   for(var j=0; j<n; j++){
7    v[j][0] = ((Math.sin((j / n) * (2 * Math.PI)) * (L0 / 2)) + 7)};
8   for(var k=0; k<n; k++){
9    v[k][1] = ((Math.cos((k / n) * (2 * Math.PI)) * (L0 / 2)) + 7)};
10  return v;
11 };
```

Für die Zufallsadjazenzmatrix wird in der Funktion *setAdjacency* der Wert *Probability* aus dem Eingabefeld "Probability of Adjacency" ausgelesen (Z. 14). Aus der Knotenanzahl n, die aus der Länge des Arrays v berechnet wird (Z. 15), wird die $n \times n$-Matrix a definiert (Z. 16-22). Die Einträge der Hauptdiagonalen werden wie bei der Einheitsmatrix mit dem Wert "1", alle anderen mit einem Zufallswert der Funktion *Math.random*() gefüllt (Z. 24-30). *Math.random*() gibt zufällige Werte im Intervall $[0, 1]$ aus. In Zeile 32-45 werden alle Elemente, bis auf die Hauptdiagonalelemente, von a betrachtet und geprüft, ob sie kleiner oder gleich $1 - WS$ sind, wobei WS der angegebene Wahrscheinlichkeitswert ist. Falls dies zutrifft, wird dieser Stelle $a_{[j][i]} = a_{[i][j]}$ eine "1" zugeordnet, andernfalls eine "0". Das Matrixelement $a_{[j][i]}$ wird $a_{[i][j]}$ gleichsetzt, so dass die Wahrscheinlichkeit für eine Kante zwischen zwei Knoten gemäß dem eingegebenem Wert realisiert wird. Die resultierende Adjazenzmatrix ist symmetrisch.

Erstellung der Zufallsadjazenzmatrix

```
12 function SetAdjacency(v){
13  var L0 = 10;
14  var WS = eval(document.getElementById("Probability").value);
15  var n = v.length;
16  var a = [];
17  for(var j=0; j<n; j++){a[j] =[];};
18  for(var j=0; j<n; j++){
19   for(var i=0; i<n; i++){
20    a[j][i] = [];
21   }
22 };
23
24  for(var j=0; j<n; j++){
25   for(var i=0; i<n; i++){
26    if(i==j){a[j][i] = 1;
27    }
28     else{a[j][i] = Math.random();}
29   }
30  }
31
32  for(var j=0; j<n; j++){
33   for(var i=0; i<n; i++){
34    if(i==j){a[j][i] = 1;}
35     if(a[j][i] <= 1-WS){
36      a[j][i] = 1;
37      a[i][j] = a[j][i];
38     }
39    else{a[j][i] = 0;
40     a[i][j] = a[j][i];
41    }
42   }
43  };
44  return a;
45 };
```

Die Zeichnung des Graphen ist in der Funktion *Placement* (Z. 46-63) definiert. Hier wird auf die Gestalt der Knoten (Z. 49-53) und Kanten (Z. 54-60) Einfluss genommen. Um die Laufzeit dieser Funktion zu verkürzen, wird nicht jedes Element einzeln nacheinander gezeichnet, sondern gleich das komplette Anfangslayout des Graphen dargestellt. Dies wird geschafft, in dem das Update des JSXGraph-Zeichenfläche zunächst unterdrückt (Z. 47) und erst später (Z. 63] wieder zugelassen wird. Die Funktion (Z. 66-71) verknüpft lediglich die drei Funktionen *PositionVertices*, *SetAdjacency* und *Placement* und ist über den Button "set graph" in der HTML-Hauptdatei aufrufbar. Dabei werden die Adjazenzmatrix a sowie Informationen über die nummerierten Knoten p in der Variable *graph* gespeichert (HTML-Datei, Z. 24-25).

Zeichnung des Graphen

```
46 function Placement(v,a){
47  board.suspendUpdate();
48  var n = a.length;
49  var p = [];
50  for (var l=0; l<n; l++) {
51   p[l] = board.create('point', [v[l][0],v[l][1]],
52   {name:'v'+ l, fillColor:'grey', strokeColor:'black'});
53  };
54  for(var r=0; r<n; r++){
55   for(var s=0; s<n; s++){
56    if(a[r][s]==1){
57     board.create('segment', [p[r],p[s]], {strokeColor:'black'});
58    }
59   }
60  };
61 board.unsuspendUpdate();
62  return p;
63 };
64
65
66 function SetGraph(){
67  var v = PositionVertices();
68  var a = SetAdjacency(v);
69  var p = Placement(v,a);
70  return [p,a];
71 };
```

8.3 Scramble

Die Funktion *Scramble* wird mit dem gleichnamigen Button betätigt. Die Informationen der Knoten werden aus der Variablen *graph* ausgelesen (Z. 74-83), den Knoten neue zufällige Variablen zugeordnet und dorthin verschoben (Z. 84-88).

Scramble-Funktion

```
72 function Scramble(graph){
73  var L0 = 10;
74  var p = graph[0];
75  var n = p.length;
76  var v = [];
77  for(var w=0;w<n; w++){
78   v[w] = [];
79  };
80  for(var h=0; h<n; h++) {
81   v[h][0] = p[h].X();
82   v[h][1] = p[h].Y();
83  };
84  for(var x=0; x<n; x++){
85   v[x][0] = 10 + (L0 - 20) * Math.random();
86   v[x][1] = 10 + (L0 - 20) * Math.random();
87   p[x].moveTo([v[x][0], v[x][1]]);
88   };
89 };
```

8.4 Floyd-Algorithmus

Zur Berechnung der kürzesten Wege zwischen allen Knoten wird in Zeile 90-107 der Floyd-Algorithmus verwendet. An dieser Stelle wird aus der Adjazenzmatrix a eine Distanzmatrix.

Floyd-Algorithmus

```
90 function Floyd(a){
91  var n = a.length;
92  for(var x= 0; x<n; x++){
93   for(var y=0; y<n; y++){
94    if(a[x][y] > 0){
95     for(var j=0; j<n; j++){
96      if(a[y][j] >0){
97       if(a[x][j] == 0 || a[x][y]+a[y][j] < a[x][j]){
98       a[x][j] = a[x][y] + a[y][j];
99       a[j][x] = a[x][j];
100      }
101     }
102    }
103   }
104  }
105 };
106 return a;
107 };
```

8.5 Prüfung auf Zusammenhang

Die Funktion *Connection* (Z. 108-117) dient zur Überprüfung, ob der gegebene Graph zusammenhängend ist. Falls eines der Matrixelemente von a identisch "0" ist, so ist der Graph nicht-zusammenhängend und die Funktion gibt den Wert $false$ zurück.

Prüfung auf Zusammenhang des Graphen

```
108 function Connection(a){
109  var n = a.length;
110  for(var x= 0; x<n; x++){
111   for(var y=0; y<n; y++){
112    if(a[x][y] == 0){
113     return false;
114    }
115   }
116  };
117 };
```

8.6 Weitere Funktionen

Mit der Funktion *maxElement* (Z. 118-127) wird aus der Distanzmatrix a der größte Wert aller Einträge wiedergegeben. Die Variable m wird dazu auf "0" gesetzt (Z. 119) und mit Hilfe der Funktion $Math.max()$ mit jedem Element der Matrix a verglichen. Ist der betrachtete Eintrag größer als m, so übernimmt m diesen neuen Wert, andernfalls behält die Variable ihren aktuellen Wert.

Suche nach dem maximalen Element

```
118 function maxElement(a){
119  var x, y, m=0;
120  var n= a.length;
121   for(var x=0; x<n; x++){
122    for(var y=0; y<n; y++){
123    m = Math.max(m,a[x][y]);
124    }
125   }
126  return m;
127 };
```

Die Härten der einzelnen Federn sind abhängig von den Längen der graphentheoretischen Wege in a und der Konstanten K, die hier erstmal beliebig gewählt wurde. Die einzelnen Federhärten werden durch die $n \times n$ - Matrix f ausgegeben (Z. 128-138).

Berechnung der Federhärten

```
128  function Strength(a){
129  var K = 10000;
130  var x, y, n = a.length, f=[];
131  for(x=0; x<n; x++){
132   f[x] = [];
133   for(y=0; y<n; y++){
134    f[x][y] = K / (a[x][y] * a[x][y]);
135    }
136  }
137  return f;
138 };
```

Ähnlich berechnen sich die Federlängen der einzelnen Federn in der $n \times n$ - Matrix w (Z. 139-149).

Berechnung der Federlängen

```
139  function SpringLength(a,L){
140  var n = a.length;
141  var x, y, w=[];
142   for(x=0; x<n; x++){
143    w[x] = [];
144     for(y=0; y<n; y++){
145      w[x][y] = L * a[x][y];
146     }
147   }
148  return w;
149 };
```

Die Funktionen *DerivateX* (Z. 150-163) und *DerivateY* (Z. 165-176) berechnen die partielle Ableitungen der Gesamtenergie nach der x- und y-Koordinate für den Knoten m:

$$\frac{\partial E}{\partial x_m} = \sum_{i \neq m} k_{mi} \left\{ (x_m - x_i) - \frac{l_{mi}(x_m - x_i)}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{\frac{1}{2}}} \right\}$$

$$\frac{\partial E}{\partial y_m} = \sum_{i \neq m} k_{mi} \left\{ (y_m - y_i) - \frac{l_{mi}(y_m - y_i)}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{\frac{1}{2}}} \right\}$$

Berechnung der partiellen Ableitungen der Gesamtenergie nach x und y

```
150  function DerivateX(m,k,l,v) {
151  var dx=0, i, d0, d1, d2, d3;
152  var n = v.length;
153   for(i=0; i<n; i++){
154    if(i != m){
155     d0 = v[m][0] - v[i][0];
156     d1 = v[m][1] - v[i][1];
157     d2 = Math.sqrt(d0 * d0 + d1 * d1);
158     d3 = (l[m][i] * d0) / d2;
159     dx += k[m][i] * (d0 - d3);
160    }
161   }
162  return dx;
163 };
164
165 function DerivateY(m,k,l,v) {
166  var dy=0, i, d0, d1;
167  var n = v.length;
168   for(i=0; i<n; i++){
169    if(i != m){
170     d0 = v[m][0] - v[i][0];
171     d1 = v[m][1] - v[i][1];
172     dy += k[m][i] * (d1 - ((l[m][i] * d1) /
          Math.sqrt(d0 * d0 + d1 * d1)));
173    }
174   }
175  return dy;
176 };
```

Die Funktionen *DerivateX* und *DerivateY* werden in der *Delta*-Funktion benötigt (Z. 177-194). Für jeden Knoten wird ein *Delta*-Wert wie folgt berechnet:

$$\Delta_m = \sqrt{\left\{\frac{\partial E}{\partial x_m}\right\}^2 + \left\{\frac{\partial E}{\partial y_m}\right\}^2}$$

Diese Werte werden später beim Aufrufen der *Delta*-Funktion im Array D (Z. 287-290) gespeichert. Die Funktion *maxImprovement* (Z. 196-203) bestimmt den größten Wert aus D, ähnlich wie bei *maxElement*. Da jedoch nicht der größte Wert, sondern der dazugehörige Knoten gesucht wird, vergleicht die Funktion $VerticeMax$-$Improvement$ diesen Wert mit jedem Element von D. Ist einer dieser Werte von D identisch mit dem größten *Delta*-Wert, so wird die Nummer des Knotens ausgegeben (Z. 205-212). Das ist der Knoten der den momentanen größten Fortschritt verspricht.

Suche nach dem Knoten mit dem größten Fortschritt

```
177  function Delta(s,k,l,v){
178  var n = v.length;
179  var d, d0, d1, d2;
180  var DY = [];
181   for(var m=0; m<n; m++){
182    DY[m] = DerivateY(m,k,l,v);
183   };
184  var DX = [];
185   for(var m=0; m<n; m++){
186    DX[m] = DerivateX(m,k,l,v);
187   };
188   d0 = DX[s] * DX[s];
189   d1 = DY[s] * DY[s];
190   d2 = d0 + d1;
191   d  = Math.sqrt(d2);
192
193  return d;
194 };
195
196 function maxImprovement(D){
197  var m, s=0;
198  var n = D.length;
199   for(var m=0; m<n; m++){
200    s = Math.max(s,D[m]);
201   }
202  return s;
203 };
204
205 function VerticeMaxImprovement(D,s){
206  var t;
207  var n = D.length;
208   for (var t=0; t<n; t++){
209    if(s == D[t])
210     return t;
211   }
212 };
```

Die doppelten partiellen Ableitungen der Gesamtenergie E_{xx}, E_{yy} und E_{xy} werden mit den Funktionen $DerivateXX$, $DerivateYY$ und $DerivateXY$ (Z. 213-255) folgendermaßen berechnet:

$$E_{xx} = \frac{\partial^2 E}{\partial x_m^2} = \sum_{i \neq m} k_{mi} \left\{ 1 - \frac{l_{mi}(y_m - y_i)^2}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{\frac{3}{2}}} \right\}$$

$$E_{yy} = \frac{\partial^2 E}{\partial y_m^2} = \sum_{i \neq m} k_{mi} \left\{ 1 - \frac{l_{mi}(x_m - x_i)^2}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{\frac{3}{2}}} \right\}$$

$$E_{xy} = \frac{\partial^2 E}{\partial x_m \partial y_m} = \frac{\partial^2 E}{\partial y_m \partial x_m} = \sum_{i \neq m} k_{mi} \frac{l_{mi}(x_m - x_i)(y_m - y_i)}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{\frac{3}{2}}}$$

Berechnung der doppelten partiellen Ableitungen der Gesamtenergie

```
213  function DerivateXX(m,k,l,v) {
214  var dxx=0, i, d0, d1, d2, d3;
215  var n = v.length;
216   for(i=0; i<n; i++){
217    if(i != m){
218     d0 = v[m][0] - v[i][0];
219     d1 = v[m][1] - v[i][1];
220     d2 = l[m][i] * d1 * d1;
221     d3 = Math.sqrt(d0 * d0 + d1 * d1);
222     dxx += k[m][i] * (1 - (d2 / Math.pow(d3,3)));
223    }
224   }
225  return dxx;
226 };
227
228 function DerivateYY(m,k,l,v) {
229  var dyy=0, i, d0, d1, d2, d3;
230  var n = v.length;
231   for(i=0; i<n; i++){
232    if(i != m){
233     d0 = v[m][0] - v[i][0];
234     d1 = v[m][1] - v[i][1];
235     d2 = l[m][i] * d0 * d0;
236     d3 = Math.sqrt(d0 * d0 + d1 * d1);
237     dyy += k[m][i] * (1 - (d2 / Math.pow(d3,3)));
238    }
239   }
240  return dyy;
241 };
```

```
242 function DerivateXY(m,k,l,v) {
243  var dxy=0, i, d0, d1, d2, d3;
244  var n = v.length;
245   for(i=0; i<n; i++){
246    if(i != m){
247     d0 = v[m][0] - v[i][0];
248     d1 = v[m][1] - v[i][1];
249     d2 = l[m][i] * d0 * d1;
250     d3 = Math.sqrt(d0 * d0 + d1 * d1);
251     dxy += k[m][i] * (d2 / Math.pow(d3,3));
252    }
253   }
254  return dxy;
255 };
```

Die Funktionen *SlideX* und *SlideY* (Z. 257-273) bestimmen in wie weit der gewählte Knoten in x- und y-Richtung verschoben wird. Die Berechnung geht aus diesen Formeln hervor:

$$\delta_x = -\frac{E_x}{E_{xx}} - \frac{E_{xy}}{E_{xx}} \left(\frac{-E_y + \frac{E_x \cdot E_{xy}}{E_{xx}}}{-\frac{E_{xy}^2}{E_{xx}} + E_{yy}} \right)$$

$$\delta_y = \frac{-E_y + \frac{E_x \cdot E_{xy}}{E_{xx}}}{-\frac{E_{xy}^2}{E_{xx}} + E_{yy}}$$

Berechnung der Verschiebung

```
257  function SlideY(m,DX,DY,DXX,DYY,DXY ){
258  var d, d0, d1, d2, d3;
259   d0 = (DXY[m] * DX[m]) / DXX[m];
260   d1 = d0 - DY[m];
261   d2 = (DXY[m] * DXY[m]) / DXX[m];
262   d3 = DYY[m] - d2;
263   d  = d1 / d3;
264  return d;
265 };
266
267 function SlideX(m,DX,DXX,DXY,SY){
268  var d, d0, d1;
269   d0 = DX[m] / DXX[m];
270   d1 = (DXY[m] * SY) / DXX[m];
271   d  = -d0 - d1;
272  return d;
273 };
```

8.7 Algorithmusfunktion

Die Funktion *SpringEmbedder* verknüpft einige der bereits definierten Funktionen und ruft diese auf, um das Endlayout zu berechnen und zu erzeugen. Die Knoten werden hier in ihre nach dem Spring-Embedder-Verfahren optimale Lage verschoben (Z. 273-358). In Zeile 277-290 werden die Werte der Funktionen *DerivateX*, *DerivateY*, *Delta* für alle Knoten berechnet und in den Arrays DX, DY und D gespeichert. Es wird der Knoten m ausgewählt, der den größten *Delta*-Wert hat, d. h. den größten Fortschritt bei der Bewegung in Richtung Energieminimum verspricht (Z. 292-293). Für diesen Knoten m werden die Funktionswerte von *DerivateXX*, *DerivateYY*, *DerivateXY*, *SlideX* und *SlideY* berechnet und die neuen Koordinaten des Knotens in v gespeichert (Z. 295-309). Die Knoten werden solange verschoben bis der Wert von *maxImprovement* unter den vorgegebenen *stopp*-Wert fällt, also je nach Wahl des *stopp*-Wertes keine signifikante Verbesserung des Layouts mehr eintritt. Außerdem wird jeder ausgewählte Knoten solange verschoben bis sein *Delta*-Wert unter den *stopp*2-Wert fällt. Dieser Knoten muss sich erst in der momentan optimalen Lage befinden, bevor der nächste Knoten zur Verrückung gewählt wird (Z. 312-358).

Algorithmusfunktion

```
273  function SpringEmbedder(p,k,l,v) {
274  var stopp = 0.01;
275  var stopp2 = 0.1;
276  var n = v.length;
277  var DX = [];
278  for(var m=0; m<n; m++){
279   DX[m] = DerivateX(m,k,l,v);
280  };
281
282  var DY = [];
283  for(var m=0; m<n; m++){
284   DY[m] = DerivateY(m,k,l,v);
285  };
286
287  var D = [];
288  for(var m=0; m<n; m++){
289   D[m] = Delta(m,k,l,v,n);
290  };
291
292  var s = maxImprovement(D);
293  var m = VerticeMaxImprovement(D,s);
294
295  var DXX = [];
296  DXX[m] = DerivateXX(m,k,l,v);
297
298  var DYY =[];
299  DYY[m] = DerivateYY(m,k,l,v);
300
301
302  var DXY =[];
303   DXY[m] = DerivateXY(m,k,l,v);
304
```

```
305  var SY = SlideY(m,DX,DY,DXX,DYY,DXY);
306  var SX = SlideX(m,DX,DXX,DXY,SY);
307
308  v[m][0] = v[m][0] + SX;
309  v[m][1] = v[m][1] + SY;
310
311
312  while (maxImprovement(D) > stopp){
313   for(var m=0; m<n; m++){
314    DX[m] = DerivateX(m,k,l,v);
315   };
316
317   for(var m=0; m<n; m++){
318    DY[m] = DerivateY(m,k,l,v);
319   };
320
321   for(var m=0; m<n; m++){
322    var n = v.length;
323     D[m] = Delta(m,k,l,v,n);
324   };
325
326   s = maxImprovement(D);
327   m = VerticeMaxImprovement(D,s);
328
329   DXX[m] = DerivateXX(m,k,l,v);
330   DYY[m] = DerivateYY(m,k,l,v);
331   DXY[m] = DerivateXY(m,k,l,v);
332
333   SY = SlideY(m,DX,DY,DXX,DYY,DXY);
334   SX = SlideX(m,DX,DXX,DXY,SY);
335
336   v[m][0] = v[m][0] + SX;
337   v[m][1] = v[m][1] + SY;
338
339
340   while(Delta(m,k,l,v) > stopp2){
341
342    DX[m] = DerivateX(m,k,l,v);
343    DY[m] = DerivateY(m,k,l,v);
344    DXX[m] = DerivateXX(m,k,l,v);
345    DYY[m] = DerivateYY(m,k,l,v);
346    DXY[m] = DerivateXY(m,k,l,v);
347
348    SY = SlideY(m,DX,DY,DXX,DYY,DXY);
349    SX = SlideX(m,DX,DXX,DXY,SY);
350
351    v[m][0] = v[m][0] + SX;
352    v[m][1] = v[m][1] + SY;
353   }
354  };
355  for(var i=0; i<n; i++){
356  p[i].moveTo([v[i][0], v[i][1]]);
357  };
358 };
```

8.8 Hauptfunktion

Die Hauptfunktion *Main* (Z. 359-401) beinhaltet den Floyd-Algorithmus (Z. 375), die Prüfung auf Zusammenhang, die Erweiterung auf nicht-zusammenhängende Graphen (Z. 377-386), die Berechnung der Federhärten und -längen und ruft die Algorithmusfunktion *SpringEmbedder* auf. Es wird in den Zeilen 377-400 zwischen den zwei Fällen unterschieden: Zusammenhängende Graphen (Z. 393-400) und nicht-zusammenhängende Graphen (Z. 377-392). Bei nicht-zusammenhängenden Graphen muss die Distanzmatrix a so geändert werden, dass die Nullen durch die folgende Konstante ersetzt werden:

$$\frac{max_{ij}d_{ij}}{1.5}$$

Daraufhin erfolgt in beiden Fällen die Berechnung der Federhärte sowie Federlänge bis schließlich die Algorithmusfunktion *SpringEmbedder* aufgerufen wird.

Hauptfunktion

```
359  function Main(graph,a){
360  var K = 10000
361  var L0 = 10;
362  var stopp = 0.0001;
363  var p = graph[0];
364  var a = graph[1];
365  var n = a.length;
366  var v = [];
367  for(var w=0; w<n; w++){
368   v[w] = [];
369  };
370   for(var h=0; h<n; h++) {
371    v[h][0] = p[h].X();
372    v[h][1] = p[h].Y();
373   };
374
375  var a = Floyd(a);
376
377  if(Connection(a,n) == false){
378   var L0 = 10;
379   var L = L0 / maxElement(a);
380   for(var x= 0; x<n; x++){
381    for(var y=0; y<n; y++){
382     if(a[x][y] == 0){
383      a[x][y] = L0 / (L * 1.5);
384     }
385    }
386   };
387
388  var k = Strength(a);
389  var l = SpringLength(a,L);
390   D = SpringEmbedder(p,k,l,v);
```

```
391     return D;
392   }
393   else{
394     var L = L0 / maxElement(a);
395     var k = Strength(a);
396     var l = SpringLength(a,L);
397
398     D = SpringEmbedder(p,k,l,v);
399     return D;
400   };
401 };
```

9 Bewertung

9.1 Erfüllung der Ästhetikkriterien

Die Ästhetikkriterien werden allgemein durch den implementierten Alorithmus gut erfüllt. Der Zufallsgraph, dessen Knoten gleichmäßig auf einem Kreis verteilt sind, wird durch die Anwendung des Spring-Embedder-Verfahrens gut übersichtlich dargestellt (Abb. 21-22). Das Anfangs- sowie das Endlayout ist vollständig in der Zeichenfläche abgebildet und füllt diese entsprechend gut aus. Der Grund dafür ist, dass die Konstante L_0, die etwa die Seitenlänge des Displays darstellt, in die Berechnung der natürlichen Federlängen eingeht (Kap. 5).
Bei Betrachtung vieler unterschiedlicher Graphen fällt jedoch auf, dass das Kriterium Planarität weniger berücksichtigt wird als die einheitlichen Kantenlängen. Aufgrund dessen erhält man bei Graphen, bei denen durchaus eine planare Darstellung möglich wäre, oft ein Layout mit Kantenkreuzungen, das allerdings durch die recht einheitlichen Kantenängen symmetrische Strukturen verdeutlicht (Abb. 21-22). Wie in dem hier gezeigten Beispiel (Abb. 21-23) bleibt das Verfahren sozusagen in einem lokalen Energieminimum stecken. Dies ist kein Fehler der beschriebenen Implementierung, sondern, wie in Kapitel 5, berechnet der Algorithmus nach Kamada & Kawai (1989) nur die lokale und nicht globale Minimierung der Gesamtenergie. Bei diesem Graphen ist es jedoch möglich die Knoten per Mausklick oder per Scramble-Button so zu verschieben, dass nach Durchlaufen des Spring-Embedder-Verfahrens ein perfektes Layout erzeugt wird (Abb. 23).
Wie man an diesem Beispiel klar erkennt, kann das Anfangslayout durchaus Auswirkungen auf das Resultat haben.

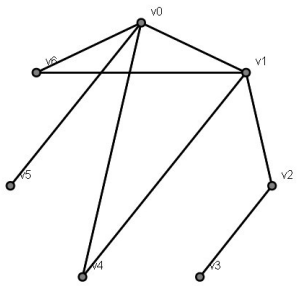

Abb. 21: Zufallsgraph

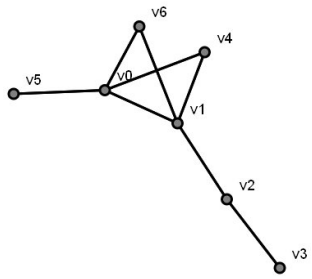

Abb. 22: Layout nach Verwendung des Algorithmus

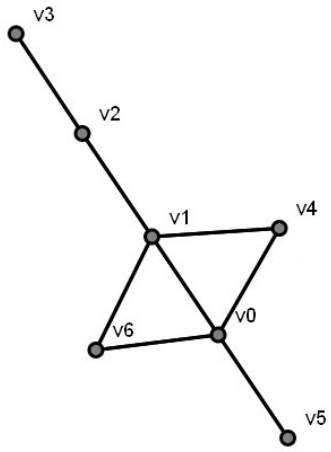

Abb. 23: Layout nach Verschiebung per Mausklick und Verwendung des Algorithmus

9.2 Anwendung

9.2.1 Mit Zufallsgraphen

Das beschriebenen Programm ist ganz intuitiv anwendbar und erfordert keinerlei Kenntnisse über den Algorithmus oder JSXGraph. Es wird lediglich ein unterstützter Webbrowser (Kap. 6) benötigt, in dem die zugehörige HTML-Datei (Anhang: Algorithm_KK.html) geöffnet wird. Damit von der HTML-Datei auf die zugehörige JavaScript-Datei (Anhang: Algorithm_KK.js) automatisch zugegriffen werden kann, müssen beide im gleichen Ordner gespeichert sein.
Auf der noch leeren JSXGraph-Zeichenfläche kann jetzt ein Zufallsgraph mit den gewünschten Werten für "number of vertices" und "probability of adjacency" durch Betätigung des "set graph"-Buttons erstellt werden. Um den Algorithmus auszuführen und ein ansprechendes Layout zu erhalten, wird der "do algorithm"-Button gedrückt. Möchte der Anwender die Knotenpositionen zufällig verändern, nutzt er den "scramble"-Button oder verschiebt die Knoten per Mausklick. Der Algorithmus kann mit dem Button "do algorithm" auf das neue Anfangslayout angewendet werden und möglicherweise ein anders Endlayout erhalten. Mit "reload" wird die JSXGraph-Zeichenfläche geleert und es kann ein neuer Graph gezeichnet werden.

In das Eingabefeld "probability of adjacency" muss ein Wert im Intervall $]0, 1[$ eingegeben werden. Je größer die Knotenanzahl, desto kleiner empfiehlt es sich den Wahrscheinlichkeitswert zu wählen, um einen Zufallsgraphen zu erzeugen, bei dem ein ansprechendes Layout überhaupt möglich ist.
Die Knotenanzahl ist generell auf zweistellige, positive Werte beschränkt. Eine Eingabe zwischen 1-99 ist zwar möglich, aber nicht immer sinnvoll. Aufgrund der Display- und Knotengröße sollten 50 Knoten nicht überschritten werden. Die Betrachtung sehr kleiner Graphen (unter 5 Knoten) ist natürlich möglich, wegen der wenig unterschiedlichen Zufallsgraphen in der Regel aber uninteressant. Für Spielereien befinden sich am besten im Bereich von 5-30 Knoten und bei einer Wahrscheinlichkeit zwischen 0.8 (bei 5 Knoten) und 0.2 (bei 30 Knoten).

9.2.2 Mit vorgegebenen Adjazenzmatrizen

Neben Zufallsgraphen können auch eigene Adjazenzmatrizen genutzt werden. Hierfür werden die Dateien Algorithm_KK_2.html und Algorithm_KK_2.js (Anhang: CD-Rom) verwendet. Diese Dateien unterscheiden sich nur geringfügig von den in Kapitel 8 erläuterten Dateien. Die wesentlichen Änderungen beruhen auf dem Weglassen der Eingabefelder sowie der Initialisierung der Adjazenzmatrix, da diese vom Anwender durch das Einbinden einer zusätzlichen JavaScript-Datei (z. B. Algorithm_KK_2_Adjacency.js, Anhang: CD-Rom) gegeben wird. Die Betätigung der Buttons erfolgt wie bei der Anwendung mit Zufallsgraphen.

9.3 Laufzeit

Die Laufzeit des betrachteten Verfahrens hängt von mehreren Komponenten ab: Knotenanzahl, Kantenanzahl, stopp und stopp2.
Große Zeitersparnisse bringen, wie in Kapitel 8 beschrieben, die Befehle *board.suspendUpdate* und *board.unsuspendUpdate* in der Funktion *Placement*. Dies führt dazu, dass die Knoten und Kanten des Anfangslayout nicht nacheinander gezeichnet werden, sondern gleich als komplettes Layout erscheinen. Außerdem wird in der Algorithmusfunktion *SpringEmbedder* darauf verzichtet die einzelnen Knotenverschiebungen auf der Zeichenfläche darzustellen. Auch hier wird erst das fertige Endlayout gezeigt.
Die Aufrufe der Funktionen *DerivateX* und *DerivateY* beanspruchen mit jeweils etwa 44 %, fast die gesamte der Laufzeit. Die Aufrufe der *Delta*-Funktion benötigen etwa 10% der Laufzeit. Alle anderen Funktionen liegen unter 1 % der benötigten Aufrufe.
Aus diesem Grund wurde die Verwendung der *DerivateX* und *DerivateY* genauer betrachtet. Es fällt auf, dass die beiden Funktionen nochmals in der *Delta*-Funktion für jeden Knoten unnötigerweise berechnet werden. Durch Erstetzen von $Delta(s,k,l,v)$ durch $Delta(s,k,l,v,DY,DX)$ (Z. 1-9) können erhebliche Laufzeitverbesserungen erzielt werden. Die bereits berechneten Wert der Ableitungsfunktionen werden mit Hilfe der Arrays DX und DY übergeben.

Laufzeitverbesserung durch verbesserte Delta-Funktion

```
1 function Delta(s,k,l,v,DY,DX){
2  var n = v.length;
3  var d,d0,d1,d2;
4   d0 = DX[s] * DX[s];
5  d1 = DY[s] * DY[s];
6  d2 = d0 + d1;
7  d = Math.sqrt(d2);
8  return d;
9 };
```

Beispielsweise verringert sich die Zahl der Aufrufe (berechnet mit Firebug 1.5.4) für die Berechnung des Layouts aus Abbildung 24 von 1.533.717 auf 165.829.

Die Wahl der determinierenden Variablen *stopp* und *stopp*2 hat auch einen großen Einfluss auf die Laufzeit. Je größer diese sind, desto seltener müssen die While-Schleifen der *SpringEmbedder*-Funktion durchlaufen werden. Das erspart zwar Zeit, allerdings erfolgt dies auf Kosten des Layouts. Die Wahl der Konstanten K hat keine signifikanten Auswirkungen auf die benötigte Zeit.

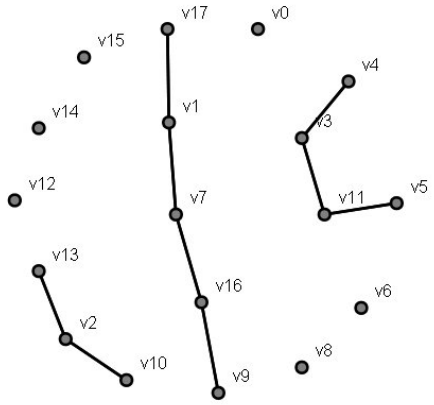

Abb. 24: Darstellung eines nicht-zusammenhängende Graphen

9.4 Verwendung der Konstanten

Um eine perfekte Darstellung der Graphen zu erzielen müssen die Konstanten K, *stopp* und *stopp*2 richtig gewählt werden. Setzt man beispielsweise die Konstanten *stopp* und *stopp*2 auf den Wert 0.0001 und variiert die die Variable K, dann erhält man folgende Resultate:

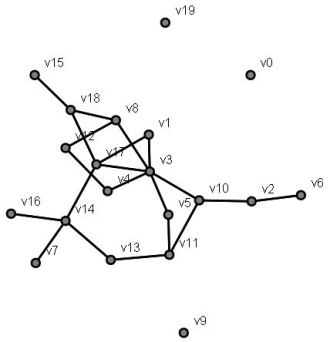

Abb. 25: Darstellung mit $K = 0.0001$, $stopp = stopp2 = 0.0001$

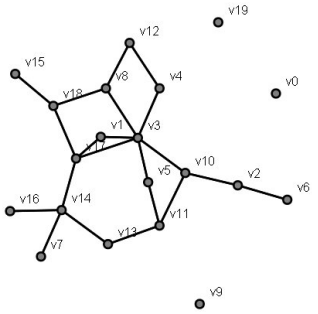

Abb. 26: Darstellung mit $K = 1$, $stopp = stopp2 = 0.0001$

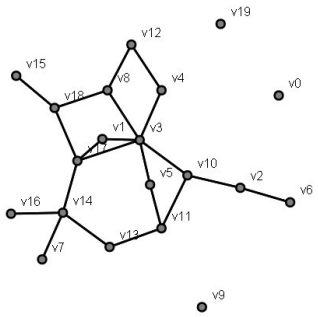

Abb. 27: Darstellung mit $K = 100000$, $stopp = stopp2 = 0.0001$

Das Festhalten der Konstanten $K = 1$ und variieren der *stopp*-Werte ergeben die folgenden 3 Layouts (Abb. 28-30). Richtig gute Darstellungen liefern Abbildung 26, 27 und 29 mit ganz unterschiedlich gewählten Konstanten. Die Erfahrung mit dem Umgang dieses Programms zeigt, dass die Wahl der Konstanten stark von der Struktur des jeweiligen Graphen abhängt, welche das genau sind, bleibt unklar. Generell gilt, dass die *stopp*-Variablen in der Regel sehr klein sein müssen (etwa kleiner 1), um eine optimale Lage der Knoten berechnen zu können.

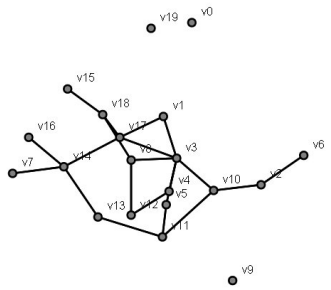

Abb. 28: Darstellung mit $K = 1$, $stopp = stopp2 = 0.0001$

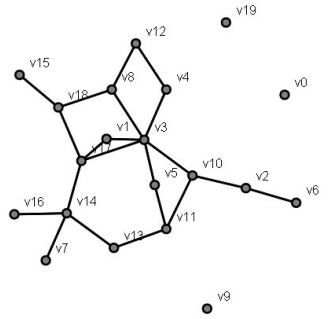

Abb. 29: Darstellung mit $K = 1$, $stopp = stopp2 = 1$

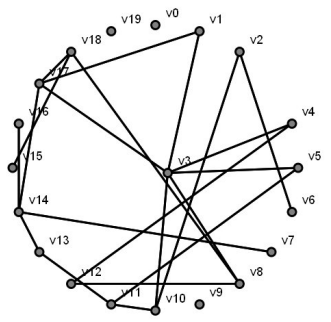

Abb. 30: Darstellung mit $K = 1$, $stopp = stopp2 = 10000$

10 Ausblick

Die gezeigte erweiterte Variante des Spring-Embedder-Verfahrens nach Kamada & Kawai (1989) lässt noch viele Verbesserungsmöglichkeiten zu. Ein generelles Problem dieses Verfahrens ist die richtige Wahl der Konstanten K, *stopp* und *stopp*2. Für die optimale Anwendung wäre es hilfreich, einen Zusammenhang zwischen den gegebenen Informationen über den Graphen, wie z. B. Adjazenz und Knotenanzahl, und den Konstanten herzustellen, um eine perfekte Darstellung zu erhalten.
Wie bei jedem Programm ist auch hier eine Laufzeitverbesserung, vor allem für Graphen ab 30 Knoten, wünschenwert. Beispielsweise könnte ein weiterer Spring-Embedder-Algorithmus speziell für große Graphen implementiert werden, der dann bei Bedarf aufgerufen wird.
Desweiteren ist es eine Überlegung wert, das Ästhetikkriterium Planarität stärker zu gewichten.

11 Zusammenfassung

Graphen dienen in vielen Bereichen der Wissenschaft dazu, komplexe Zusammenhänge und Strukturen darzustellen. Von großer Bedeutung ist dabei die ästhetisch schöne und übersichtliche Darstellung des Graphen. Im Bereich "Graph Drawing", einem Teilgebiet der Theoretischen Informatik, werden dazu Algorithmen entworfen und optimiert.
In der vorliegenden Arbeit wird auf das sogenannte Spring-Embedder-Verfahren eingegangen. Der zusammenhängende Graph wird als physikalisches Modell betrachtet, dessen Gesamtenergie durch Verschiebung der Knoten minimiert wird. Die Kanten werden zu Federn (engl. "spring") und die Knoten zu sich gegenseitig abstoßenden geladenen Teilchen.
Implementiert wurde der Algorithmus nach Kamda & Kawai (1989) in JSXGraph, einer auf JavaScript basierenden Cross-Browser-Bibliothek, die zur Visualisierung dynamischer Mathematik dient.
Das Verfahren wurde auf nicht-zusammenhängende Graphen durch Einsetzen zusätzlicher Federn erweitert. Besonderes Augenmerk wurde auf eine möglichst kurze Laufzeit gerichtet. Die Anwendung des Programms erfordert keinerlei Vorkenntnisse. Es wird die Möglichkeit geboten, sowohl bestehende Adjazenzmatrizen auszuwerten als auch Zufallsgraphen mit gewünschter Knotengröße und Adjazenzwahrscheinlichkeit zu nutzen. Die empfohlene, maximale Knotengröße liegt bei etwa 30 Knoten. Die Ästhetikkriterien werden von den resultierenden Layouts im Allgemeinen gut erfüllt.

12 Literatur

Baptist, P., Ehmann, M., Miller, C., Wassermann, A., Bocka, D., Neidhardt, W., Raab, D., Ulm, V., Frischholz, M., Höniger, E. & Heubeck, W. (2010): GEONExT. - Lehrstuhl für Mathematik und ihre Didaktik, Universität Bayreuth. - URL: http://geonext.uni-bayreuth.de/ (Stand: 08.07.2010).

Brandenburg, F. J., Brandes, U., Di Battista, G., Eades, P., Eppstein, D., De Fraysseix, H., Ganser, E., Liotta, G., Nishizeki, T., Rosenstiehl, P., Tamassia, R. & Tollis, I. G. (2010): Graphdrawing. - URL: http://www.graphdrawing.org (Stand: 08.07.2010).

Brandenburg, F. J.; Himsolt, M. & Rohrer, C. (1996): An Experimental Comparison of Force-directed and Randomized Graph Drawing Algorithms. - In: Brandenburg, F. J. [Editor]: Graph drawing. - Lecture Notes in Computer Science, **1027**: 76-87, Berlin (Springer).

Brandenburg, F. J., Jünger, M. & Mutzel, P. (1997): Algorithmen zum automatischen Zeichnen von Graphen. - In: Informatik-Spektrum, **20**(4): 199-207, Berlin (Springer).

Davidson, H. & Harel, D. (1996): Drawing Graphs Nicely Using Simulated Annealing. - In: ACM Transactions on Graphics (TOG) **15**(4): 301-311, New York (ACM).

De Fraysseix, H., Pach, J. & Pollack, R. (1990): How to draw a planar graph on a grid. - In: Combinatorica, **10**(1): 41-51, Budapest (Akadémiai Kiaó - Springer).

Di Battista, G., Eades, P., Tammassia, R. & Tollis, I. G. (1994): Algorithms for Drawing Graphs: An Annotated Bibliography. - In: Computational Geometry Theory Application **4**(5): 235-282 (Elsevier B.V.).

Eades, P. (1984): A heuristic for graph drawing, Congressus Numerantium **42**: 149-160, Winnipeg (Utilitas Mathematica Publishing Inc.).

Floyd, R. W. (1962): Algorithm 97: Shortest Path. - In: Perlis, A. J. [Editor]: Communications of the ACM, **5**(6): 345, New York (ACM).

Forster, M. (1999): Zeichnen ungerichteter Graphen mit gegebenen Knotengrößen durch ein Springembedder-Verfahren. - Dipl.-Arbeit, Universität Passau: 89 S., Passau.

Frick, A., Ludwig, A. & Mehldau, H. (1995): A Fast Adaptive Layout Algorithm for Undirected Graphs. - In: Proceedings of the DIMACS International Workshop on Graph Drawing: 388-403, London (Springer).

Fruchterman, T. M. J. & Reingold, E. M. (1991): Graph Drawing by Force-directed Placement. - In: Software - Practice and Experience **21**(11): 1129-1164, New York (John Wiley & Sons, Inc.).

Gerhäuser, M. (2010): Interaktive Geometrie mit JavaScript. - URL: http://www.michael-gerhaeuser.de/JSXGraph.23.0.html (Stand: 08.07.2010).

Jungnickel, D. (1987): Graphen, Netzwerke und Algorithmen. - Zürich (B. I. Wissenschaftsverlag - Bibliographisches Institut).

Kamada, T. & Kawai, S. (1989): An algorithm for drawing general undirected graphs. - In: Information Processing Letters **31**(1): 7-15, North-Holland (Elsevier B. V.).

Kaufmann, M. & Wagner, D. [Eds.](2001): Drawing Graphs. Methods and Models. - Berlin (Springer).

Koch, S. (2009): JavaScript - Einführung, Programmierung und Referenz - inklusive Ajax. - Auflage 5, Heidelberg (dpunkt.verlag GmbH).

Mutzel, P. (2005): Zeichnen gerichteter Graphen. Universität Dortmund. - URL: http://ls11-www.cs.uni-dortmund.de/people/gutweng/AE-07/schichten.pdf (Stand: 16.08.2010).

Nöllenburg, M. (2009): Algorithmen zur Visualisierung von Graphen - Symmetrien in serienparallelen Graphen, geradlinige planare Gitterzeichnungen. - Vorlesung im Sommersemester 2009, Karlsruher Institut für Technologie, Universität Karlsruhe. - URL: http://i11www.iti.uni-karlsruhe.de/teaching/sommer2009/graphdrawing/index (Stand: 16.08.2010).

Tunkelang, D. (1994): A Practical Approach to Drawing Undirected Graphs. Technischer Bericht: CS-94-161, School of Computer Science, Carnegie Mellon University, Pittsburgh.

Tunkelang, D. (1999): A numerical optimization approach to general graph drawing. Technischer Bericht: CMU-CS-98-189, School of Computer Science, Carnegie Mellon University, Pittsburgh. - URL: http://reports-archive.adm.cs.cmu.edu/anon/1998/CMU-CS-98-189.pdf (Stand: 25.05.2010).

Universität Konstanz (2004): Java Demo Applet (Spring Embedder). - URL: http://www.inf.uni-konstanz.de/algo/lehre/ss04/gd/demo.html (Stand: 21.08.2010).

Vohrer, U. (2007): Was sind Carbon Nanotubes? Frauenhofer IGB. - URL: http://www.igb.fraunhofer.de/www/gf/grenzflmem/cnt/bilder/Einl_C3_168.jpg (Stand: 16.8.2010).

Wagner, D. & Brandes, U. (2009): Algorithmen zur Visualisierung von Graphen. Vorläufiges Skript zur Vorlesung. - Unveröffentlichtes Manuskript, 94 S. (Stand: 13.08.2009).

Wassermann, A., Ehmann, M., Miller, C., Valentin, B., Gerhäuser, M. & Wilfahrt, P. (2010): JSXGraph. Dynamic Mathematics with JavaScript. Lehrstuhl für Mathematik und ihre Didaktik, Universität Bayreuth. - URL: http://jsxgraph.uni-bayreuth.de (Stand: 08.07.2010).

A Danksagung

Diese Bachelorarbeit entstand am Lehrstuhl für Mathematik und ihre Didaktik der Universität Bayreuth unter der Leitung von Herrn Prof. Dr. Wassermann. Besonders möchte ich mich bei ihm für die Vergabe des Themas, die gute Betreuung und die Anregungen zur Implementierung bedanken. Für eine perfekte Lernumgebung während des einführenden und begleitenden Seminars danke ich außerdem Frau Valentin, Herrn Dr. Ehmann, Herrn Gerhäuser und Herrn Wilfahrt.

B Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit mit dem Thema:

"Implementierung eines Spring-Embedder-Verfahrens in JSXGraph"

selbständig verfasst und keine anderen Hilfsmittel als die angegebenen benutzt habe. Alle Ausführungen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Schriften entnommen sind, habe ich in jedem einzelnen Falle durch Angabe der Quelle als Entlehnung kenntlich gemacht. Diese Arbeit wurde noch nicht an einer anderen Hochschule zur Erlangung eines akademischen Grades eingereicht und wurde auch noch nicht veröffentlicht.

Bayreuth, den 26. August 2010

Lena Freudenberger

C Anhang: CD-Rom

- Bachelorarbeit: Implementierung eines Spring-Embedder-Verfahrens in JSX-Graph (Bachelor.pdf)
- HTML-Hauptdatei (Algorithm_KK.html) mit zugehöriger JavaScript-Datei (Algorithm_KK.js)
- HTML-Hauptdatei (Algorithm_KK_2.html) mit zugehörigen JavaScript-Dateien (Algorithm_KK_2.js) und (Algorithm_K_Adjacency.js)
- Versionen mit Laufzeitbesserungen: Algorithm_KK_runtime.html, Algorithm_KK_runtime.js, Algorithm_KK_2_runtime.html und Algorithm_KK_2_runtime.js
- JavaScript-Dateien mit Adjazenzmatrizen (Ordner Matrix)