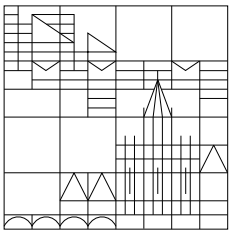
Graphenzeichnen mit hardwarebeschleunigter MDS

Masterarbeit

eingereicht von
Daniel Kaiser



Fachbereich
Informatik und Informationswissenschaft

1. **Gutachter:** Prof. Dr. Ulrik Brandes
2. **Gutachter:** Dr. Andreas Karrenbauer

Konstanz, 2011

Inhaltsverzeichnis

Zusammenfassung

Diese Masterarbeit behandelt die kombinierte Multidimensionale Skalierung (kombinierte MDS), ein Verfahren zum Zeichnen von Graphen, welches Pivot-MDS mit der ausgedünnten Stressmajorisierung verknüpft. Dabei wird das von Pivot-MDS generierte Layout, welches sich sehr effizient berechnen lässt, als Initiallayout für die ausgedünnte Stressmajorisierung verwendet. Dieses Initiallayout reduziert zum einen die Wahrscheinlichkeit der Stressmajorisierung, in lokalen Minima der Stressfunktion zu enden, zum anderen reduziert es die Anzahl von Iterationen, welche die Stressmajorisierung benötigt, um ein gutes Layout zu finden. Diese Kombination führt zu einem laufzeiteffizienten Verfahren, welches beim Zeichnen von Graphen hochwertige Ergebnisse liefert. Weiter wird eine eigene parallelisierte Implementierung der kombinierten MDS vorgestellt, welche die Hardwarebeschleunigung der Grafikkarte nutzt. Diese Implementierung erreicht im Vergleich zu effizienten Implementierungen anderer Verfahren gute Laufzeiten und Ergebnisse. Ferner bietet diese Implementierung die Möglichkeit, zusätzlich zu zweidimensionalen auch dreidimensionale Graphenlayouts zu erstellen, welche mittels eines eigens erstellen Programms visualisiert werden können.

1 Einleitung

Verschiedene Objekte können miteinander in Verbindung stehen. Eine Möglichkeit, diese Verbindung auf abstrakter Ebene zu beschreiben, stellen Graphen dar. Graphen modellieren diese Objekte als Knoten, während ihre Verbindung zueinander durch Kanten zwischen den entsprechenden Knoten repräsentiert wird. Praktische Beispiele, die sich durch Graphen repräsentieren lassen, sind

- Schienennetzwerke, wobei Bahnhöfe (Knoten) durch Schienenwege (Kanten) miteinander verbunden sind,
- Stromnetzwerke, bei denen Haushalte, Verteilerstationen und Kraftwerke (Knoten) über Stromleitungen (Kanten) zueinander in Bezug stehen und
- Leiterplattenlayouts, bei denen Bauteile (Knoten) über Leitungen (Kanten) verbunden sind.

Um die Struktur der Beziehung aller Objekte, die durch einen Graphen modelliert wird, zu veranschaulichen, besteht begründetes Interesse, diese Graphen durch eine Zeichnung zu visualisieren. Beim obigen Beispiel des Schienennetzwerkes würde aus einer solchen Visualisierung ein Linienplan hervorgehen. Da die Zeichnung eines Graphen in den meisten Fällen dazu dient, Menschen Informationen über die Beziehung der Objekte zu zeigen, sollte sie je nach Anwendungsgebiet verschiedene Eigenschaften aufweisen, um dafür adäquat zu sein. Würden alle Knoten an zufälliger Position gezeichnet, so könnten Menschen bei größeren Graphen keine Struktur erkennen. Eine wichtige Eigenschaft, die in vielen Anwendungsgebieten wünschenswert ist, ist eine gute Repräsentation der Distanz zweier Knoten im Layout, was bedeutet, dass die Knoten so positioniert werden müssen, dass deren geometrische Abstände in der Zeichnung den Distanzen im Graphen entsprechen. Die Distanz zweier Knoten ist dabei die Anzahl von Kanten, über die man gehen muss, um von einem zum anderen Knoten zu gelangen.
Für das Beispiel des Schienennetzwerkes bedeutet dies, dass Bahnhöfe im dazugehörenden Linienplan umso näher beieinander liegen, desto weniger weitere Bahnhöfe zu durchfahren sind, um von einem Bahnhof zum anderen zu gelangen. Wie weit diese Bahnhöfe auseinander liegen, wird dabei nicht beachtet. Eine weitere Eigenschaft, welche die Struktur eines Graphen für Menschen besser erkenntlich macht, ist eine möglichst geringe Anzahl von Kantenschnitten. Der Plan des Londoner U-Bahn-Netzes ist ein Beispiel für einen Linienplan, der diese Eigenschaften aufweist. Die Position der Stationen in diesem Plan entspricht ebenfalls nicht den topologischen Positionen der Stationen. Eine geringe Anzahl von Kantenschnitten wird durch das Positionieren der Knoten entsprechend ihrer Distanzen zusätzlich erreicht, da Knoten nahe bei ihren Nachbarn gezeichnet werden, was wiederum zu kurzen Kantenlängen führt und damit Kantenschnitte unwahrscheinlicher macht.
Allgemein sorgt eine gute Repräsentation der Distanz zweier Knoten im Layout für eine gute, als ästhetisch empfundene Visualisierung der Struktur des Graphen und damit der Strukur der modellierten Objektbeziehungen. Da auch sehr große Graphen, beispielsweise das Stromnetz eines Landes, visualisiert werden sollen und dies von Hand sehr viel Zeit kosten würde, ist es ferner wünschenswert, Graphen automatisch zu zeichnen.

Mittels multidimensionaler Skalierung (MDS) lassen sich Layouts generieren, welche die eben beschriebenen Eigenschaften aufweisen. Das im Rahmen dieser Masterarbeit verwendete MDS-Verfahren ist eine Kombination aus klassischer MDS [24] und Stressmajorisierung [9], welches sich, wie in [5] gezeigt, am besten dazu eignet, effizient eine Zeichnung eines Graphen zu erstellen, die seine Knotendistanzen geometrisch möglichst genau repräsentiert. Dieses Verfahren wird im weiteren Verlauf als kombinierte MDS bezeichnet. Damit die kombinierte MDS auch bei großen Graphen in annehmbarer Zeit durchzuführen ist, werden ausgedünnte Varianten der klassischen MDS und der Stressmajorisierung verwendet. Die benutzte ausgedünnte Variante der klassischen MDS ist Pivot-MDS, welche in [4] vorgestellt worden ist. Im Rahmen dieser Masterarbeit wurde eine eigene parallele Implementierung der kombinierten MDS angefertigt, welche die Hardwarebeschleunigung der Grafikkarte nutzt. Die Implementierung schneidet im Vergleich zu anderen aktuellen Implementierungen von Graphenzeichenverfahren sowohl in Bezug auf Layoutqualität als auch in Bezug auf Laufzeit gut ab. Sie bietet die Möglichkeit, zusätzlich zu zweidimensionalen auch dreidimensionale Layouts von Graphen anzufertigen, die sich mittels des eigens entwickelten Visualisierungsprogramms anzeigen lassen. Dieses Visualisierungsprogramm bietet ferner die Option, durch die dreidimensionale Ansicht des Graphen zu navigieren.

Die vorliegende Masterarbeit teilt sich in insgesamt acht Abschnitte auf. Nach dieser Einleitung folgt in Abschnitt 2 zunächst die Behandlung der mathematischen Grundlagen der verwendeten MDS-Verfahren und eine detailierte Beschreibung dieser. Abschnitt 3 zeigt die kombinierte MDS, Vorteile, welche dieses Verfahren mit sich bringt und Möglichkeiten es zu parallelisieren. Für die eigene parallele Implementierung dieses Verfahrens auf der Grafikkarte wurde CUDA, eine von Nvidia entwickelte Architektur, verwendet, in welche Abschnitt 4 eine Einführung gibt und Möglichkeiten aufzeigt, CUDA-Programme zu beschleunigen. Abschnitt 5 beschreibt diese Implementierung und begründet anhand von Laufzeitanalysen, warum ausgewählte Verfahren und Verbesserungsmöglichkeiten in die Implementierung eingegangen sind. Weiter zeigt Abschnitt 5 von der eigenen Implementierung generierte Layouts und veranschaulicht anhand dieser zuvor getroffene Aussagen. Abschnitt 6 stellt andere schnelle parallele Implementierungen von Graphzeichenverfahren vor und vergleicht diese mit der eigenen Implemtierung in Bezug auf Laufzeiten und generierte Layouts. Eine Zusammenfassung der Ergebnisse und Möglichkeiten zur weiteren Verbesserung der eigenen Implementierung finden sich in Abschnitt 7.

2 Multidimensionale Skalierung

Dieser Abschnitt erklärt Verfahren der Multidimensionalen Skalierung und baut auf [4], [22] und [9] auf. Dabei wurden die dort enthaltenen Informationen um eigene Herleitungen und Beweise erweitert.

2.1 Mathematischer Ausgangspunkt

Ziel der Multidimensionalen Skalierung (MDS) ist, eine Menge V von Elementen $\{v_1, \ldots, v_n\}$, die aus einem metrischen Raum $\mathbb{R}^h$ stammen und von denen lediglich ihre paarweisen Verschiedenheiten d_{ij} bekannt sind, möglichst ohne Informationsverlust in ihren Verschiedenheiten auf Koordinaten $x_1, \ldots, x_n$ in einem euklidischen Zielraum $\mathbb{R}^d$ abzubilden.
Für diese Aufgabe ist zunächst die Matrix $D = (d_{ij}) \in \mathbb{R}^{n \times n}$ gegeben, die als Einträge die paarweisen Verschiedenheiten der Elemente v_i und v_j beinhaltet. Alle Einträge sind auf Grund der Eigenschaften eines metrischen Raums positiv. Ferner ist D symmetrisch und hat eine Nulldiagonale, da auf dieser die Abstände eines Elements zu sich selbst enthalten sind. Nun ist eine Matrix $X = [x_1, \ldots, x_n]^T \in \mathbb{R}^{n \times d}$ zu bestimmen, deren Einträge zeilenweise die gesuchten Koordinaten der abbgebildeten $v_i \in V$ im euklidischen Zielraum $\mathbb{R}^d$ enthalten. $\|x_i - x_j\|$ gibt dabei den euklidischen Abstand zweier Bildpunkte x_i und x_j an, welcher der ursprünglichen paarweisen Verschiedenheit d_{ij} möglichst genau entspricht.

Für die dabei vorkommenden Räume und Dimensionalitäten gilt:

- *Quellraum und dessen Dimensionalität h*: Quellraum ist der Raum $\mathbb{R}^h$, aus dem die Elemente $v_1, \ldots, v_n \in V$ stammen. Er muss eine Metrik haben; diese muss jedoch nicht euklidisch sein. Seine Dimensionalität ist für MDS nicht von Bedeutung.

- *Intrinsische Dimensionalität der Verschiedenheiten*: Die intrinsische Dimensionalität der Verschiedenheiten wird mit r bezeichnet und entspricht der Anzahl der Dimensionen, die derjenige Unterraum des h-dimensionalen Quellraums hat, in dem sich alle Elemente $v_1, \ldots, v_n \in V$ befinden. Damit gilt $r \leq h$. Außerdem gilt $r < n$, da sich n Elemente einem Unterraum zuordnen lassen, der höchstens $n - 1$ Dimensionen hat.

- *Zielraum und dessen Dimensionalität*: Der Zielraum ist der Raum $\mathbb{R}^d$, in den abgebildet wird. Seine Metrik muss euklidisch sein. Der Anwender bestimmt dessen Dimensionalität d.

Angenommen, die Verschiedenheiten von $v_1, \ldots, v_n$ seien euklidische Abstände.
Falls $r \leq d$ gilt, können $v_1, \ldots, v_n$ so auf Koordinaten im Zielraum abgebildet werden, dass die aus der Abbildung resultierenden Abstände genau den paarweisen Verschiedenheiten entsprechen. MDS führt in diesem Fall Koordinaten aus bekannten paarweisen euklidischen Abständen zurück, wobei keine Information verloren geht.

Falls allerdings $r > d$ gilt, projiziert MDS die durch die Abstände implizierten höherdimensionalen Koordinaten von $v_1, \ldots, v_n$ auf Koordinaten im Zielraum. Der dabei in den Verschiedenheiten entstehende Informationsverlust wird minimiert, indem die d einflussreichsten der r-vielen intrinsischen Dimensionen projiziert werden. Das Verfahren zur Auswahl der einflussreichsten Dimensionen stellt Abschnitt 2.4 vor.

Wenn die Verschiedenheiten von $v_1, \ldots, v_n$ keine euklidischen Abstände sind, bildet MDS diese mit möglichst geringem Fehler in den Verschiedenheiten auf Koordinaten im Zielraum ab. Auch diese Abbildung ist eine Projektion. Je nachdem, wie stark die Verschiedenheiten von euklidischen Abständen abweichen, geht mehr oder weniger Information verloren. Wenn $r > d$ gilt, spielt dies für den Informationsverlust ebenfalls eine Rolle. Genauere Erläuterungen hierzu finden sich ebenfalls in 2.4.

Gutes Ergebnis: Als gutes Ergbnis wird im Rahmen dieser Arbeit eine Menge von Koordinaten $x_1, \ldots, x_n$ im euklidischen Zielraum bezeichnet, deren paarweise euklidische Abstände $\|x_i - x_j\|$ mit möglichst geringem Informationsverlust den paarweisen Verschiedenheiten der Elemente $v_1, \ldots, v_n \in V$ entsprechen.

2.2 Anwendung für Graphenzeichnen

Im Rahmen dieser Arbeit dient MDS dem Zeichnen von Graphen. Ein Graph $G = (V, E)$ ist hier ein Tupel bestehend aus einer Menge von Knoten V und einer Menge von Kanten $E \subseteq V \times V$. In diesem Zusammenhang sei n die Anzahl der Knoten, m die Anzahl der Kanten. Kanten repräsentieren paarweise Verbindungen von Knoten. Die Distanz eines Knoten zu sich selbst sei Null, die Distanz zweier durch eine Kante verbundenen Knoten sei eins und die Distanz zweier beliebiger Knoten entspreche der minimalen Anzahl von Kanten über die sie indirekt verbunden sind. Knoten entsprechen den in 2.1 eingeführten Elementen des Quellraums, womit die in 2.1 eingeführten Verschiedenheiten im Kontext des Graphenzeichnens zu den Distanzen äquivalent sind. Im Rahmen dieser Arbeit werden ausschließlich ungerichtete, schleifenfreie Graphen betrachtet, da Distanzen in einem gerichteten Graphen keiner Metrik genügen und Schleifen keinen Einfluss auf die geometrische Position der Knoten haben sollten.
Aus den paarweisen Distanzen der Knoten $v \in V$ werden deren Koordinaten für eine zwei- oder dreidimensionale Zeichnung berechnet. Zunächst ist es nötig, die Distanzen zu bestimmen. Möglich ist dies durch n Breitensuchen, von denen jede den Abstand von einem der Knoten $w \in V$ zu allen anderen bestimmt. Die asymptotische Laufzeit einer Breitensuche ist $\mathcal{O}(m)$. Daher benötigt die Bestimmung aller Distanzen $\mathcal{O}(nm)$, womit sie für große Graphen nicht in akzeptabler Zeit durchführbar ist. Die Abschnitte 2.5 und 2.7 erläutern MDS-Verfahren, welche lediglich die Distanzen einer ausgewählten Knotenmenge zu allen anderen Knoten benötigen. Sie sind für große Graphen geeignet, falls die ausgewählte Knotenmenge eine konstante Größe hat.

Für die meisten Graphen sind die Distanzen keine euklidischen Abstände, weshalb eine Zeichnung ohne Informationsverlust nicht möglich ist. Abbildung 2.1 veranschaulicht dies.
Da auch meistens die Anzahl der intrinsischen Dimensionen der Distanzen größer ist als zwei

oder drei, geht weitere Information durch Projektion verloren. Je weniger Information verloren geht, desto besser ist die resultierende Zeichnung.

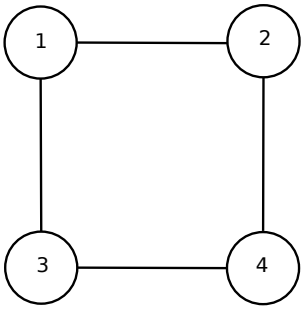


Abbildung 2.1: Bei diesem Graphen können die Knotendistanzen nicht fehlerfrei auf euklidische Abstände abgebildet werden. Die Distanzen von benachbarten Knoten sind jeweils 1, die Distanzen von 1 zu 4 und von 2 zu 3 jeweils 2. Unter der zusätzlichen Bedingung, dass die Abstände von 1 zu 4 und von 2 zu 3 wie in der Zeichnung gleich groß sein sollen, wäre ihr Abstand $\sqrt{2}$. Setzte man hingegen zusätzlich zur Bedingung, dass die Abstände von Nachbarn 1 betragen, den Abstand von 1 zu 4 auf 2, würde der Abstand von 3 zu 4 Null betragen.

2.3 Verwendete Notation

Dieser Abschnitt stellt kurz die wichtigsten im weiteren Verlauf der Arbeit verwendeten Notationen vor und erläutert ihren mathematischen Hintergrund.

Skalarprodukt: Die verwendete Schreibweise für das Skalarprodukt zweier Vektoren a und b ist $\langle a, b \rangle$.

Matrixelemente: Die Elemente einer Matrix $A \in \mathbb{R}^{m \times n}$ werden mit a_{ij} für alle $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, n\}$ bezeichnet. Dabei ist a_{ij} das Element in Zeile i und Spalte j. Die Elemente eines Matrixprodukts AB bzw. einer transponierten Matrix werden $[AB]_{ij}$ bzw. $[A^T]_{ij}$ geschrieben.

Vektoren einer Matrix: Die Zeilenvektoren einer Matrix A werden $a_{i\bullet}$, die Spaltenvektoren $a_{\bullet j}$ geschrieben. Eine Ausnahme bilden die Zeilenvektoren von Koordinatenmatrizen, z.B. X. Sie heißen x_i statt $x_{i\bullet}$, da die Matrix über diese Vektoren definiert wurde und die Vektoren auch außerhalb des Matrixkontextes Verwendung finden. Die Spaltenvektoren der Eigenvektormatrix U heißen u_i anstatt $u_{\bullet j}$.

2.4 Klassische MDS

Als klassische MDS bezeichnet man das erste MDS-Verfahren, welches in [24] vorgestellt wurde. Der bei der klassischen MDS verwendete Algorithmus, welcher (näherungsweise[1]) jene $x_1, \ldots, x_n$ findet, die die Gleichung

$$d_{ij} = \|x_i - x_j\| \tag{2.1}$$

erfüllen, lässt sich durch Umformen ebendieser Gleichung herleiten. Ziel der Umformung ist es, x_i ohne Abhängigkeit von den übrigen Koordinaten darzustellen.
Da Abstände in einem euklidischen Raum translationsinvariant sind, kann o.B.d.A. angenommen werden, dass die Koordinaten, welche rekonstruiert werden sollen, im Ursprung zentriert sind, sodass

$$\sum_{i=1}^{n} x_i = 0 \tag{2.2}$$

gilt. Diese Tatsache wird später genutzt.
Zunächst sind folgende Umformungsschritte nötig:

$d_{ij}^2 = \|x_i - x_j\|^2$	beidseitiges Quadrieren
$d_{ij}^2 = \langle x_i - x_j, x_i - x_j \rangle$	Die quadrierte Länge eines Vektors entspricht dem Skalarprodukt des Vektors mit sich selbst.

Durch Anwenden der binomischen Formal erhält man den Ausdruck

$$d_{ij}^2 = \langle x_i, x_i \rangle - 2\langle x_i, x_j \rangle + \langle x_j, x_j \rangle, \tag{2.3}$$

der aufgelöst nach $\langle x_i, x_j \rangle$ die Beziehung

$$\langle x_i, x_j \rangle = -\frac{1}{2}\left(d_{ij}^2 - \langle x_i, x_i \rangle - \langle x_j, x_j \rangle\right) \tag{2.4}$$

liefert.

Koordinatenunabhängige Darstellung von $\langle x_i, x_j \rangle$

Die Skalarprodukte $\langle x_i, x_j \rangle$ in (2.4) gilt es nun ohne Abhängigkeit von $x_1, \ldots, x_n$ auszudrücken. Dies kann man mit Hilfe einer Doppelzentrierung der Matrix $D^{(2)}$ erreichen, wobei $D^{(2)} = (d_{ij}^2)$ gilt. Dazu werden der Spaltendurchschnitt, der Zeilendurchschnitt und der Gesamtdurchschnitt von D verwendet, die gemäß

[1] Falls die Verschiedenheiten euklidische Abstände sind, können die Koordinaten, wie in Abschnitt 2.1 gezeigt, ohne Informationsverlust in den Verschiedenheiten bestimmt werden. Ansonsten findet MDS eine Näherung. Für die folgenden Gleichungen gilt die Gleichheit in (2.1) deshalb nur, falls d_{ij} euklidische Abstände sind. Abschnitt 2.4 erklärt diese Näherung.

$\frac{1}{n}\sum_{i=1}^{n} d_{ij}^2$ Durchschnitt der Spalte j

$\frac{1}{n}\sum_{j=1}^{n} d_{ij}^2$ Durchschnitt der Zeile i

$\frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}^2$ Gesamtdurchschnitt der Matrix $D^{(2)}$

definiert sind. Durch Einsetzen der rechten Seite von Gleichung (2.3) ergibt sich für den Durchschnitt der Spalte j der Ausdruck

$$\frac{1}{n}\sum_{i=1}^{n} d_{ij}^2 = \frac{1}{n}\sum_{i=1}^{n}\langle x_i, x_i\rangle - 2\langle x_i, x_j\rangle + \langle x_j, x_j\rangle. \tag{2.5}$$

Auf Grund der Ursprungszentrierung (2.2) ergeben alle x_i aufaddiert in jeder Komponente Null. Daher gilt auch für die Summe $\sum_{i=1}^{n}\langle x_i, x_j\rangle = 0$, da wenn $\sum_{i=1}^{n} a_i = 0$ gilt, $\sum_{i=1}^{n} a_i b = 0$ zutrifft. Da die beiden Faktoren $\frac{1}{n}$ und -2 dafür keine Rolle spielen, erhält man für den Spaltendurchschnitt den Term

$$\frac{1}{n}\sum_{i=1}^{n}\langle x_i, x_i\rangle + \langle x_j, x_j\rangle + \underbrace{\frac{1}{n}\sum_{i=1}^{n} -2\langle x_i, x_j\rangle}_{=0 \text{ wegen Ursprungszentrierung}}.$$

Weil x_j innerhalb einer Spalte und damit innerhalb dieser Summe gleich bleibt, kann $\langle x_j, x_j\rangle$ ausgeklammert werden, sodass

$$\langle x_j, x_j\rangle + \frac{1}{n}\sum_{i=1}^{n}\langle x_i, x_i\rangle$$

folgt. Für den Zeilendurchschnitt gilt analog:

$$\langle x_i, x_i\rangle + \frac{1}{n}\sum_{j=1}^{n}\langle x_j, x_j\rangle.$$

Der Gesamtdurchschnitt lässt sich nun über den Durchschnitt der Zeilendurchschnitte herleiten und liefert den Ausdruck

$$\frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}^2 = \frac{1}{n}\sum_{i=1}^{n}\left(\langle x_i, x_i\rangle + \frac{1}{n}\sum_{j=1}^{n}\langle x_j, x_j\rangle\right).$$

Teilt man die Summe wie folgt auf, lässt sich der rechte Teil wie unter der Klammer in (2.5) dargestellt vereinfachen, da die innere Summe von i unabhängig ist; es gilt nämlich $\frac{1}{n}\sum_{i=1}^{n} k = k$,

k entspricht hier der inneren Summe. Ferner findet eine Umbenennung des Index' der inneren Summe von j in i statt. Man erhält somit für die rechte Seite den Ausdruck

$$\frac{1}{n}\sum_{i=1}^{n}\langle x_i, x_i\rangle + \underbrace{\frac{1}{n}\sum_{i=1}^{n}\frac{1}{n}\sum_{j=1}^{n}\langle x_j, x_j\rangle}_{\frac{1}{n}\sum_{i=1}^{n}\langle x_i, x_i\rangle}.$$

Durch Addieren der beiden Teile vereinfacht sich die rechte Seite zu

$$\frac{2}{n}\sum_{i=1}^{n}\langle x_i, x_i\rangle.$$

Zusammenfassend ergeben sich nachfolgende Ausdrücke

$\frac{1}{n}\sum_{i=1}^{n} d_{ij}^2 = \langle x_j, x_j\rangle + \frac{1}{n}\sum_{i=1}^{n}\langle x_i, x_i\rangle$ Durchschnitt der Spalte j

$\frac{1}{n}\sum_{j=1}^{n} d_{ij}^2 = \langle x_i, x_i\rangle + \frac{1}{n}\sum_{j=1}^{n}\langle x_j, x_j\rangle$ Durchschnitt der Zeile i

$\frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}^2 = \frac{2}{n}\sum_{i=1}^{n}\langle x_i, x_i\rangle$ Gesamtdurchschnitt der Matrix D.

Mit Hilfe dieser Durchschnitte lässt sich das Skalarprodukt $\langle x_i, x_j\rangle$ gemäß

$$\langle x_i, x_j\rangle = -\frac{1}{2}\left(d_{ij}^2 - \frac{1}{n}\sum_{i=1}^{n} d_{ij}^2 - \frac{1}{n}\sum_{j=1}^{n} d_{ij}^2 + \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}^2\right)$$

ohne Abhängigkeit von $x_1, \ldots, x_n$ darstellen. Die Korrektheit dieser Darstellung lässt sich zeigen, indem man die rechten Seiten der in obiger Zusammenfassung gezeigten Gleichungen einsetzt und auflöst, woraus

$$\begin{aligned}
&-\frac{1}{2}(d_{ij}^2 - \overbrace{\langle x_j, x_j\rangle + \frac{1}{n}\sum_{i=1}^{n}\langle x_i, x_i\rangle}^{\frac{1}{n}\sum_{i=1}^{n} d_{ij}^2} - \overbrace{\langle x_i, x_i\rangle + \frac{1}{n}\sum_{j=1}^{n}\langle x_j, x_j\rangle}^{\frac{1}{n}\sum_{j=1}^{n} d_{ij}^2} + \overbrace{\frac{2}{n}\sum_{i=1}^{n}\langle x_i, x_i\rangle}^{\frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}^2})\\
&= \underbrace{-\frac{1}{2}(d_{ij}^2 - \langle x_j, x_j\rangle - \langle x_i, x_i\rangle}_{\text{Ausgangsdarstellung von } \langle x_i, x_j\rangle} \underbrace{- \sum_{i=1}^{n}\langle x_i, x_i\rangle - \sum_{j=1}^{n}\langle x_j, x_j\rangle + \frac{2}{n}\sum_{i=1}^{n}\langle x_i, x_i\rangle)}_{0}
\end{aligned}$$

folgt. Von nun an wird eine weitere Matrix $B = (b_{ij}) \in \mathbb{R}^{n\times n}$ verwendet, deren Einträge die Skalarprodukte $\langle x_i, x_j\rangle$ bilden, sodass

$$b_{ij} := \langle x_i, x_j\rangle = -\frac{1}{2}\left(d_{ij}^2 - \frac{1}{n}\sum_{i=1}^{n} d_{ij}^2 - \frac{1}{n}\sum_{j=1}^{n} d_{ij}^2 + \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}^2\right) \tag{2.6}$$

gilt.

Spektralzerlegung von B

Da B die Matrix der Skalarprodukte $\langle x_i, x_j \rangle$ ist, lässt sie sich mittels der Gleichheit

$$B = XX^T \tag{2.7}$$

ausdrücken. Deshalb kann man X mittels Spektralzerlegung von B rekonstruieren. Weil B eine symmetrische Matrix ist, also $b_{ij} = \langle x_i, x_j \rangle = \langle x_j, x_i \rangle = b_{ji}$ gilt, sind alle Eigenwerte von B reell. Außerdem sind die Einträge von B reell, weshalb es möglich ist, orthonormale Eigenvektoren $u_1, \ldots, u_n$ von B zu bestimmen. Orthonormale Eigenvektoren genügen den Bedingungen $\|u_i\| = 1$ und $\langle u_i, u_j \rangle = 0$ für alle i, j mit $i \neq j$. Die dazugehörigen Eigenwerte seien o.B.d.A $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$. Damit kann B mittels

$$B = U\Lambda U^{-1} = U\Lambda U^T \tag{2.8}$$

zerlegt werden. In dieser Zerlegung ist U die Matrix mit den Eigenvektoren von B als Spaltenvektoren ($U = [u_1, \ldots, u_n] \in \mathbb{R}^{n \times n}$) und Λ die Matrix mit den entsprechenden Eigenwerten auf der Diagonalen ($\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_n) \in \mathbb{R}^{n \times n}$).
Da die Eigenvektoren von B linear unabhängig sind, gilt $U^{-1} = U^T$.

Falls die Verschiedenheiten von $v_1, \ldots, v_n$ euklidische Abstände sind, gibt es, wie in Abschnitt 2.1, keinen Informationsverlust in den Verschiedenheiten, insbesondere nicht in Gleichung 2.7. B ist damit das Produkt einer Matrix mit sich selbst (Cholesky-Zerlegung) und folglich positiv semidefinit. Deshalb sind die Eigenwerte von B größer oder gleich Null. Die Anzahl der Eigenwerte, welche nicht Null sind, entspricht r, der Anzahl der intrinsischen Dimensionen der Verschiedenheiten von $v_1, \ldots, v_n$. Dieser Zusammenhang lässt sich an der Struktur von $B = XX^T$ erkennen. Die Anzahl der Vektoren, die den Spaltenraum von X aufspannen, entspricht r und ebenfalls dem Rang von X. Ein Matrixprodukt $B = XX^T$ hat den gleichen Rang wie die Matrix X; der Rang einer Matrix entspricht der Anzahl ihrer verschiedenen Eigenwerte. Damit hat B r-viele Eigenwerte.
Weil $r \leq n - 1$ gilt, gibt es mindestens einen Nulleigenwert. Die notwendige Existenz eines Nulleigenwerts lässt sich an der Struktur der Matrix B erkennen.
Auf Grund der Ursprungszentrierung gilt $\sum_{i=1}^{n} \langle x_i, x_j \rangle = 0$, daher ist die Summe der Zeilenvektoren von B Null. Damit befindet sich mindestens $(1, \ldots, 1) \in \mathbb{R}^n$ im linken Nullraum von B, womit gezeigt ist, dass mindestens ein Eigenwert von B den Wert Null hat.

Jeder Eigenwert von B, der größer als Null ist, repräsentiert eine intrinsische Dimension. Die Dimensionen werden an Hand dieser Eigenwerte folgendermaßen eingeteilt:

- *Beachtete (einflussreichste) Dimensionen:* Die einflussreichsten Dimensionen sind jene, die zu den d größten Eigenwerten gehören. Mittels dieser bestimmt MDS die Koordinaten im Zielraum, welcher d-dimensional ist.

- *Nicht beachtete Dimensionen:* Die weiteren $r - d$ intrinsischen Dimensionen, die zu den übrigen Eigenwerten > 0 gehören, werden nicht beachtet. Je größer die entsprechenden Eigenwerte sind, desto mehr Information geht verloren.

Falls die Verschiedenheiten von $v_1, \ldots, v_n$ keine euklidischen Abstände sind, entstehen Fehler, da jene nur näherungsweise auf diese abgebildet werden können. B entspricht deshalb nur näherungsweise XX^T, womit B nicht mehr positiv semidefinit ist und daher negative Eigenwerte haben kann. Die negativen Eigenwerte repräsentieren den nicht euklidischen Anteil der Verschiedenheiten. Jeder negative Eigenwert repräsentiert eine Dimension, deren Hinzufügen zu den Ergebniskoordinaten dazu führt, dass ihre Abstände nicht mehr euklidisch sind. Deshalb gibt es in diesem Fall eine weitere Gruppe von Dimensionen:

- *"Nicht euklidische" Dimensionen:* Dimensionen, die von einem negativen Eigenwert repräsentiert werden, beachtet MDS nicht, da sie für eine Abbildung auf euklidische Koordinaten nicht verwendbar sind. Je größer die negativen Eigenwerte, desto größer der Informationsverlust.

Herleitung der Koordinatenmatrix X

Aus den Gleichungen (2.7) und (2.8) lässt sich X über die Beziehung

$$B = XX^T = U\Lambda U^{-1} = U\Lambda U^T$$

herleiten. Ein bestimmtes x_{ij} kommt in den Elementen der i-ten Zeile und in denen der j-ten Spalte von B vor (in $b_{i\bullet}$ und $b_{\bullet j}$), insbesondere in b_{ii}. Für b_{ii} gilt

$$b_{ii} = \sum_{k=1}^{d} x_{ik}^2 = \sum_{l=1}^{n} \lambda_l u_{il}^2.$$

Da der euklidische Raum, in den abgebildet werden soll, d Dimensionen hat, spielen, wie oben erwähnt, nur die größten d Eigenvektoren für die Koordinaten eine Rolle. Deshalb kann man $\lambda_{d+1} = \cdots = \lambda_n = 0$ annehmen, sodass

$$b_{ii} = \sum_{k=1}^{d} x_{ik}^2 = \sum_{k=1}^{d} \lambda_k u_{ik}^2 + \underbrace{\sum_{l=d+1}^{n} \lambda_l u_{il}^2}_{0}$$

folgt. Daraus lässt sich eine Lösung für x_{ij} herleiten[2]:

$$\begin{aligned} x_{ij}^2 &= \lambda_j u_{ij}^2 \\ x_{ij} &= \sqrt{\lambda_j} u_{ij}. \end{aligned}$$

[2]Dies schließt nicht aus, dass es noch weitere Lösungen gibt. Sämtliche Multiplikationen von X mit einer Spaltenpermutationsmatrix P wären z.B. weitere Lösungen für X.

Für die Matrix X ergibt sich damit

$$X = U_{(d)}\Lambda_{(d)}^{\frac{1}{2}}, \tag{2.9}$$

wobei $\Lambda_{(d)} \in \mathbb{R}^{d\times d}$ die Diagonalmatrix mit den d größten Eigenwerten sei und $U_{(d)} \in \mathbb{R}^{n\times d}$ die Matrix mit den dazugehörenden Eigenvektoren als Spaltenvektoren. Für die Spaltenvektoren $x_{\bullet j}$ von X gilt damit die Gleichung

$$x_{\bullet j} = \sqrt{\lambda_j}u_j \,, \forall j \in \{1, \ldots, d\}. \tag{2.10}$$

Wie oben erwähnt, sind die gesuchten Koordinaten die Zeilenvektoren von X.

Interpretation als Projektion

Wie bereits in Abschnitt 2.1 beschrieben, projiziert MDS die Verschiedenheiten in einen euklidischen Raum, falls die Anzahl ihrer intrinsischen Dimensionen größer ist als die Dimensionalität des euklidischen Zielraums oder falls es nicht euklidische intrinsische Dimensionen gibt. Ansonsten rekonstruiert MDS die Koordinaten verlustfrei aus den Verschiedenheiten. Zur Veranschaulichung lassen sich diese beiden Schritte trennen:

1. Die Projektion besteht darin, aus B die Beiträge der Eigenvektoren zu entfernen, welche zu nicht beachteten Dimensionen gehören:

$$B_p = B - \sum_{i=d+1}^{n-1} \lambda_i u_i u_i^T .$$

2. B_p ist damit positiv semidefinit und hat d Eigenwerte, die größer Null sind[3]. Die Zeilenvektoren der Matrix X, welche die Cholesky-Zerlegung $B_p = XX^T$ liefert, entsprechen den gesuchten Koordinaten.

Das Ergebnis mit maximalem Informationsgehalt erhält man, wenn durch die Projektion alle euklidischen intrinsischen Dimensionen erhalten bleiben. Man entfernt also aus B den Einfluss der nicht euklidischen Komponenten der Verschiedenheiten. In diesem Fall ist B_p diejenige positiv semidefinite Matrix, welche B am ähnlichsten ist.

Die von der klassischen MDS minimierte Fehlerfunktion Strain

Die Lösung aus (2.9) minimiert die Funktion

$$\text{Strain}(X) = \sum_{i=1}^{n}\sum_{j=1}^{n}\left(b_{ij} - \langle x_i, x_j\rangle\right)^2 . \tag{2.11}$$

Falls die Verschiedenheiten von $v_1, \ldots, v_n$ euklidische Abstände mit intrinsischer Dimensionalität d sind, ist dies direkt ersichtlich, da in diesem Fall $B = XX^T$ und damit $\text{Strain}(X) = 0$ gilt. Falls

[3] Außer wenn $r < d$ gilt.

nicht, lässt sich die Korrektheit wie folgt zeigen.
Zunächst wird B zerlegt, wie im vorhergehenden Abschnitt gezeigt. Für B_p ergibt sich damit der Ausdruck

$$B_p = B - \sum_{i=d+1}^{n-1} \lambda_i u_i u_i^T = XX^T.$$

Die Komponenten der Matrizen B_p, B und $B - B_p$ lassen sich über

$$\begin{aligned} B &= \operatorname{diag}(\lambda_1, \ldots, \lambda_n) \\ B_p &= \operatorname{diag}(\lambda_1, \ldots, \lambda_d, 0, \ldots, 0) \in \mathbb{R}^{n \times n} \\ \sum_{i=d+1}^{n-1} \lambda_i u_i u_i^T &= \operatorname{diag}(\lambda_{d+1}, \ldots, \lambda_n, 0, \ldots, 0) \in \mathbb{R}^{n \times n} \end{aligned}$$

in der Orthonormalbasis der Eigenvektoren von B schreiben. Strain(X) ist die quadrierte Hilbert-Schmidt-Norm von $B - XX^T$:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \left(b_{ij} - \langle x_i, x_j \rangle \right)^2 = \left\| B - XX^T \right\|^2.$$

Da die quadrierte Hilbert-Schmidt-Norm für alle Orthonormalbasen gleich ist und $B - XX^T = \sum_{i=d+1}^{n-1} \lambda_i u_i u_i^T$ gilt, folgt die Gleichung

$$\left\| B - XX^T \right\|^2 = \left\| \operatorname{diag}(\lambda_{d+1}, \ldots \lambda_n, 0, \ldots, 0) \right\|^2 = \sum_{i=d+1}^{n-1} \lambda_i^2.$$

Die letzte Summe ist bedingt durch die Auswahl dieser Eigenwerte minimal, da alle negativen Eigenwerte mit eingebunden werden müssen und von den positiven Eigenwerten die kleinsten enthalten sind. Damit ist gezeigt, dass die klassische MDS Strain(X) minimiert.

Wie oben bereits erwähnt, bilden MDS-Verfahren die Verschiedenheiten auf Koordinatenabstände im Zielraum ab. Für die Zielkoordinaten eines Elements spielen deshalb die Verschiedenheiten zu allen anderen Elementen eine Rolle. Die klassische MDS minimiert, wie die Fehlerfunktion Strain(X) zeigt, die Summe über die (quadrierten) Fehler der Koordinatenskalarprodukte. Jeder Zielraumabstand $\|x_i - x_j\|$ der Elemente v_i, v_j wird in dieser Funktion durch das Skalarprodukt $\langle x_i, x_j \rangle$ repräsentiert. Diese Multiplikation der Koordinaten verstärkt den Informationsverlust bezüglich des Abstands $\|x_i - x_j\|$ umso mehr, je größer die Komponenten von x_i und x_j sind. Da sich aus diesem Grund große Abstände stärker auf den Fehler auswirken, bildet die klassische MDS jene großen Abstände genauer ab, weil das wiederum diesen Fehler am stärksten minimiert.

Berechnung

Die Potenziteration ist für die Berechnung von Eigenvektoren eine gute Wahl, sofern nur wenige Eigenvektoren berechnet werden sollen [25]. Dies ist im Anwendungsgebiet Graphenzeichnen

der Fall, da meistens $d \in \{2, 3\}$ gilt. Die Potenziteration bestimmt iterativ das Eigenpaar mit dem betragsmäßig größten Eigenwert. Durch mehrmaliges Anwenden lässt sich jeweils das Eigenpaar mit dem betragsmäßig nächstgrößten Eigenwert berechnen. Da nur wenige Eigenvektoren benötigt werden und, wie oben erwähnt, MDS für Graphen, deren entsprechende Matrix B große negative Eigenwerte hat, nicht geeignet ist, ist es sehr wahrscheinlich, die Eigenvektoren in der gewünschten Reihenfolge zu erhalten. Außerdem ist es möglich, schrittweise mehr Dimensionen hinzuzuziehen, sofern man den letzten berechneten Eigenwert für groß genug befindet.

Um den Eigenvektor u_1 mit dem betragsmäßig größten Eigenwert λ_1 einer Matrix A zu bestimmen, wird in der ersten Iteration ein Initialvektor $u_1^{[0]}$ mit der Matrix A multipliziert. Dieser Initialvektor darf weder der Nullvektor sein, noch darf er orthogonal zu einem der Eigenvektoren der Matrix A stehen. Jede weitere Iteration multipliziert den Ergebnisvektor der vorherigen Iteration mit A. Um zu große Werte zu vermeiden, normalisiert das Verfahren den Ergebnisvektor nach jeder Iteration gemäß

$$u_1^{[t]} = \frac{Au_1^{[t-1]}}{||Au_1^{[t-1]}||}.$$

Es kann gezeigt werden, dass

$$\lim_{t\to\infty} u_1^{[t]} = u_1$$

gilt [25]. Die Potenziteration stoppt, sobald sich das Skalarprodukt von $u_1^{[t]}$ mit $u_1^{[t-1]}$ in einer bestimmten ϵ-Umgebung um 1 befindet, sodass

$$\langle u_1^{[t]}, u_1^{[t-1]} \rangle = 1 - \epsilon$$

erfüllt ist. $u_1^{[t]}$ ist normalisiert; ein normalisierter Vektor multipliziert mit sich selbst ergibt 1. Das heißt die Potenziteration stoppt, wenn der Unterschied zwischen $u_1^{[t]}$ und $u_1^{[t-1]}$ klein genug ist.
Für den zu u_1 gehörigen Eigenwert λ_1 gilt die Gleichung

$$\lambda_1 = \lim_{t\to\infty} ||Au_1^{[t]}||.$$

Anschaulich erklärt funktioniert die Potenziteration aus folgendem Grund:
Die Multiplikation einer Matrix A mit einem Vektor $u_1^{[0]}$ ist eine lineare Abbildung dieses Vektors. Er wird dadurch in die Richtungen der Eigenvektoren der Matrix verschoben. Je größer der Eigenwert eines Eigenvektors, desto größer ist der Einfluss seiner Richtung auf die Verschiebung von u_1. Jede Iteration verschiebt $u_1^{[t]}$ folglich am stärksten in Richtung des Eigenvektors mit dem betragsmäßig größten Eigenwert. Nach mehrmaliger Wiederholung dominiert die Richtung dieses Eigenvektors immer stärker, sodass $u_1^{[t]}$ näherungsweise in seine Richtung zeigt. Damit ist der Eigenvektor gefunden, seine Länge spielt keine Rolle.
Die Länge von $Au_1^{[t]}$ ist für ein genügend großes t eine gute Näherung an λ_1, da nach der Definition des Eigenwertproblems $Au_1 = \lambda_1 u_1$ gilt und weil u_1 normiert ist, entspricht seine Länge nach der Multiplikation mit A dem Eigenwert λ_1.

Die weiteren Eigenpaare mit betragsmäßig kleineren Eigenvektoren lassen sich analog bestimmen. Um das d-te Eigenpaar zu berechnen, muss zunächst der Beitrag der ersten $d-1$ Eigenvektoren gemäß

$$A_d = A - \sum_{i=1}^{d-1} \lambda_i u_i u_i^T$$

aus der Matrix entfernt werden. Danach lässt sich das Verfahren auf die Matrix A_d anwenden. Diese Vorgehensweise ist aber nur dann günstig, wenn die Matrix, deren Eigenwerte man berechnen möchte, wenig Nulleinträge enthält, wovon man bei B ausgehen kann.

Pseudocode

Algorithmus 2.1 zeigt Pseudocode, welcher die klassische MDS beschreibt.

Algorithmus 2.1 : Klassische MDS

Input:

- $D \in \mathbb{R}^{n \times n}$, Matrix von paarweisen Verschiedenheiten d_{ij}
- d, gewünschte Anzahl der Dimensionen des Zielraums

Output: $X \in \mathbb{R}^{n \times d}$, Koordinatenmatrix mit Zeilenvektoren $x_1, \ldots, x_n \in \mathbb{R}^d$

Berechnen von B:
Berechne $D^{(2)}$
for $j \in \{1, \ldots, n\}$ **do**
  $\text{Sds}[j] \leftarrow \frac{1}{n} \sum_{i=1}^{n} [D^{(2)}]_{ij}$ //Spaltendurchschnitt
end
for $i \in \{1, \ldots, n\}$ **do**
  $\text{Zds}[i] \leftarrow \frac{1}{n} \sum_{j=1}^{n} [D^{(2)}]_{ij}$ //Zeilendurchschnitt
end
$\text{Gds} \leftarrow \frac{1}{n} \sum_{i=1}^{n} \text{Zds}[i]$ //Gesamtdurchschnitt
foreach $b_{ij} \in B \in \mathbb{R}^{n \times n}$ **do**
  $b_{ij} = -\frac{1}{2} \left([D^{(2)}]_{ij} - \text{Sds}[j] - \text{Zds}[i] + \text{Gds} \right)$
end

Spektralzerlegung:
$u_1, \ldots, u_d \in \mathbb{R}^n \leftarrow$ Zufallsinitialisierung
$i \leftarrow 1$
while $i \leq d$ **do**
  while $\langle u_i, u_i^{[t-1]} \rangle \neq 1 - \epsilon$ **do**
    $u_i^{[t-1]} \leftarrow u_i$
    $u_i \leftarrow \frac{Bu_i}{\|Bu_i\|}$
  end
  $\lambda_i = \|Bu_i\|$
  $B \leftarrow B - \lambda_i u_i u_i^T$
  if $\lambda_i \geq 0$ **then**
    $i \leftarrow i + 1$
  end
end

Rekonstruieren von X:
for $j \in \{1, \ldots, d\}$ **do**
  $x_{\bullet j} \leftarrow \sqrt{\lambda_j}\, u_j$
end

Asymptotische Laufzeit

Berechnung von $D^{(2)}$	$\mathcal{O}(n^2)$, da jedes der $n \times n$ Elemente von D quadriert werden muss.
Doppelzentrierung von $D^{(2)}$	$\mathcal{O}(n^2)$, dabei benötigen der Zeilen- bzw. Spaltendurchschnitt jeweils $\mathcal{O}(n^2)$ und der Gesamtdurchschnitt $\mathcal{O}(n)$, da dieser aus dem Durchschnitt der Zeilendurchschnitte berechnet wird.
Berechnung von B gesamt	$\mathcal{O}(n^2)$
Potenziteration für einen Eigenvektor	$\mathcal{O}(cn^2)$, dabei benötigt die Multiplikation einer $n \times n$ Matrix mit einem n-elementigen Vektor $\mathcal{O}(n^2)$. c ist die Anzahl der Iterationen.
Entfernen des Beitrags der ersten d Eigenvektoren	$\mathcal{O}(dn^2)$, d-maliges Berechnen eines "outer product" der entsprechenden Eigenvektoren
Potenziteration gesamt	$\mathcal{O}(n^2)$, angenommen d und c sind konstant.
Isolieren von X	$\mathcal{O}(dn)$, jeder der d n-elementigen Ergebnisvektoren muss mit dem entsprechenden Eigenwert multipliziert werden.
Gesamt	$\mathcal{O}(n^2)$

2.5 Pivot-MDS

Pivot-MDS betrachtet lediglich die paarweisen Verschiedenheiten einer ausgewählten Menge von Pivotelementen $p_1, \ldots, p_q \in P \subset V, q << n$ und der Elemente $v_1, \ldots, v_n$. Statt der Matrix aller paarweisen Verschiedenheiten $D \in \mathbb{R}^{n \times n}$ wird eine reduzierte Matrix $D_P \in \mathbb{R}^{n \times q}$ verwendet. D_P besteht folglich aus einer Teilmenge der Spalten von D. Für ein konstantes q reduziert sich die Laufzeit im Vergleich zur klassischen MDS um eine Größenordnung. Das Ziel ist analog zur klassischen MDS über

$$d_{P,ij} \approx \|x_i - x_j\| \; \forall i \in V, j \in P \tag{2.12}$$

definiert. Statt der Matrix $B = (b_{ij}) \in \mathbb{R}^{n \times n}$ verwendet Pivot-MDS eine Matrix $C = (c_{ij}) \in \mathbb{R}^{n \times q}$:

$$c_{ij} := \langle x_i, x_j \rangle = -\frac{1}{2}\left(d^2_{P,ij} - \frac{1}{n}\sum_{i=1}^{n} d^2_{P,ij} - \frac{1}{q}\sum_{j=1}^{q} d^2_{P,ij} + \frac{1}{nq}\sum_{i=1}^{n}\sum_{j=1}^{q} d^2_{P,ij}\right). \tag{2.13}$$

Sämtliche Herleitungsschritte sind analog zu denen der klassischen MDS. Wichtig ist, dass der Spaltenindex j bei Pivot MDS nicht wie bei der klassischen MDS von 1 bis n (über die Spalten von D) läuft, sondern nur von 1 bis q (über die Spalten von D_P). Weiter ist es wichtig zu beachten, dass $d_{P,ij}$ nicht die Verschiedenheit der Elemente v_i, v_j enthält, sondern die Verschiedenheit von v_i, p_j. Welchem Element $v \in V$ p_j entspricht, hängt von der Auswahl der Pivotelemente ab.

Auswahl der Pivotelemente

Dieser Teilabschnitt stellt Möglichkeiten zur Auswahl der Pivotelemente vor.

Zufallsauswahl: Dabei werden die Pivotelemente zufällig aus der Menge der Elemente ausgewählt. Vorteil dieser Strategie ist der geringe Zeitaufwand. Als Nachteil tritt dagegen auf, dass diese Strategie zu einer schlechten Verteilung der Pivotelemente über die Menge der Elemente führen kann.

min-max-Auswahl: Dabei wählt man zunächst ein Pivotelement p_0 zufällig aus. Die weiteren Pivotelemente p_i werden bestimmt, indem zunächst zu jedem Element v_j die minimale paarweise Verschiedenheit zu den Pivotelementen $p_0, \ldots, p_{i-1}$ bestimmt wird (min). Das Element v_j, dessen entsprechende Verschiedenheit am größten ist, ist das nächste Pivotelement p_i (max). Der Zeitaufwand dieser Methode ist größer als bei der Zufallsauswahl, sie sorgt jedoch für eine gute Verteilung der Pivotelemente. Für das Zeichnen von Graphen können die Pivotelemente während den Breitensuchen zur Distanzberechnung bestimmt werden.

Approximieren der Eigenwerte von B

Pivot-MDS verwendet die Matrix C, um die Eigenwerte von B zu approximieren. Dazu wird ausgenutzt, dass die Eigenwerte einer Matrix A denen der Matrix AA^T mit quadrierten Eigenwerten entsprechen. Es ist also ausreichend zu zeigen, dass die Eigenvektoren von CC^T eine Näherung derer von BB^T sind; CC^T ist wie B und BB^T aus dem Raum $\mathbb{R}^{n\times n}$.
Bei einer optimalen Menge P von Pivotelementen stimmte CC^T bis auf einen Proportionalitätsfaktor c mit BB^T überein, sodass die Gleichung

$$CC^T = cBB^T$$

erfüllt wäre. Die Eigenvektoren von CC^T wären zu denen von BB^T identisch, wobei die Eigenwerte von CC^T denen von BB^T multipliziert mit c entsprächen.
Im Normalfall weicht jedes Element $[CC^T]_{ij}$ um einen Fehlerwert ϵ_{ij} von dieser Proportionalität ab, was sich über

$$[CC^T]_{ij} = (c + \epsilon_{ij})[BB^T]_{ij}$$

darstellen lässt. Je kleiner $\sum_{ij\in\{1,\ldots,n\}} \epsilon_{ij}$ ist, desto besser war die Pivotauswahl und desto besser ist die Näherung der Eigenwerte von CC^T an die von BB^T und damit an jene von B. Diese Fehlersumme beschreibt den zusätzlichen Informationsverlust, zu dem Pivot-MDS im Vergleich zur klassischen MDS führt.

Effiziente Spektralzerlegung von CC^T

Wichtig zu erklären ist die Berechnung der Eigenwerte von CC^T. Würde man auf dieser Matrix eine Potenziteration durchführen, so entspräche die Laufzeit derjenigen zur Berechnung der Eigenwerte von B, da CC^T die gleichen Dimensionen hat.
Für die Herleitung der für eine Laufzeitreduktion nötigen Schritte sind die Äquivalenzen

$$Aq_1^{[t-1]} = A^t q_1^{[0]} = A^c q_1^{[t-c]}, c \leq t$$

geeignet. Sie lassen sich unter Berücksichtigung der Assoziativität der Matrixmultiplikation zu

$$\underbrace{A(\dots A(\underbrace{Aq_1^{[0]}}_{q_1^{[1]}}))\dots)}_{q_1^{[t-1]}} = \underbrace{A\dots AA}_{A^t} q_1^{[0]} = \underbrace{A(..\underbrace{(AA)}_{A^2}..)}_{A^c} \underbrace{A(..A(\underbrace{Aq_1^{[0]}}_{q_1^{[1]}}))..)}_{q_1^{[t-c]}}$$

herleiten. Dies lässt sich folgendermaßen anschaulich erklären: Wenn eine Matrix A mit sich selbst multipliziert wird ($A^2 = AA^T$), bleiben ihre Eigenvektoren gleich, jedoch mit quadrierten Eigenwerten. Multipliziert man A^2 mit einem Vektor q, so wird dieser stärker in die Richtung des Eigenvektors mit dem betragsmäßig größten Eigenwert verschoben, als multiplizierte man A mit diesem Vektor. Dadurch ist intuitiv klar, dass eine ausreichend hohe Potenz einer Matrix A multipliziert mit q, q fast ausschließlich in Richtung dieses Eigenvektors von A verschiebt, womit q eine Näherung von diesem ist.

Damit lässt sich die Potenziteration von CC^T wie folgt schreiben:

$$CC^T u_1^{[t-1]} = \left(CC^T\right)^t u_1^{[0]} = \underbrace{CC^T \dots CC^T}_{(CC^T)^t} u_1^{[0]} = \underbrace{C\left(C^TC\right)\dots\left(C^TC\right)C^T}_{(CC^T)^t} u_1^{[0]}.$$

Es ist nun ersichtlich, dass

$$C \lim_{t\to\infty} \frac{C^TCu_1^{[t]}}{\|C^TCu_1^{[t]}\|} = \lim_{t\to\infty} \frac{CC^Tu_1^{[t]}}{\|CC^Tu_1^{[t]}\|}$$

gilt. Auf das anfängliche Multiplizieren mit C^T kann verzichtet werden, da $u_1^{[0]}$ beliebig ist, bis auf die oben erwähnten Einschränkungen.[4] Die Potenziteration kann damit auf C^TC durchgeführt werden. C^TC ist aus dem Raum $\mathbb{R}^{q\times q}$, was die Laufzeit um zwei Größenordnungen reduziert.

Herleitung der Koordinatenmatrix X

Nach den Regeln der Singulärwertzerlegung entsprechen die Eigenvektoren von C^TC den rechten Singulärvektoren von C, die Eigenvektoren von CC^T den linken Singulärvektoren. Die gewünschten Eigenvektoren von CC^T, welche eine Näherung der Eigenvektoren von B^2 sind, können daher über

$$\sigma_i u_i^l = Cu_i^r$$

[4] C^T kann jedoch eine gute Initialisierung sein.

berechnet werden, wobei u_i^l die linken und u_i^r die rechten Singulärvektoren von C seien und σ_i die Singulärwerte von C, für die die Gleichung $\sigma_i = \sqrt{\lambda_i}$ gilt[5]. Für die Berechnung ist zum einen eine Multiplikation mit C nötig, die bereits aus der obigen Erklärung der Potenziteration ersichtlich ist, zum anderen muss u_i^r durch σ_i^2 geteilt werden um die Eigenvektoren von CC^T zu erhalten[6]. Daher gilt für die Eigenvektoren von CC^T die Gleichung

$$u_{CC^T{}_i} = \frac{1}{\lambda_i} u_{C^T C_i} C.$$

Wurden die Eigenvektoren mit der im vorangegangenen Teilabschnitt beschriebenen Potenziteration berechnet, müssen sie also mit $\frac{1}{\lambda_i}$ multipliziert werden.
Mit Hilfe der Eigenvektoren von CC^T ist die Herleitung der Koordinatenmatrix X analog zur klassischen MDS und damit über

$$X = U_{(d)} \Lambda_{(d)}^{\frac{1}{4}}$$

durchzuführen. Dabei enthält die Matrix $U_{(d)}$ die Eigenvektoren $u_{CC^T{}_1}, \ldots, u_{\Lambda_{(d)_d}^{\frac{1}{4}}}$ und $\Lambda_{(d)}^{\frac{1}{4}}$ die vierten Wurzeln der Eigenwerte von CC^T. Die vierte Wurzel ist nötig, da die Eigenwerte von CC^T eine Näherung der Eigenwerte von B^2 sind, womit aus diesen die Wurzel gezogen werden muss, um eine Näherung an die Eigenwerte von B zu erhalten und aus diesen muss, wie bei der klassischen MDS, wiederum die Wurzel gezogen werden, wodurch sich insgesamt die vierte Wurzel ergibt.

Pseudocode

Algorithmus 2.2 und 2.3 zeigen Pseudocode, welcher Pivot-MDS beschreibt.

[5] λ_i sind dabei die Eigenwerte von CC^T. Sie sind nach der Definition des Eigenwertproblems gleich den Eigenwerten von $C^T C$.

[6] Da die Eigenwerte von CC^T und $C^T C$ den quadrierten Singulärwerten von C entsprechen, muss durch σ_i^2 $(= \lambda_i)$ anstatt σ_i geteilt werden.

Algorithmus 2.2 : Pivot-MDS

Input:

- $D_p \in \mathbb{R}^{n \times q}$, Matrix von paarweisen Verschiedenheiten der Pivotelemente von allen Elementen
- d, gewünschte Anzahl der Dimensionen des Zielraums

Output: $X \in \mathbb{R}^{n \times d}$, Koordinatenmatrix mit Zeilenvektoren $x_1, \ldots, x_n \in \mathbb{R}^d$

Berechnen von C:
Berechne $D_p^{(2)}$
for $j \in \{1, \ldots, q\}$ **do**
  $\text{Sds}[j] \leftarrow \frac{1}{n} \sum_{i=1}^{n} [D_p^{(2)}]_{ij}$
end
for $i \in \{1, \ldots, n\}$ **do**
  $\text{Zds}[i] \leftarrow \frac{1}{n} \sum_{j=1}^{q} [D_p^{(2)}]_{ij}$
end
$\text{Gds} \leftarrow \frac{1}{n} \sum_{i=j}^{q} \text{Sds}[j]$
foreach $c_{ij} \in C \in \mathbb{R}^{n \times q}$ **do**
  $c_{ij} \leftarrow -\frac{1}{2} \left([D_p^{(2)}]_{ij} - \text{Sds}[j] - \text{Zds}[i] + \text{Gds} \right)$
end

Berechnen von $C^T C$

Spektralzerlegung:
$u_1, \ldots, u_d \in \mathbb{R}^n \leftarrow$ Zufallsinitialisierung
$i \leftarrow 1$
while $i \leq d$ **do**
  while $\langle u_i, u_i^{[t-1]} \rangle \neq 1 - \epsilon$ **do**
    $u_i^{[t-1]} \leftarrow u_i$
    $u_i \leftarrow \frac{C^T C u_i}{\|C^T C u_i\|}$
  end
  $\lambda_i \leftarrow \|C^T C u_i\|$
  $C^T C \leftarrow C^T C - \lambda_i u_i u_i^T$
  if $\lambda_i \geq 0$ **then**
    $i \leftarrow i + 1$
  end
end

Algorithmus 2.3 : Pivot-MDS Fortsetzung

for $j \in \{1, \ldots, d\}$ **do**
  $u_j \leftarrow Cu_j$
end

Rekonstruieren von X:
for $j \in \{1, \ldots, d\}$ **do**
  $x_{\bullet j} \leftarrow \frac{1}{\sqrt[4]{\lambda_j}} u_j$
end

Asymptotische Laufzeit

Berechnung von $D_p^{(2)}$	$\mathcal{O}(qn)$, da jedes der $n \times q$ Elemente von D quadriert werden muss.
Doppelzentrierung von $D_p^{(2)}$	$\mathcal{O}(qn)$, dabei benötigen der Zeilen- bzw. Spaltendurchschnitt jeweils $\mathcal{O}(qn)$ und der Gesamtdurchschnitt $\mathcal{O}(q)$, da dieser aus dem Durchschnitt der Spaltendurchschnitte berechnet wird.
Berechnung von C gesamt	$\mathcal{O}(qn)$
Berechnung von C^TC	$\mathcal{O}(q^2n)$
Potenziteration für einen Eigenvektor	$\mathcal{O}(cq^2)$, dabei benötigt die Multiplikation einer $q \times q$ Matrix mit einem q-elementigen Vektor $\mathcal{O}(q^2)$. c ist die Anzahl der Iterationen.
Entfernen des Beitrags der ersten d Eigenvektoren	$\mathcal{O}(dq^2)$, d-maliges Berechnen eines "outer product" der entsprechenden Eigenvektoren.
Potenziteration gesamt	$\mathcal{O}(q^2)$, angenommen d und c sind konstant
Multiplikation der Eigenvektoren mit C	$\mathcal{O}(dqn)$, jeder der d q-elementigen Eigenvektoren muss mit der $n \times q$ Matix C multipliziert werden.
Isolieren von X	$\mathcal{O}(dn)$, jeder der d n-elementigen Ergebnisvektoren muss mit dem entsprechenden Eigenwert multipliziert werden.
Gesamt	$\mathcal{O}(n)$, angenommen, q ist konstant.

2.6 Stressmajorisierung

Die klassische MDS minimiert, wie in Abschnitt 2.4 beschrieben, den Fehler der Skalarprodukte der Koordinaten. Falls die Verschiedenheiten der Elemente $v_1, \ldots, v_n$ keine euklidischen Abstände sind, führt dies zu einem schlechteren Ergebnis als würden die Abstände selbst optimiert werden. MDS-Verfahren, welche direkt die euklidischen Abstände möglichst genau anpassen, sind unter dem Begriff "Distanz Skalierung" zusammengefasst. Die Stressmajorisierung, welche Thema dieses Abschnitts ist, gehört dazu. Diese Verfahren versuchen ebenfalls für das in Abschnitt 2.1 beschriebene MDS-Problem

$$d_{ij} = \|x_i - x_j\| \; \forall i, j \in \{1, \ldots, n\} \tag{2.14}$$

eine möglichst gute Näherung zu finden, jedoch ohne den Umweg über Skalarprodukte. Wie in Abschnitt 2.1 beschrieben, ist eine exakte Lösung nur möglich, wenn die Verschiedenheiten d_{ij} Abstände in einem euklidischen Raum sind und die intrinsische Dimension dieser Verschiedenheiten nicht größer als die des Zielraums ist, in dem sich die Koordinaten $x_1, \ldots, x_n$ befinden.
Die zur Lösung verwendete Vorgehensweise ist die Minimierung folgender Funktion, welche den Unterschied zwischen d_{ij} und $\|x_i - x_j\|$ für alle ungeordneten Paare $\{i, j\}$ ausdrückt. Diese Funktion wird Stress genannt und ist über

$$\sigma(X) = \sum_{i=1}^{n} \sum_{j=1}^{i-1} \left(d_{ij} - \|x_i - x_j\|\right)^2 \tag{2.15}$$

definiert. Das globale Minimum dieser Funktion ist die bestmögliche Näherung der euklidischen Abstände der Zielkoordinaten an die Verschiedenheiten d_{ij}.

Gewichtungen

Im Gegensatz zur klassischen MDS kann die Stressmajorisierung mit beliebigen Gewichtungen umgehen, indem die Stressfunktion $\sigma(X)$ um einen Gewichtungsfaktor ω_{ij} erweitert wird, sodass die Stressfunktion die Gestalt

$$\sigma(X) = \sum_{i=1}^{n} \sum_{j=1}^{i-1} \omega_{ij} \left(d_{ij} - \|x_i - x_j\|\right)^2 \tag{2.16}$$

annimmt. Der Beitrag eines jeden Unterschieds einer Verschiedenheit vom entsprechenden Koordinatenabstand kann ein eigenes Gewicht erhalten. Für gewöhnlich setzt man ω_{ij} abhängig von der entsprechenden Verschiedenheit d_{ij}. Folgende Beispiele sollen dies verdeutlichen:

- $\omega_{ij} = d_{ij}$: Je größer die Verschiedenheit, desto stärker wird sie gewichtet.
- $\omega_{ij} = 1$: Jede Verschiedenheit wird gleich gewichtet.

- $\omega_{ij} = d_{ij}^{-\alpha}$: Je kleiner die Verschiedenheit, desto stärker wird sie gewichtet; für größere α umso mehr.

Die Wahl des Gewichts ist ein Kompromiss zwischen lokaler und globaler Genauigkeit des Ergbnislayouts. Lokale Genauigkeit bedeutet, dass Knoten im Verhältnis zu ihrer lokalen Umgebung (Nachbarschaft) gut positioniert sind, sodass der euklidische Abstand der Bildpunkte benachbarter Knoten möglichst genau ihren paarweisen Distanzen entspricht, während globale Genauigkeit meint, dass das Ergbnislayout möglichst gut die Struktur des Gesamtgraphen repräsentiert.
Für das Zeichnen von Graphen ist meistens $\omega_{ij} = d_{ij}^{-\alpha}$ sinnvoll, da Knoten, welche in der Nähe eines anderen Knotens sind, einen möglichst großen Einfluss auf dessen Position haben sollten. Es wurde gezeigt, dass $\alpha = 2$ die besten Ergebnisse liefert [22].

Finden der Majorante

Da die Stressfunktion lokale Minima haben kann, ist das Finden des globalen Minimums nur mit großem Aufwand möglich. Weil keine explizite Form für die Minima von $\sigma(X)$ bekannt ist, muss das Problem iterativ gelöst werden. Die Stressmajorisierung verwendet dafür eine Majorante der Stressfunktion, zu deren Herleitung auf $\sigma(X)$ (Gleichung 2.16) die binomische Formel gemäß

$$\sigma(X) = \sum_{i=1}^{n} \sum_{j=1}^{i-1} \omega_{ij} d_{ij}^2 + \omega_{ij} \|x_i - x_j\|^2 - 2\omega_{ij} d_{ij} \|x_i - x_j\|$$

angewandt wird. Die Komponenten lassen sich in getrennte Summen aufteilen. Bei der ersten Komponente handelt es sich um einen Term, der unabhängig von den Koordinaten ist und damit für eine bestimmte Matrix von Verschiedenheiten D konstant bleibt, sodass sich

$$\sigma(X) = \underbrace{\sum_{i=1}^{n} \sum_{j=1}^{i-1} \omega_{ij} d_{ij}^2}_{Konstante\ k} + \sum_{i=1}^{n} \sum_{j=1}^{i-1} \omega_{ij} \|x_i - x_j\|^2 - 2 \sum_{i=1}^{n} \sum_{j=1}^{i-1} \omega_{ij} d_{ij} \|x_i - x_j\|$$

ergibt. Da das Produkt eines Vektors mit sich selbst geteilt durch seine Länge wieder seine Länge ergibt, also $\frac{\langle a,a \rangle}{\|a\|} = \|a\|$ gilt[7], folgt die Beziehung

$$\sigma(X) = k + \sum_{i=1}^{n} \sum_{j=1}^{i-1} \omega_{ij} \|x_i - x_j\|^2 - 2 \sum_{i=1}^{n} \sum_{j=1}^{i-1} \omega_{ij} d_{ij} \frac{\langle x_i - x_j, x_i - x_j \rangle}{\|x_i - x_j\|}.$$

Unter Berücksichtigung der Cauchy-Schwarz-Ungleichung

$$\|x\| \, \|y\| \geq \langle x, y \rangle \tag{2.17}$$

[7]Das Produkt eins Vektors mit sich selbst ist das Quadrat seiner Länge $\langle a, a \rangle = \|a\|^2$.

lässt sich eine Majorante $\tau(X, Y)$ der Gestalt

$$\tau(X, Y) = k + \sum_{i=1}^{n} \sum_{j=1}^{i-1} \omega_{ij} \|x_i - x_j\|^2 - 2 \sum_{i=1}^{n} \sum_{j=1}^{i-1} \omega_{ij} d_{ij} \frac{\langle x_i - x_j, y_i - y_j \rangle}{\|y_i - y_j\|} \tag{2.18}$$

finden. Für jede Koordinatenmatrix $Y = [y_1, \ldots, y_n]^T \in \mathbb{R}^{n \times d}$ gilt $\sigma(X) \leq \tau(X, Y)$ mit Gleichheit, falls $X = Y$ erfüllt ist, was nun gezeigt wird.
Da sich $\sigma(X)$ und $\tau(X, Y)$ lediglich im letzten Term unterscheiden, muss dafür nur dieser betrachtet werden. Weil der letzte Term ein negatives Vorzeichen hat, muss er bei $\tau(X, Y)$ kleiner sein, sodass

$$\sum_{i=1}^{n} \sum_{j=1}^{i-1} \omega_{ij} d_{ij} \frac{\langle x_i - x_j, y_i - y_j \rangle}{\|y_i - y_j\|} \overset{!}{\leq} \sum_{i=1}^{n} \sum_{j=1}^{i-1} \omega_{ij} d_{ij} \frac{\langle x_i - x_j, x_i - x_j \rangle}{\|x_i - x_j\|}$$

gilt.

$$\frac{\langle x_i - x_j, y_i - y_j \rangle}{\|y_i - y_j\|} \overset{!}{\leq} \frac{\langle x_i - x_j, x_i - x_j \rangle}{\|x_i - x_j\|} \; \forall i, j \in \{1, \ldots, n\}$$

Da, wie oben erwähnt, das Produkt eines Vektors mit sich selbst geteilt durch seine Länge wieder seine Länge ergibt, gilt die Ungleichung

$$\frac{\langle x_i - x_j, y_i - y_j \rangle}{\|y_i - y_j\|} \leq \|x_i - x_j\|.$$

Durch beidseitiges Multiplizieren mit $\|y_i - y_j\|$ ergibt sich die Beziehung

$$\langle x_i - x_j, y_i - y_j \rangle \leq \|x_i - x_j\| \, \|y_i - y_j\|,$$

was nach der Cauchy-Schwarz Ungleichung korrekt ist.
Damit gilt $\sigma(X) \leq \tau(X, Y) \; \forall Y \in \mathbb{R}^{n \times d}$.

Die Funktion $\tau(X, Y)$ hat auf Grund des festen, bekannten Y den Vorteil, dass sie eine konvexe Funktion ist, sie ist nämlich eine quadratische Funktion der x_{ij} und hat damit keine lokalen Minima sondern, nur ein globales Minimum. Dieses findet sich durch Nullsetzen der Ableitung von $\tau(X, Y)$ und Auflösen des resultierenden linearen Gleichungssystems.
Die Stressmajorisierung geht dabei folgendermaßen vor:

1. Ausgehend von einer Initialkonfiguartion $X^{[0]}$ berechnet die Stressmajorisierung in einer ersten Iteration das X, welches $\tau(X, X^{[0]})$ minimiert. Wegen der oben beschriebenen Eigenschaften von $\tau(X, Y)$ gilt die Ungleichung

$$\min_X \sigma(X) \leq \min_X \tau(X, X^{[0]}) \leq \underbrace{\tau(X^{[0]}, X^{[0]}) = \sigma(X^{[0]})}_{\text{konstant}}.$$

2. Das in der aktuellen Iteration berechnete $X^{[t]}$, welches $\tau(X, X^{[t-1]})$ minimiert, wird in der Iteration $t + 1$ als neue feste Konfiguration Y verwendet. Es ist garantiert, dass der Stress

von Iteration zu Iteration nicht zunimmt, was über

$$\underbrace{\min_X \tau(X, X^{[t]})}_{\text{Berechnung in Iteration } i+1} \leq \underbrace{\min_X \tau(X, X^{[t-1]})}_{\text{Berechnung in der aktuellen Iteration}} = \tau(X^{[t]}, X^{[t-1]})$$

ausgedrückt werden kann.

Wichtig zu beachten ist, dass die Stressmajorisierung zwar in jeder Iteration das globale Minimum der Majorante berechnet, welches sich immer mehr einem Minimum der eigentlichen Stressfunktion nähert; dieses kann jedoch ein lokales Minimum der Stressfunktion sein. Deshalb ist die Ergebnisgüte der Stressmajorisierung von der Startkonfiguration $X^{[0]}$ abhängig. Weiter bestimmt die Startkonfiguration die Dimensionalität des Zielraums.

Matrixschreibweise der Majorante

Dieser Abschnitt zeigt die Korrektheit der Matrixschreibweise

$$\tau(X, Y) = k + \mathrm{Tr}(X^T L^\omega X) - 2\mathrm{Tr}(X^T L^Y Y) \tag{2.19}$$

von $\tau(X, Y)$. Dabei ist $\mathrm{Tr}(A) = \sum_{i=1}^{n} a_{ii}$ die Spur einer Matrix A, $L^\omega \in \mathbb{R}^{n\times n}$ ist die gewichtete Laplace'sche Matrix und $L^Y \in \mathbb{R}^{n\times n}$ eine von Y abhängige Laplace'sche Matrix. Diese Matrizen sind über

$$L^\omega = \begin{cases} -\omega_{ij} & i \neq j \\ \sum\limits_{k\in\{1,\dots,n\}\setminus\{i\}} \omega_{ik} & i = j \end{cases}$$

$$L^Y = \begin{cases} -\omega_{ij} d_{ij} \mathrm{inv}\left(\|y_i - y_j\|\right) & i \neq j \\ -\sum\limits_{k\in\{1,\dots,n\}\setminus\{i\}} [L^Y]_{ik} & i = j \end{cases}$$

definiert, wobei

$$\mathrm{inv}(x) = \begin{cases} \frac{1}{x} & x \neq 0 \\ 0 & x = 0 \end{cases}$$

gilt. Die erste Komponente ist in beiden Darstellungen k. Die Gleichheit der zweiten Komponenten

$$\mathrm{Tr}(X^T L^\omega X) \stackrel{!}{=} \sum_{i=1}^{n} \sum_{j=1}^{i-1} \omega_{ij} \|x_i - x_j\|^2$$

lässt sich erkennen, wenn die Elemente der Matrix $L^\omega X$ unter Verwendung der Definition der gewichteten Laplace'schen Matrix über

$$([L^\omega X]_{ij}) = \sum_{k=1}^{n} [L^\omega]_{ik} x_{kj} = \underbrace{\left(\sum_{k\in\{1,\dots,n\}\setminus\{i\}} \omega_{ik}\right) x_{ij}}_{\text{Summand für } i=k} + \sum_{k\in\{1,\dots,n\}\setminus\{i\}} -\omega_{ik} x_{kj}$$

ausgedrückt werden. Die Elemente der Matrix $X^T L^\omega X$ ergeben sich damit zu

$$([X^T L^\omega X]_{ij}) = \sum_{l=1}^{n} \underbrace{x_{li}}_{[X^T]_{il}} [L^\omega X]_{lj} = \sum_{l=1}^{n} x_{li} \left(\left(\sum_{k \in \{1,\dots,n\} \setminus \{l\}} \omega_{lk} \right) x_{lj} + \sum_{k \in \{1,\dots,n\} \setminus \{l\}} -\omega_{lk} x_{kj} \right).$$

$X^T L^\omega X$ ist eine $d \times d$-Matrix. Für ihre Spur gilt der Ausdruck

$$\mathrm{Tr}(X^T L^\omega X) = \sum_{i=1}^{d} [X^T L^\omega X]_{ii} = \sum_{i=1}^{d} \sum_{l=1}^{n} x_{li} \left(\left(\sum_{k \in \{1,\dots,n\} \setminus \{l\}} \omega_{lk} \right) x_{li} + \sum_{k \in \{1,\dots,n\} \setminus \{l\}} -\omega_{lk} x_{ki} \right).$$

Das innere x_{li} kann in die Summe gezogen werden. Durch anschließende Vereinigung der beiden innersten Summen ergibt sich für die rechte Seite der Gleichung der Term

$$\sum_{i=1}^{d} \sum_{l=1}^{n} x_{li} \left(\sum_{k \in \{1,\dots,n\} \setminus \{l\}} \omega_{lk} x_{li} - \omega_{lk} x_{ki} \right).$$

Das äußere x_{li} lässt sich in die innerste Summe ziehen, sodass man

$$\sum_{i=1}^{d} \sum_{l=1}^{n} \sum_{k \in \{1,\dots,n\} \setminus \{l\}} \omega_{lk} x_{li}^2 - \omega_{lk} x_{ki} x_{li}$$

erhält. Die innere Doppelsumme läuft über alle geordneten Paare $(l,k) : l \neq k; l,k \in \{1,\dots,n\}$. Ihre Summanden lassen sich umsortieren, indem man die zu den Paaren (l,k) und (k,l) gehörigen Summanden nebeneinander schreibt. Somit kann man eine Doppelsumme über alle ungeordneten Paare $\{l,k\} : l \neq k; l,k \in \{1,\dots,n\}$ verwenden, deren Summanden die Ausdrücke enthält, in denen $\{l,k\}$ vorkommt.[8] Dadurch ergibt sich der Ausdruck

$$\sum_{i=1}^{d} \sum_{l=1}^{n} \sum_{k=1}^{l-1} \omega_{lk} x_{li}^2 - \omega_{lk} x_{ki} x_{li} + \underbrace{\omega_{kl}}_{\omega_{lk}} x_{ki}^2 - \underbrace{\omega_{kl}}_{\omega_{lk}} x_{li} x_{ki}.$$

Es wird davon ausgegangen, dass die Gewichte in Abhängigkeit von den Verschiedenheiten gewählt wurden. Damit gilt auf Grund der in Abschnitt 2.1 eingeführten Symmetrieeigenschaft $d_{ij} = d_{ji}$, dass die Gewichte ebenfalls dieser Symmetrieeigenschaft genügen, also $\omega_{ij} = \omega_{ji}$ erfüllt ist. Nach Ausklammern von ω_{lk} und Addieren der gleichen Komponenten ergibt sich

$$\sum_{i=1}^{d} \sum_{l=1}^{n} \sum_{k=1}^{l-1} \omega_{lk} \left(x_{li}^2 + -2 x_{ki} x_{li} + x_{ki}^2 \right),$$

woraus durch inverses Anwenden der binomischen Formel und Umordnen der Summe der Ausdruck

$$\sum_{l=1}^{n} \sum_{k=1}^{l-1} \sum_{i=1}^{d} \omega_{lk} \left(x_{li} - x_{ki} \right)^2$$

[8] Die anfänglichen Summanden kann man sich als $n \times n$-Matrix mit "leerer" Diagonale vorstellen. Die Umformung entspricht einem elementweisen Addieren der transponierten oberen Dreiecksmatrix mit der unteren Dreiecksmatrix. Die Ergebniselemente sind Summanden der gewünschten Doppelsumme.

gewonnen werden kann. Die Summe über die quadrierten Komponenten eines Vektors entspricht der quadrierten Länge dieses Vektors. Dies beachtend und durch Umbennen der Indizes ergibt sich schließlich die gewünschte Form

$$\sum_{i=1}^{n}\sum_{j=1}^{i-1}\omega_{ij}\|x_i - x_j\|^2.$$

Damit ist die Gleichheit gezeigt.

Der Beweis der Gleichheit der dritten Komponenten

$$\mathrm{Tr}(X^T L^Y Y) \overset{!}{=} \sum_{i=1}^{n}\sum_{j=1}^{i-1}\omega_{ij}d_{ij}\frac{\langle x_i - x_j, y_i - y_j\rangle}{\|y_i - y_j\|}$$

ist dem eben aufgeführten sehr ähnlich. Zunächst sollen wieder die Elemente des ersten Matrixprodukts $L^Y Y$ dargestellt werden, was sich über

$$([L^Y Y]_{ij}) = \sum_{k=1}^{n}[L^Y]_{ik}y_{kj} = \underbrace{\left(-\sum_{k\in\{1,\dots,n\}\setminus\{i\}} -\omega_{ik}d_{ik}\mathrm{inv}\left(\|y_i - y_k\|\right)\right)y_{ij}}_{\text{Summand für } i=k} + \sum_{k\in\{1,\dots,n\}\setminus\{i\}} -\omega_{ik}d_{ik}\mathrm{inv}\left(\|y_i - y_k\|\right)y_{kj}$$

bewerkstelligen lässt. Dies beachtend ergibt sich für die $n \times n$-Matrix $X^T L^Y Y$ der Ausdruck

$$([X^T L^Y Y]_{ij}) = \sum_{l=1}^{n}\underbrace{x_{li}}_{[X^T]_{il}}[L^Y Y]_{lj} =$$

$$\sum_{l=1}^{n} x_{li}\left(\left(-\sum_{k\in\{1,\dots,n\}\setminus\{l\}} -\omega_{lk}d_{lk}\mathrm{inv}\left(\|y_l - y_k\|\right)\right)y_{lj} + \sum_{k\in\{1,\dots,n\}\setminus\{l\}} -\omega_{lk}d_{lk}\mathrm{inv}\left(\|y_l - y_k\|\right)y_{kj}\right).$$

Ebenfalls analog zum obigen Beweis gilt für die Spur von $X^T L^Y Y$ die Gleichung

$$\mathrm{Tr}(X^T L^Y Y) = \sum_{i=1}^{d}[X^T L^Y Y]_{ii} =$$

$$\sum_{i=1}^{d}\sum_{l=1}^{n} x_{li}\left(\left(-\sum_{k\in\{1,\dots,n\}\setminus\{l\}} -\omega_{lk}d_{lk}\mathrm{inv}\left(\|y_l - y_k\|\right)\right)y_{li} + \sum_{k\in\{1,\dots,n\}\setminus\{l\}} -\omega_{lk}d_{lk}\mathrm{inv}\left(\|y_l - y_k\|\right)y_{ki}\right).$$

y_{li} und das Minus werden in die Summe gezogen, die innersten Summen vereinigt. Somit erhält man

$$\sum_{i=1}^{d}\sum_{l=1}^{n} x_{li}\left(\sum_{k\in\{1,\dots,n\}\setminus\{l\}} \omega_{lk}d_{lk}\mathrm{inv}\left(\|y_l - y_k\|\right)y_{li} - \omega_{lk}d_{lk}\mathrm{inv}\left(\|y_l - y_k\|\right)y_{ki}\right).$$

Anschließend wird x_{li} in die Summe gezogen, woraus

$$\sum_{i=1}^{d}\sum_{l=1}^{n}\sum_{k\in\{1,\dots,n\}\setminus\{l\}} \omega_{lk}d_{lk}\mathrm{inv}\left(\|y_l-y_k\|\right)y_{li}x_{li} - \omega_{lk}d_{lk}\mathrm{inv}\left(\|y_l-y_k\|\right)y_{ki}x_{li}$$

folgt. Durch Ausklammern ergibt sich der Ausdruck

$$\sum_{i=1}^{d}\sum_{l=1}^{n}\sum_{k\in\{1,\dots,n\}\setminus\{l\}} \omega_{lk}d_{lk}\mathrm{inv}\left(\|y_l-y_k\|\right)\left(y_{li}x_{li}-y_{ki}x_{li}\right).$$

Das folgende Umordnen der Summe ist analog zur entsprechenden Stelle im vorausgegangenen Beweis. Die Erklärung dazu steht ebenfalls dort. Damit folgt der Term

$$\sum_{i=1}^{d}\sum_{l=1}^{n}\sum_{k=1}^{l-1}\omega_{lk}d_{lk}\mathrm{inv}\left(\|y_l-y_k\|\right)\left(y_{li}x_{li}-y_{ki}x_{li}\right)+\underbrace{\omega_{kl}d_{kl}\mathrm{inv}\left(\|y_k-y_l\|\right)}_{\omega_{lk}d_{lk}\mathrm{inv}(\|y_l-y_k\|)}\left(y_{ki}x_{ki}-y_{li}x_{ki}\right).$$

Die weiteren Schritte führen schrittweises Ausklammern durch und lassen sich über

$$\begin{aligned}
&\sum_{i=1}^{d}\sum_{l=1}^{n}\sum_{k=1}^{l-1}\omega_{lk}d_{lk}\mathrm{inv}\left(\|y_l-y_k\|\right)\left(\left(y_{li}x_{li}-y_{ki}x_{li}\right)+\left(y_{ki}x_{ki}-y_{li}x_{ki}\right)\right)\\
&\sum_{i=1}^{d}\sum_{l=1}^{n}\sum_{k=1}^{l-1}\omega_{lk}d_{lk}\mathrm{inv}\left(\|y_l-y_k\|\right)\left(x_{li}\left(y_{li}-y_{ki}\right)-x_{ki}\left(-y_{ki}+y_{li}\right)\right)\\
&\sum_{i=1}^{d}\sum_{l=1}^{n}\sum_{k=1}^{l-1}\omega_{lk}d_{lk}\mathrm{inv}\left(\|y_l-y_k\|\right)\left(\left(x_{li}-x_{ki}\right)\left(y_{li}-y_{ki}\right)\right)
\end{aligned}$$

darstellen. Durch Umordnen der Summe lässt sich der Ausdruck

$$\sum_{l=1}^{n}\sum_{k=1}^{l-1}\sum_{i=1}^{d}\omega_{lk}d_{lk}\mathrm{inv}\left(\|y_l-y_k\|\right)\left(\left(x_{li}-x_{ki}\right)\left(y_{li}-y_{ki}\right)\right)$$

gewinnen. Die Summe der Produkte der Elemente zweier Vektoren entspricht ihrem Skalarprodukt. Dies beachtend erhält man mit

$$\sum_{i=1}^{n}\sum_{j=1}^{i-1}\omega_{ij}d_{ij}\mathrm{inv}(\|y_i-y_j\|)\langle x_i-x_j,y_i-y_j\rangle$$

die gewünschte Form. Damit ist die Gleichheit gezeigt.

Ableiten der Majorante

Um $\tau(X, Y)$ abzuleiten, soll die im vorausgegangenen Abschnitt eingeführte Matrixschreibweise verwendet werden. Die Konstante k ist für die Ableitung nicht von Bedeutung; die beiden weiteren Terme lassen sich separat ableiten. Für die Ableitung ist es nicht nötig, die Elemente der Laplace'schen Matrizen durch die jeweiligen Definitionen zu ersetzen.

Für den zweiten Term gilt der Ausdruck

$$\mathrm{Tr}(X^T L^\omega X) = \sum_{i=1}^{d}\sum_{l=1}^{n}\underbrace{[X^T]_{il}}_{x_{li}}\sum_{k=1}^{n}[L^\omega]_{lk}x_{ki} = \sum_{i=1}^{d}\sum_{l=1}^{n}\sum_{k=1}^{n}x_{li}[L^\omega]_{lk}x_{ki}.$$

Nun sollen die partiellen Ableitungen nach x_{ij} gefunden werden. Dazu fasst man die Summanden, welche x_{ij} enthalten, zu einer getrennten Summe zusammen. Alle anderen Summanden sind für die entsprechende partielle Ableitung nicht relevant, da sie konstant sind. Die partielle Ableitung

$$\frac{\partial}{\partial x_{ij}}\quad\underbrace{\sum_{k=1}^{n}x_{ij}[L^\omega]_{ik}x_{kj}}_{\text{Summand der äußren Doppelsumme mit } l=i,i=j}\quad+\underbrace{\sum_{i\in\{1,\dots,n\}\setminus\{j\}}\sum_{l\in\{1,\dots,n\}\setminus\{i\}}\sum_{k=1}^{n}x_{li}[L^\omega]_{lk}x_{ki}}_{\text{konstant, da von } x_{ij} \text{ unabhängig}}$$

lässt sich auf den Ausdruck

$$\frac{\partial}{\partial x_{ij}}\sum_{k=1}^{n}x_{ij}[L^\omega]_{ik}x_{kj}$$

reduzieren. In diesem Ausdruck wird der Summand, welcher x_{ij}^2 enthält, aus der Summe gezogen, sodass sich

$$\frac{\partial}{\partial x_{ij}}x_{ij}^2[L^\omega]_{ij} + \sum_{k\in\{1,\dots,n\}\setminus\{i\}}x_{ij}[L^\omega]_{ik}x_{kj}$$

ergibt. Die Summanden lassen sich einzeln ableiten, was zu

$$2x_{ij}[L^\omega]_{ij} + \sum_{k\in\{1,\dots,n\}\setminus\{i\}}[L^\omega]_{ik}x_{kj} = 2[L^\omega X]_{ij}$$

führt. Alle diese partiellen Ableitungen zusammenfassend erhält man die Gleichung

$$\frac{\Delta}{\Delta X}\mathrm{Tr}(X^T L^\omega X) = 2L^\omega X.$$

Die Vorgehensweise beim Ableiten des dritten Terms ist ähnlich. Man verwendet

$$Tr(X^T L^Y Y) = \sum_{i=1}^{d}\sum_{l=1}^{n}\underbrace{[X^T]_{il}}_{x_{li}}\sum_{k=1}^{n}[L^Y]_{lk}y_{ki} = \sum_{i=1}^{d}\sum_{l=1}^{n}\sum_{k=1}^{n}x_{li}[L^Y]_{lk}y_{ki}$$

und isoliert ebenfalls die für die partiellen Funktionen von x_{ij} konstanten Summanden. Auf diese Weise ergibt sich

$$\frac{\partial}{\partial x_{ij}} \underbrace{\sum_{k=1}^{n} x_{ij}[L^Y]_{ik}y_{kj}}_{\text{Summand der äußren Doppelsumme mit } l=i,i=j} + \underbrace{\sum_{i\in\{1,\dots,n\}\setminus\{j\}} \sum_{l\in\{1,\dots,n\}\setminus\{i\}} \sum_{k=1}^{n} x_{li}[L^Y]_{lk}y_{ki}}_{\text{konstant, da von } x_{ij} \text{ unabhängig}},$$

was zu

$$\frac{\partial}{\partial x_{ij}} \sum_{k=1}^{n} x_{ij}[L^Y]_{ik}y_{kj}$$

vereinfacht werden kann. Da alle übrigen Summenden ein x_{ij} enthalten, können sie alle auf dieselbe Art und Weise abgeleitet werden. Man erhält daher

$$\sum_{k=1}^{n} [L^Y]_{ik}y_{kj} = [L^Y Y]_{ij},$$

woraus sich

$$\frac{\Delta}{\Delta X}\mathrm{Tr}(X^T L^Y Y) = L^Y Y$$

ergibt. Die Ableitungen der beiden Terme lassen sich nun zusammenfügen[9] und es ergibt sich

$$\frac{\Delta}{\Delta X}\tau(X,Y) = 2L^\omega X - 2L^Y Y = L^\omega X - L^Y Y.$$

Elementweises Minimieren

Die Nullstellen dieser Ableitung sollen in Abhängigkeit von den zugehörigen x_{ij} dargestellt werden. Mit den im vorigen Abschnitt beschriebenen elementweisen Darstellungen von $L^\omega X$ und $L^Y Y$ ergibt sich die Äquivalenz

$$L^\omega X - L^Y Y = 0$$

$$\Leftrightarrow \left(\sum_{k\in\{1,\dots,n\}\setminus\{i\}} \omega_{ik}\right) x_{ij} + \sum_{k\in\{1,\dots,n\}\setminus\{i\}} -\omega_{ik}x_{kj} -$$

$$\left(\sum_{k\in\{1,\dots,n\}\setminus\{i\}} \omega_{ik}d_{ik}\mathrm{inv}\left(\|y_i - y_k\|\right) y_{ij} + \sum_{k\in\{1,\dots,n\}\setminus\{i\}} -\omega_{ik}d_{ik}\mathrm{inv}\left(\|y_i - y_k\|\right) y_{kj}\right) = 0.$$

Der Term, welcher x_{ij} enthält, wird auf die andere Seite gebracht. Da alle Summen gleich viele Summanden haben, lassen sich die übrigen Summanden zusammenfassen, sodass man den Ausdruck

$$-\left(\sum_{k\in\{1,\dots,n\}\setminus\{i\}} \omega_{ik}\right) x_{ij} = \sum_{k\in\{1,\dots,n\}\setminus\{i\}} -\omega_{ik}x_{kj} - \omega_{ik}d_{ik}\mathrm{inv}\left(\|y_i - y_k\|\right) y_{ij} + \omega_{ik}d_{ik}\mathrm{inv}\left(\|y_i - y_k\|\right) y_{kj}$$

[9] Da der Term $\mathrm{Tr}(X^T L^Y Y)$ in Gleichung (2.19) einen Faktor 2 aufweist, muss dieser auch hier übernommen werden.

erhält. Nun lässt sich x_{ij} isolieren und es ergibt sich

$$x_{ij} = \frac{\sum\limits_{k\in\{1,\dots,n\}\setminus\{i\}} \omega_{ik}x_{kj} + \omega_{ik}d_{ik}\mathrm{inv}\left(\|y_i - y_k\|\right)y_{ij} - \omega_{ik}d_{ik}\mathrm{inv}\left(\|y_i - y_k\|\right)y_{kj}}{\sum\limits_{k\in\{1,\dots,n\}\setminus\{i\}} \omega_{ik}},$$

woraus durch Ausklammern von ω_{ik}

$$x_{ij} = \frac{\sum\limits_{k\in\{1,\dots,n\}\setminus\{i\}} \omega_{ik}\left(x_{kj} + d_{ik}\mathrm{inv}\left(\|y_i - y_k\|\right)y_{ij} - d_{ik}\mathrm{inv}\left(\|y_i - y_k\|\right)y_{kj}\right)}{\sum\limits_{k\in\{1,\dots,n\}\setminus\{i\}} \omega_{ik}}$$

folgt. Durch weiteres Ausklammern erhält man

$$x_{ij} = \frac{\sum\limits_{k\in\{1,\dots,n\}\setminus\{i\}} \omega_{ik}\left(x_{kj} + d_{ik}\left(y_{ij} - y_{kj}\right)\mathrm{inv}\left(\|y_i - y_k\|\right)\right)}{\sum\limits_{k\in\{1,\dots,n\}\setminus\{i\}} \omega_{ik}}$$

als Darstellung des Minimums der Majorante $\tau(X, Y)$ in Abhängigkeit der einzelnen x_{ij}. Aus dieser Darstellung ergibt sich für $x_{ij}^{[t]}$ der Ausdruck

$$x_{ij}^{[t]} = \frac{\sum\limits_{k\in\{1,\dots,n\}\setminus\{i\}} \omega_{ik}\left(x_{kj}^{[t]} + d_{ik}\left(x_{ij}^{[t-1]} - x_{kj}^{[t-1]}\right)\mathrm{inv}\left(\|x_i^{[t-1]} - x_k^{[t-1]}\|\right)\right)}{\sum\limits_{k\in\{1,\dots,n\}\setminus\{i\}} \omega_{ik}}. \tag{2.20}$$

Um $x_{ij}^{[t]}$, die Koordinate des Elements v_i in der Dimension j zum Zeitpunkt der t-ten Iteration, einzeln berechnen zu können, muss diese unabhängig von allen anderen Koordinaten aus $X^{[t]}$ sein. Dafür müssen die $x_{kj}^{[t]}$ durch die entsprechenden Konstanten $x_{kj}^{[t-1]}$, welche aus der vorherigen Iteration bekannt sind, ersetzt werden. Diese Methode wurde in [18] vorgestellt und liefert

$$x_{ij}^{[t]} = \frac{\sum\limits_{k\in\{1,\dots,n\}\setminus\{i\}} \omega_{ik}\left(\overbrace{x_{kj}^{[t-1]}}^{\text{statt } x_{kj}^{[t]}} + d_{ik}\left(x_{ij}^{[t-1]} - x_{kj}^{[t-1]}\right)\mathrm{inv}\left(\|x_i^{[t-1]} - x_k^{[t-1]}\|\right)\right)}{\sum\limits_{k\in\{1,\dots,n\}\setminus\{i\}} \omega_{ik}}. \tag{2.21}$$

Dass diese Vorgehensweise den Stress der gesamten Konfiguration minimiert, lässt sich zeigen, indem die Ungleichung

$$\tau(X_{x_{ij}^{[t]}}, X^{[t-1]}) \leq \tau(X, X^{[t-1]})$$

verwendet wird. In dieser Ungleichung gelte die Notation

$$A_{b_{ij}} = \begin{cases} a_{kl} & \forall k \in \{1,\dots,m\}, l \in \{1,\dots,n\} : k \neq i, l \neq j \\ b_{ij} & k = i, l = j \end{cases},$$

$A, B \in \mathbb{R}^{m \times n}$. Wenn in X ein partiell minimierendes x_{ij} festgesetzt wird, bedeutet dies, dass sich der Stress der gesamten Konfiguration minimiert. Daher erhält man durch Einsetzen eines partiell minimierenden $x_{ij}^{[t]}$ in die Konfiguration $X^{[t-1]}$ eine resultierende Konfiguration $X^{[t-1]}_{x_{ij}^{[t]}}$, welche den Stress mindestens so gut minimiert wie $X^{[t-1]}$. Deswegen gilt

$$\min_X \tau(X, X^{[t-1]}) = \tau(X^{[t]}, X^{[t-1]}) \leq \min_{x_{ij}} \tau(X^{[t-1]}_{x_{ij}}, X^{[t-1]}) = \tau(X^{[t-1]}_{x_{ij}^{[t]}}, X^{[t-1]}) \leq \tau(X^{[t-1]}, X^{[t-1]}),$$

was bedeutet, dass es möglich ist, durch Bestimmen des einen minimierenden x_{ij} den Stress der gesamten Konfiguration zu minimieren, wenn alle Elemente von X außer einem fixiert sind. Gleichung (2.21) berechnet $\min_{x_{ij}} \tau(X^{[t-1]}_{x_{ij}}, X^{[t-1]})$ und damit ein solches x_{ij}.
Die elementweise Minimierung ist für die ausgedünnte Stressmajorisierung sowie für die Beschleunigung des beiliegenden Anwendungsprogramms von essenzieller Bedeutung, was die Abschnitte 2.7 und 5 zeigen.

Anschaulich erklärt ist jedes Element k bestrebt, dass das Element i zu ihm den Abstand d_{ik} aufweist. Es gibt deshalb in jeder Iteration für jede Dimension eine Wunschposition an, an der es das Element i idealerweise in Bezug auf seine eigene Position anordnen würde. Diese Wunschposition ist über

$$\underbrace{x_{kj}^{[t]}}_{\text{derzeitige eigene Position}} + \underbrace{d_{ik}\left(x_{ij}^{[t-1]} - x_{kj}^{[t-1]}\right) \text{inv}\left(\|x_i^{[t-1]} - x_k^{[t-1]}\|\right)}_{\text{derzeitige Näherung des zur Dimension } j \text{ gehörigen Anteils von } d_{ik}}$$

definiert. Der Quotient $d_{ik}\text{inv}\left(\|x_i^{[t-1]} - x_k^{[t-1]}\|\right)$ gibt die Abweichung des aktuellen zum gewünschten Abstand an. Das Produkt dieses Quotienten mit dem aktuellen Abstand in der Dimension j entspricht dem für den nächsten Zeitpunkt gewünschten Abstand in dieser Dimension. Wird dieses Produkt auf die aktuelle Position des Elements k addiert, erhält man die von k für i gewünschte Position. Das Vorzeichen von $\left(x_{ij}^{[t-1]} - x_{kj}^{[t-1]}\right)$ zeigt, ob i innerhalb der Dimension j "vor" oder "nach" k platziert werden soll.

Abbruchbedingung der Iteration

Es ist möglich, nach einer gewissen konstanten Anzahl von Iterationen abzubrechen. Weiter kann die Änderung des Stresswertes von Iteration zu Iteration als Abbruchkriterium verwendet werden, sodass die Iteration abbricht, falls die Änderung unter ein bestimmtes ϵ fällt. Eine weitere Möglichkeit stellt Abschnitt 3.2 vor.

Pseudocode

Algorithmus 2.4 zeigt Pseudocode, welcher die Stressmajorisierung beschreibt.

Algorithmus 2.4 : Stressmajorisierung

Input:

- $D \in \mathbb{R}^{n \times n}$, Matrix von paarweisen Verschiedenheiten d_{ij}
- $\Omega \in \mathbb{R}^{n \times n}$, Matrix der Gewichte ω_{ij} von d_{ij}
- Initialkonfiguration $X^{[0]} \in \mathbb{R}^{n \times d}$

Output: $X \in \mathbb{R}^{n \times d}$, Koordinatenmatrix mit Zeilenvektoren $x_1, \ldots, x_n \in \mathbb{R}^d$

$X \leftarrow X^{[0]}$
for #*Iterationen* **do**
  $X^{[t-1]} \leftarrow X$
  for $j \in \{0, \ldots, d\}$ **do**
    for $i \in \{0, \ldots, n\}$ **do**
      $x_{ij} \leftarrow \frac{\sum_{k \in \{1,\ldots,n\} \setminus \{i\}} \omega_{ik}\left(x_{kj}^{[t-1]} + d_{ik}\left(x_{ij}^{[t-1]} - x_{kj}^{[t-1]}\right)\mathrm{inv}\left(\|x_i^{[t-1]} - x_k^{[t-1]}\|\right)\right)}{\sum_{k \in \{1,\ldots,n\} \setminus \{i\}} \omega_{ik}}$
    end
  end
end

Asymptotische Laufzeit

Bestimmen des Gesamtgewichts für ein Element	$\mathcal{O}(n)$ Das Gesamtgewicht eines Elements v_i ist die Summe der Gewichte, durch die beim Bestimmen der Wunschpositionen für v_i geteilt wird. Da die Summe alle Gewichte ω_{ik} summiert, ist die Laufzeit $\mathcal{O}(n)$.
Bestimmen der Wunschposition eines Elements für ein anders	$\mathcal{O}(d)$, da bis auf die Bestimmung des Abstands $\|x_i^{[t-1]} - x_k^{[t-1]}\|$, welche $\mathcal{O}(d)$ benötigt, alle Berechnungen konstant sind.
Bestimmen aller paarweisen Wunschpositionen	$\mathcal{O}(dn^2)$, da jedes Element für jedes andere eine Wunschposition angibt.
Gesamt	$\mathcal{O}(n^2)$ Für c Iterationen und d Dimensionen ergibt sich $\mathcal{O}(cd^2n^2)$. Angenommen d und c sind konstant, resultiert $\mathcal{O}(n^2)$.

2.7 Ausgedünnte Stressmajorisierung

Pivot-MDS beschleunigt die klassische MDS durch Verwenden von nur wenigen Spalten der Matrix D. Auf ähnliche Art und Weise lässt sich die Stressmajorisierung beschleunigen. Da es bei der Stressmajorisierung möglich ist, innerhalb einer Iteration die neue Position $x_{ij}^{[t+1]}$ eines Elements v_i unabhängig von den anderen $x^{[t+1]} \in X^{[t+1]}$ zu berechnen, kann D für jedes Element die Verschiedenheiten zu einer unterschiedlichen (gleich großen) Teilmenge von V enthalten[10]. Die ausgedünnte Stressmajorisierung ermöglicht es deshalb, folgende Elementmengen zu bestimmen:

1. *Pivotelemente*: Eine Menge von Pivotelementen, welche wie oben beschrieben ausgewählt werden können. Diese sind für alle Elemente gleich. Die Anzahl der Pivotelemente sei q.

2. *Aktive Nachbarschaft*: Eine Menge von Elementen, die sich in der Nachbarschaft des jeweiligen Elements befinden. Die Anzahl an Elementen in einer aktiven Nachbarschaft sei a.

Unter der Bezeichnung *aktive Elemente* sollen diese beiden Mengen zusammengefasst werden. Es existieren $(q + a)$-viele aktive Elemente. Der Name ausgedünnte Stressmajorisierung wurde anstatt Pivot Stressmajorisierung gewählt, da eine ausgedünnte Matrix D_b benutzbar ist, welche nicht von einer festen Pivotmenge P abhängt.

[10] Bei Pivot-MDS ist dies für jedes Element die feste Teilmenge $P \subset V$.

Die ausgedünnte Stressmajorisierung minimiert den ausgedünnten Stress, welcher über die Funktion

$$\sigma_a(X) = \sum_{i=1}^{n} \sum_{k=1}^{(q+a)} \omega_{b,ik} \left(d_{b,ij} - \|x_i - x_{b,k}\| \right)^2 \tag{2.22}$$

definiert ist, wobei D_b die Matrix ist, welche die paarweisen Verschiedenheiten der aktiven Elemente zu allen Elementen enthält, Ω_b die entsprechende Gewichtungsmatrix und $x_{b,k}$ die Koordinaten des k-ten aktiven Knotens im euklidischen Zielraum darstellen.
Die ausgedünnte Stressmajorisierung positioniert die Knoten so, dass sie in Bezug auf die aktiven Knoten möglichst gut liegen, was bedeutet, dass ihr euklidischer Abstand zu den aktiven Knoten mit möglichst geringem Informationsverlust den jeweiligen Distanzen entspricht.

Für das Zeichnen von Graphen ist die ausgedünnte Stressmajorisierung, wie Pivot-MDS, ein Verfahren, was sich auf Grund der um eine Größenordnung verringerten Anzahl von benötigten Distanzen auch für große Graphen anwenden lässt. Die Möglichkeit, eine aktive Nachbarschaft auswählen zu können, erlaubt es lokale Strukturen in der Zeichnung besser darzustellen.

Pseudocode

Algorithmus 2.5 zeigt Pseudocode, welcher die ausgedünnte Stressmajorisierung beschreibt.

Algorithmus 2.5 : Ausgedünnte Stressmajorisierung

Input:

- $D_b \in \mathbb{R}^{n\times(q+a)}$, Matrix von paarweisen Verschiedenheiten der aktiven Elemente von allen Elementen
- $\Omega_b \in \mathbb{R}^{n\times(q+a)}$, Matrix der Gewichtungen $\omega_{b,ij}$ von $d_{b,ij}$
- Initialkonfiguration $X^{[0]} \in \mathbb{R}^{n\times d}$

Output: $X \in \mathbb{R}^{n\times d}$, Koordinatenmatrix mit Zeilenvektoren $x_1, \ldots, x_n \in \mathbb{R}^d$

$X \leftarrow X^{[0]}$
for #*Iterationen* **do**
  $X^{[t-1]} \leftarrow X$
  for $j \in \{0, \ldots, d\}$ **do**
    for $i \in \{0, \ldots, n\}$ **do**
      $x_{ij} \leftarrow \frac{\sum_{k\in\{1,\ldots,(q+a)\}} \omega_{b,ik}\left(x_{b,kj}^{[t-1]} + d_{b,ik}\left(x_{ij}^{[t-1]} - x_{b,kj}^{[t-1]}\right)\mathrm{inv}\left(\|x_i^{[t-1]} - x_{b,k}^{[t-1]}\|\right)\right)}{\sum_{k\in\{1,\ldots,(q+a)\}} \omega_{b,ik}}$
    end
  end
end

Asymptotische Laufzeit

Bestimmen des Gesamtgewichts für ein Element	$\mathcal{O}(a + q)$ Das Gesamtgewicht eines Elements v_i ist die Summe der Gewichte, durch die beim Bestimmen der Wunschpositionen für v_i geteilt wird. Da die Summe alle Gewichte $\omega_{p\ ik}$ summiert, es also für jedes aktive Element einen Summanden gibt, ist die Laufzeit $\mathcal{O}(a + q)$.
Bestimmen der Wunschposition eines Pivotelements für ein anders	$\mathcal{O}(d)$, da bis auf die Bestimmung des Abstands $\|x_i^{[t-1]} - x_j^{[t-1]}\|$, welche $\mathcal{O}(d)$ benötigt, alle Berechnungen konstant sind.
Bestimmen aller paarweisen Wunschpositionen	$\mathcal{O}(dn(a + q))$, da jedes Pivotelement für jedes Element eine Wunschposition angibt.
Gesamt	$\mathcal{O}(n)$ Für c Iterationen und d Dimensionen ergibt sich $\mathcal{O}(cd^2n(a + q))$. Angenommen d, c und $(a + q)$ sind konstant, resultiert $\mathcal{O}(n)$.

2.8 Vergleich der MDS-Verfahren

Der entscheidende Unterschied von klassischer MDS und Pivot-MDS zur (ausgedünnten) Stressmajorisierung ist die minimierte Fehlerfunktion. Die klassische MDS[11] minimiert den Fehler der Koordinatenskalarprodukte und bildet deshalb, wie in Abschnitt 2.4 erklärt, große Verschiedenheiten genauer ab. Die Stressmajorisierung lässt es zu, Gewichtungen für die aktiven Knoten zu bestimmen, womit man Einfluss darauf hat, welche Verschiedenheiten bevorzugt werden sollen, beziehungsweise die Möglichkeit, sämtliche Verschiedenheiten gleichwertig zu behandeln. Da die Stressmajorisierung deshalb das in 2.1 beschriebene MDS-Problem mit geringerem Fehler bezüglich des Informationsverlustes in den Verschiedenheiten löst, sind die von ihr generierten Layouts besser als die der klassischen MDS, was auch zu einem für besser empfundenen Layout führt, wie sich anhand der Abbildungen in Abschnitt 5.5 erkennen lässt.
Die Stressmajorisierung ist im Gegensatz zu Pivot-MDS von einer Initialkonfiguartion anhänigig.
Pivot-MDS hat den Vorteil, dass es in der Praxis, trotz gleicher asymptotischer Laufzeiten, deutlich schneller ist als die ausgedünnte Stressmajorisierung, da die konstante Anzahl c von Iterationen bei Pivot-MDS auf der $k \times k$-Matrix ausgeführt wird, was zu ck^2 führt. Bei der ausgedünnten Stressmajorisierung findet die gesamte Berechung innerhalb einer Iteration statt, womit sich ckn ergibt. Außerdem reicht bei der Potenziteration für gewöhnlich eine kleinere Anzahl von Iteratio-

[11] im weiteren Verlauf dieses Teilabschnitts stellvertretend für Pivot-MDS

nen aus. Die Laufzeittests in Abschnitt 5 zeigen dies. Die Auswahl eines dieser Verfahren stellt also einen Kompromiss zwischen Ergebnisgüte und Laufzeit dar.

3 Kombinierte MDS

Die kombinierte MDS ist ein Verfahren, welches geeignet ist, große Graphen in kurzer Zeit mit gutem Ergebnis zu zeichnen und sowohl die Vorteile von Pivot-MDS als auch der ausgedünnten Stressmajorisierung nutzt. Abschnitt 5.5 zeigt Laufzeiten und Ergebnisse für verschiedene Beispielgraphen.

3.1 Integration der vorgestellten Verfahren

Die kombinierte MDS berechnet zunächst die von Pivot-MDS und der ausgedünnten Stressmajorisierung benötigten Distanzen. Die Distanzen aller Knoten zu den Pivotknoten müssen dabei nur einmal berechnet werden, da Pivot-MDS und die ausgedünnte Stressmajorisierung die gleichen Pivotknoten verwenden. Nach der Distanzberechnung wird Pivot-MDS ausgeführt. Pivot-MDS hat hier die Aufgabe, ein Layout des Graphen zu finden, welches die globale Struktur des Graphen gut repräsentiert. Abschnitt 2.4 zeigt, warum Pivot-MDS dazu geeignet ist. Die ausgedünnte Stressmajorisierung benutzt die berechneten Koordinaten nun als Initialkonfiguration, was mehrere Vorteile hat. Da das Initiallayout eine gute Repräsentation der gobalen Graphstruktur ist, ist es für die ausgedünnte Stressmajorisierung unwahrscheinlicher, in einem lokalen Minimum zu enden. Dies führt mit einer größeren Wahrscheinlichkeit zu guten Ergebnissen. Weiter führt das Initiallayout dazu, dass für ein gutes Ergebnis weniger Iterationen nötig sind.
Weil die Stressmajorisierung bestrebt ist, eine durchschnittliche Kantenlänge von 1 zu erreichen, skaliert die kombinierte MDS das von Pivot-MDS generierte Layout auf diese Durchschnittskantenlänge, bevor es als Initiallayout für die ausgedünnte Stressmajorisierung verwendet wird. Dies spart weitere Iterationen, welche die ausgedünnte Stressmajorisierung ansonsten benötigen würde um eine entsprechende Skalierung durchzuführen.
Da die Laufzeit von Pivot-MDS deutlich geringer ist als die der ausgedünnten Stressmajorisierung und diese auf Grund der Initialisierung weniger Iterationen benötigt, ist die kombinierte MDS schneller als die ausgedünnte Stressmajorisierung. Wie viele Iterationen gespart werden können, hängt von der Güte des durch Pivot-MDS generierten Layouts ab, welche wiederum vom Graphen abhängt. Abschnitt 5.5 zeigt den dadurch erreichten Geschwindigkeitsgewinn anhand verschiedener Beispielgraphen.

3.2 Abbruchkriterium der ausgedünnten Stressmajorisierung

Ein mögliches Abbruchkriterium für die ausgedünnte Stressmajorisierung ist, nach einer gewissen Anzahl von Iterationen die Änderung des ausgedünnten Stresses seit der letzten Messung zu berechnen und, falls die Änderung kleiner als ein bestimmtes ϵ ist, abzubrechen. Nachteil davon ist, dass öfteres Bestimmen dieses Stresswertes relativ viel Laufzeit kostet, da hierfür über alle Wunschpositionen aller aktiven Knoten für alle anderen Knoten summiert werden muss. Zu seltenes Bestimmen der Stressänderung führt allerdings dazu, dass sehr wahrscheinlich mehr Iterationen als nötig ausgeführt werden.
Die kombinierte MDS löst dies mittels einer Bewertung der Ergebnisqualität von Pivot-MDS und führt bei guter Qualität wenige Iterationen durch, bei schlechter mehr. Als Bewertungskriterium

dienen dabei die während Pivot-MDS bestimmten Eigenwerte. Zusätzlich werden die beiden nächstgrößten Eigenwerte berechnet. Der Mehraufwand an Laufzeit dafür ist zu vernachlässigen. Angenommen, es soll ein zweidimensionals Layout generiert werden, ist das Layout von Pivot-MDS dann gut, wenn die ersten beiden Eigenwerte möglichst dominant sind. Sind der dritte und vierte Eigenwert ähnlich groß wie der erste und zweite, so ist das Ergebnis von Pivot-MDS ziemlich schlecht. Die Stressmajorisierung benötigt in diesem Fall deutlich mehr Iterationen, um diese auszugleichen, wozu sich Beispiele und Erläuterungen in Abschnitt 5.5 finden. Die Funktion, welche die Anzahl der nötigen Iterationen für die ausgedünnte Stressmajorisierung aus den Eigenwerten berechnet, beachtet diese Tatsache und ist in der Implementierung als

$$\text{iter\_count}(\lambda_1, \lambda_2, \lambda_3, \lambda_4) = \max\left(20, 100/\frac{\lambda_1 + \lambda_2}{\lambda_3 + \lambda_4}\right)$$

definiert. Für ein dreidimensionales Layout kann man analog vorgehen.

3.3 Möglichkeiten zur Parallelisierung

Die kombinierte MDS lässt sich gut parallelisieren und damit durch hoch parallele Hardware wie Grafikkarten deutlich beschleunigen. Dieser Abschnitt beschreibt, welche Möglichkeiten es zur Parallelisierung der kombinierten MDS gibt. Abschnitt 5 zeigt, welche die Implementierung umsetzt, warum und wie.

Distanzberechnung

Sowohl die Berechnung der Distanzen aller Knoten zu den Pivotknoten als auch der Distanzen der Knoten zu ihrer aktiven Nachbarschaft lassen sich parallelisieren.
Für die Berechnung der Distanzen aller Knoten zu den Pivotknoten sind q (Anzahl Pivotknoten) viele Breitensuchen von jedem Pivotknoten aus nötig. Da diese unabhängig voneinander sind, lassen sie sich parallel ausführen.
Eine Möglichkeit die Breitensuche selbst zu parallelisieren, stellen [13] und [14] vor. Der in diesen Papern vorgestellte Algorithmus verwendet anstatt einer Warteschlange (queue) als Datenstruktur ein Array *frontier*, welches die aktuelle Breitensuchebene verwaltet. Die Breitensuche läuft dabei iterativ ab. Vor der ersten Iteration setzt die Wurzel ihren Eintrag in *frontier* auf *true* und trägt im Array, welches den Abstand zur Wurzel speichert, den Wert Null ein. In jeder Iteration prüft jeder Knoten unabhängig von den anderen, ob sein Eintrag in *frontier true* ist. Ist dies der Fall, deaktiviert er seinen Eintrag in *frontier* und markiert sich selbst als besucht. Für alle zu ihm adjazenten Knoten, welche noch nicht als besucht markiert sind, trägt er seine Entfernung um eins erhöht in das Abstandsarray ein und aktiviert ihre Einträge in *frontier*. Sobald nach einer Iteration in *frontier* keine aktiven Einträge existieren, terminiert der Algorithmus. Die Äbstände sämtlicher Knoten zur Wurzel befinden sich nun im Abstandsarray. Die Iterationen dieses Algorithmus' müssen sequenziell ablaufen, da jede Iteration von der jeweils vorhergehenden abhängig ist.
Der Nachteil dieser Methode ist, dass bis zu $\mathcal{O}(n^2)$ viele Rechenschritte nötig sind, da im schlechtesten Fall jede Breitensuch-Ebene nur einen Knoten enthält und es damit n Ebenen gibt, in denen

jeweils n Knoten rechnen. Eine optimale Version kommt mit $\mathcal{O}(m+n)$ Rechenoperationen aus, da jeder Knoten und jede Kante einmal betrachtet werden muss. Die vorgestellte parallele Methode ist also nicht arbeitseffizient (*workefficient*).
Die Paper [20] und [21] stellen arbeitseffiziente, parallele Breitensuchalgorithmen für CUDA vor. Sie durchlaufen ebenfalls pro Iteration eine Breitensuchebene. Die Datenstruktur, welche die aktuelle Breitensuch-Ebene verwaltet, ist jedoch eine Warteschlange. Somit gibt es nicht für jeden Knoten einen Eintrag, der entweder "true" oder "false" ist; die Knoten, welche in der nächsten Iteration aktiv sein sollen, werden aufeinanderfolgend in dieser Warteschlange gespeichert. Dies ist eine Aufgabe, die für Grafikkarten nicht direkt geeignet ist.
Für eine effiziente Implementierung sind einige Hürden zu überwinden. Da die Anzahl an Nachbarn von Knoten zu Knoten verschieden ist, sollte man dafür sorgen, dass die Recheneinheiten trotzdem in etwa gleich viel Arbeit haben (load balancing). Weiter ist es nicht trivial, die Warteschlange parallel zu füllen, wenn die Knoten verschiedene Zahlen von Nachbarn hineinschreiben wollen. Die in den Papern beschriebenen Möglichkeiten und eine eigene Implementierung, welche [21] verwendet, stellt Abschnitt 5.2 vor.

Für die Berechnung der Distanzen der Knoten zu ihrer aktiven Nachbarschaft sind n (Anzahl Knoten) viele Breitensuchen nötig. Diese terminieren jedoch, nachdem sie a (Anzahl aktiver Nachbarn) viele Knoten erreicht haben. Auch diese Breitensuchen lassen sich unabhängig voneinander ausführen. Eine einzelne dieser Breitensuchen ist auf die oben beschriebene Weise parallelisierbar; der Algorithmus sollte jedoch prüfen, ob bereits genug Knoten besucht wurden und gegebenenfalls terminieren.

Pivot-MDS

Pivot-MDS ist fast vollständig parallelisierbar. Die für die Doppelzentrierung benötigten Zeilen- und Spaltendurchschnitte lassen sich unabhängig voneinander bestimmen. Beim Berechnen der Matrix C^TC und der anschließenden Potenziteration handelt es sich um eine Matrix-Matrix-Multiplikation bzw. um Matrix-Vektor-Multiplikationen, also um Aufgaben, die für die Parallelisierung bestens geeignet sind, da es möglich ist, jedes Ergebniselement unabhängig von den anderen zu berechnen.

Ausgedünnte Stressmajorisierung

Die Iterationen der ausgedünnten Stressmajorisierung müssen sequentiell ausgeführt werden. Bei der zweiten for-Schleife, welche über sämtliche Knoten i läuft und deren neue Position bestimmt (s. Algorithmus 2.5), handelt es sich um eine parallelisierbare for-Schleife. Sämtliche dieser Positionen lassen sich unabhängig voneinander bestimmen. Es ist ein noch höherer Parallelisierungsgrad erreichbar, da jede Wunschposition von einem Knoten k für einen Knoten i (ein Summand der Summe in Algorithmus 2.5) unabhänigig von allen anderen Wunschpositionen berechnet werden kann. Um die Anzahl der sequenziellen Schritte, die für das anschließende Aufaddieren der Summanden nötig ist, von $\mathcal{O}(q + a)$ ($(q + a)$ = Anzahl aktiver Knoten) auf $\mathcal{O}(log(q+a))$ zu reduzieren, verwendet die Implementierung die "sum-reduce"-Methode, welche

beim Vorstellen der Implementierung in Abschnitt 5 erklärt wird.

Weitere Parallelisierung

Es ist möglich, die Berechnung der Distanzen der Knoten zu ihrer aktiven Nachbarschaft, welche nur für die ausgedünnte Stressmajorisierung gebraucht werden, gleichzeitig mit Pivot-MDS auszuführen.

4 CUDA

CUDA “Compute Unified Device Architecture” ist eine von Nvidia entwickelte Architektur für die Ausführung allgemeiner, datenparalleler Algorithmen auf Grafikkarten. Ein Algorithmus ist datenparallel, wenn er das gleiche Programm parallel auf viele Datenelemente anwendet. Vereinfacht kann man sich einen solchen Algorithmus als for-Schleife vorstellen, welche über ein Datenarray iteriert und deren Iterationen unabhängig voneinander, d.h. in beliebiger Reihenfolge, ausführbar sind. Ein Beispiel dafür ist die map-Funktion aus Haskell, welche dieselbe Operation auf eine Liste von Daten anwendet. Zu dieser Klasse von Algorithmen gehört auch die Hauptaufgabe der Grafikkarte, das Rendern, welches lineare Transformationen parallel auf viele Vertices bzw. Pixel anwendet. Bei Grafikkarten dient ein weitaus größerer Anteil der Transistoren dem Berechnen als dies bei einer CPU der Fall ist. Abbildung 4.1 veranschaulicht das. Ein großer Teil der Transistoren einer CPU sind für Caching und Kontrolllogik wie “out of order execution” oder “branch prediction” zuständig. Den fehlenden Cache kompensiert die Grafikkarte durch “Verstecken der Speicherlatenz”, was in Abschnitt 4 erklärt wird.

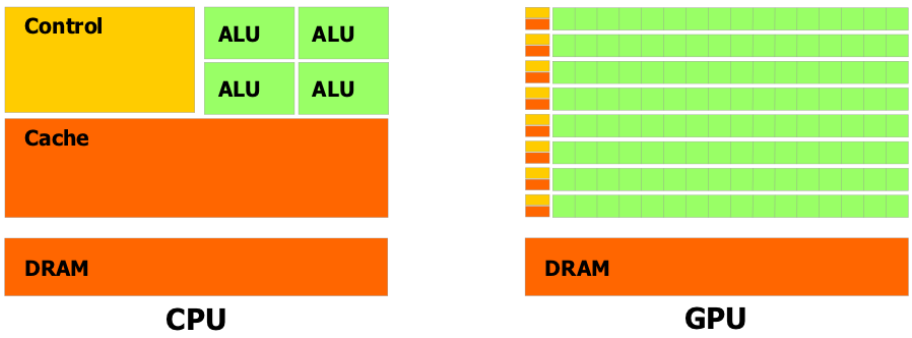


Abbildung 4.1: Verteilung der Transistoren bei CPU und GPU. Entnommen aus [3].

Alle Grafikkarten von Nvidia ab der “8000er-Serie” sind CUDA-fähig. Eine CUDA-fähige Grafikkarte besitzt mehrere hochparallele Streaming Multiprozessoren (SM). Jeder dieser verfügt über eine Anzahl von Registerplätzen und shared memory. Weiter gibt es einen globalen Speicherbereich, auf welchen alle SMs zugreifen können. Die dazu gehörige Softwareabstraktion ist eine Threadhierarchie:

- *Thread:* Ein Thread wendet ein Programm auf ein bestimmtes Datenelement an.[12] Ein SM führt gleichzeitig eine Menge von Threads aus. Jeder Thread kann eigene Registerplätze haben, auf die es nur ihm möglich ist zuzugreifen.
- *Threadblock:* Ein Threadblock ist eine Gruppe von Threads. Wie viele Threads ein Threadblock maximal enthalten kann, hängt von der Grafikkarte ab (512 oder 1024). Ein gesamter

[12] Man kann die Granularität auch gröber wählen.

Threadblock befindet sich garantiert auf einem SM. Daher ist es für Threads innerhalb eines Blocks möglich auf den gleichen “shared memory” Bereich zuzugreifen. Es können sich mehrere Threadblocks gleichzeitig in einem SM befinden. Die Threads eines Blocks haben IDs, anhand derer sie sich innerhalb des entsprechenden Blocks eindeutig identifizieren lassen. Sie können innerhalb eines Blocks ein-, zwei- oder dreidimensional organisiert sein. Die IDs haben die entsprechende Anzahl an Koordinaten.

- *Grid:* Die Threadblocks werden zu einem Grid zusammengefasst. Alle Threads können auf den globalen Speicherbereich zugreifen. Die Blocks bekommen ebenfalls ein-, zwei- oder dreidimensionale IDs. Damit lassen sich einzelne Threads über Block-ID und Thread-ID eindeutig identifizieren. Im Gegensatz zu einem Block, bei dem die Gesamtzahl der Threads ungeachtet der Dimensionalität beschränkt ist, ist im Grid die Größe jeder Dimension beschränkt (bei den meisten Grafikkarten $2^{16} = 65536$).

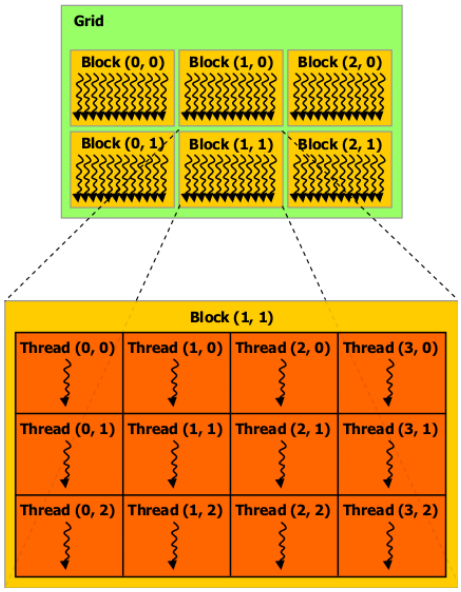


Abbildung 4.2: Veranschaulichung der Threadhierarchie. Entnommen aus [3].

CUDA stellt CUDA C zur Verfügung, eine erweitertes C, welches es ermöglicht mittels sogenannter “Kernelfunktionen” Code auf der Grafikkarte auszuführen.[13] Kernelfunktionen lassen sich wie gewöhnliche Funktionen aus dem Code, welcher auf der CPU läuft (Host-Code)[14] aufrufen. Beim Aufruf spezifiziert man mittels einer speziellen Syntax die Dimensionalität und Größe des Grids und seiner Threadblöcke. Jeder Kernelfunktion ist ein Grid zugeordnet[15], wobei die

[13] Es unterstützt jedoch nicht den gesamten Sprachumfang von C. Funktionspointer sind nicht möglich. Grafikkarten vor der Fermi-Generation unterstützen keine Rekursion.

[14] Den CUDA C Code, welcher auf der Grafikkarte läuft, bezeichnet man als Device-Code.

[15] Ab der Fermi-Generation können mehrere Kernelfunktionen und damit Grids parallel ausgeführt werden. Ältere Karten sind dazu nicht im Stande.

Threadblöcke innerhalb eines Grids die gleiche Größe und Dimensionalität haben müssen. Jeder Thread führt den Code der Kernelfunktion aus. Innerhalb der Kernelfunktion kann man den aktuellen Thread mittels Block- und Thread-ID identifizieren. Dieses Schema kann man sich mit Hilfe der oben erwähnten parallelisierten for-Schleife vorstellen, wobei der Code innerhalb der Schleife dem Code der Kernelfunktion entspricht und jede Iteration von einem eigenen Thread ausgeführt wird. Auf den Index der Iteration lässt sich mittels der Block- und Thread-ID zugreifen.
Die Datenelemente eines Algorithmus müssen entsprechend gruppiert werden. Sowohl die Threads innerhalb eines Threadblocks als auch die Threadblöcke selbst lassen sich in beliebiger Reihenfolge ausführen. CUDA-C-Programme skalieren mit der Anzahl der SMs, welche eine Grafikkarte zur Verfügung stellt. Je mehr SMs vorhanden sind, desto mehr Threadblöcke lassen sich gleichzeitig ausführen. Mit Hilfe von Funktionen lassen sich im Host-Code Stellen im globalen Speicher der Grafikkarte reservieren. Daten vom Hauptspeicher kann man dorthin kopieren bzw. von dort in den Hauptspeicher transferieren. Dies ermöglicht die Kommunikation von CPU und Grafikkarte. Die Threads eines Threadblocks lassen sich innerhalb der Kernelfunktion synchronisieren, da sie sich garantiert in einem SM befinden. Daten im globalen Speicher sind über Kernelfunktionsaufrufe hinweg persistent[16]. Deshalb ist es möglich, alle Threads durch Aufteilen einer Kernelfunktion in zwei zu synchronisieren.

Für die Ausführung werden Blocks in Warps aufgeteilt. Ein Warp ist eine Gruppe von 32 Threads. Hat ein Block weniger Threads, so werden leere aufgefüllt. Die Warps enthalten Threads mit aufsteigenden IDs. Alle 32 Threads starten gleichzeitig und führen synchron die gleiche Instruktion aus. Sie können jedoch verschiedenen Zweigen einer if-Verzeigung folgen.
Der Ausführungskontext eines Warps bleibt so lange erhalten, bis die Threads, die dieser enthält, ihre Berechnungen beendet haben. Somit kann der Warp-Scheduler Kontexwechsel ohne Kosten durchführen.

Der Rest dieses Abschnitts zeigt Möglichkeiten, CUDA-C-Code zu beschleunigen. Diese wurden für die Implementierung der kombinierten MDS verwendet. Die Abschnitte 5.3 und 5.4 zeigen, wie diese umgesetzt wurden und welchen Geschwindigkeitsvorteil sie bringen. Die Liste der vorgestellten Beschleunigungsmöglichkeiten ist nicht vollständig. Die CUDA-fähigen Grafikkarten sind je nach Eigenschaften einer “compute capability”-Klasse zugeordnet. Für die Implementierung wurde eine GeForce 250 GTS mit compute capability 1.1 verwendet, später auch eine GeForce GTX 550 Ti mit compute capability 2.1. Es war jedoch zeitlich nicht möglich, die neuen Features von compute capability 2.1 zu nutzen. Abschnitt 5 erläutert dies und zeigt Laufzeiten. Der Effekt der einzelnen Optimierungen variiert je nach verwendeter Grafikkarte und compute capability. Eine gute Erklärung dieser und weiterer Optimierungen liefert [2]. In [3] befindet sich eine Übersicht aktueller Grafikkarten und deren compute capability.

[16] Im globalen Speicher der Grafikkarte reservierte Speicherstellen werden im Host-Code freigegeben.

Belegungsgrad: Es sollte ein maximaler Belegungsgrad angestrebt werden, das heißt, zu jeder Zeit sollte sich die maximale Anzahl aktiver Threads in einem SM befinden und alle SMs genutzt werden (Bei der Nvidia GTS 250 sind dies 768 Threads pro SM und 16 SMs; insgesamt maximal 12288 aktive Threads.). Dafür müssen die Blocks folgende Hardwaregrenzen beachten (in Klammern stehen jeweils die entsprechenden Größen für die Nvidia GTS 250):

- Maximale Anzahl aktiver Blocks pro SM (8)
- Gesamtgröße der einem SM zur Verfügung stehenden Registerplätze (8192 KB)
- Größe des shared memory Bereichs, der einem SM zur Verfügung steht (16384 KB)

Daher sollte jeder Block mindestens $\frac{max.\ Threadzahl}{max.\ Blockzahl}$ viele Threads haben. Wenn Blöcke mehr Threads haben und damit weniger als die maximale Anzahl an Threadblöcken aktiv ist, ist dies unproblematisch, solange $aktiveBlocks * Threads\ pro\ Block = max.\ Threadzahl$ gilt. Optimalerweise ist die Anzahl der pro Block verbrauchten Registerplätze $\frac{max.\ Registerpltze}{aktive\ Blocks}$ und des pro Block verbrauchten shared memory $\frac{max.\ shared\ memory}{aktive\ Blocks}$. Außerdem ist zu beachten, dass sich diese Beschränkungen gegenseitig beeinflussen, z.B. hängen Registerplätze und shared memory pro Block von der Anzahl der Blocks ab. Ein optimaler Belegungsgrad in nur selten zu erreichen. Je nach Algorithmus sollte man den besten Kompromiss finden.

Instruktionsdurchsatz: Der Instruktionsdurchsatz bezeichnet die Anzahl der Recheninstruktionen, welche die Grafikkarte pro Zeiteinheit durchführen kann. Diese sollte so nah wie möglich am Maximum liegen. Dafür ist unter anderem der eben beschriebene Belegungsgrad wichtig.

Divergente Verzweigungen: Wie oben erwähnt, führen alle Threads eines Warps parallel die gleiche Instruktion aus. Falls nur ein Teil der Threads innerhalb eines Warps einem Zweig einer if-Verzweigung folgt, müssen die anderen Threads warten. Für einen hohen Instruktionsdurchsatz ist es daher gut, wenn ein Warp einheitlich dem gleich Zweig folgt.

Verstecken der Speicherlatenz: Wenn alle Warps auf den Speicher warten, können keine weiteren Instruktionen ausgeführt werden. Damit die Grafikkarte die Speicherlatenz verstecken kann, ist es wichtig, deutlich mehr Threads zu verwenden als gleichzeitig aktiv sein können. Das erlaubt dem Warp-Scheduler, Threads zu deaktivieren, während sie auf den Speicher warten und Threads, welche rechnen wollen, zu aktivieren. Wie oben in diesem Abschnitt erwähnt, benötigt ein Kontextwechsel fast keine Zeit.

Speicherdurchsatz: Der Speicherdurchsatz bezeichnet die Menge von Daten, welche die Grafikkarte pro Zeiteinheit aus dem Speicher liest. Um diesen zu erhöhen, sollte möglichst Speicher mit hoher Bandbreite verwendet werden. Das Transferieren von Daten aus dem Hauptspeicher in den Speicher der Grafikkarte ist verhältnismäßig sehr langsam. Daher sollte man dies so selten wie möglich benutzen und beachten, dass Daten, die man in den globalen Speicher der Grafikkarte geladen hat, über Kernelfunktionsaufrufe persistent sind.

Der globale Speicher der Grafikkarte hat eine deutlich geringere Bandbreite als die shared memory Bereiche und die Register. Für eine Erhöhung des Speicherdurchsatzes ist es daher wichtig, die Zugriffe auf den globalen Speicher zu reduzieren. Ein Threadblock, der mehrmals auf dieselben Daten zugreift, sollte diese in seinem shared memory Bereich zwischenspeichern. Wenn einzelne Threads Daten öfters benötigen, sollten sie in Registern gespeichert werden.

Zusammenhängende Speicherzugriffe: Wenn Threads mit aufeinander folgenden IDs auf zusammenhängende, an entsprechenden Grenzen ausgerichtete Speicherbereiche zugreifen, benötigt dies lediglich einen Zugriff auf den globalen Speicher. Beispielsweise können 16 aufeinander folgende Threads 16 floats ($16 * 4 = 64$ Bytes) mit einem einzigen Speicherzugriff lesen bzw. schreiben, wenn diese hintereinander im Speicher liegen und an einer 64-Byte Grenze ausgerichtet sind. Durch das Ausnutzen zusammenhängender Speicherzugriffe lässt sich der Speicherdurchsatz enorm erhöhen.

Bankkonflikte: Der shared memory Bereich ist in Bänke unterteilt. Jede Bank kann gleichzeitig nur einen Datensatz adressieren. Wenn mehrere Threads in einem Warp gleichzeitig Daten über die gleiche Bank laden wollen, müssen diese Speicherzugriffe serialisiert werden, was den Speicherdurchsatz des shared memory reduziert. Dies wird als Bankkonflikt bezeichnet. Bänke sind für vier Adressen, d.h. für 32-Bit Werte zuständig. Wenn die Threads in einem Warp auf verschiedene Bänke zugreifen, treten keine Bankkonflikte auf.

5 Implementierung der kombinierten MDS

Der Code, welcher auf der CPU läuft, ist in C geschrieben, der Code, welcher auf der Grafikkarte läuft, in CUDA-C. Die Implementierung verwendet keine weiteren Bibliotheken und keine aufwändigen Datenstrukturen. Um der Einfachheit und Geschwindigkeit willen sind die einzigen verwendeten zusammengesetzten Datenstrukturen mit *malloc* reservierte Speicherbereiche und *structs* aus C. Diese Speicherbereiche dienen dazu, eine Menge einfacher Datentypen hintereinander im Speicher abzulegen. Sie werden zur Vereinfachung im folgenden als Arrays bezeichnet, auch wenn es keine C-Arrays sind. Die Matrizen wurden ebenfalls als eindimensionale Speicherbereiche umgesetzt. Sie liegen in der row-major Form vor, das heißt, die einzelnen Zeilen der Matrix liegen hintereinander im Speicher.
Der folgende Teilabschnitt gibt eine Übersicht und Erklärungen zu den in der Implementierung verwendeten Datenstrukturen. Die danach folgenden Teilabschnitte zu den einzelnen Schritten der kombinierten MDS beschreiben jeweils zunächst die Implementierung und vergleichen anschließend Laufzeiten. Eine CPU-Implementierung dient jeweils zur Veranschaulichung des Geschwindigkeitsvorteils, welchen die Grafikkarte bringt. Die Implementierung benutzt nur einen Kern der CPU. Das für die Implementierung verwendete System besteht aus einer Intel Core 2 Duo E6650 @ 2,3 GHz CPU und einer Nvidia GTS 250 GPU. Das System entspricht in etwa den in [7] und [10] verwendeten Systemen, was den Laufzeitvergleich von [7] und [10] mit der kombinierten MDS in Abschnitt 6 erleichtert. Später wurde zusätzlich ein aktuelles System, bestehend aus einer AMD Phenom II X4 850 @ 3,3 GHz CPU und einer Nvidia GTX 550 Ti GPU, verwendet; die Implementierung benutzt jedoch nicht die von der neuen Grafikkarte zur Verfügung gestellten Features und ist nicht für diese Grafikkarte optimiert, weshalb die Laufzeiten für Graphen, für die die Distanzberechnung auf der CPU ausgeführt wird, auf ersterem System etwas besser sind.
Die Laufzeiten für die Distanzberechnung wurden mit ersterem System bestimmt, für Pivot-MDS und die ausgedünnte Stressmajorisierung mit letzterem. Für die Messungen wurde neben der C-Zeitfunktion der *Compute Visual Profiler* von Nvidia benutzt. Eine ausführliche Erklärung dazu findet sich im zugehörigen Benutzerhandbuch [1].
Tabelle 5.1 listet die verwendeten Beispielgraphen auf.

Graph	**#Knoten**(n)	**#Kanten**(m)	**Layout-Vorschau**
smallworld	1000	3000	
data	2851	15093	
3elt	4720	13722	
uk	4824	6837	
add32	4960	9462	
bcsstk33	8738	291583	
flower_050	9030	131241	
grid_rnd_100	9497	17849	
snowflake_C	9701	9700	
whitaker3	9800	28989	
sierpinski_08	9843	19683	
spider_C	10000	22000	
crack	10240	30380	
4elt	15606	45878	
bcsstk31	35588	572914	
t60k	60005	89440	
wing	62032	121544	
598a	110971	741934	
fe_ocean	143437	409593	
m14b	214765	1679018	
auto	448695	3314611	

Tabelle 5.1: Übersicht der im Rahmen dieser Arbeit verwendeten Beispielgraphen, ihrer Knoten- und Kantenanzahl sowie einer Vorschau ihres von der kombinierten MDS generierten Layouts.

5.1 Verwendete Datenstrukturen

adj_lists_offset und adj_lists_edges: Diese Arrays dienen der effizienten Speicherung des Graphen innerhalb einer in [13] und [14] vewendeten Methode, welche Abbildung 5.1 veranschaulicht. Dabei ist adj_lists_edges ein Speicherbereich, welcher die Adjazenzlisten aller Knoten mit aufsteigender ID[17] hintereinander enthält. Das Array *adj_lists_offset* enthält an Stelle i den Index j, sodass die Adjazenzliste des Knotens mit der ID i in *adj_lists_edges* an Stelle j beginnt. Damit benötigt ein Graph asymptotisch $\mathcal{O}(m+n)$ Speicher. Es ist jedoch zu bedenken, dass bei den behandelten ungerichteten Graphen jede Kante, repräsentiert durch Anfangs- bzw. Endknoten, in zwei Adjazenzlisten vorkommt und damit zweimal Speicher in *adj_lists_edges* benötigt. Der von *adj_lists_edges* benötigte Speicherplatz ist daher $2 * e * sizeof(int)$, wobei e in diesem Fall die Anzahl der Kanten und $sizeof(int)$ der Speicherbedarf für eine Ganzzahl ist. *adj_lists_offset* benötigt $(n+1) * sizeof(int)$ Speicher. Die letzte Speicherstelle zeigt hinter das Ende von *adj_lists_edges* und wird für eine Abbruchbedingung beim Iterieren über die Adjazenzlisten benötigt.

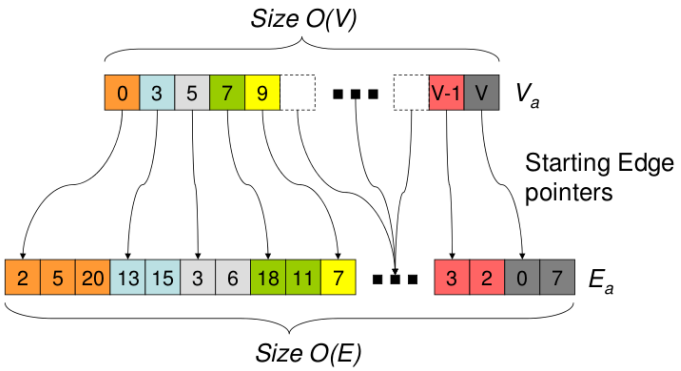


Abbildung 5.1: Veranschaulichung der Graph-Datenstruktur. Entnommen aus [14].

nodes: Ein Array von *structs*, welche die Koordinaten der Knoten des Graphen speichern. Für den 2D-Fall enthält die Struktur die Werte *x_pos* und *y_pos* für die x- bzw. y-Koordinate des Knotens, im 3D-Fall kommt *z_pos* für die z-Koordinate hinzu. Die Knoten-IDs verwaltet *nodes* implizit, an der Stelle i befinden sich die Koordinaten des Knotens mit der ID i. Der benötigte Speicherplatz ist $n * sizeof(node\_struct)$, wobei *node_struct* $d * sizeof(int)$ benötigt. Ferner gilt $d \in \{2,3\}$, je nach Layout-Dimensionalität.

dist_matrix_p und dist_matrix_n: Diese Matrizen speichern die Distanzen aller Knoten zu den aktiven Knoten. Dabei enthält *dist_matrix_p* die Distanzen zu den Pivotelementen und *dist_matrix_p* die zur aktiven Nachbarschaft. Die i-te Zeile dieser Matrizen enthält die Distanzen des Knotens i zu den Pivotknoten bzw. der aktiven Nachbarschaft. Daher ist dist_matrix_p eine $n \times q$-Matrix (q ist die Anzahl der Pivotknoten) und benötigt $n * q * sizeof(int)$ Speicher;

[17] Wie die ID zu Stande kommt, ist bei der Datenstruktur *nodes* erklärt.

dist_matrix_n hat die Dimensionalität $n \times a$ (a ist die Anzahl der Knoten in der aktiven Nachbarschaft) und damit einen Speicherbedarf von $n * a * sizeof(int)$.

pivot_idx und neigh_idx: Da in den Distanzmatrizen die Spalten den jeweils nächsten Pivotknoten bzw. Knoten aus der aktiven Nachbarschaft enthalten, jedoch nicht bekannt ist, welche Knoten-ID diese haben, werden Datenstrukturen benötigt, welche die Abbildung von Spalten-ID auf die ID der entsprechenden aktiven Knoten repräsentieren. Diesen Zweck erfüllen *pivot_idx* und *neigh_idx*. Da die Pivotknoten für jeden Knoten gleich sind und daher eine Spalte von *dist_matrix_p* die Distanzen zu einem bestimmten Pivotknoten enthält, ist *pivot_idx* ein Array, welches an Stelle j die Knoten-ID i des Pivotknotens enthält, dessen zugehörige Distanzen die j-te Spalte von *dist_matrix_p* beinhaltet. Da jeder Knoten eine eigene aktive Nachbarschaft hat, sind den Spalten von *dist_matrix_n* (meistens) mehr als ein aktiver Knoten zugewiesen. Um die ID dieser aktiven Knoten zu ermitteln, ist eine Matrix der gleichen Dimensionalität nötig. Diese Rolle übernimmt die Matrix *neigh_idx* in der Implementierung. An der Stelle (i, j) enthält sie die ID desjenigen aktiven Knotens, der mit der Position (i, j) in *dist_matrix_n* assoziiert ist.

5.2 Distanzberechnung

Distanzen zu Pivotknoten

Wie in Abschnitt 3.3 erwähnt, stellt [20] einen arbeitseffizienten Breitensuchalgorithmus für CUDA vor. Für jede Breitensuchebene wird eine Iteration ausgeführt, welche

- die Knoten aus der Warteschlange liest,
- den Abstand dieser Knoten zur Wurzel speichert (Nummer der Ebene),
- ihren Status auf markiert setzt
- und deren nicht markierte Nachbarn für die nächste Iteration in die Warteschlange schreibt.

Zu beachten ist, dass die Warteschlange im Gegensatz zur seqeuenziellen Version der Breitensuche zu einem bestimmten Zeitpunkt nur Knoten der gleichen Breitensuchebene enthält. Aus diesem Grund bezeichnet [20] diese Warteschlange auch als *frontier*. Die Warteschlange enthält jeweils die aktiven[18] Knoten der in [14] verwendeten *frontier*. Die Warteschlange ist außerdem als Array realisiert, um auf alle Elemente in asymptotisch konstanter Zeit zugreifen zu können. Um ein Schreiben an ihr Ende zu ermöglichen, hat sie einen Zähler, welcher die Anzahl der Elemente speichert.
Jeder aktive Knoten wird in einem eigenen Thread bearbeitet. Um die jeweiligen Nachbarn in eine Warteschlange im globalen Speicher zu schreiben, müssten diese Schreibzugriffe sequentialisiert werden. Der vorgestellte Algorithmus vermeidet dies durch eine Hierarchie von Warteschlangen. Abbildung 5.2 veranschaulicht diese Hierarchie.

[18] Ein Knoten ist in einer Iteration aktiv, wenn er sich in der zugehörigen Breitensuchebene befindet. Nicht zu verwechseln ist dies mit der aktiven Nachbarschaft.

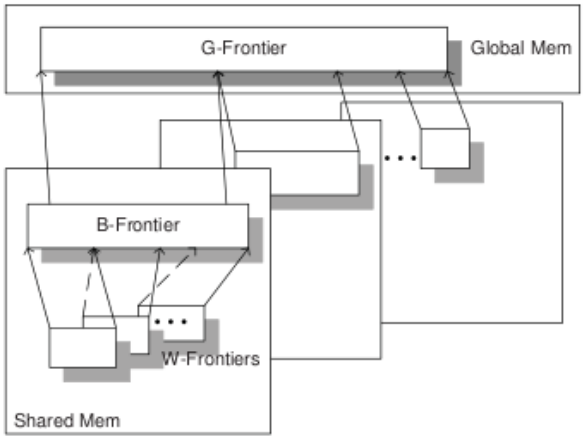


Abbildung 5.2: Veranschaulichung Warteschlangenhierarchie. Entnommen aus [20].

Die Warteschlangenhierarchie basiert auf der CUDA-Threadhierarchie. Die globale Warteschlange (in Abbildung 5.2 G-Frontier) befindet sich im globalen Speicherbereich und enthält nach jeder Iteration alle Knoten, die in der nächsten Iteration aktiv sind. Jeder Block verwaltet eine blockinterne Warteschlange (in Abbildung 5.2 B-Frontier), welche sich im shared-memory Bereich befindet. Die unterste Ebene dieser Hierarchie bilden Warp-Warteschlangen (in Abbildung 5.2 W-Frontier), welche sich ebenfalls im shared-memory befinden.
Zunächst schreiben die Threads die jeweiligen Nachbarn in die Warp-Warteschlangen. Obwohl der Name dies vermuten lässt, gibt es nicht pro Warp eine Warp-Warteschlange. Deren Anzahl ist abhängig von der Zahl der Streamingprozessoren pro Multiprozessor. Wie in Abschnitt 4 erwähnt, führt ein Multiprozessor die Threads eines Warps in Gruppen dieser Anzahl aus, wobei die Gruppen in der Reihenfolge der Thread-IDs ihrer Threads ausgeführt werden. Die Reihenfolge, in der die Warps ausgeführt werden, ist dabei irrelevant. Abbildung 5.3 veranschaulicht, wie dieses Scheduling zum Aufbau der Warp-Warteschlange ausgenutzt wird.

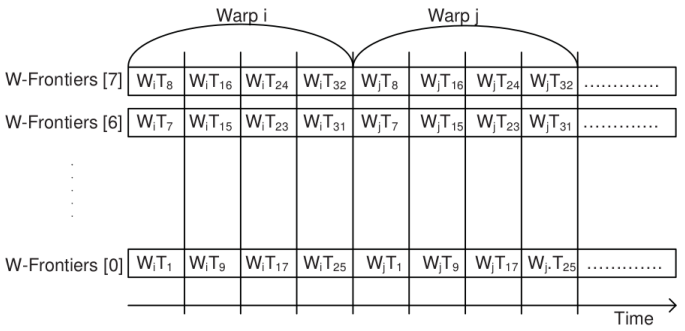


Abbildung 5.3: Veranschaulichung des Warp-Scheduling, welches zum Aufbau der Warp-Warteschlange ausgenutzt wird. Entnommen aus [20].

Threads, die zur gleichen Zeit ausgeführt werden, schreiben jeweils in eine andere Warp-Warteschlange und aktualisieren den entsprechenden Elementzähler. Dadurch vermeidet der Algorithmus Kollisionen. Nachdem die Threads alle Knoten für die nächste Iteration in Warp-Warteschlangen geschrieben haben, berechnet ein Thread pro Block anhand deren Elementzähler die Bereiche der Block-Warteschlange, in denen diese Elemente gespeichert werden sollen. Das Bestimmen dieser Bereiche muss ein einzelner Thread seriell ausführen, um Kollisionen zu vermeiden. Das Übertragen der Elemente von den Warp-Warteschlangen in die Block-Warteschlange erfolgt parallel. Das Übertragen der Elemente von den Block-Warteschlangen in die globale Warteschlange geschieht analog. Um Kollisionen bei der Bereichsbestimmung zu vermeiden, werden atomare Operationen benutzt.
Diese Hierarchie wird ebenfalls für eine effiziente Synchronisation verwendet. Es ist nicht nötig, für jede Iteration die Kernelfunktion erneut zu starten und dadurch eine Synchronisation nach jeder Iteration zu erzwingen. Der Algorithmus erreicht dies, indem er unabhängig von der Anzahl der aktiven Knoten die maximale Anzahl an Threads pro Block startet. Solange die Anzahl der aktiven Knoten die maximale Threadzahl pro Block nicht übersteigt, genügt ein Block. Dieser Block führt nach jeder Iteration eine blockinterne Synchronisierung durch. Ist die Anzahl der aktiven Knoten größer als die maximale Threadzahl pro Block, aber kleiner als die Anzahl maximal aktiver Threads auf der Grafikkarte, so können sich diese Threads über den globalen Speicher synchronisieren. Um die nötige Anzahl von Threads zu starten, muss die Kernelfunktion einmal beim Überschreiten dieser Grenze erneut aufgerufen werden. Erst wenn die globale Warteschlange noch mehr Knoten enthält, ist nach jeder Iteration ein erneutes Aufrufen der Kernelfunktion nötig, solange die Anzahl der Elemente in der Warteschlange nicht wieder unter die entsprechende Grenze fällt.

Der Nachteil dieses Algorithmus' ist die starke Architekturabhängigkeit beim Schreiben in die Warp-Warteschlangen. Für Grafikkarten mit compute capability $1.x$ ist der Algorithmus sehr effizient. Bei compute capability $2.x$ entspricht die Anzahl von SPs pro SM der Warpgröße. Damit wäre jedem Thread eine eigene Warteschlange zugeordnet, was zu einer ineffizienten sequenziellen Bereichsberechnung pro Thread führen würde. Die Anzahl der insgesamt sequenziell ausgeführten Schritte erhöht sich bei diesem Algorithmus von compute capability $1.x$ auf $2.x$ nicht. Bei $1.x$ ist ein Teil davon jedoch durch die Hardware bedingt und damit unausweichlich. Auf Grund dieser Architekturabhängigkeit wurde [20] nicht im Rahmen dieser Arbeit implementiert.

Der in [21] vorgestellte Algorithmus benutzt wie [20] eine als Array realisierte Warteschlange, nutzt jedoch den vollen Parallelisierungsgrad von compute capability $2.x$. Die Offsets zum Vermeiden des sequenziellen Füllens dieser Warteschlange werden mittels Präfixsummen berechnet. Das Paper [21] unterscheidet zwischen einer Knotenwarteschlange und einer Kantenwarteschlange. Erstere dient der Speicherung der in einer Iteration aktiven Knoten, während letztere in jeder Iteration die Nachbarn der aktiven Knoten speichert. Sie wird als Kantenwarteschlange bezeichnet, da sie die Zielknoten der von aktiven Knoten ausgehenden Kanten enthält und somit die im aktuellen Level durchlaufenen Kanten repräsentiert. Jede Iteration der Breitensuche ist in zwei Phasen unterteilt:

- *Expansion:* Die Expansionsphase liest die aktiven Knoten aus der Knotenwarteschlange und bestimmt deren Nachbarn. Sie expandiert damit die Knotenwarteschlange und generiert die nächste Kantenwarteschlange.

- *Kontraktion:* Die Kontraktionsphase liest die Knoten aus der Kantenwarteschlange und entfernt Knoten, die bereits als besucht markiert wurden. Den übrigen Knoten ordnet sie die aktuelle Ebene als Distanz zur Wurzel zu und markiert sie als besucht. Sie kontrahiert damit die Kantenwarteschlange und generiert die nächste Knotenwarteschlange.

Das Paper [21] stellt mehrere Methoden vor, die dieses Konzept nutzen. Zunächst soll der Aufbau der einzelnen Methoden erläutert werden, anschließend die Umsetzung der einzelnen Schritte. Allen Methoden ist gemeinsam, dass die Kernelfunktion, welche die Aufgaben einer Iteration implementiert, zur Synchronisation nach jeder Iteration erneut aufgerufen wird. Die vorgestellten Methoden sind:

- *Expansions-Kontraktions-Methode:* Hierbei bekommt die Kernelfunktion eine Knotenwarteschlange als Eingabe und führt zunächst eine Expansion dieser aus. Danach kontrahiert sie die generierte Kantenwarteschlange. Da jede Iteration die Knotenwarteschlange an die nächste weitergibt, ist diese im globalen Speicher realisiert (in [21]: out-of-core node queue). Der Hostcode startet für jeden Knoten in der Knotenwarteschlange einen Thread. Die Kantenwarteschlange wird nur innerhalb der Kernelfunktion verwendet und nicht vollständig realisiert (in [21]: in-core edge queue).

- *Kontraktions-Expansions-Methode:* Diese Methode funktioniert, abstrakt erklärt, analog zur Expansions-Kontraktions-Methode, wobei Knoten- und Kantenwarteschlange vertauscht sind. Der Hostcode startet für jede Kante in der Knotenwarteschlange einen Thread.

- *Getrennte-Phasen-Methode:* Hierbei gibt es zwei Kernelfunktionen, von denen jeweils eine die Expansions- bzw. Kontraktionsphase übernimmt. Es ist damit nötig sowohl die Knotenwarteschlange, als auch die Kantenwarteschlange im globalen Speicher zu realisieren. Pro Iteration finden zwei globale Synchronisierungen statt. Für den Expansionskernel startet der Hostcode eine der Knotenzahl entsprechende Anzahl von Threads, für den Kontraktionskernel eine der Kantenzahl entsprechende.

- *Hybrid-Methode:* Diese Methode setzt in Abhängigkeit der Anzahl an Elementen den *Kontraktions-Expansions-Kernel* oder die *Getrennte-Phasen-Methode* ein; ersteren falls die Anzahl nicht größer ist als die Anzahl maximal aktiver Threads, letztere anderenfalls.

Bereits auf dieser abstrakten Ebene lassen sich Vor- und Nachteile der Methoden erkennen. Bei der Expansions-Kontraktions-Methode ist nur die Knotenwarteschlange im globalen Speicher realisiert, was globalen Speicherplatz und Zugriffe auf diesen spart. Die Kontraktions-Expansions-Methode erreicht einen höheren Parallelisierungsgrad. Dies ist jedoch nur von Vorteil, wenn die Kantenwarteschlange nicht deutlich mehr Elemente enthält als die maximale Threadanzahl, da die Threads bei dieser Methode weniger Arbeit haben. Die Getrennte-Phasen-Methode nutzt die Rechenwerke der Grafikkarte am besten, da für die beiden Phasen jeweils eine

angepasste Anzahl von Threads gestartet wird. Dies ist jedoch nur von Vorteil, wenn die Kantenwarteschlange deutlich mehr Elemente enthält als die maximale Threadanzahl. Diese Methode benötigt den meisten globalen Speicher und Zugriffe darauf. Die Hybrid-Methode vereinigt die Vorteile der Kontraktions-Expansions-Methode und der Getrennte-Phasen-Methode. Wie Tests in [21] zeigen und in der eigenen Implementierung ergeben haben, sind die Vorteile der Kontraktions-Expansions-Methode gegenüber der Expansions-Kontraktions-Methode überwiegend, weshalb letztere in der Hybridmethode nicht verwendet wird. Die nachfolgenden Unterpunkte beschreiben die in [21] aufgezeigten Möglichkeiten für eine effiziente Implementierung der Expansions- und Kontraktionsphase.

Parallele Präfixsumme: Um atomare Operationen beim Berechnen von Offsets für das parallele Füllen von Warteschlangen zu vermeiden, verwendet [21] Präfixsummen. Die Präfixsumme über ein Array a ist folgendermaßen definiert: präfix_summe$[i] = \sum_{k=0}^{i-1} a[k]$. Deren Berechnung lässt sich mit Hilfe von CUDA parallelisieren [15].

Expansion (Bestimmen der Nachbarn): Die naheliegende Methode zum Bestimmen der Nachbarn ist, dass jeder Thread die Nachbarn eines Knotens bestimmt. Dies hat jedoch entscheidende Nachteile. Zum einen führt es zu einer ungleichen Lastverteilung, da Knoten sehr unterschiedliche Anzahlen von Nachbarn haben können, zum anderen bestimmt ein Teil der Threads die Nachbarn seriell, während ein anderer Teil inaktiv ist. Wird die Kontraktion vor der Expansion durchgeführt, so sind Threads inaktiv, deren Knoten als besucht markiert wurden. In beiden Fällen sind die Threads inaktiv, deren Knoten Duplikate sind. Warum Duplikate vorkommen und wie diese gefiltert werden, ist weiter unten in diesem Abschnitt erklärt.
Ziel ist es, eine bessere Lastverteilung zu erreichen, potenziell jeden Nachbarn parallel zu bestimmen und damit möglichst keine inaktiven Threads zu haben. Daher behandelt der Algorithmus die Knoten je nach Anzahl der Nachbarn auf verschiedene Art und Weise.

Feingranular-blockbasierend: Diese Methode wird für Knoten angewandt, die weniger Nachbarn haben als die Anzahl von Threads pro Warp. Jeder Thread speichert in einer Schleife, die über den Adjazenzlisten-Indexbereich seines Knotens läuft, die einzelnen Adjazenzlisten-Indizes in ein Array im shared-memory Bereich. Dabei kann auf Grund von verschiedenen Nachbarzahlen das oben erwähnte Problem der inaktiven Threads auftreten, was aber an dieser Stelle akzeptabel ist, wenn die Knoten nur wenige Nachbarn haben, da in diesem Fall nicht viele Threads inaktiv sind und nicht für lange Zeit, weil die aktiven Threads wenig Arbeit haben.
Die Offsets, mit denen die Threads die jeweiligen Indizes in das shared-memory-Array schreiben, werden mittels der oben erwähnten Präfixsumme bestimmt. Anschließend können alle Threads des Blocks jeweils einen Index aus dem shared-memory-Array lesen und den zugehörigen Knoten aus der Adjazenzliste laden. Somit werden die Nachbarn parallel geladen. Da die Größe des shared-memory-Array dafür optimalerweise der Anzahl an Threads pro Block entspricht, findet die Expansion in einer Schleife statt, die in jeder Iteration entsprechend viele Indizes in das shared-memory-Array schreibt und danach parallel die jeweiligen Knoten lädt. Diese Methode nutzt die Hardware beim Lesen der Nachbarn optimal aus. Da aufeinander folgende Threads Daten lesen, welche hintereinander im globalen Speicher liegen, finden effiziente zusammenhän-

gende Speicherzugriffe statt. Sie ist aber, wie bereits erwähnt, nur für Knoten mit einer Nachbarzahl geeignet, welche die Anzahl an Threads pro Warp nicht übersteigt, da sonst eine sehr geringe Zahl von Threads, im schlechtesten Fall nur ein einziger, Indizes für die Nachbarn seiner Knoten berechnet.
Algorithmus 5.1 zeigt kommentierten Pseudocode für diese Methode.
Am Anfang ist jedem Knoten ein Thread zugeordnet, der dessen Adjazenzlisten-Indizes in ein shared-memory-Array schreibt. Danach wird die Arbeit, das eigentliche Einlesen der Nachbarn, gleichmäßig auf alle Threads eines Blocks aufgeteilt (load balancing). Die Verbindung von einem Knoten zu einem Thread gibt es dabei nicht mehr. Die Nachbarn eines Knotens werden dennoch in dem Block bestimmt, zu dem der Thread, dem der Knoten anfangs zugeteilt war, gehörte.

Folgendes setzt Algorithmus 5.1 als bekannt voraus:

- node, der vom aktuellen Thread behandelte Knoten
- thread_id, die ID des aktuellen Threads
- BLOCK_THREADS, die Anzahl an Threads pro Block
- prefix_sum(), eine Funktion, welche die Präfixsumme auf dem als erstes Argument übergebenen Array berechnet. Im zweiten Argument speichert sie die Gesamtsumme der Arrayelemente.

Grobgranular-warpbasierend: Die warpbasierende Methode ist für Knoten mit einer Nachbarzahl, die größer als die Anzahl von Threads pro Warp ist, geeignet. Sie bestimmt die Adjazenzlisten-Indizes parallel. Dafür teilt jeweils ein Thread, dessen Nachbarn noch nicht bestimmt wurden, seinen Indexbereich dem gesamten Warp, in dem er sich befindet, mit. Dieser Thread kontrolliert damit sein Warp. Jeder Thread dieses Warps bestimmt einen Index jenes Bereichs und lädt den entsprechenden Nachbarn aus der Adjazenzliste. Da der Indexbereich des kontrollierenden Threads bei der Anwendung des warpbasierenden Verfahrens meistens größer als die Anzahl an Threads pro Warp ist, läuft dies in einer Schleife, bis alle Indizes des Bereichs bestimmt wurden. Das Verfahren läuft insgesamt solange in einer äußeren Schleife, bis jeder Thread des Warps, dem ein aktiver Knoten mit Nachbarn zugeordnet ist, einmal die Kontrolle übernommen hat. Falls der kontrollierende Knoten genug Nachbarn hat, finden zusammenhängende Speicherzugriffe statt, da Threads mit aufeinander folgenden IDs Daten lesen, die hintereinander im Speicher stehen. Für Knoten mit geringerer Nachbarzahl ist diese Methode weniger effizient, da es in diesem Fall inaktive Threads beim Lesen der Nachbarn aus dem globalen Speicher gibt und bei zu wenigen Nachbarn keine zusammenhängenden Speicherzugriffe möglich sind. Algorithmus 5.2 zeigt kommentierten Pseudocode für diese Methode. Für Knoten, welche mehr Nachbarn haben als Threads pro Block, kann man diese Methode statt auf Warps analog auf ganze Blocks anwenden (grobgranular-blockbasierend).

Algorithmus 5.1 : Expansion feingranular-blockbasierend, vgl. [21]

```
//Arrays im shared-memory-Bereich
shared shared_queue[BLOCK_THREADS]
shared prefix_sum_buffer[BLOCK_THREADS]
//Speicherplatz für die Gesamtsumme im shared-memory-Bereich
shared total_sum

//Bestimmung des Indexbereichs
from ← adj_list_offset[node]
to ← adj_list_offset[node + 1]
/*Initialisierung der entsprechenden Stelle des Array für die Präfixsumme mit der Anzahl
der Nachbarn, für die der aktuelle Thread zuständig ist*/
prefix_sum_buffer[thread_id] ← to - from
//Berechnung der Präfixsumme auf dem übergebenen Array
prefix_sum(prefix_sum_buffer, total_sum);

/*Das Array enthält nun an Stelle der ID jedes Threads die Anzahl der Nachbarn, die alle
Threads mit geringerer ID zusammen haben. Dies ist der Offset, ab dem der aktuelle
Thread die Indizes seiner Nachbarn in die Warteschlange im shared-memory-Bereich
schreibt. total_sum ist nun die Anzahl der Nachbarn, die dieser Block behandeln muss.*/

block_progress ← 0; //Anzahl der von diesem Block bereits behandelten Nachbarn
offset ← prefix_sum_buffer[thread_id]
//Die folgende Schleife läuft, solange es noch nicht behandelte Nachbarn gibt.
while (remain ← total_sum - block_progress) > 0 do
    /*In der folgenden Schleife berechnen die Threads die Indizes der Nachbarn ihrer
    Knoten und schreiben diese in das shared-memory-Array. Sie läuft, solange noch Platz
    im Array ist und solange der jeweilige Thread noch Nachbarn hat*/
    while (offset < block_progress + BLOCK_THREADS) && (from < to) do
        shared_queue[offset - block_progress] ← from
        offset++ //Stelle, an die der Index des nächsten Nachbarn geschrieben wird
        from++ //Index des nächsten Nachbarn
    end
    synchronize()
    if thread_id < min(remain, BLOCK_THREADS) then
        neighbour ← adj_lists_edges[thread_id]
        /*An dieser Stelle kann der gefundene Nachbar weiterverarbeitet werden. Dies ist
        nicht Teil dieses Pseudocodes*/
    end
    block_progress ← block_progress + BLOCK_THREADS
    synchronize()
end
```

Folgendes setzt Algorithmus 5.2 als bekannt voraus:

- node, der vom aktuellen Thread behandelte Knoten
- thread_id, die ID des aktuellen Threads
- WARP_SIZE, die Anzahl an Threads pro Warp
- WARPS, die Anzahl an Warps pro Block
- BLOCK_THREADS, die Anzahl an Threads pro Block
- warp_id, die ID des aktuellen Waps, warp_id = BLOCK_THREADS *modulo* WARP_SIZE
- lane_id, die Nummer eines Threads innerhalb seines Warps, lane_id = BLOCK_THREADS ÷ WARP_SIZE

Falls die Kontraktion vor der Expansion ausgeführt worden ist, ist es nötig, anschließend sämtliche gefundenen Nachbarn in die globale Kantenwarteschlange zu schreiben, während die Expansions-Kontraktions-Methode die gefundenen Nachbarn direkt behandelt.
Nachbarn, die mittels der feingranular-blockbasierenden Methode gefunden worden sind, schreiben die Threads direkt nach dem Einlesen aus der Adjazenzliste parallel als Gruppe, deren Größe der Anzahl an Nachbarn in der Warteschlange im shared-memory entspricht, in die Kantenwarteschlange im globalen Speicher. Da aufeinander folgende Threads dabei Daten hintereinander in diesen Speicher schreiben, finden, wie beim Lesen, schnelle zusammenhängende Speicherzugriffe statt. Damit es keine Kollisionen gibt, wenn mehrere Blocks gleichzeitig ihre gerade behandelte Gruppe von Nachbarn in den globalen Speicher zu schreiben suchen, erhöht jeweils ein Thread aus dem Block den Zähler der globalen Kantenwarteschlange unter Verwendung einer atomaren Operation um die Größe der Gruppe, die geschrieben werden soll. Den Wert den der Zähler davor aufgewiesen hat, speichert dieser Thread im shared-memory des Blocks[19]. Jener Wert dient für alle Threads dieses Blocks als Offset, welcher addiert mit der Thread-ID den Index ergibt, an dessen Stelle dieser den ihm zugeteilten Nachbarn aus der Gruppe in die globale Kantenwarteschlange schreibt.
Nachbarn, die mittels der groben, warpbasierenden Methode gefunden worden sind, schreiben die Threads ebenfalls direkt nach dem Einlesen in die globale Kantenwarteschlange. Die Bestimmung der Offsets ist für diese jedoch komplexer. Dafür wird zuvor eine Präfixsumme über die Größen der Indexbereiche, welche der Anzahl der Nachbarn entspricht, derjenigen Knoten berechnet, deren Nachbarn die warpbasierende Methode bestimmen soll. Das Berechnen der Präfixsumme ist analog zu der für die feingranular-blockbasierende Methode. Ein Thread des Blocks addiert unter Verwendung einer atomaren Operation die Gesamtanzahl von Nachbarn, die sein Block mittels der warpbasierenden Methode bestimmt, auf den Zähler der globalen Kantenwarteschlange. Den Wert, den der Zähler davor aufgewiesen hat, schreibt er in den shared-memory

[19] Die von CUDA bereitgestellte atomare Additionsfunktion gibt den Wert zurück, welchen die entsprechende Speicherstelle im globalen Speicher zuvor aufgewiesen hat. Ein weiterer Zugriff auf diese Speicherstelle kann erst stattfinden, wenn sowohl Lesen als auch Schreiben vollzogen sind. Somit kann es hier keine “race condition” geben.

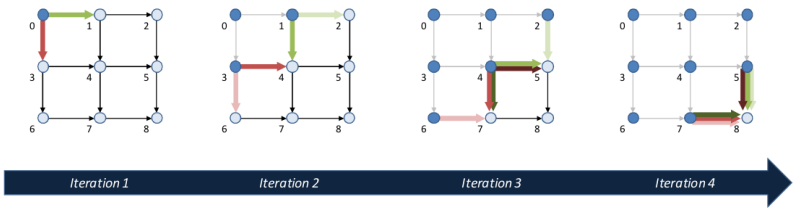
Algorithmus 5.2 : Expansion grobgranular-warpbasierend, vgl. [21]

```
//Arrays im shared-memory-Bereich
shared warp_buffer[WARPS][3]

//Bestimmung des Indexbereichs
from ← adj_list_offset[node]
to ← adj_list_offset[node + 1]

/*Die folgende Schleife läuft, solange ein Thread des aktuellen Warps nicht behandelte
Nachbarn hat.*/
while warp_any(to - from) do
    /*Jeder Thread, der nicht behandelte Nachbarn hat, versucht, seine lane_id in den
    seinem Warp zugeordneten Warpbuffer zu schreiben. Nur ein Thread wird erfolgreich
    sein.*/
    if to - from then
        warp_buffer[warp_id][0] ← lane_id
    end
    /*Der erfolgreiche Thread schreibt seinen Indexbereich in den seinem Warp
    zugeordneten Warpbuffer und erlangt damit die Kontrolle über das Warp.*/
    if warp_buffer[warp_id][0] == lane_id then
        warp_buffer[warp_id][1] ← from
        warp_buffer[warp_id][2] ← to
        /*Der Breich dieses Threads wird auf 0 gesetzt, damit er nicht nochmal behandelt
        wird.*/
        from ← to
    end

    /*Jeder Thread bestimmt nun einen Index ausgehend vom Beginn des Bereichs des
    Threads, der die Kontrolle über sein Warp hat.*/
    gather_from ← warp_buffer[warp_id][1] + lane_id
    gather_to ← warp_buffer[warp_id][2]
    /*Solange sich dieser Index im Bereich des kontrollierenden Threads befindet, lädt der
    aktuelle Thread den zu diesem Index gehörenden Knoten aus der Adjazenzliste. Falls
    der Knoten des kontrollierenden Threads mehr Nachbarn hat als die Anzahl an
    Threads pro Warp, müssen mehrere Iterationen durchgeführt werden. Auf den Index
    eines Threads wird daher vor der nächsten Iteration die Warpgröße addiert*/
    while gather_from < gather_to do
        neighbour ← adj_lists_edges[gather_from]
        gather_from ← gather_from + WARP_SIZE
        /*An dieser Stelle kann der gefundene Nachbar weiterverarbeitet werden. Dies ist
        nicht Teil dieses Pseudocodes*/
    end
end
```

Bereich des Blocks. Dieser dient als Grundoffset für das Schreiben der mit jener Methode bestimmten Nachbarn. Der Thread, welcher die Kontrolle übernimmt, schreibt zusätzlich zum Anfang und Ende seines Indexbereichs seinen Präfixsummenwert assoziiert mit seinem Warp in den shared-memory-Bereich. Dieser Präfixsummenwert entspricht der Anzahl von Elementen, die Threads mit kleinerer ID schreiben und dient als zweite Komponente für den Offset. Die Stelle, an die Threads den ersten Nachbarn ihres Aufgabenbereichs schreiben, ergibt sich aus der Summe von Grundoffset, dem entsprechenden Präfixsummenwert und der lane_id (s. Algorithmus 5.2) des Threads. Dieser Gesamtoffset wird in der while-Scheife für jeden weiteren Nachbarn um die Warpgröße erhöht.

Kontraktion: Wird die Kontraktion vor der Expansion ausgeführt, besteht sie lediglich darin, dass jeder Thread denjenigen Knoten aus der Kantenwarteschlange lädt, der an der Stelle seiner Thread-ID gespeichert ist, prüft, ob dieser bereits besucht worden ist und, falls ja, inaktiv gesetzt wird. Die Threads, die dann noch aktiv sind, sind alle für einen Knoten der impliziten, soeben kontrahierten Knotenwarteschlange zuständig. Dies ist der Grund für den oben in diesem Abschnitt erwähnten Nachteil der inaktiven Threads der Kontraktions-Expansions-Methode. Ein zusätzlicher Vorteil ist hingegen die einfache Kontraktionsphase.
Wird die Kontraktion nach der Expansion ausgeführt, so bestimmen die Threads nach dem Laden der Nachbarn aus der Adjazenzliste, welche dieser Nachbarn bereits besucht wurden. Die noch nicht besuchten Nachbarn müssen in die globale Knotenwarteschlange geschrieben werden. Da diese hintereinander stehen, also nicht durch inaktive Elemente getrennt sein sollen, ist eine weitere Präfixsumme nötig, welche die Indizes der Knotenwarteschlange bestimmt, an deren Stelle die Threads mit aktiven Knoten diese schreiben sollen. Dafür schreibt jeder Thread eine 1 in ein Array im shared-memory-Bereich, falls sein Knoten aktiv ist, anderenfalls eine 0. Die Präfixsumme über dieses Array bestimmt die gewünschten Offsets und die Anzahl von Knoten, welche dieser Block in die Knotenwarteschlange schreibt. Sie wird einmal pro Gruppe von Nachbarn, welche die feingranular-blockbasierende Methode bestimmt hat, und einmal für alle Nachbarn, die die warpbasierende Methode bestimmt hat, berechnet. Das kollisionsfreie Schreiben in die globale Knotenwarteschlange geschieht analog zur oben beschriebenen Vorgehensweise bei der globalen Kantenwarteschlange.

Duplikate entfernen: Wie oben in diesem Abschnitt bereits erwähnt, können in den Warteschlangen Duplikate vorkommen. Da alle Knoten, die in einer Breitensuchebene aktiv sind, potenziell gleichzeitig behandelt werden, schreiben Knoten, die den gleichen Nachbarn haben, diesen jeweils einmal in die Warteschlange. Abbildung 5.4 veranschaulicht dies und zeigt, dass der gleiche Knoten sehr oft in der Warteschlange vorkommen kann.
Dies verlangsamt die Ausführung der Breitensuche enorm und kann weiter zum Überlaufen der globalen Warteschlange führen. Deshalb müssen Duplikate in jeder Iteration entfernt werden. Der Algorithmus schreibt die Duplikate einer Iteration in die globale Warteschlange, da ein Löschen derer davor zu zeitaufwändig und kompliziert wäre. Das Löschen wird dadurch realisiert, dass ein Thread nach dem Lesen seines Knotens aus der Knotenwarteschlange (Expansions-Kontraktions-Methode) bzw. aus der Kantenwarteschlange (Kontraktions-Expansions-Methode) prüft, ob dieser ein Duplikat ist und sich gegebenenfalls deaktiviert. Er deaktiviert sich, in dem



BFS Iteration	*Actual Vertex-frontier*	*Actual Edge-frontier*
1	0	1,3
2	1,3	2,4,4,6
3	2,4,4,6	5,5,7,5,7,7
4	5,5,7,5,7,7	8,8,8,8,8,8,8,8

Abbildung 5.4: Veranschaulichung des gleichzeitigen Entdeckens von Knoten, welches zu Duplikaten in den Warteschlangen führt. Entnommen aus [21].

er den Indexbereich seines Knotens auf Null setzt, um im späteren Verlauf der Kernelfunktion nicht mehr beachtet zu werden. Damit ein Thread feststellen kann, ob sein Knoten ein Duplikat ist, stellt [21] zwei Heuristiken vor. Zum einen eine warpbasierende, welche mittels Hashing Duplikate innerhalb eines Warps entdeckt. Zum anderen eine, welche aktive Knoten cached, prüft, ob der Knoten eines Thread sich im Cache befindet und diesen gegebenenfalls deaktiviert. Erstere Methode ist in der eigenen Implementierung vorhanden, letztere nicht. Auf die Funktionsweise dieser beiden wird in dieser Arbeit nicht weiter eingegangen. Die Implementierung benutzt eine eigene Methode, die sich im Test für die Beispielgraphen als effizienter herausgestellt hat. Der Nachteil der beiden Methoden aus [21] ist, dass sie nicht alle Duplikate entdecken. Die eigene Methode erreicht dies durch eine getrennte Kernelfunktion. Der Hostcode ruft diese am Ende jeder Iteration mit einem Thread pro Knoten in der globalen Warteschlange auf, unabhängig ob es sich um eine Kanten- oder Knotenwarteschlange handelt. Jeder Thread benutzt die ID seines Knotens als Index für ein globales Array (welches n = #Knoten viele Elemente aufnehmen kann) an dessen Stelle er die Summe seiner Block- und Thread-ID schreibt. Dies wird pro Knoten-ID nur einem Thread gelingen. Da der Hostcode die Kernelfunktion für die nächste Breitensuchebene mit der gleichen Anzahl von Threads aufruft, kann jeder Thread nach dem Lesen seines Knotens auf der globalen Warteschlange prüfen, ob die Summe seiner Block- und Thread-ID an Stelle der ID seines Knoten im eben erwähnten Array steht. Ist dies nicht der Fall, deaktiviert er sich. Damit werden alle Duplikate aus der Warteschlange entfernt. Dies bringt den Vorteil, dass eine Knotenwarteschlange der Größe n bzw. eine Kantenwarteschlange der Größe m garantiert reicht.
Der Nachteil dieser Methode ist der zusätzliche Kernelaufruf und die Zugriffe auf den globalen Speicher. Je nach dem wie viele Duplikate vorkommen, lohnt sich die eine oder andere Methode.

Eigene Implementierung: Die eben beschriebene Methode bringt beachtliche Geschwindigkeitsvorteile bei sehr großen Graphen mit kleinem Durchmesser. Für die meisten in dieser Arbeit verwendeteten Beispielgraphen zeigt sich jedoch kein Vorteil gegenüber der CPU-Implementierung. Das Problem liegt darin, dass bei kleineren Graphen und Graphen mit großem Durchmesser die Recheneinheiten der Grafikkarte nicht ausgelastet sind. Deshalb nutzt die Implementierung in dieser Arbeit die Unabhängigkeit der q-vielen Breitensuchen. Der Aufruf der entsprechenden Kernelfunktion behandelt jeweils eine Breitensuchebene aller q Breitensuchen. Das führt zu einer deutlich höheren Auslastung der Recheneinheiten. Für diese spezielle Anwendung ist die

Expansions-Kontraktions-Methode die geeignetste. Wie oben in diesem Abschnitt erwähnt, muss man einen Kompromiss zwischen Anzahl inaktiver Threads und Höhe des Parallelisierungsgrads eingehen. Ersteres ist bei der Expansions-Kontraktions-Methode stärker gewichtet, zweiteres bei der Kontraktions-Expansions-Methode. Da der Parallelisierungsgrad durch das gleichzeitige Behandeln aller Breitensuchen hoch genug ist, führt der Vorteil, weniger inaktive Threads zu haben, zu einer besseren Laufzeit.
Da die Vermeidung inaktiver Threads wichtiger ist, wird außerdem bei der Expansion ausschließlich die feingranular-blockbasierende Methode verwendet.
Ein weiterer Vorteil der Expansions-Kontraktions-Methode, welcher in diesem Fall deutlich mehr Gewicht bekommt, ist der geringere Speicherverbrauch. Weil alle Breitensuchen unabhängige globale Warteschlangen benötigen, müssen $2q$ Warteschlangen[20] im globalen Speicher der Grafikkarte liegen.
Alle Breitensuchen potenziell gleichzeitig auszuführen hat jedoch den Nachteil, dass eine Bestimmung der Pivotknoten mittles des min-max-Verfahrens nicht mehr möglich ist. Die Implementierung wählt die Pivotknoten zufällig.
In der eigenen Implementierung wurde der Code um das Bestimmen der Pivotknoten mittels der min-max-Methode und der Pivotgewichtungen erweitert.

Vergleich mit der CPU-Version: Der Geschwindigkeitsvorteil, den die GPU-Implementierung gegenüber der CPU-Implementierung hat, ist abhängig vom Durchmesser des Graphen. Je kleiner der Durchmesser, desto mehr Knoten werden pro Breitensuchebene bearbeitet, um so höher ist der mögliche Parallelisierungsgrad und desto geeigneter ist die GPU-Implementierung. Ab einem bestimmten Durchmesser ist jede Grafikkarten-Implementierung langsamer als eine schnelle CPU-Implementierung, da die CPU-Architektur für Berechnungen mit geringem möglichen Parallelisierungsgrad geeigneter ist. Tabelle 5.2 zeigt die Laufzeiten in Sekunden, welche die Bestimmung der Distanzen zu den Pivotknoten bei den oben gezeigten Beispielgraphen benötigt. Die CPU-Implementierung und die GPU-Implementierung, welche die Breitensuchen einzeln ausführt, bestimmen die Pivotelemente nach der min-max-Methode, die GPU-Implementierung, welche die Breitensuchen zusammen ausführt, verwendet eine zufällige Pivotauswahl. Die Zeiten in Klammern bei den ersten beiden Verfahren zeigen, welche diese mit einer zufälligen Pivotauswahl benötigen. Die Laufzeitmessungen wurden auf dem System bestehend aus AMD Phenom II X4 850 @ 3,3 GHz CPU und Nvidia GTX 550 Ti GPU durchgeführt.

[20] $2q$, da jede Breitensuche eine Input- und Outputwarteschlange benötigt

Graph	**#Knoten**	**#Kanten**	CPU [s]		GPU einzeln [s]		GPU zus. [s]
smallworld	1000	3000	0.0014	(0.0013)	0.1346	(0.1377)	0.0872
data	2851	15093	0.0081	(0.0075)	0.1845	(0.1713)	0.0926
3elt	4720	13722	0.0133	(0.0111)	0.1620	(0.1658)	0.0940
uk	4824	6837	0.0116	(0.0091)	0.3227	(0.3117)	0.0954
add32	4960	9462	0.0126	(0.0104)	0.1224	(0.1204)	0.0872
bcsstk33	8738	291583	0.0864	(0.0951)	0.4609	(0.4685)	0.2984
flower_050	9030	131241	0.0472	(0.0486)	0.5196	(0.4972)	0.1482
grid_rnd_100	9497	17849	0.0240	(0.0195)	0.3327	(0.3099)	0.1055
snowflake_C	9701	9700	0.0228	(0.0142)	1.6302	(1.4346)	0.1825
whitaker3	9800	28989	0.0323	(0.0259)	0.2884	(0.2822)	0.1022
sierpinski_08	9843	19683	0.0232	(0.2057)	0.3437	(0.1324)	0.1074
spider_C	10000	22000	0.0219	(0.0195)	2.3641	(2.1270)	0.2009
crack	10240	30380	0.0337	(0.0294)	0.2268	(0.2158)	0.1113
4elt	15606	45878	0.0665	(0.0574)	0.2867	(0.2300)	0.1258
bcsstk31	35588	572914	0.2556	(0.2625)	0.6691	(0.6427)	0.4675
t60k	60005	89440	0.1755	(0.1431)	0.9664	(0.9334)	0.2562
wing	62032	121544	0.7157	(0.6063)	0.5140	(0.4443)	0.3037
598a	110971	741934	2.1235	(1.8749)	1.0647	(0.9486)	0.8473
fe_ocean	143437	409593	1.4680	(1.2612)	1.0898	(0.9176)	0.6632
m14b	214765	1679018	4.1458	(3.7710)	1.9959	(1.7814)	1.7609
auto	448695	3314611	10.8724	(9.6457)	4.2048	(3.7906)	3.5424

Tabelle 5.2: Laufzeiten, welche die CPU, die GPU bei einzelner Behandlung einer Breitensuche sowie die GPU beim Behandeln aller Breitensuchen zusammen für die Berechnung der Distanzen zu den Pivotknoten benötigen.

Distanzen zur aktiven Nachbarschaft

Zur Bestimmung der aktiven Nachbarschaft jedes Knotens v und der Distanzen von v zu dieser sind n Breitensuchen nötig, welche parallel ausgeführt werden können. In der Implementierung führt je ein Thread eine dieser Breitensuchen aus. Damit skaliert die Bestimmung der aktiven Nachbarschaft mit der Größe des Graphen. Der Hostcode muss die entsprechende Kernelfunktion nur einmal aufrufen. Die Breitensuche, welche jeder Thread ausführt, entspricht weitgehend der seriellen Version. Sie bricht jedoch ab, nachdem sie a (= Größe der aktiven Nachbarschaft) viele Knoten gefunden hat. Es gibt zwei weitere Anpassungen. Für jede Breitensuche muss es eine eigene Instanz des Arrays *visited*, welches speichert, ob ein Knoten bereits besucht wurde, geben. Für den Registerbereich ist dieses Array jedoch zu groß. Selbst im globalen Speicher wäre es für große Graphen nicht möglich, n-viele Arrays mit jeweils n Einträgen zu halten. In der Implementierung hat *visited* deshalb die konstante Größe a (= Anzahl an Knoten in der aktiven Nachbarschaft) und wird als Hashtabelle benutzt. Die verwendete Hashfunktion ist eine binäre Und-Verknüpfung mit $a - 1$. Die Implementierung verwendet bei $a = 64$ eine Hashtabelle der Größe 512. Das Verwenden einer Hashtabelle kann dazu führen, dass noch nicht markierte Knoten als besucht angenommen und damit nicht beachtet werden.
Diese Hashtabelle könnte im Registerbereich eines Threads gehalten werden, sie wäre jedoch so groß, dass nur wenige Threads pro Block aktiv sein könnten, da bereits wenige dieser Hashtabellen den einem Block zu Verfügung stehenden Registerbereich füllen würden. Die Laufzeit ist unter Verwendung des globalen Speichers, auf Grund des deutlich höheren Parallelisierungsgrads, besser.
Weiter speichern die Threads Tupel aus Knoten und deren Abstand zur Wurzel anstatt lediglich den Knoten in ihrer lokalen Warteschlange. Ansonsten müsste pro Kindknoten einmal auf dessen Distanz im globalen Speicher zugegriffen werden. Dafür wäre zusätzlich eine Abbildung von der Knoten-ID auf die entsprechende Distanz in der Matrix nötig (neigh_idx speichert die umgekehrte Abbildung).

Tabelle 5.3 zeigt die Laufzeiten, welche CPU und GPU zur Bestimmung der aktiven Nachbarschaft und deren Distanzen zu den jeweiligen Knoten benötigen. Dabei verwendet die CPU-Implementierung anstatt der Hashtabelle das volle visited-Array. Die Laufzeitmessungen wurden auf dem System bestehend aus AMD Phenom II X4 850 @ 3,3 GHz CPU und Nvidia GTX 550 Ti GPU durchgeführt.

Graph	**#Knoten**	**#Kanten**	CPU [s]	GPU [s]
smallworld	1000	3000	0.0012	0.0032
data	2851	15093	0.0042	0.0139
3elt	4720	13722	0.0065	0.0194
uk	4824	6837	0.0059	0.0181
add32	4960	9462	0.0049	0.0135
bcsstk33	8738	291583	0.0064	0.0282
flower_050	9030	131241	0.0178	0.0605
grid_rnd_100	9497	17849	0.0115	0.0359
snowflake_C	9701	9700	0.0087	0.0356
whitaker3	9800	28989	0.0149	0.0397
sierpinski_08	9843	19683	0.0096	0.0413
spider_C	10000	22000	0.0096	0.0387
crack	10240	30380	0.0169	0.0431
4elt	15606	45878	0.0256	0.1241
bcsstk31	35588	572914	0.0543	0.0946
t60k	60005	89440	0.1583	0.2176
wing	62032	121544	0.3260	0.2652
598a	110971	741934	1.0364	0.3957
fe_ocean	143437	409593	1.3813	0.4979
m14b	214765	1679018	3.1789	0.7381
auto	448695	3314611	13.6375	1.6423

Tabelle 5.3: Laufzeiten, welche die CPU und die GPU für die Berechnung der Distanzen zur aktiven Nachbarschaft benötigen.

5.3 Pivot-MDS

Den überwiegenden Teil der Rechenzeit benötigt Pivot-MDS bei großen Graphen für das Berechnen von C^TC. Die Doppelzentrierung erfordert eine Laufzeit von $\mathcal{O}(qn)$, C^TC benötigt $\mathcal{O}(q^2n)$ und die Potenziteration $\mathcal{O}(q^2)$. Auch wenn ein konstantes q (für die Tests 64) angenommen wird, so fällt q^2 als Faktor vor n bei den konkreten Laufzeiten deutlich ins Gewicht. Außerdem sind die konstanten Operationen bei C^TC Gleitkomma-Multiplikationen, welche zeitaufwändiger sind als die Additionen bei der Doppelzentrierung. Bei der Potenziteration ist die benötigte Rechenzeit unabhängig von der Größe des Graphen. Deswegen wird ihr Beitrag zur Laufzeit von Pivot-MDS mit zunehmender Graphengröße geringer.
Daher wurde im Rahmen dieser Arbeit nur die Matrixmultiplikation C^TC auf die Grafikkarte portiert. Die Beispiele unten in diesem Abschnitt zeigen das beschriebene Laufzeitverhalten.

Host Code (erster Teil): Zunächst werden die für die Doppelzentrierung der Matrix $D_p^{(2)}$ benötigten Durchschnitte und mit deren Hilfe die Matrix C bestimmt. Da die Einträge in C oft so groß sind, dass die Elemente von C^TC nicht mehr in den *float* Datentyp passen und eine Berechnung von C^TC unter Verwendung des *double* Datentyps merkbar länger dauern würde, wird C an dieser Stelle normalisiert. Die verwendete Norm ist dabei die Division aller Matrixelemente durch ihren Gesamtdurchschnitt. Durch diese Operation ändern sich die Singulärvektoren von C nicht und damit auch nicht die Eigenvektoren von C^TC. Da die Eigenwerte dadurch mit einem bestimmten Faktor skaliert werden, führt dies zu einer Skalierung des von Pivot-MDS generierten Layouts, was jedoch keine Rolle spielt, da die kombinierte MDS die Initialkonfiguration vor der ausgedünnten Stressmajorisierung unabhängig von dieser Skalierung skaliert.
Anschließend reserviert der Host Code den für die Matrixmultiplikation C^TC nötigen globalen Speicher für die Matrix C und die Ergebnismatrix C^TC auf der Grafikkarte und kopiert die Matrix C dort hin. Für das Berechnen von C^TC wird für jedes Element der Ergebnismatrix C^TC ein Thread gestartet, welcher dieses Ergebniselement berechnet. Um den *shared-memory*-Bereich gut nutzen zu können, verwendet die Implementierung "kachelweise" Matrixmultiplikation (tiled matrix multiplication [19], [3]). Dazu wird die Ergebnis-Matrix in quadratische Kacheln eingeteilt, welche eine bestimmte Menge von Elementen enthalten. Jeder Threadblock kümmert sich um eine solche Kachel. Diese Einteilung legt nahe, die Blöcke in einem zweidimensionalen Grid zu organisieren. Die Blöcke selbst sind ebenfalls zweidimensional. Die Implementierung benutzt 16 Threads pro Blockdimension, was insgesamt 256 Threads pro Block ergibt. Die Größe der Dimensionen sollten Zweierpotenzen sein und 16 ist die größte, deren Quadrat die mögliche Anzahl von Threads pro Block nicht übersteigt. Würde eine kleinere Zahl gewählt, so könnte die Anzahl der nötigen globalen Speicherzugriffe weniger reduziert werden, siehe weiter unten in diesem Abschnitt. Außerdem würde eine zu kleine Dimensionsgröße (z.B. 4) die Zahl der maximal ausführbaren Threads stark einschränken.

Device Code: Jeder Thread führt die Kernelfunktion (Device Code), die sich um die Matrix-Multiplikation C^TC kümmert, aus. In der Kernelfunktion berechnen die Threads zunächst aus der x- bzw. y-Komponente der Block- und Thread-ID den Index der Spalte beziehungsweise Reihe der Matrix für die sie zuständig sind. Aus diesen beiden Indizes ergibt sich das Ele-

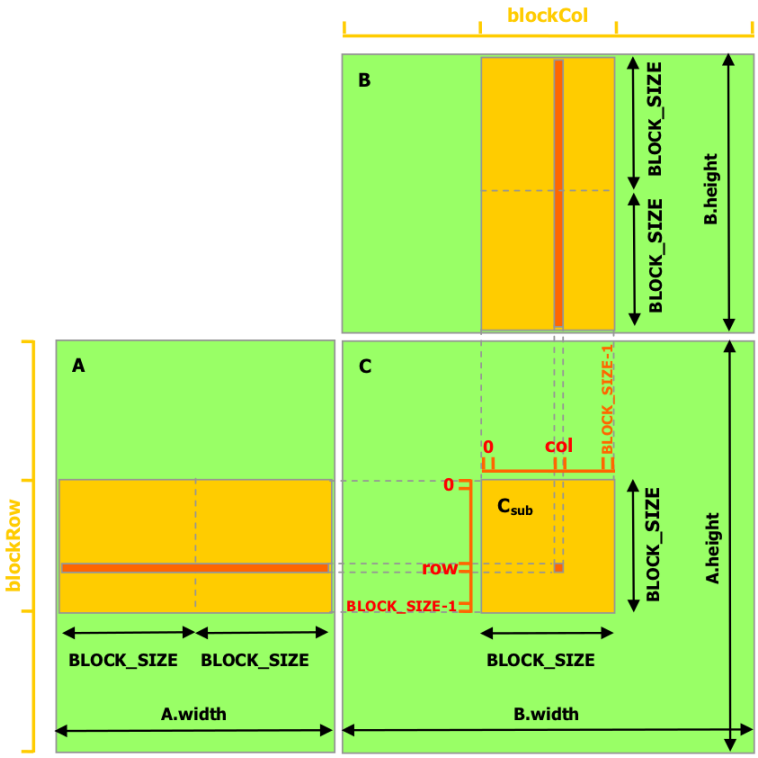


Abbildung 5.5: Veranschaulichung der “kachelweisen” Matrixmultiplikation. Entnommen aus [3].

ment der Ergebnismatrix, welches die Threads berechnen sollen. Abbildung 5.5 veranschaulicht die in dieser Kernelfunktion durchgeführte kachelweise Matrixmultiplikation. Zum besseren Verständnis soll zunächst eine einfache Version einer parallelen Matrix-Multiplikation $AB = C, A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times p}$ gezeigt werden. Dabei bildet der Thread, welcher für das Element ij der Ergebnismatrix C zuständig ist, das Skalarprodukt des Vektors, den die i-te Spalte von A darstellt, mit dem Vektor, welchen die j-te Spalte von B repräsentiert, in einer for-Schleife. Der Nachteil dabei ist, dass dafür jede Zeile von A und jede Spalte von B n-mal aus dem globalen Speicher geladen werden muss. Der *shared-memory*-Bereich wird nicht genutzt.
Auch bei der kachelweisen Matrixmultiplikation erfüllen die einzelnen Threads diese Aufgabe. Sie sind jedoch zu Blocks zusammengefasst, um den *shared-memory*-Bereich nutzen zu können. Der Block mit den Blockkoordinaten kl ist dabei für die Kachel der Ergebnismatrix C mit diesen Koordinaten zuständig. Dieser Block multipliziert die Vektoren der k-ten Kachelreihe von A mit denen der l-ten Kachelspalte von B, wozu man sich auch für die Matrizen A und B eine Einteilung in Kacheln gleicher Größe denken muss, siehe Abbildung 5.5. Die Threads innerhalb des Blocks berechnen dazu zunächst das Ergebnis der Teilmatrixmultiplikation $A_{[k][1]}B_{[1][l]}$, wobei $A_{[x][y]}$ die Kachel einer Matrix A mit den Koordinaten x, y sei. Die Ergebnisse der jeweils nächsten Teilmatrixmultiplikation werden direkt auf die Ergebnisse der jeweils vorherigen addiert. Die einzelnen Threads führen dabei eine verschachtelte for-Schleife aus. Die äußere iteriert über die

Kacheln, die innere über die Elemente der jeweiligen Kachel. Jeder Thread lädt innerhalb der äußeren for-Schleife jeweils das Element der Kacheln von A und B in den Speicher welches seiner zweidimesionalen Thread-ID entspricht. Damit wird jedes Element der Kacheln nur einmal aus dem globalen Speicher geladen, obwohl der Kachelbreite entsprechend viele Zugriffe auf dieses geschehen. Das reduziert die Anzahl der benötigten globalen Speicherzugriffe gegenüber der einfachen Version um einen Faktor, welcher der Kachelbreite entspricht.
Falls die Zeilenlänge von A bzw. die Spaltenlänge von B kein Vielfaches der Kachelbreite ist, werden die Matrizen mit weiteren Spalten bzw. Reihen aufgefüllt, die nur Nullen enthalten, sodass die Matrizen sich vollständig in Kacheln dieser Breite einteilen lassen. Dies beeinflusst das Ergebnis der Berechnung nicht und vermeidet if-Verzweigungen zur Identifikation der Threads, welche für die unvollständigen Kacheln zuständig wären. Bei diesen if-Verzweigungen folgten Threads innerhalb eines Warps verschiedenen Zweigen, was zu einer Laufzeitverschlechterung führte.

Host Code (zweiter Teil): Danach führt die Implementierung die Berechnung der Eigenvektoren mittels der Potenziteration durch, wie in Abschnitt 2.4 beschrieben.

Vergleich mit der CPU-Version: Die CPU-Implementierung von Pivot-MDS nutzt die Symmetrie von C^TC aus und berechnet jedes Element nur einmal, womit sich die Laufzeit ca. um die Hälfte reduziert. Die GPU-Implementierung berechnet die Werte doppelt, da das Berechnen einer Dreiecksmatrix nicht gut auf das Gridschema passt.
Folgende Laufzeitmessungen wurden auf dem System bestehend aus Intel Core 2 Duo E6650 @ 2,3 GHz CPU und Nvidia GTS 250 GPU durchgeführt. Pivot-MDS benötigt auf der CPU beim Graphen 4elt eine Laufzeit (ohne Distanzberechnung) von 0,17 Sekunden, für fe_ocean benötigt sie 5,34 Sekunden. Die Laufzeit der CUDA-Implementierung dafür beträgt 0,067 bzw. 0,34 Sekunden. Bei 4elt ist das eine Beschleunigung um Faktor 5, bei fe_ocean um Faktor 15. Je größer der Graph ist, desto größer fallt der Geschwindigkeitsvorteil der GPU-Version gegenüber der CPU-Version[21] aus. Dies war zu erwarten, da, wie oben beschrieben, die Multiplikation C^TC den größten Teil der konkreten Laufzeit von Pivot-MDS ausmacht.

Instruktionsdurchsatz: Die Implementierung der kachelweisen Matrixmultiplikation enthält keine divergenten Verzweigungen, welche den Instruktionsdurchsatz verschlechtern würden.

Speicherdurchsatz: Die kachelweise Matrixmultiplikation sorgt für einen guten Speicherdurchsatz. Die oben erwähnte einfache Implementierung benötigt für 4elt eine Laufzeit von 0,14, für fe_ocean von 0,99 Sekunden.

5.4 Ausgedünnte Stressmajorisierung

Host Code: Die Implementierung nutzt den maximalen Parallelisierungsgrad, den Abschnitt 3.3 beschreibt. Für die Berechnung jeder Wunschposition eines jeden aktiven Knotens für jeweils

[21] Dieses Verhalten wurde mit weiteren Graphen getestet.

alle anderen Knoten startet der Host Code einen eigenen Thread. Alle Threads, welche die Wunschpositionen der aktiven Knoten für einen bestimmten Knoten berechnen, sind zu einem Threadblock zusammengefasst. Für jede Kombination von Elementen der gesamten Knotenmenge und den aktiven Knoten existiert daher ein Thread; zu jedem Knoten gehört ein Threadblock. Diese Gruppierung ist sinnvoll, da so nur Threads innerhalb eines Blocks die neue Position eines Knotens berechnen und sie damit den schnellen *shared memory*-Bereich effizient nutzen können. Der globale Speicher ist lediglich für das Lesen bzw. Schreiben der oben beschriebenen Datenstrukturen nötig, jedoch nicht für das Speichern von Zwischenergebnissen. Das Grid ist zweidimensional organisiert. Dies liegt nicht daran, dass die Berechnung in dieser Art organisiert ist, sondern lediglich an der Tatsache, dass die Anzahl an Blöcken in einer Dimension auf 65536 beschränkt ist und ein eindimensionales Grid bei diesem Parallelisierungsgrad die Anzahl der Knoten im Graphen auf diese Zahl beschränken würde.

Device Code: Jeder Thread führt, wie eben beschrieben, die entsprechende Kernelfunktion (Device Code) aus. In der Kernelfunktion der ausgedünnten Stressmajorisierung bestimmen die Threads nicht ihre eindeutige ID. Block- und Thread-ID werden getrennt benutzt, da die Block-ID bestimmt, zu welchem Knoten w die berechnete Wunschposition gehört und die Thread-ID die Nummer des aktiven Knotens ist, welches die Wunschposition angibt. Mit Hilfe der Thread-ID und *pivot_idx* ist es daher möglich, die Knoten-ID des aktiven Knotens v zu bestimmen, für den der jeweilige Thread zuständig ist. In jedem Block kümmern sich die Threads mit den ersten q (Anzahl Pivotknoten) IDs um die Pivotknoten, die Threads mit den nächsten a (Anzahl Knoten in der aktiven Nachbarschaft) kümmern sich um die Knoten der aktiven Nachbarschaft. Da nur ein einziger Thread die Graphdistanz von einem bestimmten v zu einem bestimmten w benötigt, speichert dieser diese Distanz in einer Registervariablen. Gleiches gilt für die ID des behandelten aktiven Knotens. Da alle Threads innerhalb eines Blockes Wunschpositionen für den gleichen Knoten w berechnen, lädt der Thread mit der ID Null dessen Knoten-ID aus dem global memory und speichert sie in einer Variablen im *shared-memory*-Bereich. Für die Wunschpositionen jeder Dimension und für das Gewicht der Wunschpositionen wird jeweils ein Array im *shared memory* angelegt. Erstere initialisiert der jeweilige Thread mit Null, letzteres mit dem Gewicht, was er aus der Distanz berechnet[22]. Nach diesen Schritten ist es nötig, alle Threads eines Blocks zu synchronisieren, da die Threads im weiteren Verlauf auf die Variablen im *shared memory* zugreifen und garantiert sein muss, dass die benötigten Variablen initialisiert sind[23].
Anschließend berechnen die Threads die euklidische Distanz zwischen den ihnen zugeteilten Knoten w und v. Darauf folgt die eigentliche Berechnung der Wunschpositionen, wie in Algorithmus 2.5 beschrieben. Wichtig zu beachten ist, dass in der parallelen Version die Wunschposition nicht direkt aufsummiert, sondern in den oben erwähnten *shared-memory*-Arrays gespeichert werden. Um die Anzahl der sequentiellen Schritte zu reduzieren, die für ein Aufsummieren der

[22] Ob das Berechnen des Gewichts an dieser Stelle oder das Übergeben einer vorberechneten Gewichtsmatrix an die Kernelfunktion effizienter ist, hängt von der verwendeten Funktion ab, die aus der Distanz eine Gewichtung generiert. In diesem Fall wird $\omega = 1/d$ angenommen.

[23] Anderenfalls könnte es passieren, dass ein Thread k, welcher die Speicherstelle *array[k]* im *shared memory* initialisieren soll, vom Scheduler noch nicht aktiviert wurde, während ein weiterer Thread l bereits auf die Stelle *array[k]* zugreift (race condition).

Wunschpositionen und der Gewichtungen nötig sind, benutzt die Implementierung die *sum reduce*-Methode. Abbildung 5.6 veranschaulicht dieses Verfahren.

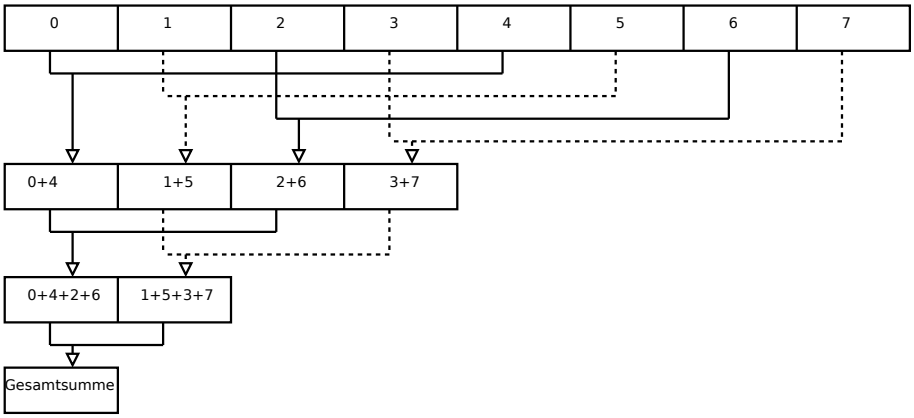


Abbildung 5.6: Veranschaulichung der *sum reduce*-Methode.

In diesem Verfahren addiert jeder Thread j mit Thread-ID kleiner $k/2$, wobei k die Anzahl Elemente im Array sei, den Wert an Arrayposition j mit dem Wert an Position $j + k/2$. Dies wird so lange wiederholt, bis das Ergebnis an der nullten Arrayposition steht. Da sich die Länge des Arrays bei jedem Schritt halbiert, reduziert sich die Anzahl der benötigten sequentiellen Schritte von k auf $log(k)$. Die Implementierung kann nur mit Arraylängen, die Zweierpotenzen entsprechen, umgehen. Da es aus in diesem Abschnitt beschriebenen Gründen nicht sinnvoll ist, andere Anzahlen von aktiven Knoten zuzulassen, spielt diese Einschränkung hier keine Rolle. Im Allgemeinen ist dieses Verfahren auch für beliebige Arraylängen benutzbar. Das Array muss dafür bis zur nächsten Zweierpotenzgrenze mit Nullen aufgefüllt werden[24]. Vor jedem der sequenziellen Schritte müssen sich die Threads synchronisieren, damit grantiert ist, dass alle Zwischenergebnisse an der erwarteten Stelle stehen. Zum Schluss berechnet der Thread mit der ID Null aus der Summe der Wunschpositionen und dem Gesamtgewicht die neue Position des Knotens, für welchen sein Block zuständig ist, und schreibt das Ergebnis zurück in den globalen Speicher.

Vergleich mit der CPU-Implementierung: Folgende Laufzeitmessungen wurden auf dem System bestehend aus Intel Core 2 Duo E6650 @ 2,3 GHz CPU und Nvidia GTS 250 GPU durchgeführt. Für die Laufzeitmessungen führt die Stressmajorisierung jeweils eine konstante Anzahl von 100 Iterationen aus. Die ausgedünnte Stressmajorisierung benötigt auf der CPU beim Graphen 4elt eine Laufzeit (ohne Distanzberechnung) von 5,6 Sekunden, für fe_ocean benötigt sie 60 Sekunden. Die Laufzeit der CUDA-Implementierung dafür beträgt 0,3 bzw. 3,2 Sekunden. Das ist jeweils eine Beschleunigung um mehr als Faktor 18.

Belegungsgrad: Der Belegungsgrad ist nahe am Optimum, wie die Ausgabe des Profilers beim

[24] Man könnte dies auch mit if-Verzweigungen lösen. Der Ansatz mit zusätzlichen Nullen ist jedoch effizienter, da er divergente Verzweigungen vermeidet.

Ausführen des Programms auf dem Graphen 4elt zeigt:

```
Kernel details: Grid size: [64 244 1], Block size: [128 1 1]
Register Ratio: 0.9375 ( 7680 / 8192 ) [10 registers per thread]
Shared Memory Ratio: 0.75 ( 12288 / 16384 ) [1616 bytes per Block]
Active Blocks per SM: 6 (Maximum Active Blocks per SM: 8)
Active threads per SM: 768 (Maximum Active threads per SM: 768)
Potential Occupancy: 1 ( 24 / 24 )
Achieved occupancy: 1 (on 16 SMs)
Occupancy limiting factor: None
```

In diesem Beispiel ist die Größe des Grids 64 in der x-Dimension und 244 in der y-Dimension. In der Implementierung ist die Blockanzahl der x-Dimension auf 64 festgesetzt. Die Größe der y-Dimension ist von der Knotenanzahl des Graphen abhängig und beträgt $\frac{64}{n} + 1$. Bei der Wahl der Anzahl der Blocks in der y-Dimension ist ein Kompromiss einzugehen. Je größer die Anzahl ausfällt, desto größere Graphen kann die Implementierung behandeln, desto größer wird jedoch auch die Anzahl der möglichen inaktiven Threads. Im Beispiel sieht man, dass für 4elt $64 * 244 = 15616$ Threads gestartet wurden. Der Graph hat jedoch nur 15606 Knoten, womit 10 Threads nichts tun. Der Laufzeitverlust dadurch ist sehr gering. Die Blockgröße ist wie erwartet 128, was der Anzahl der aktiven Knoten entspricht.
Weiter sieht man an der Ausgabe, dass die Registerplätze fast vollständig genutzt wurden. Es wäre möglich, ein weiteres *shared-memory*-Array zu nutzen, was in der Implementierung jedoch nicht nötig ist. Sämtliche SMs sind voll ausgelastet und bei großen Graphen werden genug Threads gestartet, um die Speicherlatenz zu verstecken. Würde man eine andere Anzahl aktiver Knoten wählen, so wäre der Belegungsgrad in den meisten Fällen schlechter.

Instruktionsdurchsatz: Der Instruktionsdurchsatz des Kernels ist lediglich 0,5. Dies liegt daran, dass der Speicher der Flaschenhals ist und wird daher im Rahmen des Speicherdurchsatzes erklärt. Um den Instruktionsdurchsatz zu erhöhen, vermeidet die Implementierung Divergenzen innerhalb eines Warps bei der Entscheidung, ob ein Thread für Pivotknoten oder für die aktive Nachbarschaft zuständig ist dadurch, dass, wie oben in diesem Abschnitt beschrieben, die ersten q Threads sich um die Pivotknoten und die nächsten a Threads sich um die aktive Nachbarschaft kümmern. Dabei ist es wichtig, dass q und a als Vielfache von 32, der Anzahl an Threads pro Warp, gewählt werden. Dadurch ist garantiert, dass ein Warp jeweils einheitlich einer Verzweigung folgt. Bei den anderen if-Verzweigungen folgt nur der Thread mit der ID Null dem einen Zweig, alle weiteren dem anderen. Hier ist eine divergente Verzweigung nicht zu vermeiden. Sie findet aber nur in dem Warp statt, welches den Thread mit ID Null enthält.
Den Geschwindigkeitsvorteil, der durch das Vermeiden von divergenten Verzweigungen zu erreichen ist, zeigt das folgende Szenario: Jeder Thread soll abwechselnd der anderen Verzweigung folgen. Beim Graphen 4elt führt dies mit den oben beschriebenen Einstellungen zu einer Erhöhung der Laufzeit von 0,3 auf 0,63 Sekunden, bei fe_ocean von 3,2 auf 6,5. Der Instruktionsdurchsatz reduziert sich dadurch ebenfalls um die Hälfte. Dieses Szenario ist ein Extremfall und wird in Anwendungen selten auftreten, zeigt jedoch gut, welchen Effekt divergente Verzweigungen haben können.

Durch den hohen Parallelisierungsgrad und die dadurch bedingte hohe Threadanzahl kann die Grafikkarte die Speicherlatenz bei größeren Graphen gut verstecken. Die Threadanzahl wirkt sich daher nicht limitierend auf den Instruktionsdurchsatz aus.

Speicherdurchsatz: Es wurde darauf geachtet, die Registerplätze sowie den *shared-memory*-Bereich so gut wie möglich zu nutzen (s. Beschreibung oben in diesem Abschnitt). Lädt man Knoten *w* nicht in den *shared-memory*-Bereich und Knoten *v* nicht in ein Register und lädt für jeden Zugriff die entsprechenden Daten aus dem globalen Speicher, so erhöht sich die Laufzeit bei 4elt von 0,3 auf 1,13 Sekunden und bei fe_ocean von 3,2 auf 11,7 Sekunden.
In der Implementierung sind die Speicherzugriffe auf den globalen Speicher, welche die aktiven Knoten in den *shared-memory*-Bereich laden, nicht zusammenhängend, da die Knoten-IDs von aufeinander folgenden aktiven Knoten nicht aufeinander folgend sein müssen.[25] Für die Pivotknoten könnte man das Problem lösen, indem man die Daten der Pivotknoten der Reihe nach am Anfang des *nodes*-Arrays speichert und in einem weiteren Array die Abbildung der Postion in *nodes* auf die Spalten der Distanzmatrix speichert. Da jeder Knoten jedoch eine andere aktive Nachbarschaft hat, ist diese Vorgehensweise dafür nicht möglich. An dieser Stelle könnte man eine weitere deutliche Laufzeitsteigerung der Implementierung erreichen. Um wie viel sich die Laufzeit steigern ließe, wenn zusammenhängende Speicherzugriffe möglich wären, lässt sich zeigen, indem auf einander folgende Knoten aus *nodes* als aktive Knoten verwendet werden[26]. Die Laufzeit verringert sich bei 4elt auf 0,13 Sekunden und bei fe_ocean auf 1,18 Sekunden.[27]
Sämtliche Zugriffe auf *dist_matrix_p*, *dist_matrix_n*, *pivot_idx* und *neigh_idx* sind zusammenhängend. Damit dies möglich ist, haben die Matrizen den oben beschriebenen Aufbau, bei dem die zu einem Knoten gehörigen Distanzen sich in einer Zeile befinden und in row-major Form im Speicher liegen[28]. Dieser Aufbau garantiert zusammenhängende Speicherzugriffe, da ein Block einen Knoten bearbeitet und daher auf die gleiche Zeile zugreift. Die Threads behandeln aufeinanderfolgende aktive Knoten und damit aufeinanderfolgende Spalten der Matrix, womit sie auf zusammenhängende Speicherbereiche zugreifen. Da die Threads, welche die Pivotelemente bearbeiten, die Pivotelemente in der Reihenfolge, in der sie bestimmt wurden, behandeln, sind die Speicherzugriffe auf *pivot_idx* ebenfalls zusammenhängend.
Eine Version, welche keine zusammenhängenden Speicherzugriffe zulässt, soll deren Geschwindigkeitsvorteil demonstrieren. Dazu wurden *dist_matrix_p* und *dist_matrix_n* vor der Anwendung der ausgedünnten Stressmajorisierung transponiert[29], sodass eine Spalte nun zu einem Knoten gehört. Dies verlängert die Laufzeit von 4elt 0,3 auf 0,74 Sekunden, bei bei fe_ocean von 3,2

[25] Aus diesem Grund ist die Laufzeitzunahme bei dem Beispiel, welches bei jedem Benutzen der Knotendaten diese aus dem globalen Speicher lädt, so hoch.

[26] Dies ergibt in Bezug auf das Ergebnis keinen Sinn und soll lediglich der Laufzeitdemonstration gelten.

[27] Für dieses Beispiel war es auf der GTS 250 nötig, das *nodes*-Array durch zwei getrennte Arrays für die *x*- und *y*-Position zu ersetzen, da anderenfalls kein zusammenhängender Speicherzugriff möglich war. Möglicherweise ließ das Speicherlayout der nodes structs keine zusammenhängenden Zugriffe zu. Aus diesem Grund sollte in einer späteren Version das *nodes*-Array auf diese Weise ersetzt werden.

[28] Würde man dies genau umgekehrt durchführen, also jedem Knoten eine Spalte zuordnen und in column-major Form zugreifen, so wären zusammenhängende Speicherzugriffe ebenfalls möglich.

[29] Die Laufzeit für das Transponieren wurde bei den Vergleichen nicht dazu gezählt. Die Laufzeitverschlechterung liegt allein daran, dass die Speicherzugriffe nicht zusammenhängend sind.

auf 8 Sekunden. Falls man weiter *neigh_idx* transponiert, verschlechtert sich die Laufzeit weiter bei 4elt auf 1,065, bei fe_ocean auf 10,8 Sekunden. Wären die Zugriffe auf *pivot_idx* ebenfalls nicht zusammenhängend, so würde die zu einer weiteren vergleichbaren Laufzeitverschlechterung führen, da auf diese Datenstrukturen jeweils gleich oft zugegriffen wird.

5.5 Layouts und Laufzeiten

Dieser Abschnitt zeigt von der kombinierten MDS generierte Layouts und die dafür benötigten Laufzeiten. Er vergleicht außerdem die Ergebnisqualität verschiedener vorgestellter Methoden. Ein Vergleich der Ergebnisse der kombinierten MDS mit anderen Verfahren zeigt Abschnitt 6. Die Layouts und Laufzeiten werden anhand der oben eingeführten Beispielgraphen gezeigt. Die gewählte Standardkonfiguration ist:

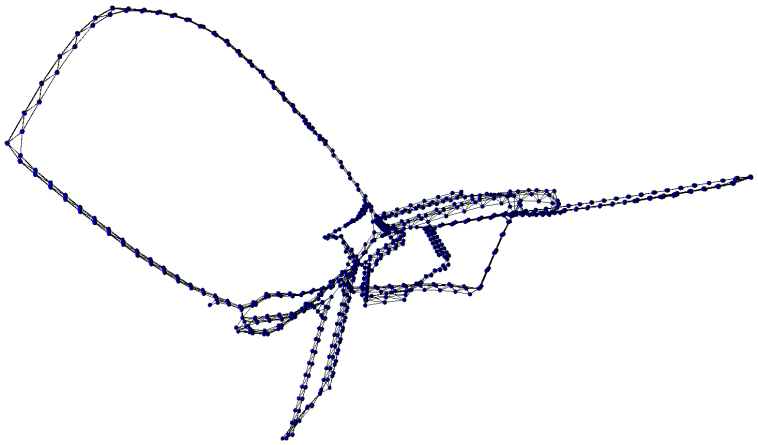
- *Anzahl aktiver Knoten:* Die Standardkonfiguration benutzt 64 Pivotknoten und jeweils 64 Knoten in den aktiven Nachbarschaften.
- *Distanzberechnung:* Für Graphen bis zu einer Größe von 62000 berechnet die CPU die Distanzen, sowohl zu den Pivotknoten, als auch zur aktiven Nachbarschaft, für größere Graphen die GPU. Hier wäre es für eine zukünftige Version besser, eine Heuristik in Abhängigkeit des Verhältnisses von Knotenzahl zum Durchmesser des Graphen zu wählen.
- *Pivotauswahl und Gewichtung:* Graphen, deren Distanzen die CPU berechnet, benutzen die min-max-Pivotmethode, während dagegen Graphen, deren Distanzen die GPU berechnet, die Zufallswahl verwenden. Bei der min-max-Pivotauswahl wird ein Pivotknoten p entsprechend der Anzahl der Knoten, die p als nächsten Pivotknoten haben, gewichtet. Bei der Zufallswahl sind sie mit $\frac{n}{q}$ gewichtet.
- *Stressmajorisierung:* Die Anzahl der Iterationen wird in Abhängigkeit der während Pivot-MDS berechneten Eigenwerte bestimmt, wie in Abschnitt 3.2 gezeigt. Sie beträgt mindestens 20. Für die Gewichtungsfaktoren gilt $\omega_{ij} = \frac{1}{d_{ij}}$.

Tabelle 5.4 zeigt die Gesamtlaufzeiten für die eben beschriebene Konfiguration, zum Vergleich die Laufzeit, welche die reine CPU-Implementierung benötigt und die Anzahl der Iterationen, welche die Stressmajorisierung ausführt. Hinter der GPU-Laufzeit stehen in Klammern die Zeiten, welche für die Teile der kombinierten MDS benötigt wurden (<Distanzbestimmung> | <Pivot-MDS> | <Stressmajorisierung>). Die Laufzeitmessungen wurden auf dem System bestehend aus AMD Phenom II X4 850 @ 3,3 GHz CPU und Nvidia GTX 550 Ti GPU durchgeführt.

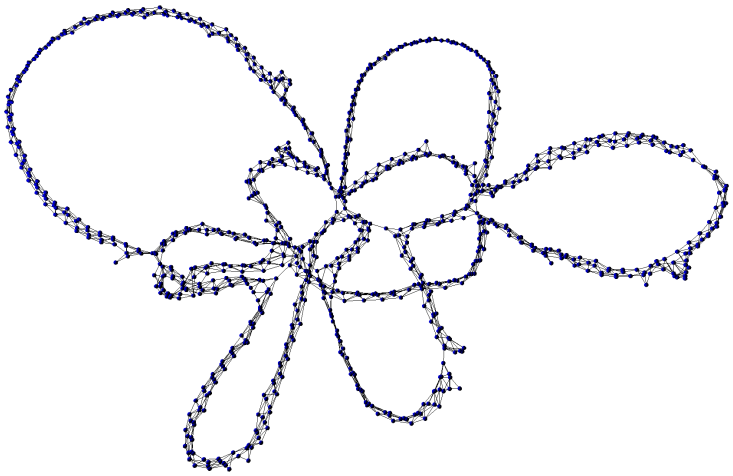
Graph	**#Knoten**	**#Kanten**	GPU [s]		#Iter.	CPU [s]
smallworld	1000	3000	0.0828	(0.0026 | 0.0821 | 0.0047)	20	0.0533
data	2851	15093	0.1091	(0.0124 | 0.0816 | 0.0131)	20	0.2274
3elt	4720	13722	0.1220	(0.0199 | 0.0832 | 0.0211)	20	0.3956
uk	4824	6837	0.1258	(0.0183 | 0.0814 | 0.0214)	20	0.3936
add32	4960	9462	0.1249	(0.0182 | 0.0775 | 0.0216)	20	0.4089
bcsstk33	8738	291583	0.2355	(0.0936 | 0.0942 | 0.0373)	20	0.7905
flower_050	9030	131241	0.3293	(0.0649 | 0.0869 | 0.1754)	99	1.9972
grid_rnd_100	9497	17849	0.1763	(0.0357 | 0.0876 | 0.0400)	20	0.7911
snowflake_C	9701	9700	0.2966	(0.0265 | 0.0983 | 0.1717)	89	1.9231
whitaker3	9800	28989	0.1945	(0.0454 | 0.1071 | 0.0419)	20	0.8300
sierpinski_08	9843	19683	0.1713	(0.0384 | 0.0881 | 0.0448)	20	0.8277
spider_C	10000	22000	0.3093	(0.0338 | 0.0976 | 0.1778)	92	2.3392
crack	10240	30380	0.1910	(0.0522 | 0.0941 | 0.0446)	20	0.8833
4elt	15606	45878	0.2609	(0.0953 | 0.0988 | 0.0668)	20	1.3513
bcsstk31	35588	572914	0.6378	(0.3106 | 0.1569 | 0.1702)	20	3.1752
t60k	60005	89440	0.7752	(0.3358 | 0.1898 | 0.2495)	20	5.1728
wing	62032	121544	1.3778	(0.5859 | 0.1277 | 0.6606)	33	9.2760
598a	110971	741934	2.2089	(1.2613 | 0.2276 | 0.7181)	20	14.2047
fe_ocean	143437	409593	2.0871	(1.1838 | 0.2970 | 0.6062)	20	14.6693
m14b	214765	1679018	4.1723	(2.5480 | 0.4424 | 1.1932)	20	27.3984
auto	448695	3314611	10.8079	(5.2665 | 0.9232 | 4.6410)	30	84.2424

Tabelle 5.4: Gesamtlaufzeiten der kombinierten MDS für CPU und GPU sowie die Anzahl an Iterationen, welche die ausgedünnte Stressmajorisierung ausführte.

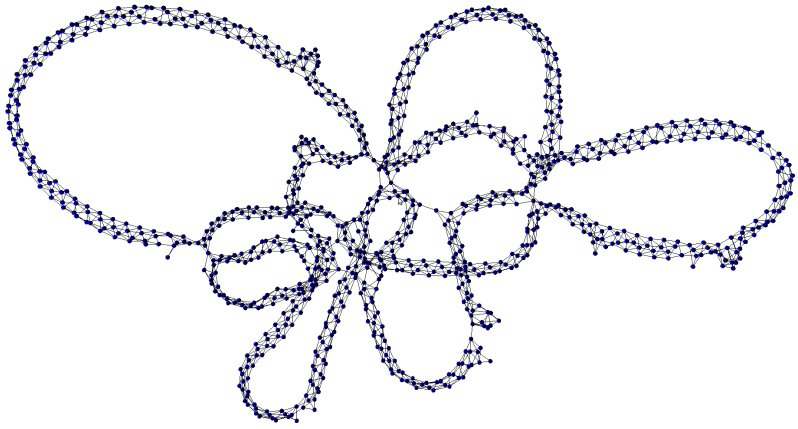
Die Abbildungen 5.7 bis 5.30 zeigen für jeden der Beispielgraphen das von Pivot-MDS generierte Layout, welches als Initiallayout für die ausgedünnte Stressmajorisierung dient und das Ergebnislayout der kombinierten MDS. Für einen Teil dieser Graphen gibt es weitere Layouts, anhand derer Ergebnisse anderer Konfigurationen gezeigt werden.
Die von Pivot-MDS generierten zweidimensionalen Layouts von flower_050, spider_C und snowflake_C sind ungenügend. Da ihre Matrizen C mindestens vier Eigenvektoren signifikanter Größe aufweisen, führt die Stressmajorisierung auf Grund der in Abschnitt 3.2 vorgestellten Abbruchbedingung bei diesen Graphen mehr Iterationen durch (s. Tabelle 5.4).
Bei den meisten Graphen ist jedoch das durch Pivot-MDS generierte Initiallayout sehr gut und nahe am Ergebnislayout der kombinierten MDS. Die Abbildungen der übrigen Graphen zeigen dies.
Falls die ausgedünnte Stressmajorisierung mit einer zufälligen Initialkonfiguration 20 Iteratationen ausführt, ist das Ergebnislayout ungenügend. Nach 100 Iterationen entspricht es in etwa dem von der kombinierten MDS generierten Layout, wofür aber deutlich mehr Laufzeit benötigt wurde. Die Abbildungen 5.21, 5.23 und 5.28 zeigen dies und veranschaulichen damit den großen Nutzen des durch Pivot-MDS generieren Initiallayouts. Weiter lässt sich an diesen Abbildungen die Abhängigkeit der Stressmajorisierung vom Initiallayout erkennen. Jeweils zwei Layouts dieser Graphen zeigen, dass die ausgedünnte Stressmajorisierung nach 20 Iterationen verschieden nah am gewünschten Ergebnislayout ist.
Für Graphen, deren empfundene Layoutgüte mehr von der in Abschnitt 2.6 beschriebenen lokalen Genauigkeit abhängt, ist eine Gewichtung der Pivotknoten, welche wie die verwendete deren Einfluss verstärkt, nicht optimal, da eine solche Gewichtung eher der globalen Genauigkeit zuträglich ist. Beispiele dafür sind smallworld und data, deren Abbildungen ein Ergbnislayout mit einer Konfiguration ohne Pivotgewichtungen zeigen.



(a) Pivot-MDS

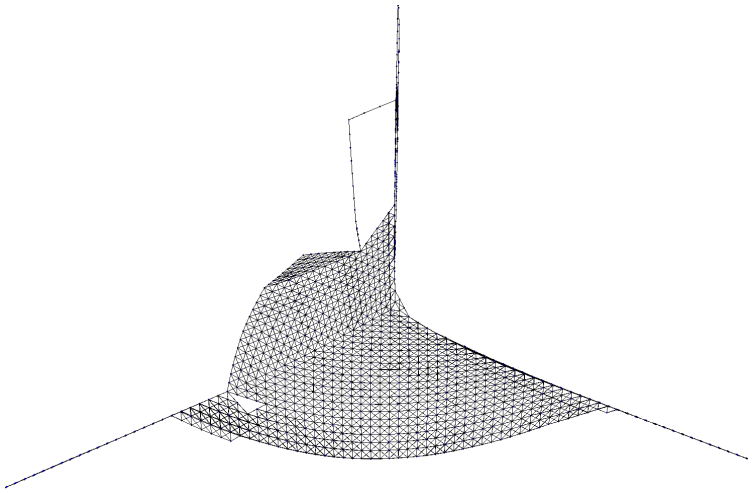


(b) kombinierte MDS

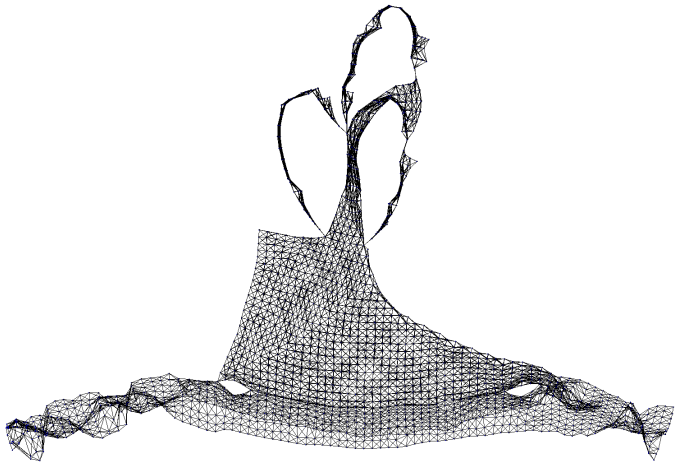


(c) kombinierte MDS mit Pivotgewichtung 1 statt $\frac{n}{q}$

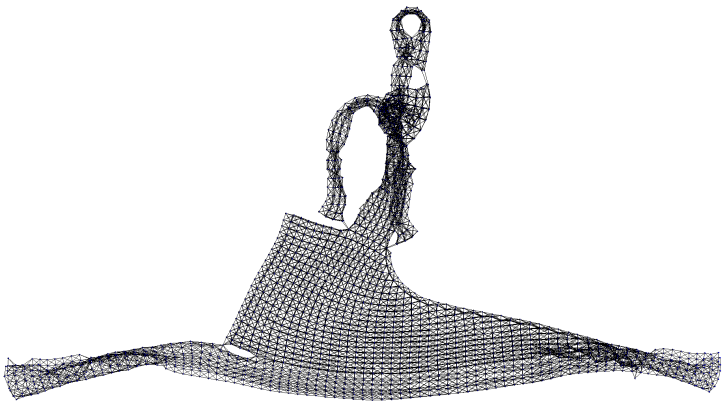
Abbildung 5.7: Layouts für Smallworld. $n = 1000$, $m = 3000$.



(a) Pivot-MDS

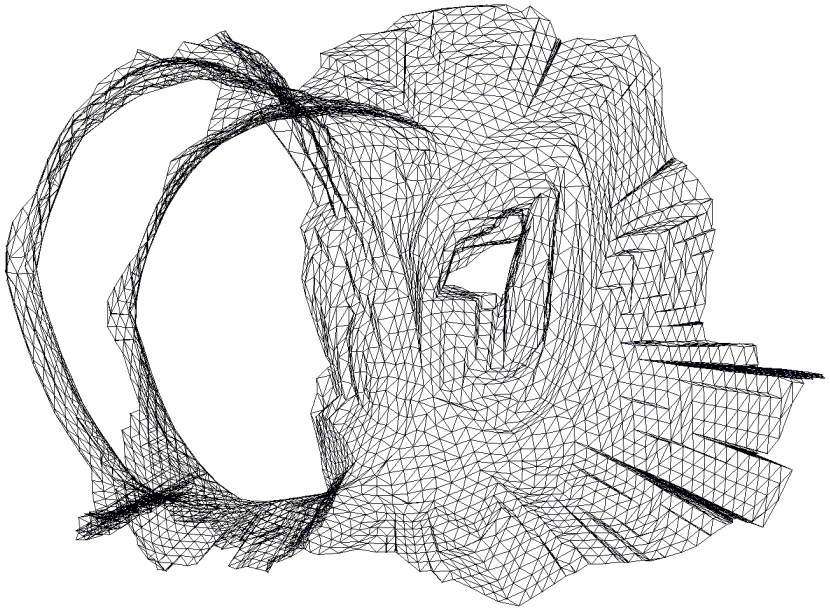


(b) kombinierte MDS

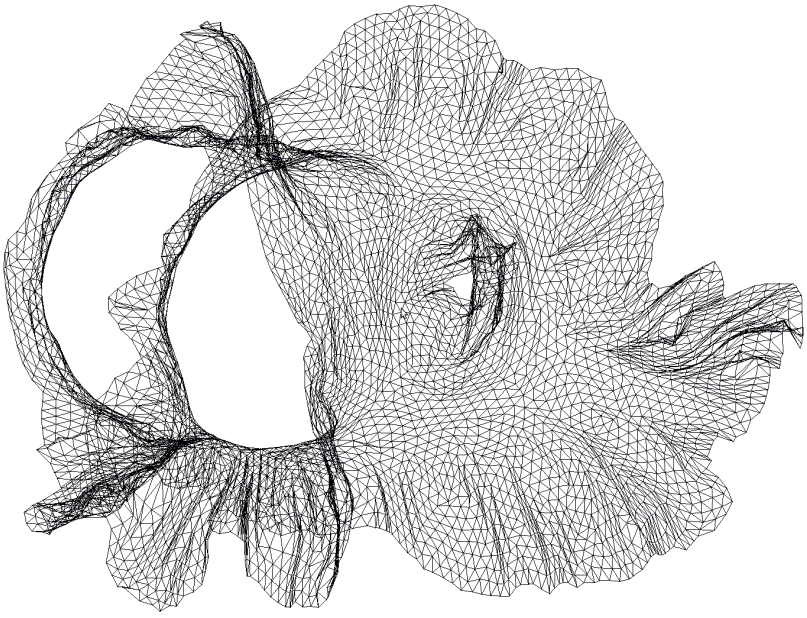


(c) kombinierte MDS mit Pivotgewichtung 1 statt $\frac{n}{q}$

Abbildung 5.8: Layouts für data. $n = 2851$, $m = 15093$.



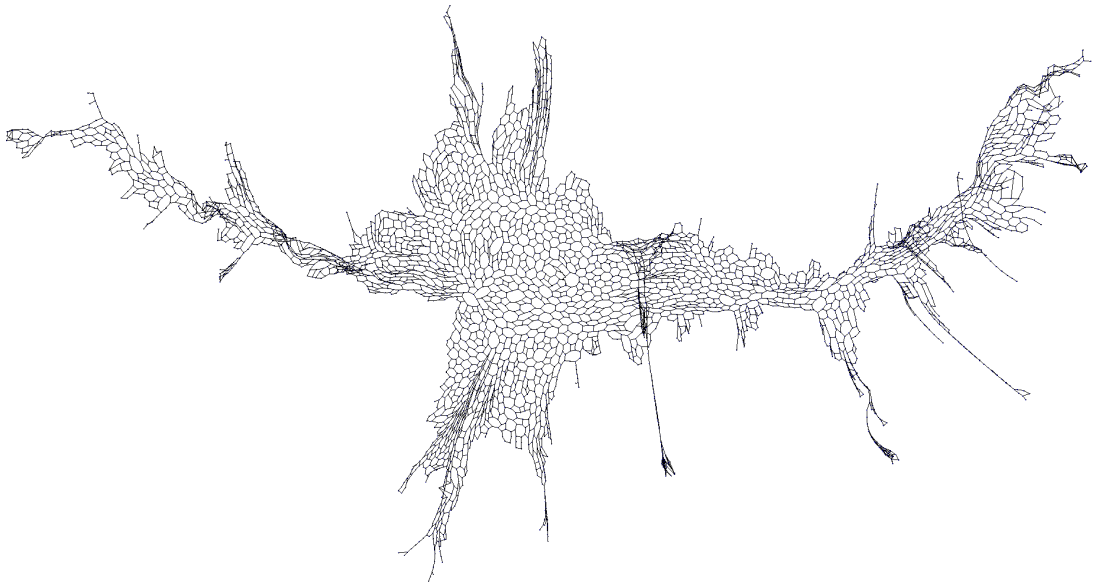
(a) Pivot-MDS



(b) kombinierte MDS

Abbildung 5.9: Layouts für 3elt. $n = 4720$, $m = 13722$.

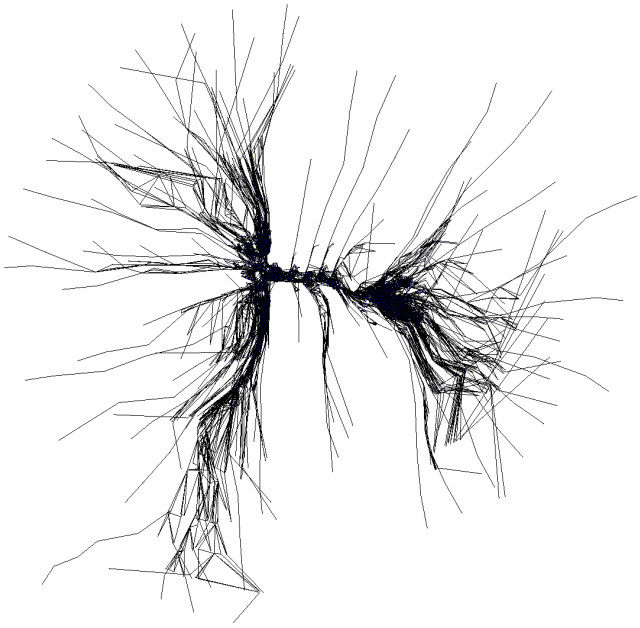
(a) Pivot-MDS



(b) kombinierte MDS

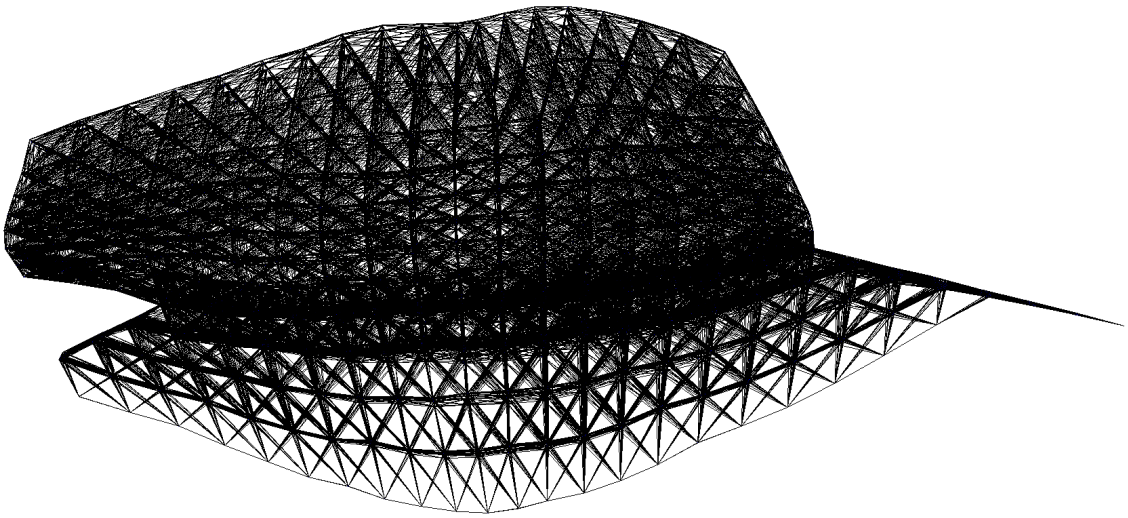
Abbildung 5.10: Layouts für uk. $n = 4824$, $m = 6837$.

(a) Pivot-MDS

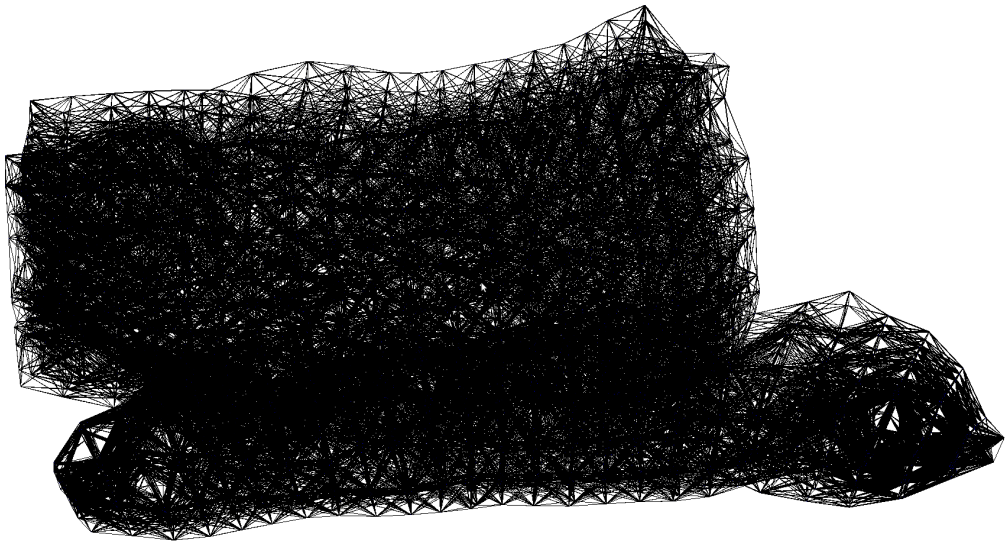


(b) kombinierte MDS

Abbildung 5.11: Layouts für add32. $n = 4940$, $m = 9462$.

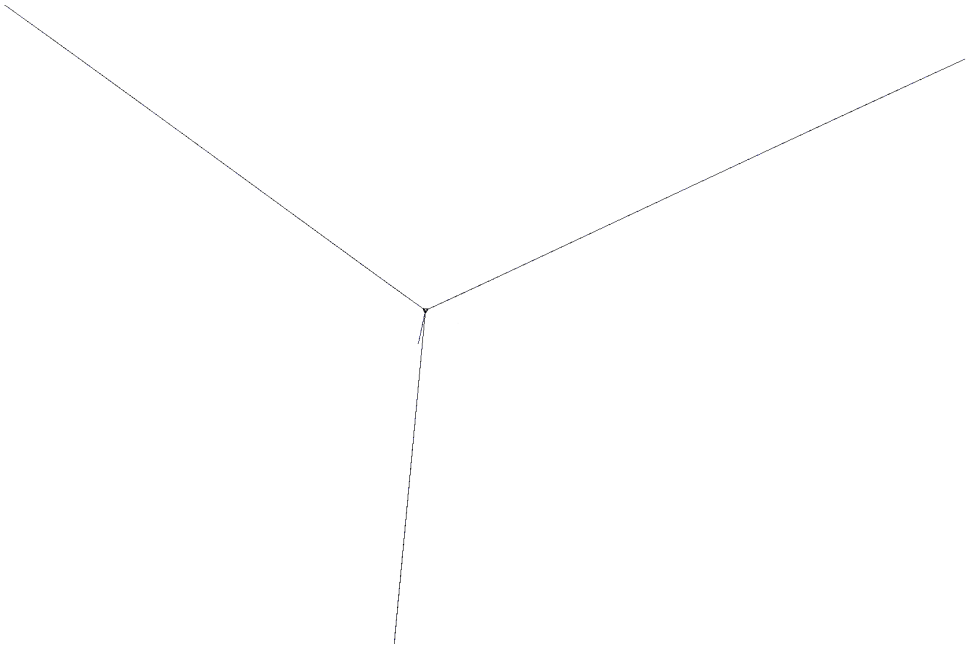


(a) Pivot-MDS

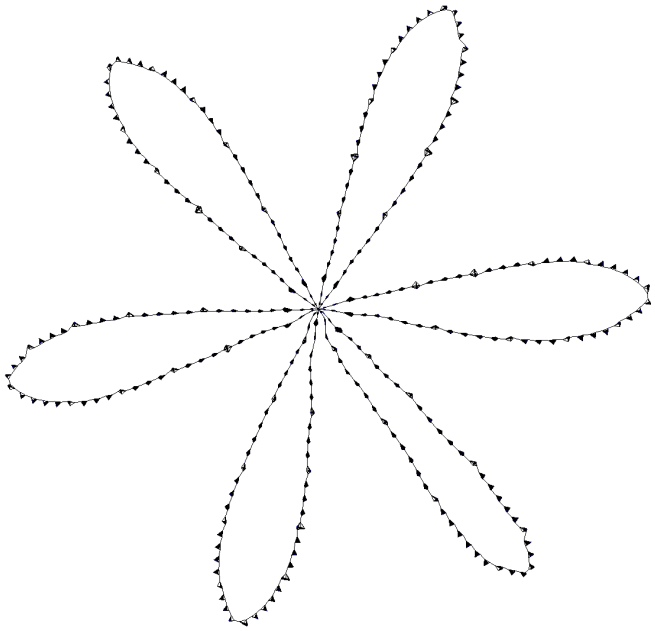


(b) kombinierte MDS

Abbildung 5.12: Layouts für bcsstk33. $n = 8738$, $m = 291583$.

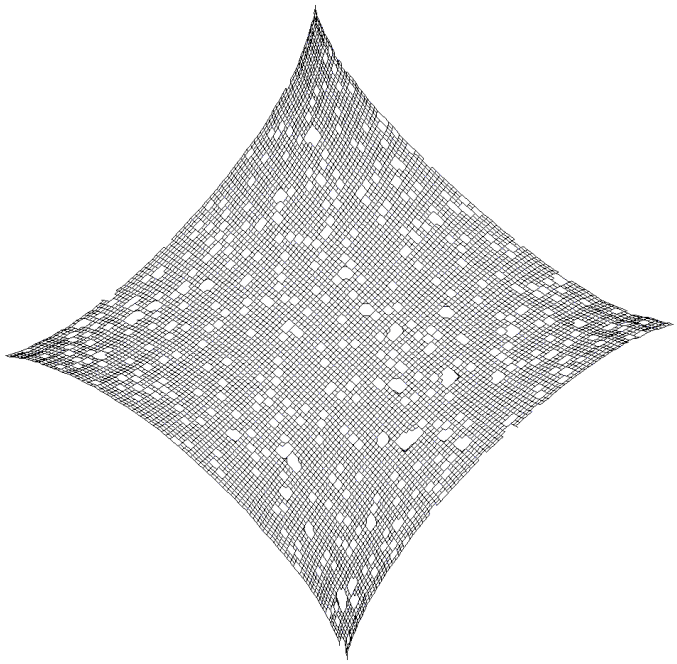


(a) Pivot-MDS

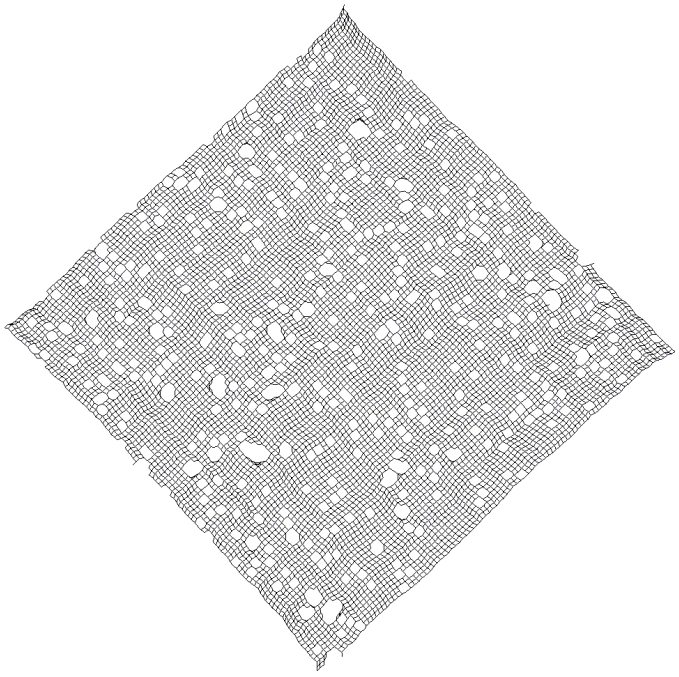


(b) kombinierte MDS

Abbildung 5.13: Layouts für flower_050. $n = 9030$, $m = 131241$.

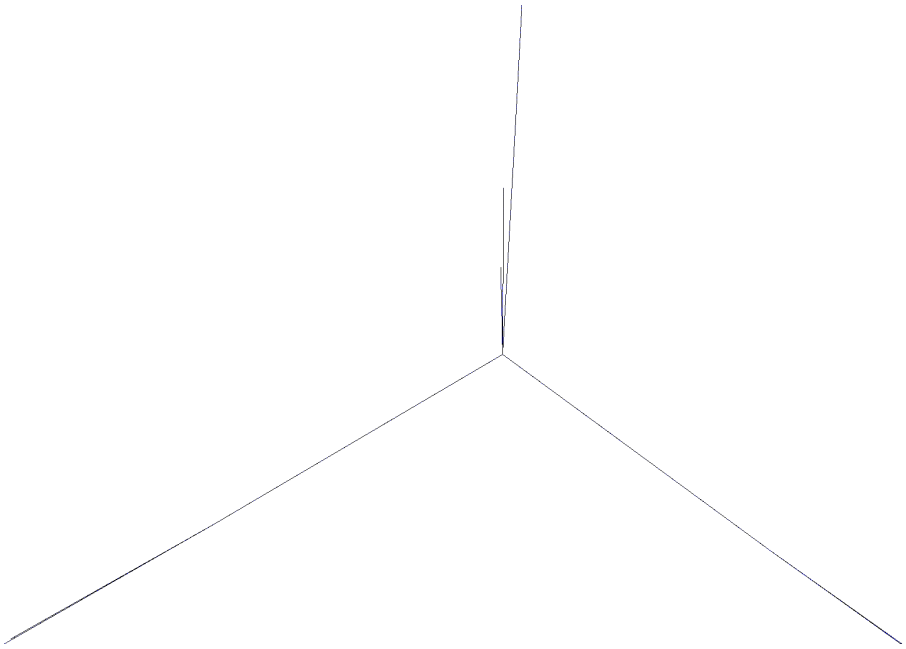


(a) Pivot-MDS

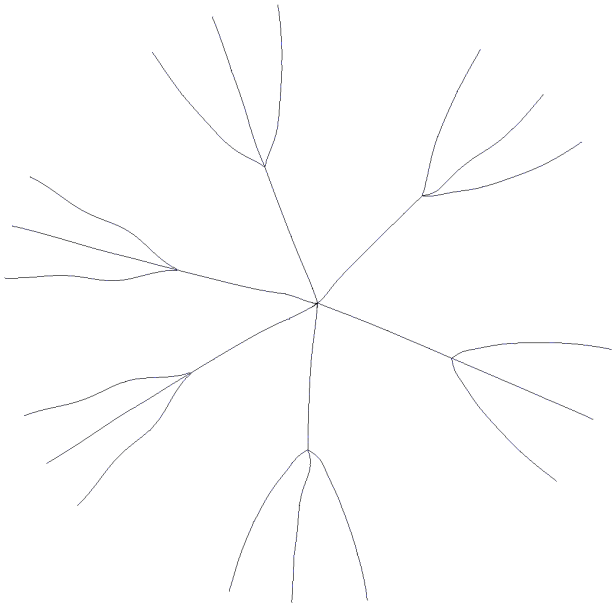


(b) kombinierte MDS

Abbildung 5.14: Layouts für grid_rnd_100. $n = 9497$, $m = 17849$.

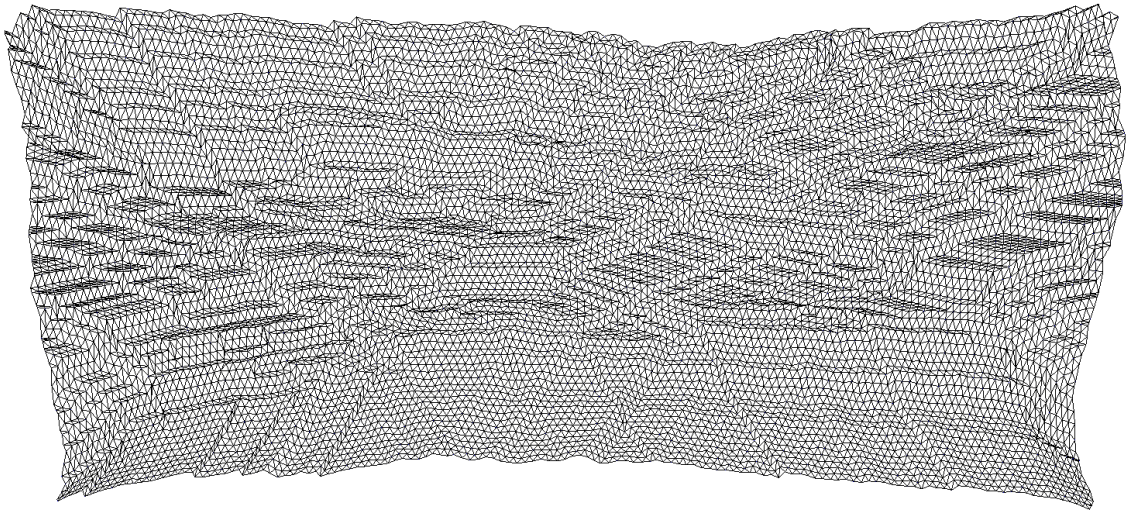


(a) Pivot-MDS

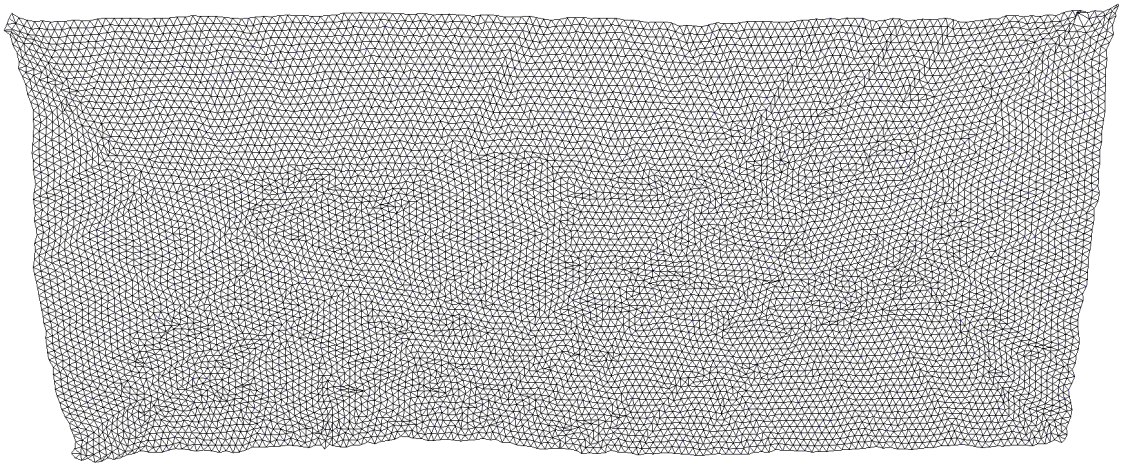


(b) kombinierte MDS

Abbildung 5.15: Layouts für snowflake_C. n = 9701, m = 9700.

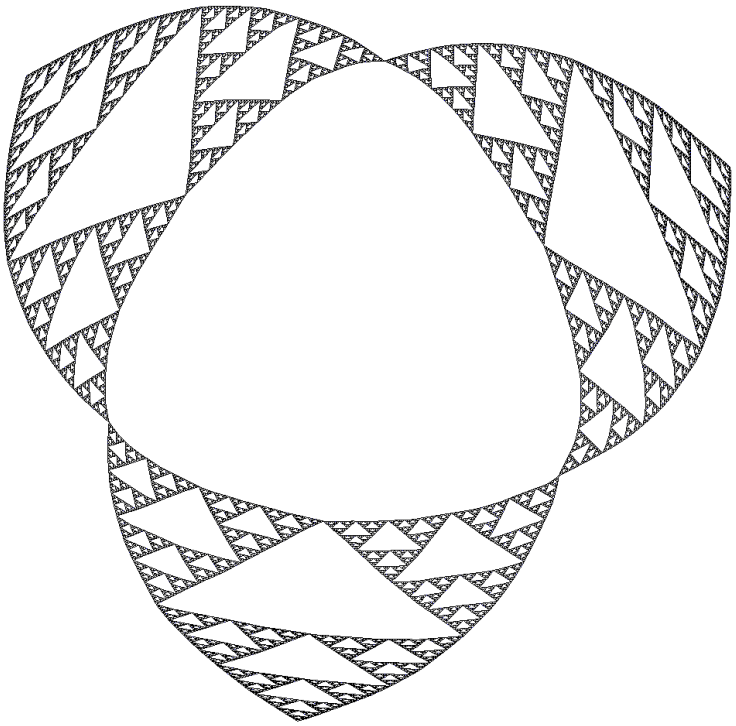


(a) Pivot-MDS

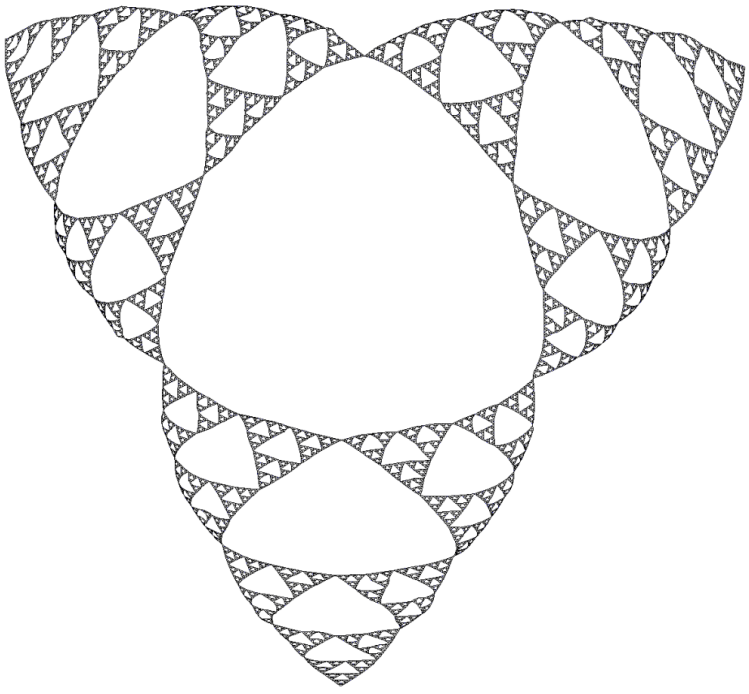


(b) kombinierte MDS

Abbildung 5.16: Layouts für whitaker3. $n = 9800$, $m = 28989$.

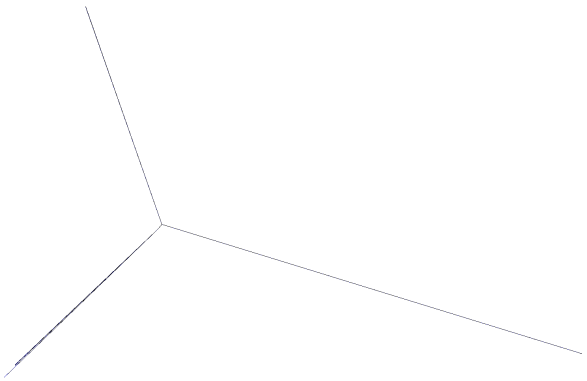


(a) Pivot-MDS

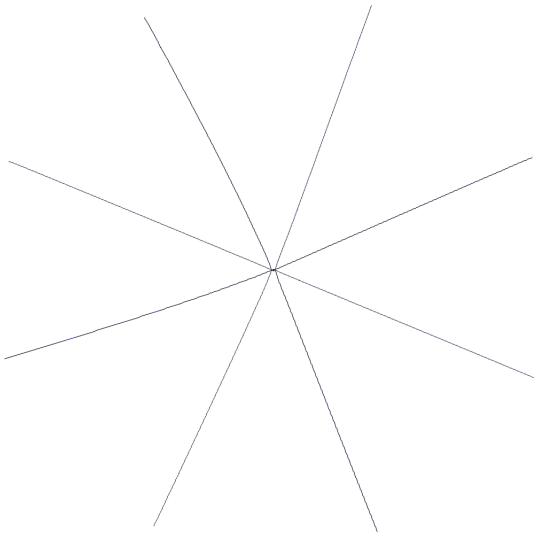


(b) kombinierte MDS

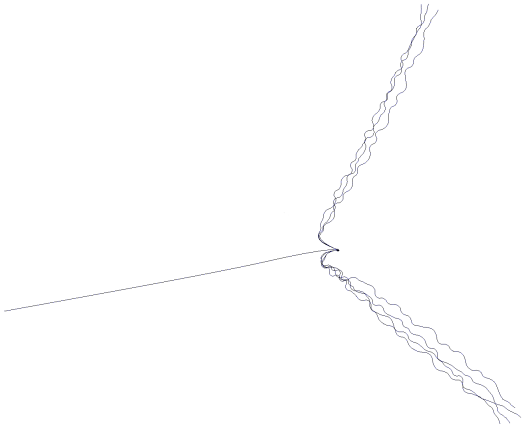
Abbildung 5.17: Layouts für sierpinski_08. $n = 9843$, $m = 19683$.



(a) Pivot-MDS

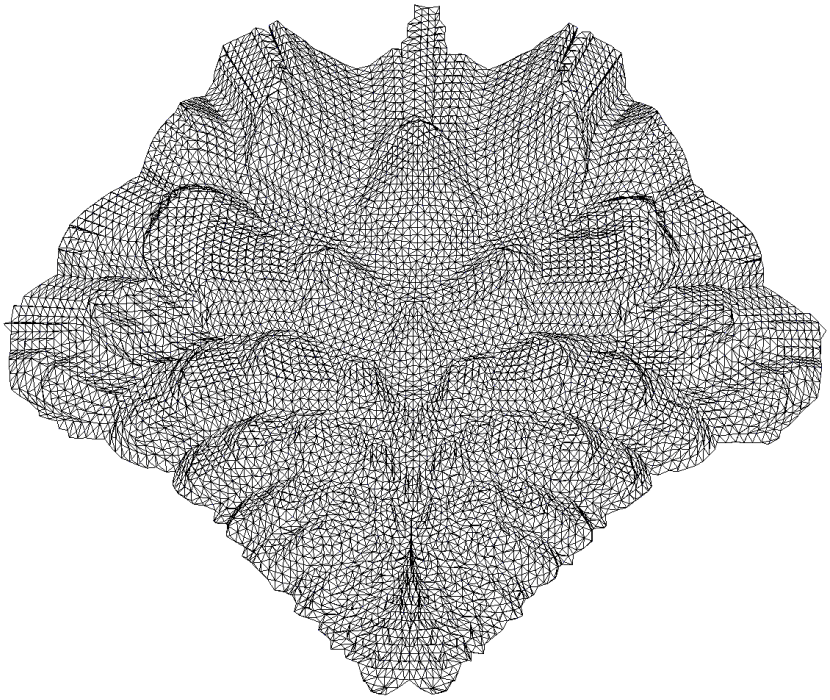


(b) kombinierte MDS

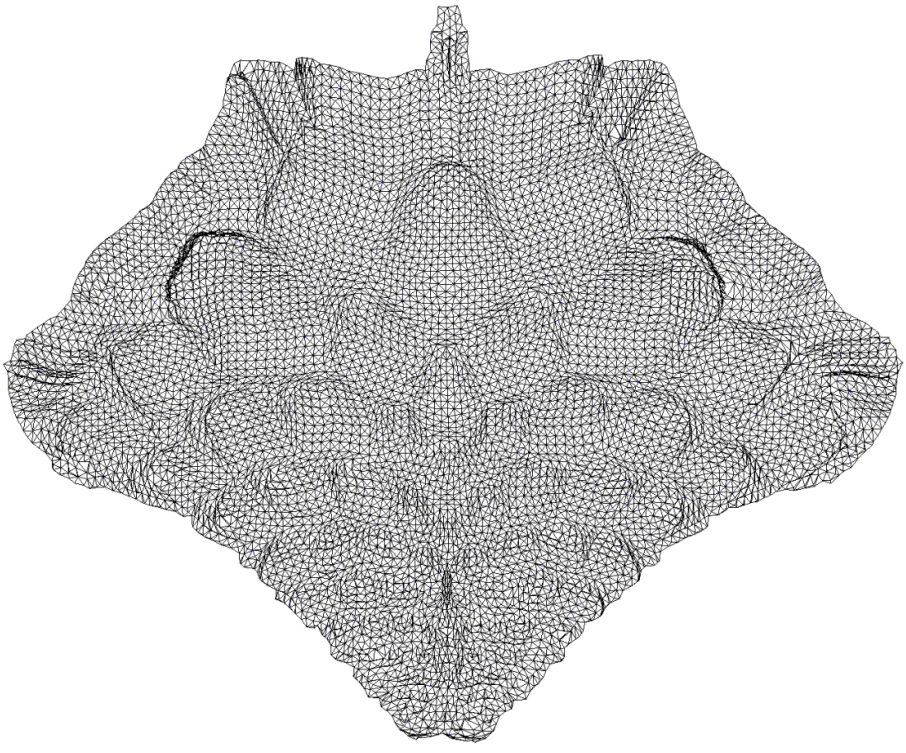


(c) kombinierte MDS mit Pivotgewichtung 1 statt $\frac{n}{q}$

Abbildung 5.18: Layouts für spider_C. $n = 10000$, $m = 22000$.

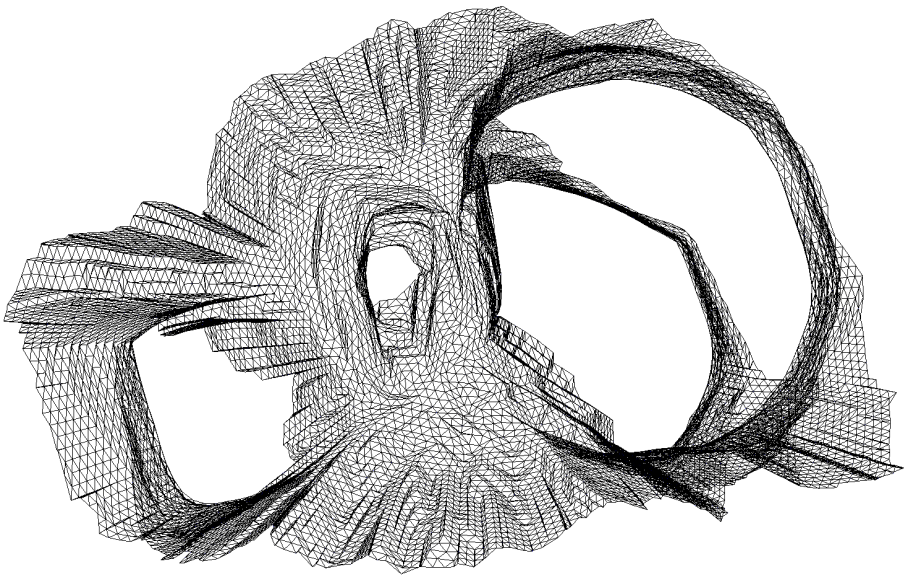


(a) Pivot-MDS



(b) kombinierte MDS

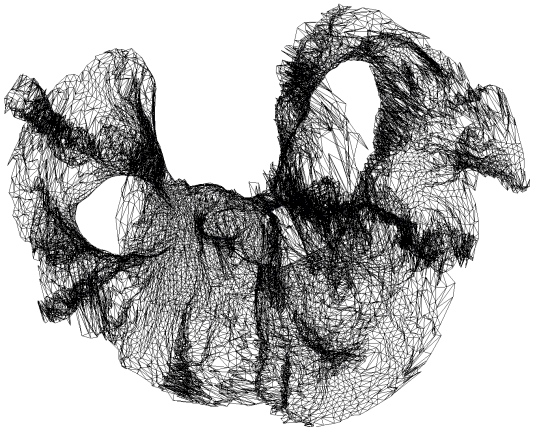
Abbildung 5.19: Layouts für crack. $n = 10240$, $m = 30380$.



(a) Pivot-MDS

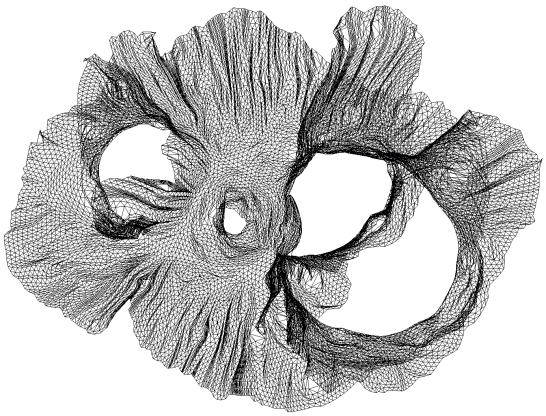
(b) kombinierte MDS

Abbildung 5.20: Layouts für 4elt. $n = 15606$, $m = 45878$.



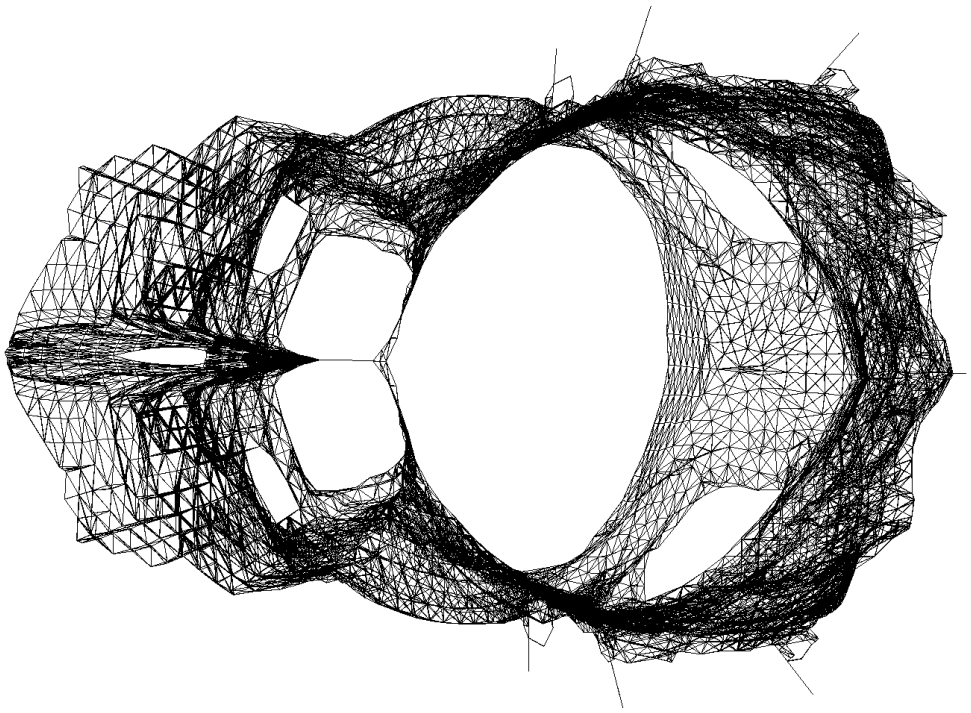
(a) ausgedünnte Stressmajorisierung 20 Iterationen

(b) ausgedünnte Stressmajorisierung 20 Iterationen

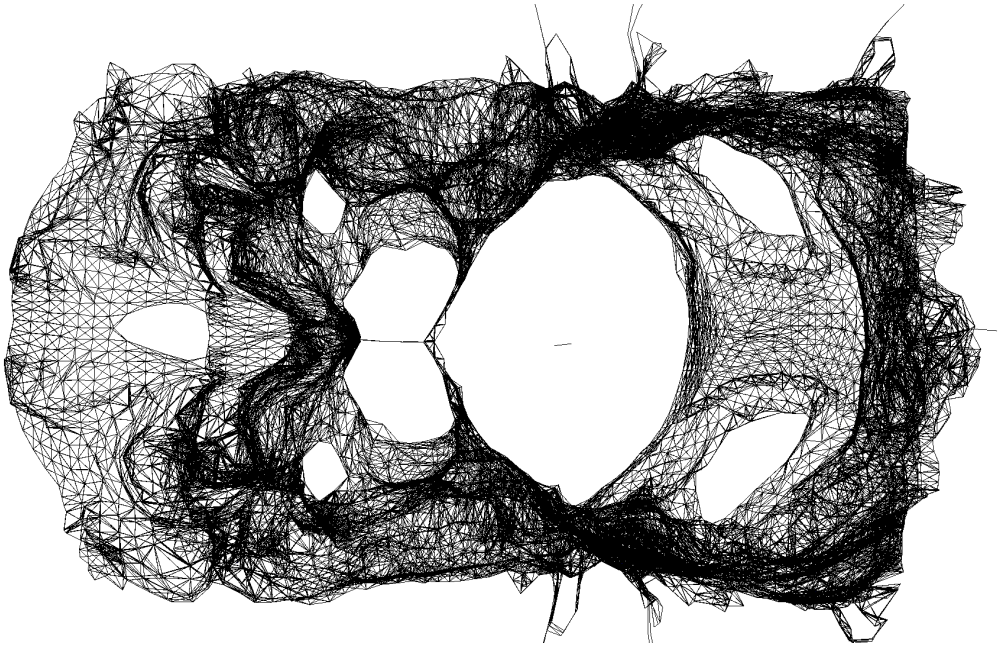


(c) ausgedünnte Stressmajorisierung 100 Iterationen, Laufzeit: 0,4802 s

Abbildung 5.21: Layouts für 4elt ohne Initialisierung durch Pivot-MDS



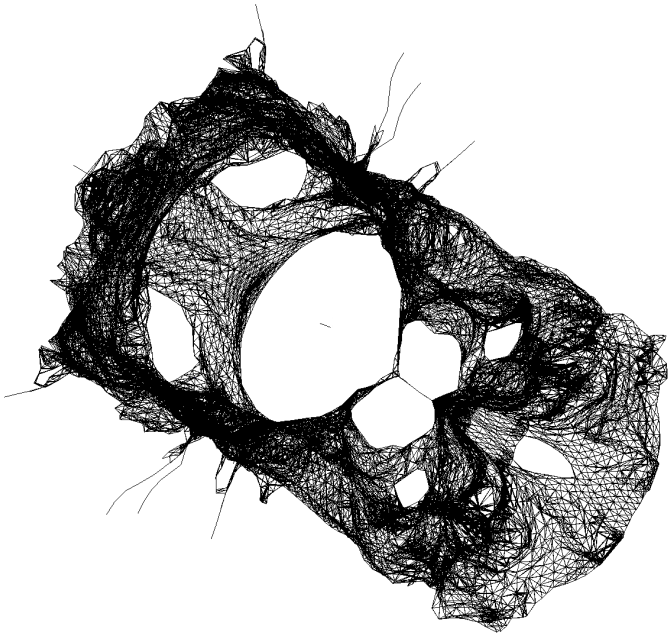
(a) Pivot-MDS



(b) kombinierte MDS

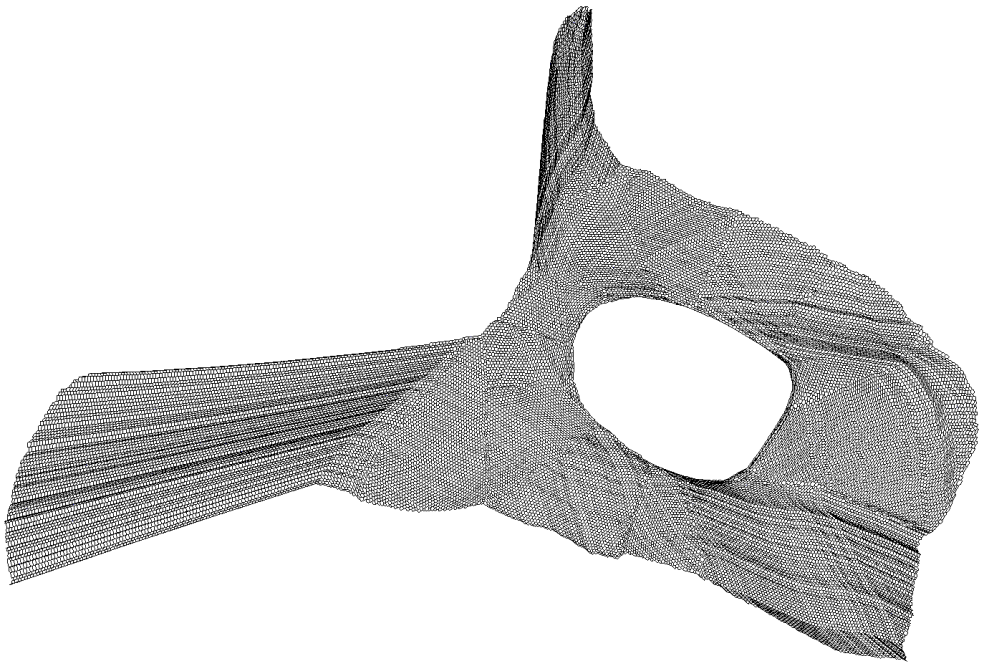
Abbildung 5.22: Layouts für bcsstk31. $n = 35588$, $m = 572914$.

(a) ausgedünnte Stressmajorisierung 20 Iterationen

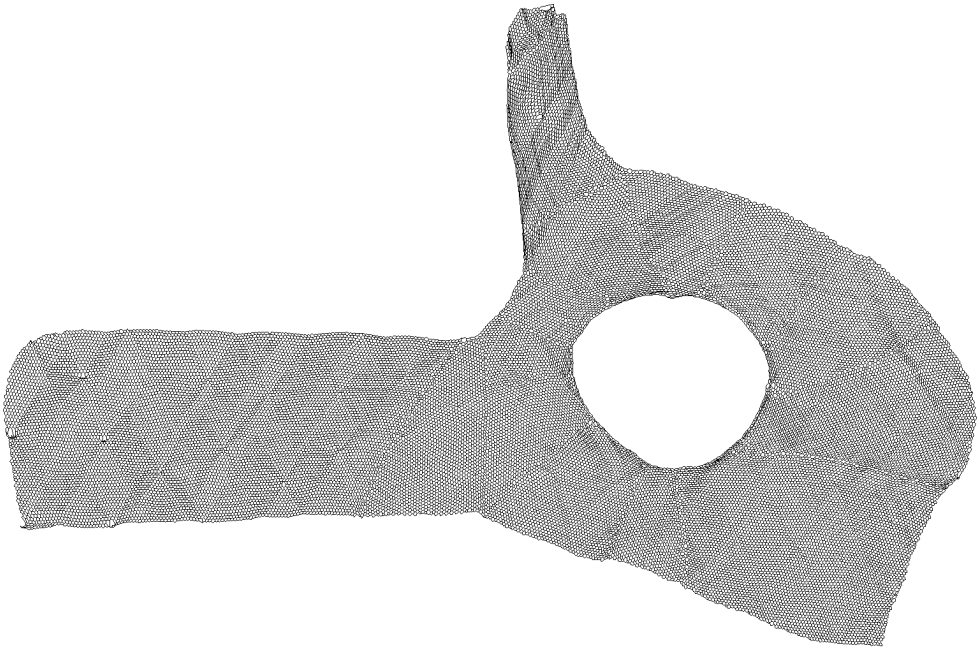


(b) ausgedünnte Stressmajorisierung 100 Iterationen, Laufzeit: 1,1860 s

Abbildung 5.23: Layouts für bcsstk31 ohne Initialisierung durch Pivot-MDS.

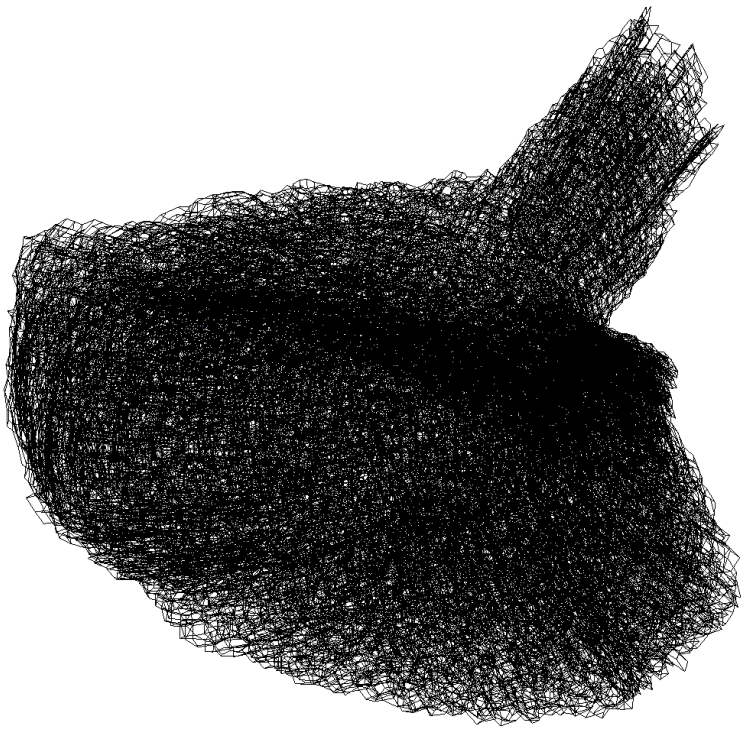


(a) Pivot-MDS

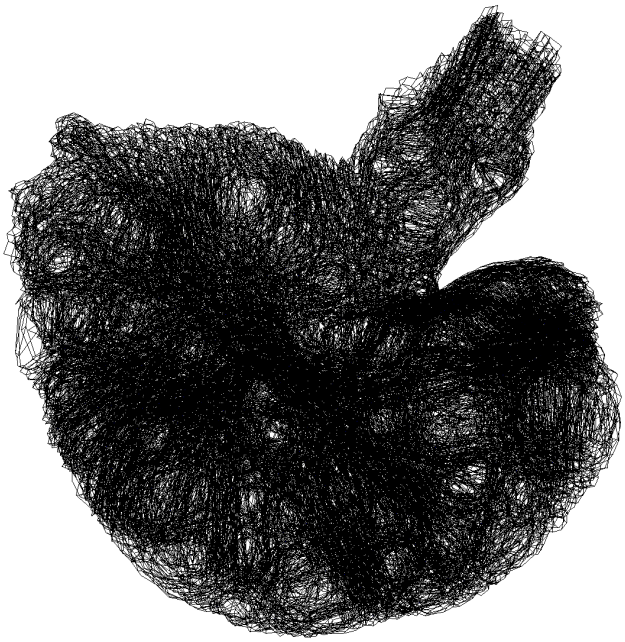


(b) kombinierte MDS

Abbildung 5.24: Layouts für t60k. $n = 60005$, $m = 89440$.



(a) Pivot-MDS



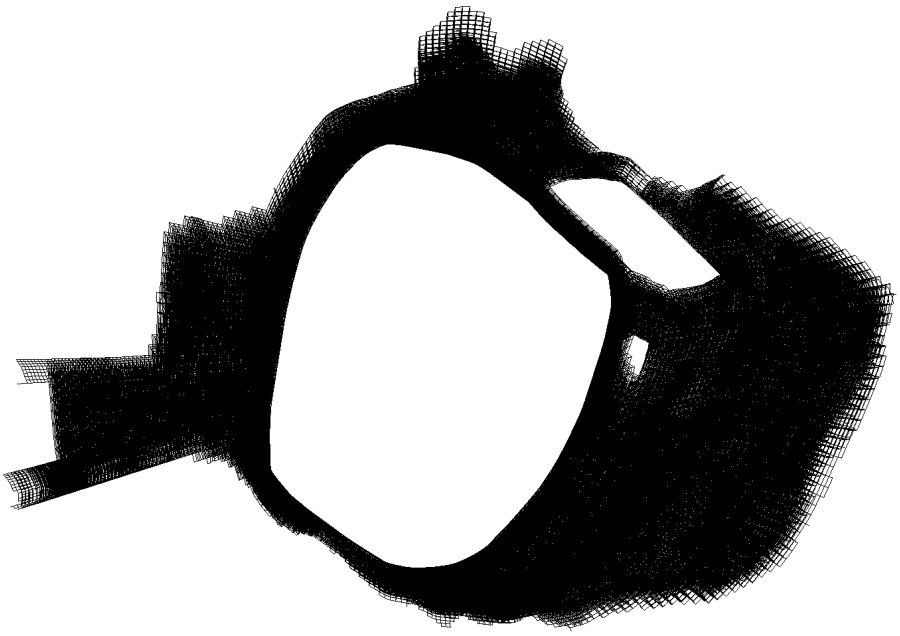
(b) kombinierte MDS

Abbildung 5.25: Layouts für wing. $n = 62032$, $m = 121544$.

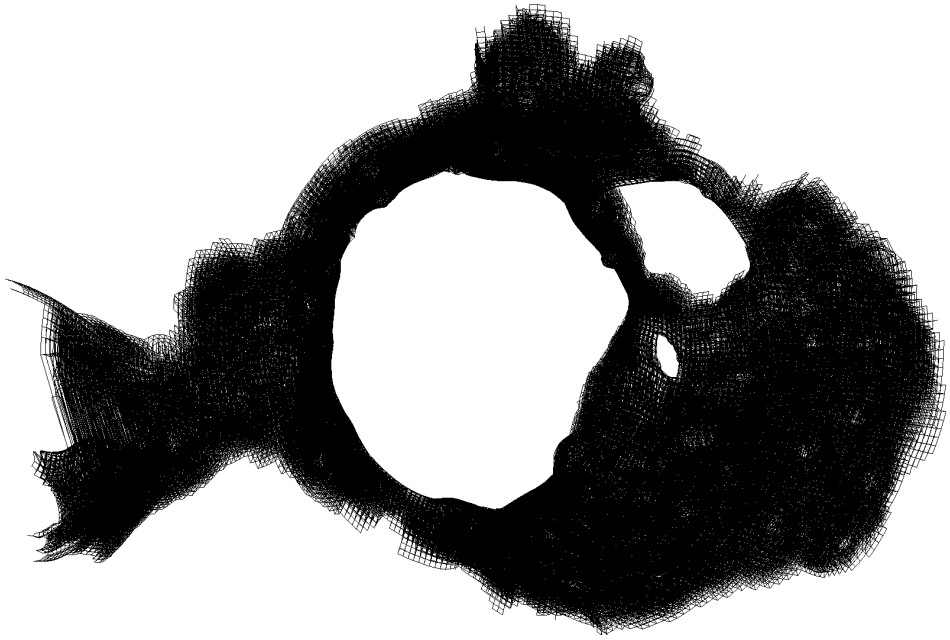
(a) Pivot-MDS

(b) kombinierte MDS

Abbildung 5.26: Layouts für 598a. $n = 110971$, $m = 741934$.



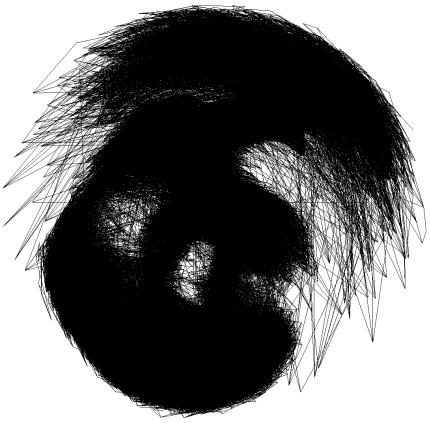
(a) Pivot-MDS



(b) kombinierte MDS

Abbildung 5.27: Layouts für fe_ocean. $n = 143437$, $m = 409593$.

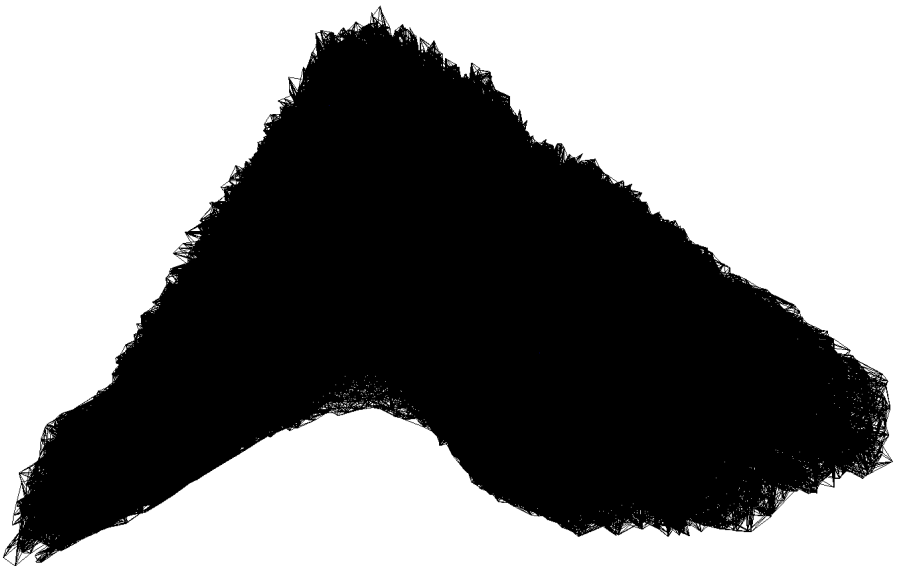
(a) ausgedünnte Stressmajorisierung 20 Iterationen



(b) ausgedünnte Stressmajorisierung 20 Iterationen

(c) ausgedünnte Stressmajorisierung 100 Iterationen, Laufzeit: 4,0096 s

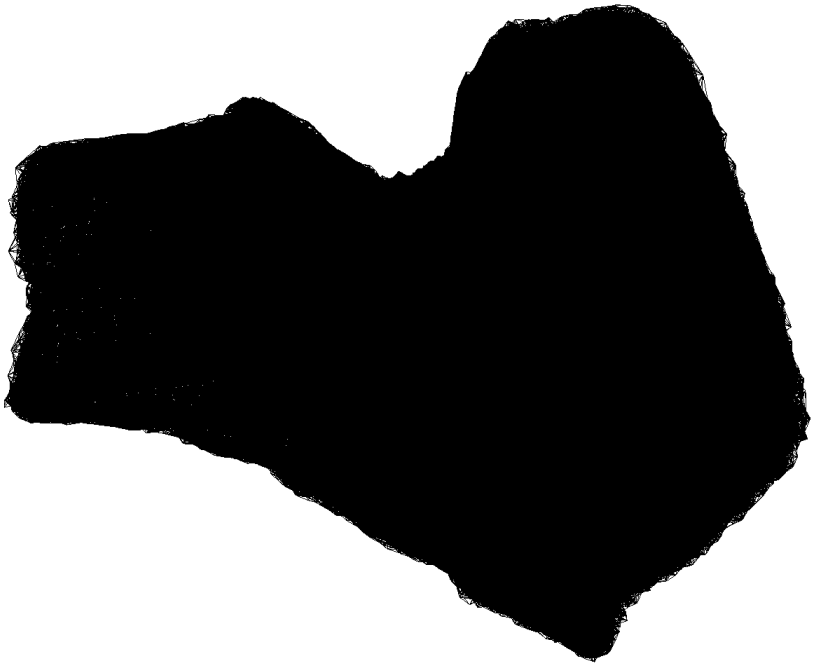
Abbildung 5.28: Layouts für fe_ocean ohne Initialisierung durch Pivot-MDS.



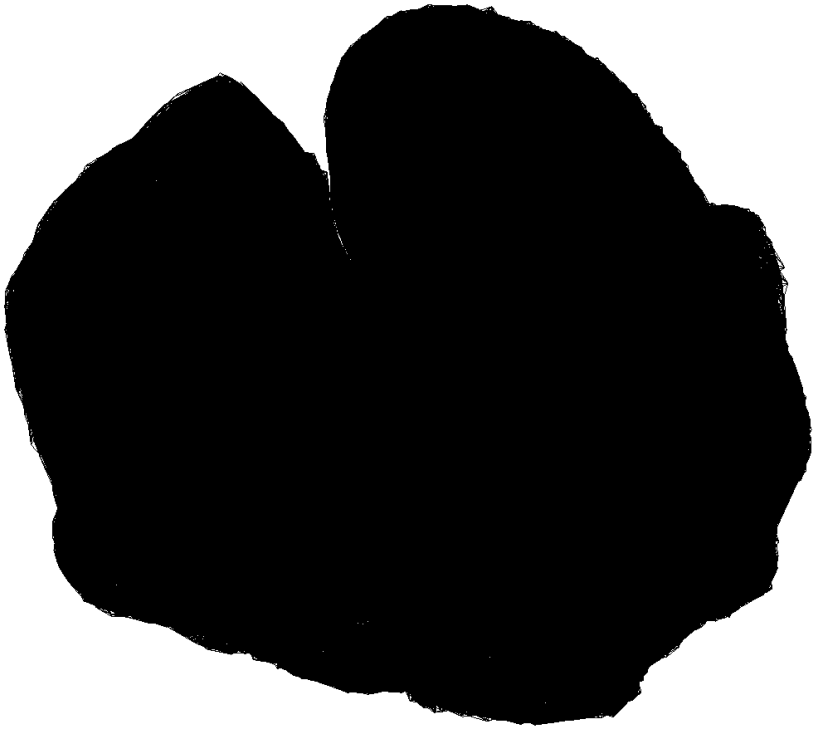
(a) Pivot-MDS

(b) kombinierte MDS

Abbildung 5.29: Layouts für m14b. $n = 214765$, $m = 1679018$.



(a) Pivot-MDS

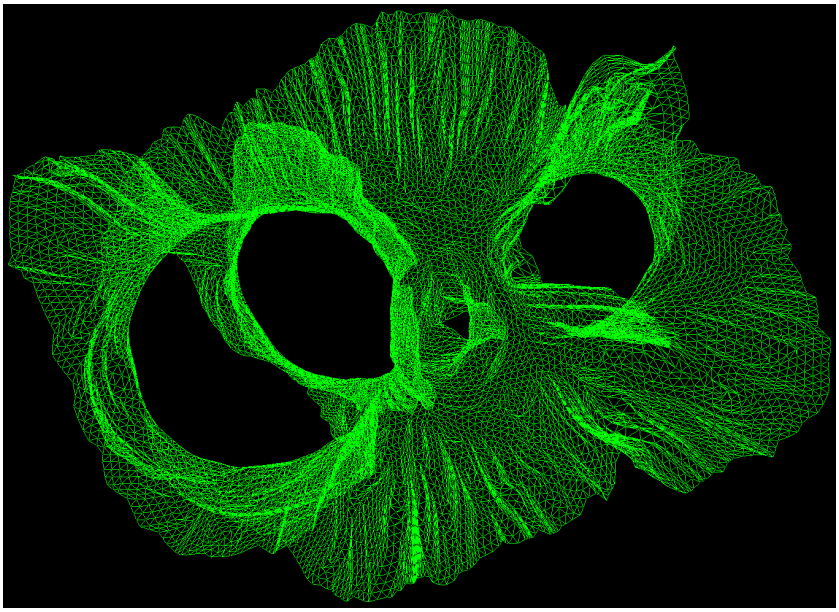


(b) kombinierte MDS

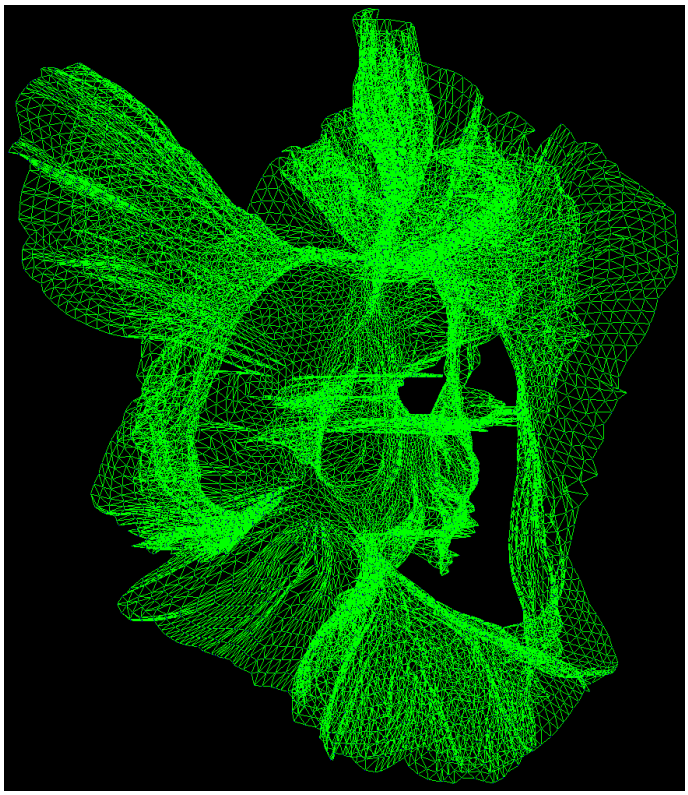
Abbildung 5.30: Layouts für auto. $n = 448695$, $m = 3314611$.

5.6 Dreidimensionale Layouts

Die eigene Implementierung der kombinierten MDS ist in der Lage, dreidimensionale Ergebniskoordinaten für die Knoten zu berechnen, welche sich mittels des im Rahmen dieser Arbeit implementierten Visualisierungsprogramms anzeigen lassen. Abbildung 5.31 enthält zwei Screenshots des Visualisierungsprogramms, welche verschiedene Ansichten des Graphen 4elt zeigen. Strukturen des Graphen, beispielsweise die "Falten" im Graphen 4elt, welche im zweidimensionalen Layout durch Projektion in eine Ebene überlagert sind, lassen sich in der dreidimensionalen Ansicht deutlich besser erkennen.
Da einzelne Screenshots nicht gut geeignet sind, diesen Vorteil dreidimensionaler Layouts zu veranschaulichen, enthält die beiliegende DVD Videos, welche das Navigieren durch die in Tabelle 5.1 aufgeführten Beispielgraphen demonstrieren.



(a) Frontalansicht (Betrachter Position auf positiver z-Achse)



(b) Ansicht von links (Betrachter Position auf negativer x-Achse)

Abbildung 5.31: dreidimensionale Layouts für 4elt.

6 Vergleich

Dieser Abschnitt vergleicht die kombinierte MDS mit anderen schnellen Verfahren zum Zeichnen von Graphen. Der Fokus liegt dabei auf Verfahren, welche für die GPU ausgelegt sind und für die eine GPU-Implementierung existiert. Diese Verfahren sind:

- Glimmer [17], [16]
- MGLG, Multilevel Graphlayout on the GPU [7]
- RMGD, Rapid Multipole Graph Drawing [10].

Der Vergleich behandelt sowohl die prinzipiellen Unterschiede und Gemeinsamkeiten dieser Verfahren als auch die Laufzeiten der Implementierungen und die von diesen generierten Graphlayouts. Die Methoden wurden effizient implementiert. Der Code, welcher auf der Grafikkarte läuft, wurde für RMGD, wie für die kombinierte MDS, in CUDA geschrieben. Glimmer und MGLG verwenden dazu die OpenGL Shading Language. Details zu den jeweiligen Implementierungen behandelt diese Arbeit nicht. Sie finden sich in [17], [16], [7] bzw. [10].

6.1 Kräftebasierende Graphenlayout-Algorithmen

Es gibt zwei bekannte Kräfte-Modelle, mit denen sich bestimmte Klassen von Graphenlayout-Algorithmen veranschaulichen lassen.

Federn-Modell: Beim Federn-Modell existiert zwischen jedem Knotenpaar eine Feder. Diese Federn haben in Ruhe eine Länge, die der Graphdistanz des Knotenpaars entspricht. Der Zustand minimaler Energie dieses Systems entspricht dem resultierenden Layout. Die Stressmajorisierung lässt sich mit diesem Modell erklären. Jede Iteration reduziert die Energie in diesem System. Sie entspricht dem Stress.

Teilchen-Modell: Im Teilchen-Modell herrschen zwei unterschiedliche Arten von Kräften. Die Knoten werden dabei als geladene Teilchen betrachtet, die sich gegenseitig abstoßen. Weiter werden Kanten als Federn modelliert. Adjazente Knoten ziehen sich daher an. Es handelt sich um ein generisches Modell, zu dessen Berechnung man sowohl physikalische Gesetze, z.B. das Coulomb'sche Gesetz für die abstoßenden Kräfte und das Hooke'sche Gesetz für die anziehenden Kräfte, als auch eigens definierte Funktionen verwenden kann. Der Zustand geringst möglicher Energie führt auch hier zum resultierenden Layout. Beispiele für das Teilchen-Modell sind der in MGLG verwendete Algorithmus von Fruchterman und Reingold [8] und die in RMGD verwendete "Fast Multilevel Multipole Method" (FM3) [12].

6.2 Multilevel-Verfahren

Glimmer, MLGL und RMGD sind Multilevel-Verfahren. Algorithmus 6.1 veranschaulicht das Schema eines rekursiv aufgebauten Multilevel-Graphlayoutalgorithmus' (der Algorithmus kann

auch iterativ aufgebaut sein). Eine Rekursionsebene (bzw. eine Iteration) soll dabei als Ebene des Multilevel-Verfahrens bezeichnet werden.

Algorithmus 6.1 : Multilevel Graphlayout Schema (vgl. Pseudocode in [16])

```
Input: nodes //Initialpositionen der Knoten
Output: nodes //Ergebnispositionen der Knoten
multilevel(nodes) begin
    if nodes = emptyset then return
    subset ← coarse(nodes)
    multilevel(subset)
    interpolate(nodes, subset)
    layout(nodes)
end
```

coarse(nodes): Der Multilevelalgorithmus baut im rekursiven Abstieg (bzw. in einer ersten Schleife) mittels einer Vergröberungsfunktion coarse() eine Hierarchie von Knotenteilmengen auf: $V_0 \supset V_1, \ldots, \supset V_c$. Die durch diese Hierarchie induzierten Graphen stellen Vergröberungen des Ausgangsgraphen dar.

interpolate(nodes, subset): Im rekursiven Aufstieg (bzw. in einer zweiten Schleife) interpoliert eine Funktion interpolate() mittels der in der vorherigen Rekursionsebene[30] (bzw. Iteration) für die Knotenmenge *subset* bestimmten Koordinaten die Koordinaten der Knotenmenge *nodes*. Dabei entspricht *subset* einer Knotenteilmenge V_{i-1} und *nodes* V_i.

layout(nodes): Eine Layoutfunktion (z.B. Stressmajorisierung) bestimmt die Knotenpositionen des durch die Knotenmenge *nodes* induzierten Graphen. Dabei dienen die zuvor interpolierten Koordinaten als Intialpositionen.

Alle der vorgestellten Verfahren benutzen eine iterative, konvergierende Layoutfunktion. Daher ist die Laufzeit des Multilevelalgorithmus nur verschwindend größer als die Zeit, die ein direktes Anwenden der Layoutfunktion auf den gesamten Graphen benötigen würde. Beim Multilevel-Schema konvergiert die Layoutfunktion auf jeder Ebene, außer der ersten, deutlich schneller, da sie ein Initiallayout verwendet, welches mit Hilfe des Layouts der jeweils tieferen Ebene interpoliert worden ist.
Das Multilevel-Verfahren hat einen entscheidenden Vorteil gegenüber dem alleinigen Anwenden der Layoutfunktion. Die meisten Layoutfunktionen bevorzugen nahe Distanzen (z.B. Stressmajorisierung mit $\alpha = -2$), da dies zu guten Ergebnissen führt. Bei dieser Vorgehensweise ist es jedoch wahrscheinlicher, in lokalen Minima zu enden. Falls die Knotenmengen auf den unteren Ebenen gut über den Graphen verteilte Knoten enthalten, findet der Multilevel-Algorithmus zunächst die globale Struktur des Graphen. Diese wird aufgrund der Interpolation an die höheren

[30] In der untersten Ebene findet keine Interpolation statt.

Ebenen weitergereicht und beeinflusst damit die Initialpositionen dieser höheren Ebenen. Dadurch ist es unwahrscheinlich, dass das Resultat stark von der globalen Struktur abweicht und in einem lokalen Minimum endet.

Die kombinierte MDS erreicht diesen Vorteil durch das Ausführen von Pivot-MDS vor der ausgedünnten Stressmajorisierung. Pivot-MDS hat dabei den gleichen Zweck wie die unteren Ebenen eines Multilevel-Layoutalgorithmus. Es generiert ein Layout, welches die globale Struktur des Graphen repräsentiert. Die ausgedünnte Stressmajorisierung hat die Aufgabe der höheren Ebenen. Sie benutzt die von Pivot-MDS berechneten Koordinaten als Initialkonfiguration, womit es wie bei der Multilevel-Vorgehensweise unwahrscheinlich ist, in lokalen Minima zu enden. Ferner konvergiert die ausgedünnte Stressmajorisierung dadurch schneller, als würde sie mit einer zufälligen Initialisierung ausgeführt, wie die höheren Multilevel-Ebenen. Da ein Multilevelalgorithmus mehrere Ebenen verwendet, entstehen "weichere" Übergänge zwischen den Layouts der einzelnen Ebenen, was die Wahrscheinlichkeit des Haltens in einem lokalen Minimum weiter senkt und für ein gutes, "geglättetes" Ergebnis sorgt. Die kombinierte MDS hat dagegen nur zwei "Ebenen". Pivot-MDS hat jedoch den Vorteil, dass es unabhängig von der Initialkonfiguration ist und sehr schnell eine gute Repräsentation der globalen Stuktur des Graphen liefert (s. Abbildungen 5.7 bis 5.30 und Tabelle 5.4). Außerdem benötigt die kombinierte MDS keine Vergröberungsschritte, was weitere Zeit spart.

6.3 Glimmer

Vergröberung: Die Vergröberung von V_i zu V_{i+1} geschieht bei Glimmer durch zufälliges Ziehen von Knoten aus V_i. Der Reduzierungsfaktor von Untermenge zu Untermenge ist $\frac{1}{8}$. Die kleinste Menge enthält mindestens 1000 Knoten. Die Optimalität dieser Zahlen für Glimmer wurde empirisch nachgewiesen [17]. Dieses zufällige Ziehen hat den Vorteil, dass es sehr wenig Laufzeit kostest; Glimmer erstellt zu Beginn eine zufällige Permutation der Knoten und entnimmt davon jeweils die benötigte Anzahl. Nachteil ist jedoch, dass in tieferen Ebenen schlechte Repräsentationen des Ausgangsgraphen wahrscheinlicher sind. Bei der kombinierten MDS entspräche diese Methode einer zufälligen Pivotauswahl.

Interpolation: Sowohl für die Interpolation als auch für das Layout verwendet Glimmer GPU Stochastic Force (GPUSF), einen eigenen Distanzskalierungsalgorithmus. Für die Interpolation wird GPUSF in jeder Ebene ebenfalls auf die Knotenmenge *nodes* angewandt, wobei jedoch nur die Positionen der Knoten $nodes \setminus subset$ verändert werden.

Layout: GPUSF benutzt, wie die ausgedünnte Stressmajorisierung, nur eine Teilmenge der Distanzen. Jedem Knoten $v_k \in V_i$ ist eine Menge naher und eine Menge zufälliger Knoten zugeordnet. Die Knoten aus diesen Mengen bestimmen die Position von v_k. Sie enthalten jeweils vier Knoten. Dies entspricht der Größe eines RGBA-Pixels und ist damit gut für die Grafikkarte geeignet. Die Mengen sind jedoch sehr klein.
Der Namensteil "Stochastic" kommt daher, dass die Menge naher Knoten nicht durch einen preprocessing-Schritt bestimmt wird, sondern durch stochastische Auswahl. Zu Beginn initiali-

siert GPUSF beide Mengen mit zufälligen Knoten. Nach jeder Iteration wird geprüft, ob Knoten der "zufälligen Menge" einen kleineren Abstand zum Knoten v_k aufweisen als Knoten der "nahen Menge". Ist dies der Fall, ersetzt GPUSF die Knoten in der nahen Menge durch die näher liegenden. Die Knoten der zufälligen Menge werden in jeder Iteration neu bestimmt. Dies hat den entscheidenen Nachteil, dass trotz der kleinen Mengen von aktiven Knoten sehr viele Distanzen bekannt sein müssen.

Für die Bestimmung der Distanzen werden bei Glimmer mehrere Vorgehensweisen beschrieben. Die erste ist nicht für das Zeichnen von Graphen, sondern für das Clustern von Punktmengen gedacht. Dabei wird davon ausgegangen, dass die höherdimensionalen Koordinaten bekannt und ihre Abstände euklidisch sind. Die Berechnung der jeweiligen höherdimesionalen Abstände findet in jeder Iteration statt, in der sie benötigt werden, was sehr viel Laufzeit kostet. Weiter ist es möglich, Distanzmatrizen einzulesen. Da $n \times n$-Matrizen schon bei Graphen mittlerer Größe nicht in den Speicher der Grafikkarte passen, stellt [17] "distance paging" vor, die Möglichkeit jeweils diejenigen Teile einer Distanzmatrix in den Speicher der Grafikkarte zu laden, die gerade benötigt werden. Außerdem stellt das Paper "distance feeding" vor, was ermöglicht, nur die benötigten Teile der Distanzmatrix zu berechnen ("lazy-evaluation"). Insgesamt benötigt Glimmer aber trotzdem eine sehr große Zahl von Distanzen, wie oben in diesem Abschnitt beschrieben. Die konkrete Berechnung der Distanzmatrizen ist nicht Teil der Beschreibung.
Im Abschnitt "future work" spricht [16] diesen Nachteil an und stellt in Aussicht, dass in einer zukünftigen Version die Elemente der nahen und zufälligen Menge in jeder Iteration gleich bleiben sollen und dass deren Größe vom Benutzer bestimmbar sein wird.

Asymptotische Laufzeit: Die asymptotische Gesamtlaufzeit von Glimmer ist in [17] mit $\mathcal{O}(n^2)$ angegebenen. Dabei nimmt es n GPUSF-Iterationen an mit jeweils einer Laufzeit von $\mathcal{O}(n)$. In den meisten Fällen benötigt Glimmer jedoch weniger Iterationen. Die asymptotisch bessere Laufzeit der Implementierung der kombinierten MDS liegt lediglich daran, dass sie eine konstante Anzahl an Iterationen vorschreibt.

Für Glimmer gibt es keine Beispiellayouts der oben aufgeführten Graphen. Die Laufzeiten sind schlechter als die von MGLG.

6.4 Multilevel Graph Layout

Vergröberung: Der in [7] vorgestellte Algorithmus MGLG (Multilevel Graphlayout on the GPU) verwendet zwei verschiedene Methoden, um Repräsentationen des Ausgangsgraphen mit geringerer Auflösung zu erzeugen.
Zunächst erzeugt eine Kantenentfernungsoperation (edge collapse) eine Hierarchie von Knotenuntermengen $V_1 \supset \cdots \supset V_c$, wobei die Implementierung $c = 3$ verwendet. Dazu bekommen die Knoten und Kanten Gewichtungen, welche mit 1 initialisiert werden. Die Kantenentfernungsoperation ersetzt mehrere adjazente Knotenpaare v_i, v_j durch jeweils einen Knoten, wobei die Gewichtung des neuen Knotens der Summe der Gewichtungen der zu diesem verschmolzenen Knoten entspricht. Paper [7] erwähnt, die Gewichtungen der Kanten seien entsprechend ange-

passt. Wahrscheinlich fällt die Kante zwischen v_i und v_j ohne Einfluss auf Kanten oder Knoten weg. Zwei Kanten, welche mit v_i bzw. v_j und mit einem gemeinsamen Knoten inzident sind, verschmelzen zu einer und erhalten die entsprechend summierte Gewichtung. Alle anderen mit v_i bzw. v_j inzidenten Kanten sind danach mit dem neuen Knoten inzident und behalten ihre Gewichtung.
Um die zu löschenden Kanten auszusuchen, sortiert der Algorithmus die Knoten zunächst nach ihrem Grad, um als erstes Knoten mit geringem Grad zu entfernen. Ausgehend vom Knoten geringsten Grades u bestimmt er die zu löschende Kante durch Maximierung von

$$\frac{w(u,v)}{w(v)} + \frac{w(u,v)}{w(u)},$$

wobei $w(u)$ das Gewicht eines Knotens u und $w(u,v)$ das Gewicht einer Kante (u,v) ist. Wie viele Kanten in einem Vergröberungsschritt gelöscht werden, ist nicht erwähnt.
Anschließend erzeugt der Algorithmus ausgehend von V_c, $c = 3$ mittels Partitionierung eine Hierarchie von Graphen mit aufsteigender Auflösung. Diese Graphen werden nicht als gröbere Versionen des Ausgangsgraphen bezeichnet, erfüllen jedoch später in den entsprechenden Ebenen deren Zweck. Die Hierarchie wird folgendermaßen erzeugt: Der von V_3 induzierte Graph wird mittels spektraler Graphpartitionierung [6] in drei Partitionen aufgeteilt. Dabei enthalten die Partitionen gleich viele Elemente, weiter trennt sie ein minimaler Schnitt. Der Algorithmus teilt die Partitionen entsprechend weiter auf, so dass eine Hierarchie von Partitionen P_i^l entsteht, wobei l für die Hierachieebene steht und i für eine Partition dieser Ebene. Die Ebene l enthält somit l^3 Partitionen. Um die Graphhierarchie zu erstellen, werden die Knoten der Partitionen einer solchen Ebene jeweils zu einem Knoten zusammengefasst. Die Kanten, die die so generierten Knoten verbinden, sind mit der Summe der Gewichtungen der Kanten zwischen den beiden Partitionen gewichtet. Falls zwischen zwei Partitionen keine Kante existiert, sind die daraus generierten Knoten nicht verbunden. Dies führt zu einer Graphhierarchie $L^0, \ldots, L^{finest}$, deren Knoten v_i^l den Partionen $P_i^0, \ldots, P_i^{finest}$ entsprechen. Dabei ist zu beachten, dass der erste generierte Graph die geringste Auflösung hat. Er hat die gleiche Rolle wie die gröbste Darstellung des Graphen in den anderen Verfahren. Die Hierarchie von Graphen, welche der zugrunde liegende Multilevel-Algorithmus nacheinander verwendet, ist $L^0, \ldots, L^{finest}, G(V_c), \ldots, G(V_1), G$, wobei $G(V_c)$ der von der Knotenmenge V_c induzierte Graph ist und G der Ausgangsgraph.

Mittels spektraler Graphpartitionierung lassen sich aus einem Graphen Repräsentationen desselben mit geringerer Auflösung erzeugen, die dessen Struktur sehr gut repräsentieren. Das Verfahren kostet jedoch viel Rechenzeit. Der Ausgangsgraph für die spektrale Graphpartitionierung in MLGL ist der bereits durch die Kantenentfernungsoperationen vergröberte Graph $G(V_c)$. Die Graphen $L^0, \ldots, L^{finest}$ können folglich den Ausgangsgraphen höchstens so gut repräsentieren wie $G(V_c)$. Es stellt sich daher die Frage, ob sich die spektrale Graphpartitionierung lohnt und wie gut die Ergebnisse ausfallen würden, wenn man für alle Ebenen eine Vergröberung mittels der Kantenentfernungsoperation verwenden würde. Eine weitere Methode zur Vergröberung verwendet RMGD. Diese wird in Abschnitt 6.5 vorgestellt.

Ein mit einer Graphpartitionierung vergleichbares Vorgehen kann die kombinierte MDS in der

Phase der ausgedünnten Stressmajorisierung erreichen, indem sie den Pivotknoten verschiedene Gewichtungen zuordnet. Die Gewichtung eines Pivotknotens p entspricht dabei der Anzahl der Knoten, deren nächstliegender (bezüglich Graphdistanz) Pivotknoten p ist. Diese Knoten sind sozusagen in einer von p repräsentierten Partition. Durch die hybride min-max-Pivotauswahlstrategie (s. Abschnitt 2.5) würde man eine recht gleichmäßige Verteilung der Partitionen über den Graphen erreichen. Es wäre interessant, auf diese Art und Weise eine Vergröberungshierarchie für ein Multilevel-Verfahren zu erstellen.

Interpolation: Bei der Interpolation zwischen den Ebenen, zu denen die Graphen $L^0, \ldots, L^{finest}$ gehören, bekommen die Knoten der höheren Ebene zunächst die Position, die ihre Partition in der tieferen Ebene hat. Danach findet eine Skalierung der Knotenpositionen statt, welche proportional zum Verhältnis der Knotenanzahl der beiden involvierten Ebenen ist [7]:

$$\{x_{\bullet 1}, x_{\bullet 2}\} \leftarrow \sqrt{\frac{|V(L^l)|}{|V(L^{l-1})|}}\{x_{\bullet 1}, x_{\bullet 2}\},$$

wobei $|V(L^l)|$ die Knotenanzahl des Graphen der Ebene l ist.
Danach verschiebt ein iterativer Algorithmus jeden Knoten v_i in jeder Iteration an eine Position, die er mittels des Durchschnitts der aktuellen Position von v_i und der durchschnittlichen Position der Nachbarn von v_i bestimmt. In der Implementierung terminiert der Algorithmus nach 50 Iterationen.
Für die Interpolation der zu $G(V_c), \ldots, G(V_1), G$ gehörigen Ebenen wird das gleiche Verfahren benutzt. Es ist nicht angegeben, welche Position aus der tieferen Ebene den Knoten der höheren zugewiesen wird. Man kann annehmen, dass ein Knoten v_i die Position des Knotens bekommt, der durch Verschmelzung von v_i mit einem anderen entstanden ist.

Layout: MGLG verwendet zwei Layoutalgorithmen. Für das Layout der unteren Ebenen, deren Graphen eine geringe Anzahl Knoten haben, ist der in Abschnitt 2.6 bereits erwähnte Algorithmus von Kamada und Kawai (KK) [18] zuständig, welcher eine Stressmajorisierung darstellt. Diese Ebenen berechnen sämtliche paarweisen Distanzen. Die oberen Ebenen layoutet der Algorithmus von Fruchterman und Reingold (FR) [8], dem das Teilchen-Modell zugrunde liegt. Das Paper [7] begründet diese Entscheidung damit, dass KK zwar weniger anfällig für lokale Minima sei, jedoch weniger ästhetische Layouts liefere als FR und dass die Berechnung aller paarweisen Distanzen in den höheren Ebenen zu aufwändig sei. Die von der kombinierten MDS generierten Layouts sind denen des KK-Algorithmus ähnlich. Ein Vergleich der Layouts findet sich am Ende dieses Abschnitts. Der Berechnung aller paarweisen Distanzen könnte man mittels einer ausgedünnten Stressmajorisierung ausweichen, zumal der FR-Algorithmus ebenfalls angepasst wurde, um die Berechnung aller paarweisen abstoßenden Kräfte zu vermeiden. Dazu verwendet MGLG KD-Bäume.

KD-Baum: Ein KD-Baum (k-dimensionaler Baum) ist ein Baum, der Partitionierungen eines k-dimensionalen Raums repräsentiert. Dabei entspricht die Wurzel des KD-Baums dem gesamten Raum und damit allen Elementen. Seine weiteren Knoten entsprechen jeweils einem Halbraum

des Raumes ihres Elternknotens und den Elementen, die sich darin befinden. Jedem Knoten ist damit ein Unterraum des Ausgangsraums zugeteilt. Die Unterräume, die von Knoten einer Ebene des KD-Baums repräsentiert werden, bilden zusammen den Ausgangsraum. Damit repräsentieren die Ebenen des KD-Baums jeweils eine Partitionierung der Elemente. Für gewöhnlich sind die Hyperebenen, welche zur Aufteilung des Raums in der Ebene l des Baumes führen, orthogonal zur Achse der Dimension l modulo Raumdimensionalität. Im zweidimensionalen Fall ist demnach die erste Hyperebene orthogonal zur x-Achse, die zweiten beiden sind orthogonal zur y-Achse und die der dritten Ebene sind wieder orthogonal zu x-Achse.

MGLG erzeugt mittels eines KD-Baums eine geometrische Partitionierung der Knoten des Graphen basierend auf den aus der darunterliegenden Ebene interpolierten Koordinaten. Dabei werden die Knoten jeweils anhand des Medians der x- bzw. y-Koordinaten aufgeteilt. Die Aufteilung stoppt, wenn eine gewünschte Partitionsgröße erreicht wurde, welche in der Laufzeitanalyse in [7] mit $\sqrt{n}$ angegeben ist. Das Paper [7] betrachtet nur die feinste Partitionierung, d.h. diejenige, die durch die unterste Ebene des KD-Baums repräsentiert wird. Damit gibt es $\sqrt{n}$ Partitionen mit jeweils $\sqrt{n}$ Knoten. Für einen Knoten v werden die paarweisen abstoßenden Kräfte, die auf ihn wirken, nur innerhalb der Partition berechnet, in der sich v befindet. Andere Partitionen wirken jeweils als Gesamtkraft, die von ihren Schwerpunkten ausgeht, auf v. Dies reduziert die für das Bestimmen der abstoßenden Kräfte benötigte asymptotische Laufzeit von $\mathcal{O}(n^2)$ auf $\mathcal{O}(n * \sqrt{n})$. Die anziehenden Kräfte werden alle berechnet, was zu einer Laufzeit von $\mathcal{O}(m)$ führt.
Anschließend verschiebt MGLG die Knoten proportional zur berechneten Gesamtkraft, die auf sie wirkt. Eine simulierte Abkühlung (simulated annealing) sorgt dafür, dass die wirkende Gesamtkraft in jeder Iteration um einen gewissen Faktor abnimmt. Das Layout ist gefunden, sobald dieses System "eingefroren" ist.
Um Laufzeit zu sparen, verwendet die oberste Ebene, welche für das Layout von G zuständig ist, FR nicht. Das Resultat sind die mittels der Positionen von $G(V_1)$ interpolierten Koordinaten.

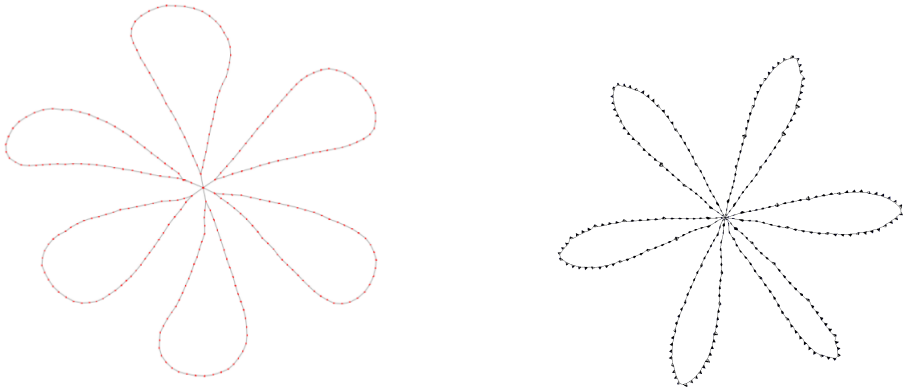
Asymptotische Laufzeit:

Spektrale Graphpartitionierung	$\mathcal{O}(n^{1,5})$
Abstoßende Kräfte	$\mathcal{O}(n^{1,5})$
Anziehende Kräfte	$\mathcal{O}(m)$
gesamt	$\mathcal{O}(n^{1,5} + m)$

Die Anzahl der Iterationen ist dabei konstant.

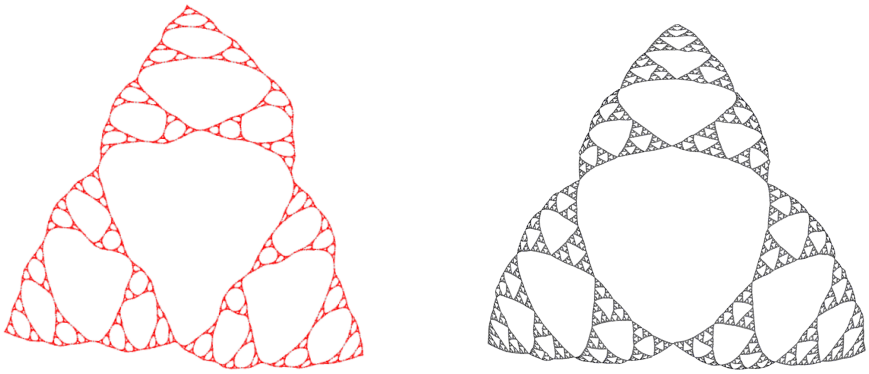
Ergebnisvergleich: Die Abbildungen 6.1 bis 6.4 vergleichen die von MGLG generierten Layouts mit denen der kombinierten MDS. Die angegebenen Laufzeiten beziehen sich auf das System, bestehend aus AMD Phenom II X4 850 @ 3,3 GHz CPU und Nvidia GTX 550 Ti GPU. Sie

sind jedoch bei Graphen, deren Distanzen auf der CPU bestimmt werden, sogar schlechter als die auf dem System mit Intel Core 2 Duo E6650 @ 2,3 GHz CPU und einer Nvidia GTS 250 GPU, da die neuen Features der Nvidia GTX 550 Ti nicht genutzt wurden und die Implementierung auf die Nvidia GTS 250 angepasst ist. Deshalb sind die Zeiten mit dem von MGLG verwendeten System vergleichbar, welches aus einer 2,4 GHz Core 2 Duo CPU und einer Nvidia 8800GTS GPU besteht.



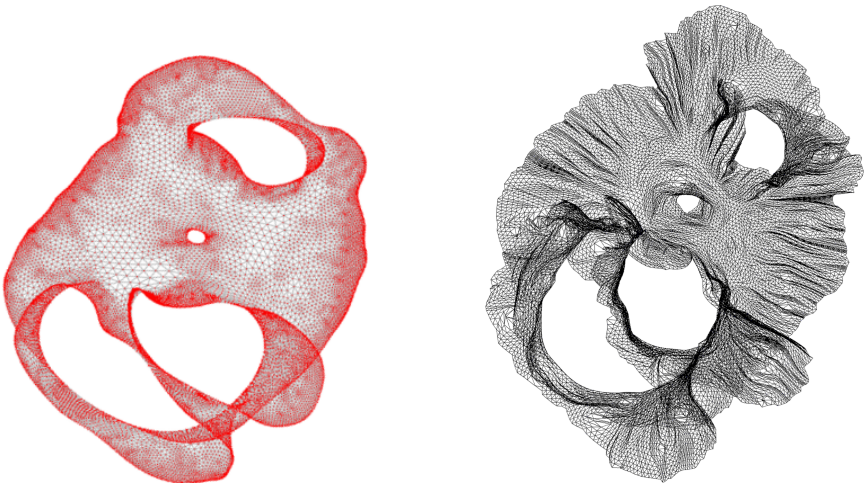
(a) MGLG, Laufzeit: 1,59 s (b) kombinierte MDS, Laufzeit: 0,3293 s

Abbildung 6.1: Layouts für flower_050 (im Paper flower_B). $n = 9030$, $m = 131241$.



(a) MGLG, Laufzeit: 2,705 s (b) kombinierte MDS, Laufzeit: 0,1713 s

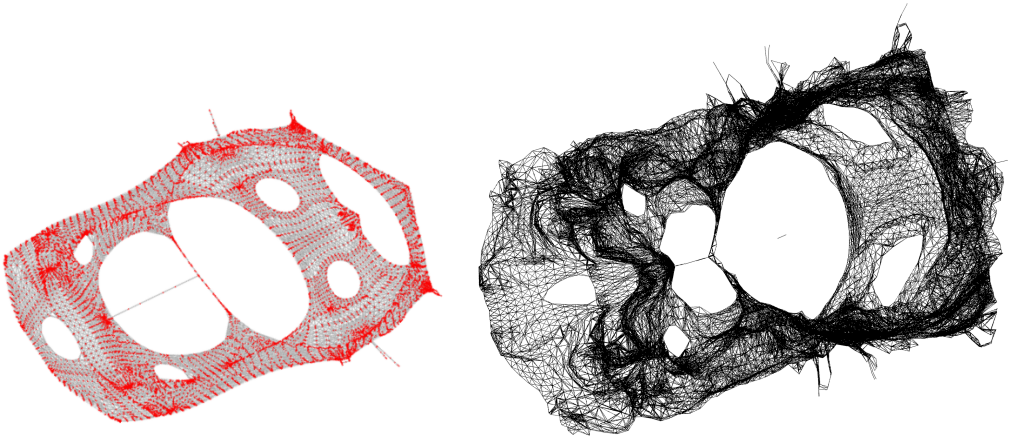
Abbildung 6.2: Layouts für sierpinski_08. $n = 9843$, $m = 19683$.



(a) MGLG, Laufzeit: 3,237 s

(b) kombinierte MDS, Laufzeit: 0,2609 s

Abbildung 6.3: Layouts für 4elt. $n = 15606$, $m = 45878$.



(a) MGLG, Laufzeit: 5,754 s

(b) kombinierte MDS, Laufzeit: 0,6378 s

Abbildung 6.4: Layouts für bcsstk31. $n = 35588$, $m = 572914$.

6.5 Rapid Multipole Graph Drawing

Vergröberung: Der in [10] vorgestellte Algorithmus RMGD (Rapid Multipole Graph Drawing) erstellt die Vergröberungshierarchie $V_1 \supset \cdots \supset V_c$ durch die Bildung maximal unabhängiger Mengen. Die Berechnung derer ist NP-vollständig, weshalb RMGD folgende Näherung benutzt: Der Algorithmus erzeugt die Menge V_{i+1} aus der Menge V_i, indem er in einer ersten Iteration einen Knoten v aus V_i markiert und in V_{i+1} einfügt. Danach werden in V_i sämtliche zu v adjazente Knoten markiert. Weitere Iterationen wählen jeweils nicht markierte Knoten aus und verfahren entsprechend mit diesen. Sind alle Knoten in V_i markiert, ist V_{i+1} gefunden. Dies führt zu einer recht guten Repräsentation der globalen Struktur in den unteren Ebenen und ist sowohl schneller als auch einfacher als die Hierarchieerstellung von MGLG.

Interpolation: Zur Interpolation verwendet RMGD die gleiche Methode wie MGLG, lediglich ohne Skalierung. Knoten in V_i werden mit der Position ihrer Elternknoten aus V_{i+1} initialisiert. Wahrscheinlich ist der Elternknoten $w \in V_{i+1}$ von $v \in V_i$ der Knoten w, den diejenige Iteration zu V_{i+1} hinzugefügt hat, welche v in V_i markiert hat.

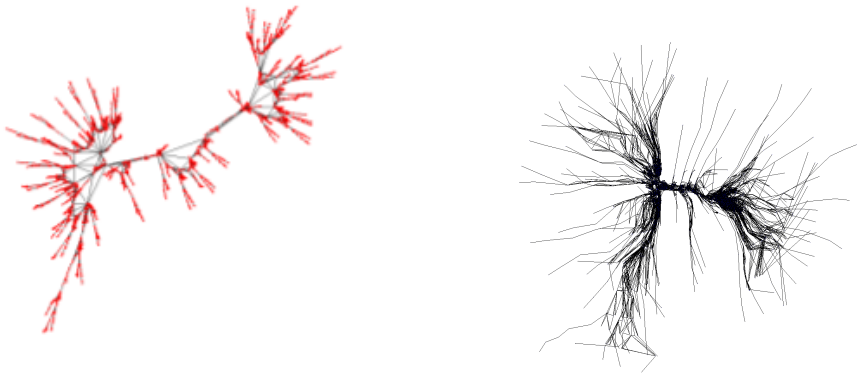
Layout: Die verwendete Layoutfunktion baut auf der in FM3 [12] verwendeten auf und basiert damit wie MGLG auf dem Teilchen-Modell. Den abstoßenden Kräften, welche auf die Knoten wirken, liegt dabei wie in FM3 das physikalische Modell der Multipolentwicklung zugrunde. Um nicht alle paarweisen abstoßenden Kräfte berechnen zu müssen, benutzt RMGD ebenfalls einen KD-Baum. Dabei werden im Gegensatz zu MLGL auch die Partitionierungen höherer Ebenen des KD-Baums benutzt. Für jeden Knoten des KD-Baums, von denen jeder einer Partition entspricht, werden die Koeffizienten der Multipolentwicklung berechnet, was in [10] bzw. [12] erklärt ist. Für die Berechnung der auf jeden Knoten v wirkenden abstoßenden Kräfte durchläuft RMGD den KD-Baum einmal von der Wurzel bis zu dem Blatt, welches diesen Knoten enthält. Bei jedem durchlaufenen Knoten k des KD-Baums wird geprüft, ob sich v in einem gewissen Radius um das geometrische Zentrum des von k repräsentierten Unterraums befindet. Ist dies nicht der Fall, so wirkt diese Partition als Gesamtkraft auf v. Anderenfalls steigt der Algorithmus in den zu k gehörigen Unterbaum ab und verfährt entsprechend weiter. Sobald dieser Algorithmus in einem Blatt angekommen ist, bestimmt er die genauen paarweisen abstoßenden Kräfte.
Durch diese Vorgehensweise benötigt die Bestimmung der paarweisen abstoßenden Kräfte eine asymptotische Laufzeit von $\mathcal{O}(n * log(n))$, falls die Blätter eine konstante Anzahl von Knoten enthalten. Diesen asymptotischen Laufzeitvorteil weist RMGD gegenüber MGLG auf, da es nicht sämtliche von den Blättern repräsentierten Partitionen für die Bestimmung der auf einen Knoten v wirkenden Kräfte heranzieht, sondern die Balanciertheit des KD-Baums ausnutzt und für eine Menge von Blättern, die v nicht enthalten und sich in einem gemeinsamen Unterbaum befinden, die zu der Wurzel dieses Unterbaums gehörige Partition verwendet. Weiter besagt [10], die von RMGD verwendete Multipolentwicklung sei weniger fehleranfällig als die in MGLG benutzte Schwerpunkt-Methode. Die zwischen adjazenten Knoten wirkenden Kräfte werden ebenfalls alle berechnet. RMGD verwendet jedoch eine andere Kraft-Funktion als MGLG.

Asymptotische Laufzeit:

Vergröberung	$\mathcal{O}(n)$
Abstoßende Kräfte	$\mathcal{O}(n * log(n))$
Anziehende Kräfte	$\mathcal{O}(m)$
gesamt	$\mathcal{O}(n * log(n) + m)$

Die Anzahl der Iterationen ist bei diesen Laufzeitangaben konstant.

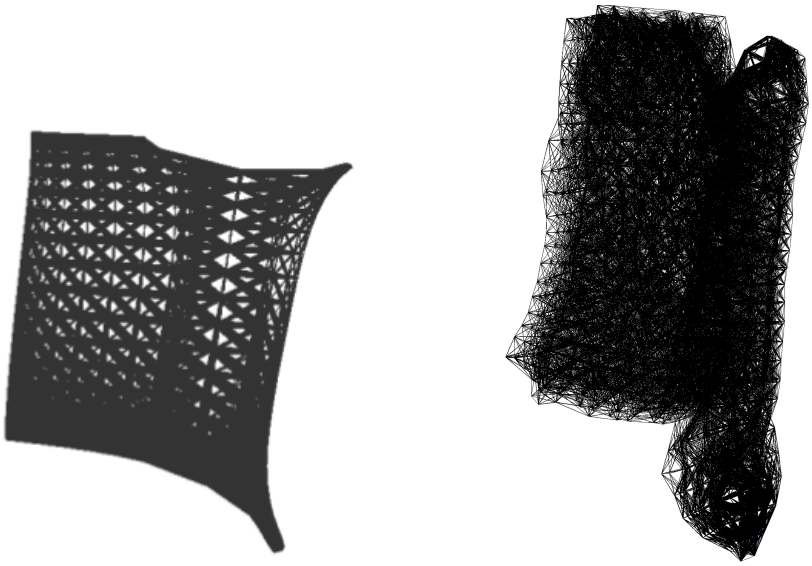
Ergebnisvergleich: Die Abbildungen 6.5 bis 6.15 vergleichen die von RMGD generierten Layouts mit denen der kombinierten MDS. Das für die kombinierte MDS verwendete System ist das gleiche wie für den Vergleich mit MGLG und kann aus dem gleichen Grund als Vergleichsbasis für das in RMGD verwendete System dienen, welches aus einer AMD Athlon 64 CPU und Nvidia GeForce 8800 GTX besteht.



(a) RMGD, Laufzeit: 1,40 s

(b) kombinierte MDS, Laufzeit: 0,1249 s

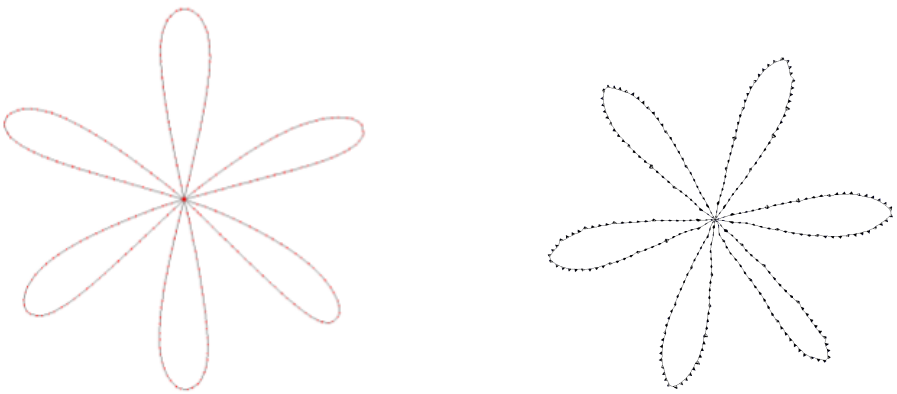
Abbildung 6.5: Layouts für add32 . $n = 4960$, $m = 9462$.



(a) RMGD, Laufzeit: 0,968 s

(b) kombinierte MDS, Laufzeit: 0,235 s

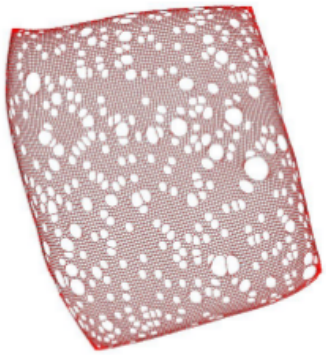
Abbildung 6.6: Layouts für bcsstk33. $n = 8738$, $m = 291583$.



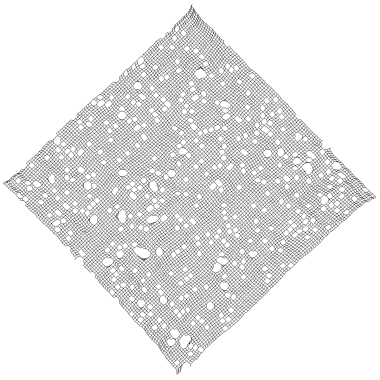
(a) RMGD, Laufzeit: 0,547 s

(b) kombinierte MDS, Laufzeit: 0,3293 s

Abbildung 6.7: Layouts für flower_050 (im Paper flower_B). $n = 9030$, $m = 131241$.

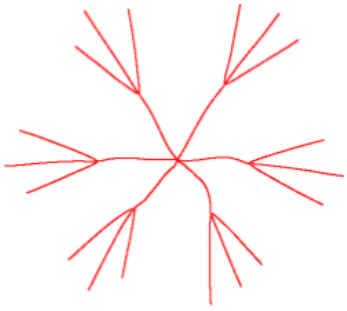


(a) RMGD, Laufzeit: 1,72 s

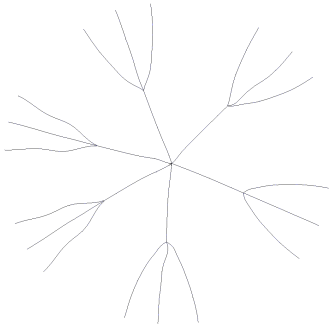


(b) kombinierte MDS, Laufzeit: 0,1736 s

Abbildung 6.8: Layouts für grid_rnd_100. $n = 9497$, $m = 17849$.

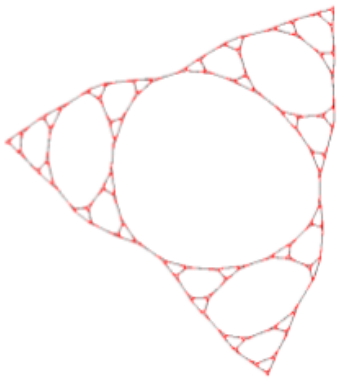


(a) RMGD, Laufzeit: 1,94 s

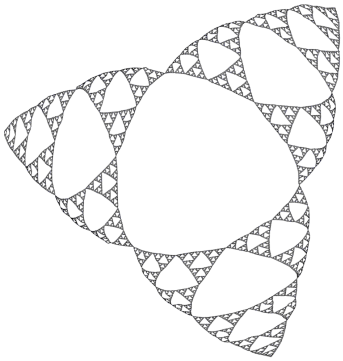


(b) kombinierte MDS, Laufzeit: 0,2966 s

Abbildung 6.9: Layouts für snowflake_C (im Paper snowflake_B). $n = 9701$, $m = 9700$.

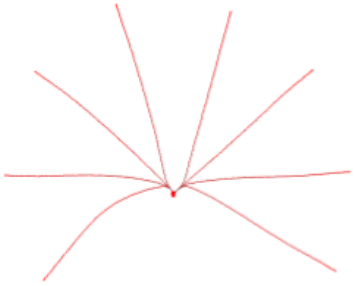


(a) RMGD, Laufzeit: 0,984 s

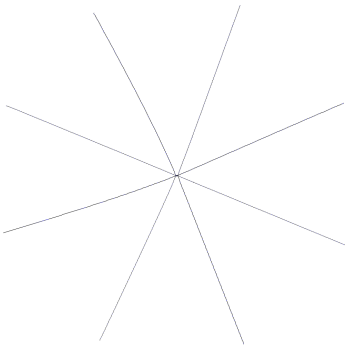


(b) kombinierte MDS, Laufzeit: 0,1713 s

Abbildung 6.10: Layouts für sierpinski_08. $n = 9843$, $m = 19683$.

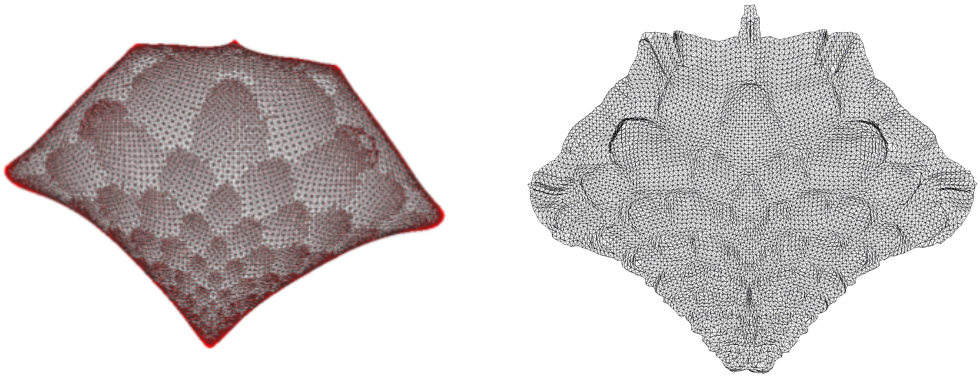


(a) RMGD, Laufzeit: 1,49 s



(b) kombinierte MDS, Laufzeit: 0,3093 s

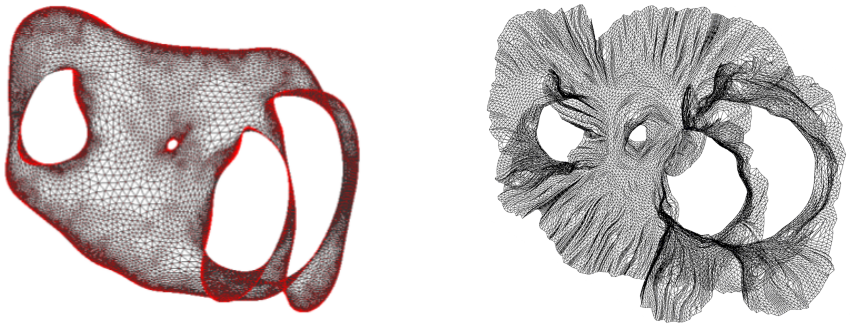
Abbildung 6.11: Layouts für spider_C (im Paper spider_B) . $n = 10000$, $m = 22000$.



(a) RMGD, Laufzeit: 0,937 s

(b) kombinierte MDS, Laufzeit: 0,191 s

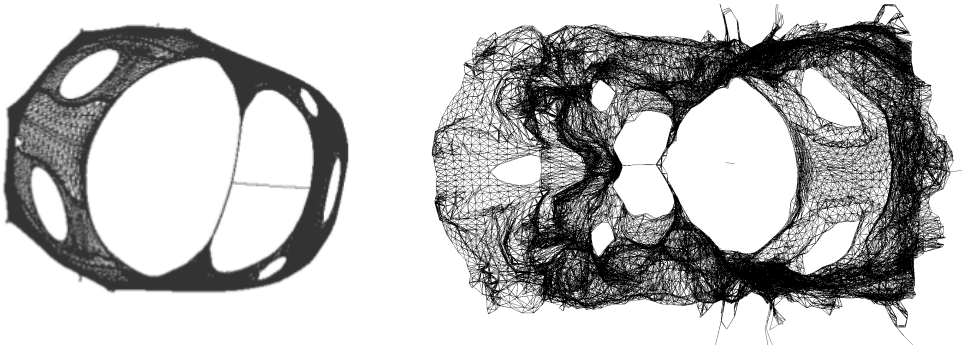
Abbildung 6.12: Layouts für crack. $n = 10240$, $m = 30380$.



(a) RMGD, Laufzeit: 1,58 s

(b) kombinierte MDS, Laufzeit: 0,2609 s

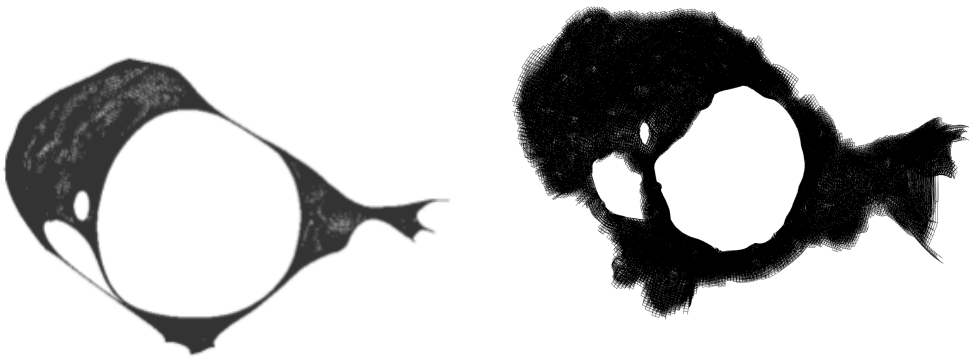
Abbildung 6.13: Layouts für 4elt. $n = 15606$, $m = 45878$.



(a) RMGD, Laufzeit: 1,31 s

(b) kombinierte MDS, Laufzeit: 0,6378 s

Abbildung 6.14: Layouts für bcsstk31. $n = 35588$, $m = 572914$.



(a) RMGD, Laufzeit: 12,07 s (b) kombinierte MDS, Laufzeit: 2,08 s

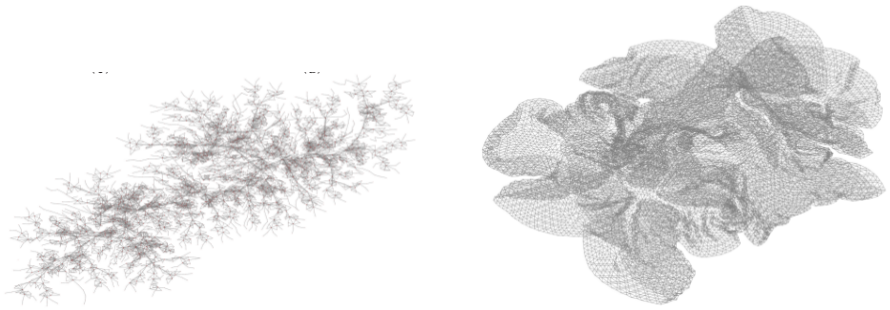
Abbildung 6.15: Layouts für fe_ocean. $n = 143437$, $m = 409593$.

6.6 Weitere Verfahren

Bei den im Folgenden kurz vorgestellten Verfahren handelt es nicht um Multilevel-Algorithmen. In [26] werden Methoden vorgestellt, welche zur Beschleunigung der Stressmajorisierung auf der Grafikkarte dienen. Da diese Methoden jedoch alle paarweisen Distanzen beachten, ist die Laufzeit deutlich höher als bei den anderen vorgestellten Verfahren (einschließlich der kombinierten MDS). Es werden nur Laufzeiten für Graphen bis zu einer Größe von 2048 Knoten und 10166 Kanten angegeben. Die beste angegebene Laufzeit beträgt für diesen größten Graphen 15,43 Sekunden. Die Layouts dürften eine sehr gute Qualität aufweisen; [26] enthält jedoch keine Beispiele. Aus diesen Gründen behandelt diese Arbeit das Paper [26] nicht genauer.

Das Paper [23] stellt einen parallelen Graphlayout-Algorithmus vor, der nicht die Grafikkarte, sondern viele CPU-Kerne als parallele Hardware benutzt. Dabei wird eine auf dem Teilchen-Modell basierende Layoutfunktion verwendet. Der Algorithmus teilt die Knoten auf die zur Verfügung stehenden Prozessoren so auf, dass die Summe der Knotengrade pro Prozessor gleich ist, was zu einer guten Lastverteilung (load balancing) führt. Außerdem wird bei der Verteilung auf Lokalität geachtet. Nahe beieinander liegende Knoten werden dem gleichen Prozessor zugeteilt. In jeder Iteration der Layoutfunktion berechnen die Prozessoren zunächst die Kräfte, die zwischen den ihnen zugeordneten Knoten wirken (lokale Kräfte), danach berechnen sie die "globalen Kräfte". Alle paarweisen Kräfte werden dabei nur in den ersten Iterationen bestimmt. In späteren Iterationen berechnen die Prozessoren zwar alle paarweisen lokalen Kräfte, die globalen Kräfte aber werden nur noch zwischen "benachbarten Prozessoren" berechnet. Die Kräfte, die von Knoten anderer Prozessoren ausgehen, werden ignoriert. Dies spart Laufzeit, da die Interprozessorkommunikation, wie bei der Grafikkarte, deutlich ineffizienter ist als der Informationsaustausch innerhalb eines Prozessors. Abbildung 16(a) zeigt, dass das Layout von add32 zwar gut ist, bei 4elt sieht man jedoch den Nachteil der Methode. Das Ignorieren der Kräfte, die von sämtlichen sich weiter weg befindlichen Knoten ausgehen, führt dazu, dass die globale Struktur des Graphen nicht mehr erkennbar ist. Bei 4elt sieht man z.B. die "Löcher" (s. Abbildung 5.20) nicht mehr.

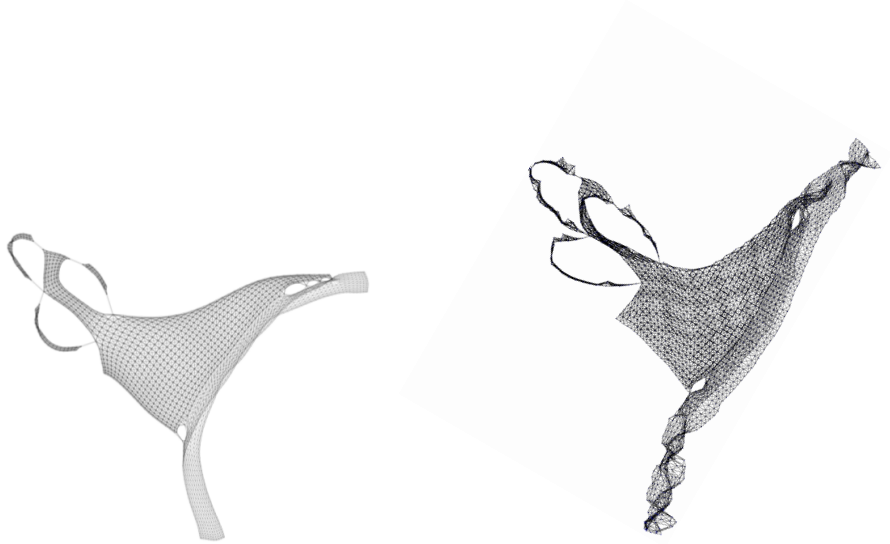
Eine Idee ist, nur die paarweisen lokalen Kräfte zu berechnen und sämtliche Knoten anderer Prozessoren jeweils als Gesamtkraft wirken zu lassen, ähnlich wie die Partitionen bei MGLG und RMGD. Der Hauptgrund, warum [23] nicht genauer behandelt wird, ist jedoch die benötigte Hardware. Die Berechnung des Layouts von 4elt ist bei einer Anzahl von 256 Prozessorkernen in etwa so schnell (0,81 s) wie die kombinierte MDS auf einer einfachen Nvidia 250GTS Grafikkarte. Außerdem ist die globale Struktur bei der größeren Prozessorzahl noch schlechter zu erkennen.



(a) add32 (b) 4elt

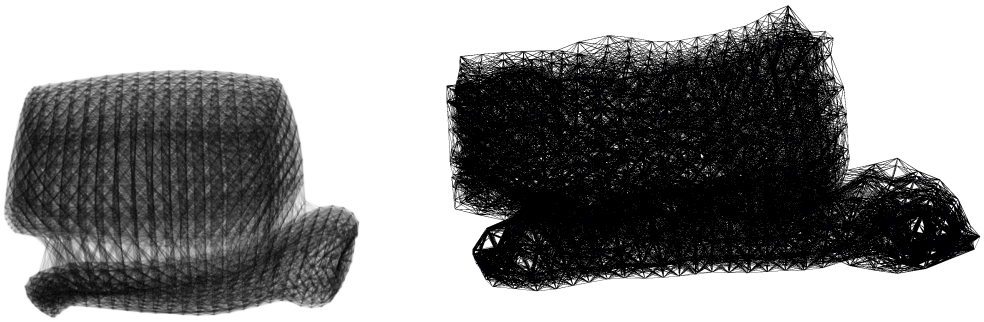
Abbildung 6.16: Layouts aus [23] mit 32 Prozessoren.

Die Diplomarbeit [11] stellt eine parallele Implementierung der Fast-Multipole-Multilevel-Methode (PFM3) für CPUs vor, deren Laufzeiten und generierte Layouts sehr gut sind. Nach Erhalt von [11] war es leider für eine Beschreibung der in [11] enthalten Implementierung zu spät. Zur Laufzeitmessung verwendet [11] ein Dual Socket Intel Xeon E5430 System. Die Abbildungen 6.17 bis 6.24 bis vergleichen die Ergebnislayouts mit denen der kombinierten MDS. Die Laufzeitmessungen für die kombinierte MDS wurden auf dem System bestehend aus AMD Phenom II X4 850 @ 3,3 GHz CPU und Nvidia GTX 550 Ti GPU durchgeführt.



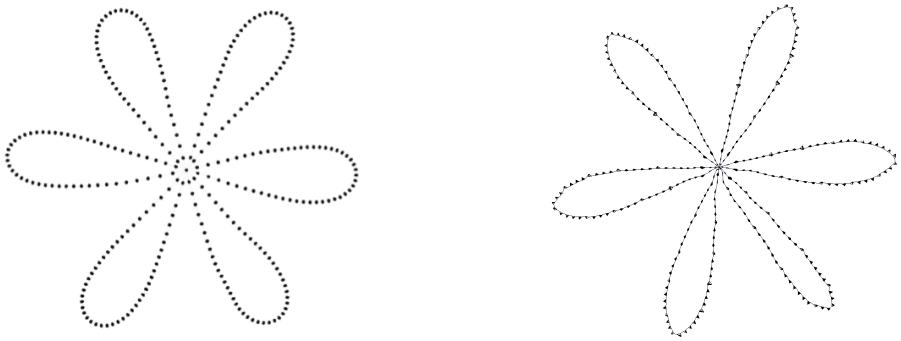
(a) PFM3, Laufzeit: 0,234 s (b) kombinierte MDS, Laufzeit: 0,109 s

Abbildung 6.17: Layouts für data. $n = 2851$, $m = 15093$.



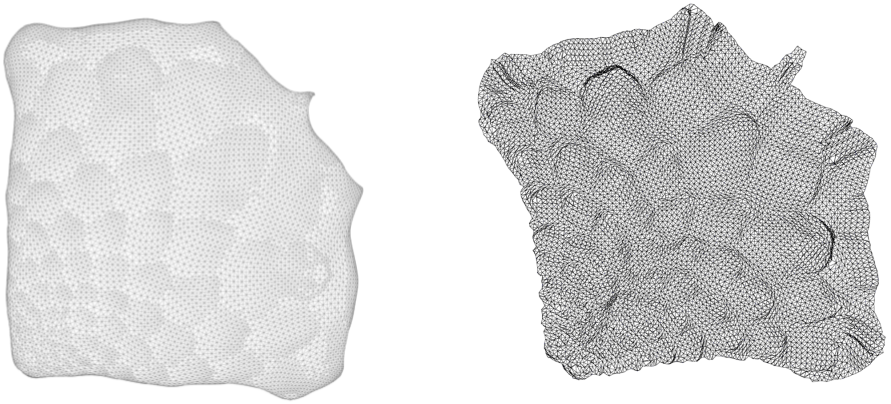
(a) PFM3, Laufzeit: 0,640 s (b) kombinierte MDS, Laufzeit: 0,235 s

Abbildung 6.18: Layouts für bcsstk33. $n = 8738$, $m = 291583$.



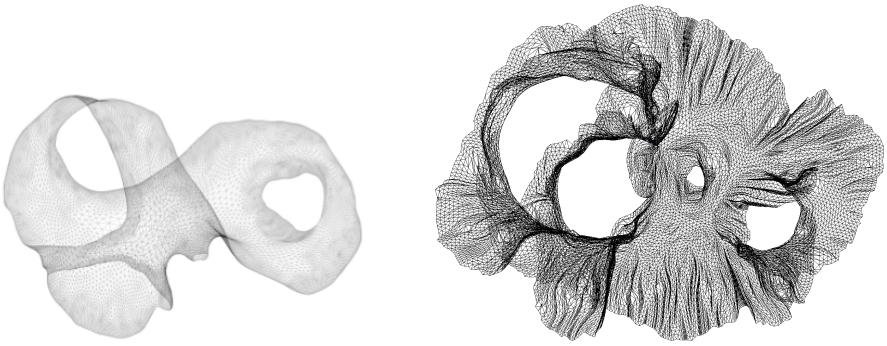
(a) PFM3, Laufzeit: 0,279 s (b) kombinierte MDS, Laufzeit: 0,329 s

Abbildung 6.19: Layouts für flower_050. $n = 9030$, $m = 131241$.



(a) PFM3, Laufzeit: 0,365 s (b) kombinierte MDS, Laufzeit: 0,191 s

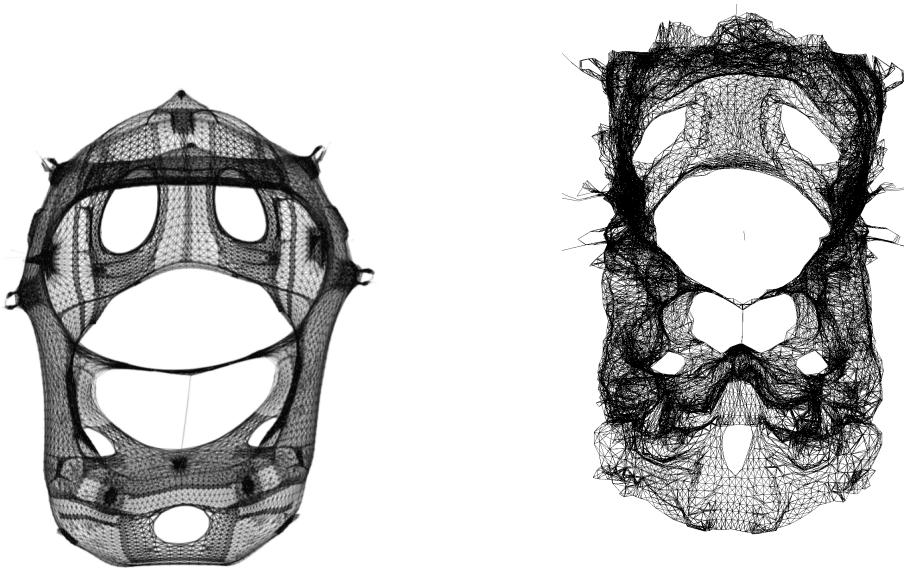
Abbildung 6.20: Layouts für crack. $n = 10240$, $m = 30380$.



(a) PFM3, Laufzeit: 0,362 s

(b) kombinierte MDS, Laufzeit: 0,260 s

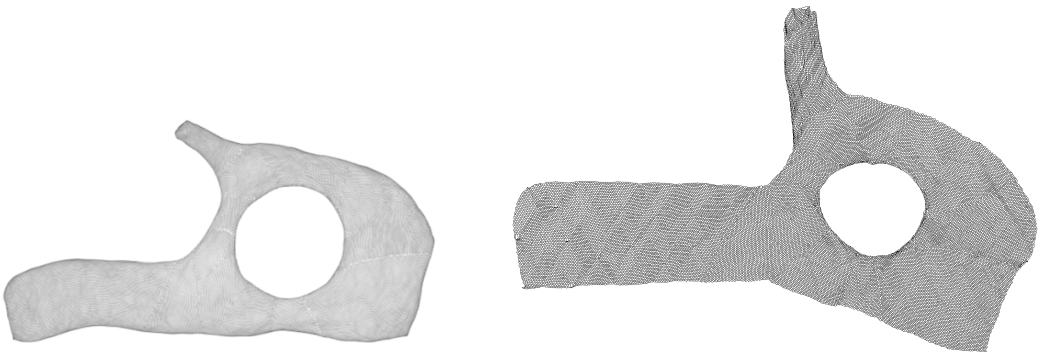
Abbildung 6.21: Layouts für 4elt. $n = 15606$, $m = 45878$.



(a) PFM3, Laufzeit: 0,977 s

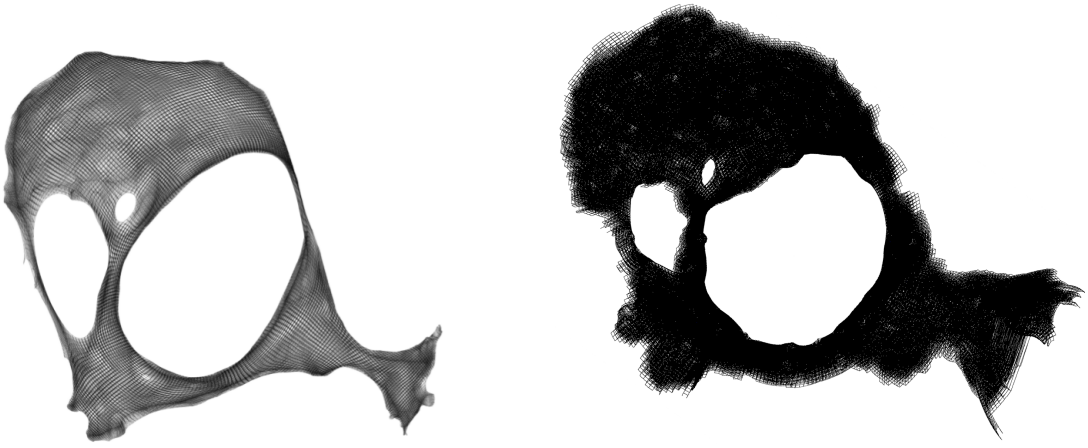
(b) kombinierte MDS, Laufzeit: 0,638 s

Abbildung 6.22: Layouts für bcsstk31. $n = 35588$, $m = 572914$.



(a) PFM3, Laufzeit: 0,836 s (b) kombinierte MDS, Laufzeit: 0,775 s

Abbildung 6.23: Layouts für t60k. $n = 60005$, $m = 89440$.



(a) PFM3, Laufzeit: 2,092 s (b) kombinierte MDS, Laufzeit: 2,087 s

Abbildung 6.24: Layouts für fe_ocean. $n = 143437$, $m = 409593$.

7 Zusammenfassung und Ausblick

Die im Rahmen dieser Masterarbeit behandelte kombinierte MDS, welche Pivot-MDS mit der ausgedünnten Stressmajorisierung verbindet, nutzt die Vorteile dieser beiden Verfahren, ist laufzeiteffizient und liefert beim Graphenzeichnen hochwertige Ergebnisse, was die Laufzeitangaben und Ergebnislayouts zeigen (s. auch [5]). Es wurde eine eigene Implementierung der kombinierten MDS geschaffen, welche deren Teile, Distanzberechnung, Pivot-MDS und ausgedünnte Stressmajorisierung, parallelisiert und dafür die Hardwarebeschleunigung der Grafikkarte mit Hilfe von CUDA nutzt. Dafür wurden die Stärken verschiedener Implementierungsmöglichkeiten gegeneinander abgewogen. Das eigens eingeführte Abbruchkriterium der ausgedünnten Stressmajorisierung nutzt während der zuvor durchgeführten Pivot-MDS generierte Informationen, um mit sehr geringem Laufzeitaufwand die Anzahl der Iterationen an den jeweiligen Graphen anzupassen. Weiter können mit Hilfe des eigens entwickelten Visualisierungsprogramms dreidimensionale Layouts von Graphen angezeigt werden, in denen man mittels dieses Visualisierungsprogramms auch navigieren kann.
Wie [5] zeigt, ist die Stressmajorisierung geeigneter als Varianten der Multipolmethode, wie sie [7] und [10] zugrunde liegen, wenn es darum geht, die Knoten eines Graphen so zu zeichnen, dass ihre euklidischen Abstände in der Zeichnung möglichst genau ihren Knotendistanzen entsprechen. Die von der kombinierten MDS generierten Layouts sind daher geeigneter als auf Varianten der Multipolmethode aufbauende Methoden um dem in der Einleitung formulierten Wunsch nach einem Layout, welches die paarweisen Knotendistanzen möglichst gut repräsentiert, nachzukommen.
Die Hardwarebeschleunigung durch die Grafikkarte sorgt für gute Laufzeiten der Implementierung. Zukünftige Arbeiten könnten Hardwarebeschleunigung in noch höherem Maße nutzen. Bei der Bestimmung der Distanzen ist es möglich, die Distanzen zu den Pivotknoten gleichzeitig mit den Distanzen zur aktiven Nachbarschaft zu berechnen. Dies ist zum einen durch das Feature der Fermi-Grafikkartengeneration, mehrere Kernelfunktion gleichzeitig starten zu können, was den Belegungsgrad der Grafikkarte weiter steigern würde, möglich, zum anderen durch die Verwendung zweier Grafikkarten. Für beide Möglichkeiten ist es nötig, mehrere CPU-Threads zu verwenden, was zusätzlich mehrere Kerne der CPU nutzten würde. Ferner soll die Implementierung für die neueste Grafikkartengeneration optimiert werden um eine weitere Laufzeitverbesserung zu erhalten.
In Bezug auf die Visualisierung soll das Programm um die Möglichkeit der Anzeige von dreidimensionalen Graphenlayouts auf stereoskopen Bildschirmen erweitert werden.

8 Anhang

8.1 Visualisierungsprogramm

Im Rahmen dieser Arbeit wurde ein eigenes Visualisierungsprogramm mittels OpenGL erstellt. Dieses ist in der Lage, sowohl die generierten zweidimensionalen als auch dreidimensionalen Layouts anzuzeigen. Die Implementierung der kombinierten MDS ist im Visualisierungsprogramm integriert, sodass es möglich ist mit einem Befehl das Layout zu berechnen und anschließend anzuzeigen. Angenommen das Programm wurde unter dem Namen layoutgraph kompiliert, lässt sich die Zeichnung eines Graphen dessen Adjazenzlisten in der Datei graph_file.cvs enthalten sind durch den Aufruf

```
./layoutgraph graph_file.cvs
```

berechnen und anzeigen. Fügt man hinter den Aufruf eine 3, so wird ein dreidimensionales Layout des Graphen berechnet und angezeigt. Das dabei verwendete Format für die Graphdatei enthält die Kopfzeile

```
<Knotenzahl> :: <Kantenzahl>,
```

welche die Anzahl der Knoten und Kanten enthält. Ohne Leerzeile folgen die Adjazenzlisten der Knoten, wobei die Zeilen $2, \ldots, n+1$ die Adjazenzlisten der Knoten $v_0, \ldots, v_{n-1}$ enthalten. Eine Adjazenzliste besteht aus den Knoten-IDs der enthaltenen Knoten, welche durch Semikola getrennt sind. Die entsprechenden Graphdateien der in Tabelle 5.1 enthaltenen Graphen befinden sich ebenfalls auf der beiliegenden DVD.

8.2 Beiliegende DVD

Auf der beiliegenden DVD finden sich der Quellcode der Implementierung der kombinierten MDS, der Quellcode des Visualisierungsprogramms, Graphdateien der in Tabelle 5.1 enthaltenen Graphen sowie Videos, welche durch das Visualisieungsprogramm angezeigte dreidimensionale Layouts der in dieser Arbeit verwendeten Beispielgraphen demonstrieren.

Literatur

[1] *Compute Visual Profiler User Guide*. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/Compute_Visual_Profiler_User_Guide.pdf.

[2] *CUDA C Best Practices Guide*. http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf.

[3] *Nvidia CUDA C Programming Guide*. http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf.

[4] Ulrik Brandes and Christian Pich. Eigensolver Methods for Progressive Multidimensional Scaling of Large Data, 2007.

[5] Ulrik Brandes and Christian Pich. An Experimental Study on Distance-Based Graph Drawing. 2009.

[6] Fan R. K. Chung. Spectral graph theory. *Regional Conference Series in Mathematics, American Mathematical Society*, Volume 92, 1997.

[7] Yaniv Frishman and Ayellet Tal. Rapid Multipole Graph Drawing on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, Volume 13 Issue 6, 2007.

[8] Thomas M. J. Fruchterman and Edward M. Reingold. Graph Drawing by Force-directed Placement. *Software—Practice and Experience*, Volume 21 Issue 11, 1991.

[9] Emden R. Gansner, Yehuda Koren, and Stephen North. Graph drawing by stress majorization. In *GRAPH DRAWING*, pages 239–250. Springer, 2004.

[10] Apeksha Godiyal, Jared Hoberock, Michael Garland, and John C. Hart. Rapid Multipole Graph Drawing on the GPU. *Lecture Notes in Computer Science*, 5417/2009, 2009.

[11] Martin Gronemann. Engineering the Fast-Multipole-Multilevel Method for multicore and SIMD architectures. Master's thesis, Technische Univertität Dortmund, 2009.

[12] Stefan Hachul and Michael J. Unger. Large-Graph Layout with the Fast Multipole Multilevel Method. *Technical report, Zentrum für Angewandte Informatik Köln*, 2005.

[13] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. *HiPC'07 Proceedings of the 14th international conference on High performance computing*, 2007.

[14] Pawan Harish, Vibhav Vineet, and P. J. Narayanan. Large Graph Algorithms for Massively Multithreaded Architectures. 2009.

[15] Mark Harris, Shubhabrata Sengupta, and John D. Owens. *GPU Gems 3*. Addison-Wesley, http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html, 2007.

[16] Stephen Ingram. Multilevel Multidimensional Scaling on the GPU. Master's thesis, The University Of British Columbia, 2007.

[17] Stephen Ingram, Tamara Munzner, and Marc Olano. Glimmer: Multilevel MDS on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, Volume 15 Issue 2, 2009.

[18] Tomihisa Kamada and Saturo Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, Volume 31, 1989.

[19] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors*. mkp, 2010.

[20] Lijuan Luo, Martin D. F. Wong, and Wen mei Hwu. An effective GPU implementation of breadth-first search. In *DAC'10*, pages 52–55, 2010.

[21] Duane Merrill, Michael Garland, and Andrew Grimshaw. High Performance and Scalable GPU Graph Traversal. *Technical Report*, CS-2011-05, 2011.

[22] Christian Pich. *Applications of Multidimensional Scaling to Graph Drawing*. PhD thesis, Univertität Konstanz, 2009.

[23] Anna Tikhonova and Kwan-Liu Ma. A Scalable Parallel Force-Directed Graph Layout Algorithm. *Eurographics Symposium on Parrallel Graphics and Visualization*, 2008.

[24] Warren S. Torgerson. Multidimensional scaling: I. Theory and method. *Psychometrika*, 1952.

[25] R. von Mises and H. Pollaczek-Geiringer. Praktische Verfahren der Gleichungsauflösung. In *ZAMM - Zeitschrift für Angewandte Mathematik und Mechanik*, volume 9, pages 152–164, 1929.

[26] Wang Yong-Xian, Li Zong-Zhe, Yao Lu, Cao Wei, and Wang Zheng-Hua. Two Improved GPU Acceleration Strategies for Force-Directed Graph Layout. *Computer Application and System Modeling (ICCASM)*, Volume 13, 2010.