UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

Experimentelle Analyse von Algorithmen zur Lösung des Bisektionsproblems in Graphen

Experimental analysis of algorithms solving the graph bisection problem

Bachelorarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Marcel Wienöbst

ausgegeben und betreut von
Prof. Dr. Maciej Liśkiewicz

mit Unterstützung von
MSc. Martin Schuster

Lübeck, den 21. Oktober 2016

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Quellen und Hilfsmittel angefertigt zu haben.

Lübeck, den 21. Oktober 2016

Zusammenfassung

In dieser Arbeit werden verschiedene Ansätze zur Lösung des NP-vollständigen Bisektionsproblems experimentell auf ihre Güte und Laufzeit untersucht. Dabei werden exakte Algorithmen sowie Heuristiken betrachtet. Im Zentrum steht der Vergleich einer Greedy-Strategie mit einem Spektralansatz. Beide Verfahren sind von großer praktischer Bedeutung und haben unterschiedliche Stärken und Schwächen. Die Algorithmen wurden in der Programmiersprache Python implementiert. Mithilfe von experimentellen Tests auf verschiedenen Graphklassen wird ein genauer Vergleich der Heuristiken ermöglicht. Diese Graphklassen wurden speziell zur Untersuchung der Algorithmen gewählt. Es wurden auch verschiedene zufällige reguläre Graphen als Testgrundlage verwendet. Für die Generierung dieser Graphen wurde ein von Steger und Wormald vorgeschlagenes Verfahren mit einer Laufzeit von $\mathcal{O}(nd^2)$ implementiert.

Abstract

This thesis deals with different approaches to the NP-complete bisection problem. The quality as well as the running time of those approaches are tested experimentally. Exact algorithms and heuristics are considered. The main focus is a comparison of a greedy and a spectral strategy. Both algorithms have been heavily used in practice and have their own strengths and weaknesses. All algorithms were implemented in the programming language Python. With experimental tests on various graph classes the performances of the algorithms are compared extensively. Those graph classes have been chosen specifically for the discussed algorithms. In particular the performance on different random regular graphs is investigated. In order to generate these graphs an approach introduced by Steger and Wormald with a running time of $\mathcal{O}(nd^2)$ has been implemented.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Beiträge und Ergebnisse dieser Arbeit . . . 2
- 1.2 Stand der Forschung . . . 3
- 1.3 Aufbau der Arbeit . . . 3

2 Grundlagen **4**

3 Exakte Verfahren **5**
- 3.1 Brute-Force . . . 5
- 3.2 Lineare Programmierung . . . 6

4 Heuristiken **10**
- 4.1 Greedy . . . 10
- 4.2 Breitensuche . . . 14
- 4.3 Breitensuche und Greedy . . . 16
- 4.4 Eigenwertansatz . . . 17

5 Generierung von zufälligen Graphen **22**
- 5.1 Zufällige reguläre Graphen . . . 22
- 5.2 Zufällige reguläre Graphen mit geringer Bisektionsweite . . . 25
- 5.3 Bipartite Graphen . . . 27
- 5.4 Kubische Kreisgraphen . . . 29

6 Resultate auf zufälligen Graphen **32**
- 6.1 Zufällige reguläre Graphen . . . 32
- 6.2 Zufällige reguläre Graphen mit geringer Bisektionsweite . . . 33
- 6.3 Bipartite Graphen . . . 35
- 6.4 Kubische Kreisgraphen . . . 36
- 6.5 Zusammenfassung . . . 36

7 Deterministische Graphen **38**
- 7.1 Leiter . . . 38
- 7.2 2D-Gitter . . . 40
- 7.3 2D-Torus . . . 42
- 7.4 Rook-Graph . . . 44
- 7.5 King-Graph . . . 46
- 7.6 Hypercube . . . 47
- 7.7 Butterfly-Netzwerk . . . 49
- 7.8 Cockroach-Graph . . . 50
- 7.9 Verbundene Binärbäume . . . 51
- 7.10 Comb-Graph . . . 53
- 7.11 Zusammenfassung . . . 54

8 Fazit und Ausblick **56**

1 Einleitung

Das Problem der Graph-Partitionierung beschreibt im Allgemeinen eine Aufteilung der Knoten in mindestens zwei Teilmengen. Diese Aufteilung soll derartig sein, dass die Anzahl an Kanten, welche Knoten aus verschiedenen Partitionen verbinden, minimiert wird. Darüber hinaus wird verlangt, dass die Partitionen von gleicher Größe sind. Ein Spezialfall der Graph-Partitionierung ist die Bisektion eines Graphen, also eine Partitionierung der Knoten in zwei Teilmengen. Die minimale Anzahl von Kanten zwischen den Partitionen über alle balancierten Bisektionen wird auch *Bisektionsweite* genannt. Die Berechnung dieser Bisektionsweite ist ein NP-vollständiges Problem für allgemeine Graphen [1], reguläre Graphen [2] und bipartite Graphen [3]. Darüber hinaus approximieren die besten bekannten Approximationsalgorithmen mit polynomieller Laufzeit die Bisektionsweite lediglich auf einen Faktor von $\mathcal{O}(\log^2(|V|))$ [4]. Solche Bisektionsweiten sind aus Anwendungssicht für praktisch geeignete Graphgrößen jedoch nicht ausreichend. In der Praxis werden daher häufig Heuristiken verwendet.

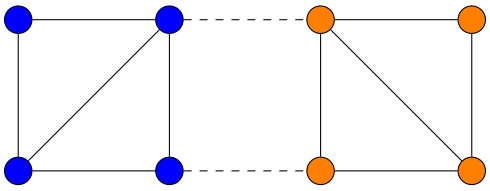

Abbildung 1: Beispielgraph mit eingezeichneter minimaler Bisektion

In Abbildung 1 ist eine optimale Bisektion für einen Beispielgraphen dargestellt. Die blauen Knoten sowie die orangefarbenen bilden jeweils eine Partition. Die gestrichelten Linien geben die Kanten zwischen diesen Partitionen an. Die Bisektionsweite dieses Beispielgraphen beträgt 2.

Das Bisektionsproblem kann einerseits als Berechnung der Bisektionsweite eines Graphen und andererseits als Berechnung der optimalen Bisektion definiert werden. In dieser Arbeit werden ausschließlich Algorithmen vorgestellt, welche eine Bisektion finden und zurückgeben. Da auch Heuristiken betrachtet werden, ist diese Bisektion jedoch nicht zwangsläufig die optimale.

Das Problem der Graph-Partitionierung ist von großer praktischer Bedeutung. In Kapitel 3 des Surveys [5] werden einige Anwendungen vorgestellt. Graphen modellieren viele verschiedene Anwendungsprobleme und sind somit die Grundlage für eine ganze Reihe von Algorithmen und deren praktischer Nutzung. Dabei ist die Partitionierung von Graphen eine fundamentale algorithmische Operation.

Eine kanonische Anwendung der Graph-Partitionierung ist die Verteilung von Aufgaben auf Prozessoren bei einer Parallelverarbeitung. Dabei soll eine Partitionierung sowohl die Arbeit etwa zu gleichen Maßen verteilen als auch die benötigte Kommunikation zwischen den Prozessoren minimieren. Mit steigender Nutzung von paralleler Verarbeitung steigt also auch die Nutzung von Bisektions- und Partitionierungsalgorithmen.

Eine weitere klassische Anwendung der Graph-Partitionierung ist im Design von Digitalschaltungen für VLSI (very large-scale integration) Systeme zu finden. Dabei soll die Partitionierung die Komplexität des Designs durch eine Aufteilung in kleinere Komponenten reduzieren. Darüber

hinaus soll die Gesamtlänge der Kabel so gering wie möglich bleiben. Daher wird hier häufig als Minimierungsfunktion die Anzahl von Verbindungen zwischen den Teilschaltungen gewählt. Im Survey [6] aus dem Jahr 1991 werden verschiedene Algorithmen zur Platzierung von VLSI-Zellen verglichen. Dabei ist die Strategie der Graph-Partitionierung mit minimalem Cut das kosteneffizienteste Verfahren. Heutzutage werden jedoch vermehrt analytische Ansätze bevorzugt, was neben dem algorithmischen Fortschritt auch mit einer Veränderung der Probleminstanzen zu begründen ist [7].

Weitere Anwendungen von Partionierungen und Bisektionen von Graphen gibt es in den Bereichen der Bildverarbeitung bzw. -segmentierung sowie komplexen Netzwerken und sogar in der Routenplanung auf Straßennetzen.

Es kann davon ausgegangen werden, dass verschiedene Arten von Graph-Partitionierung vor allem im Bereich der Parallelverarbeitung und somit auch das Bisektionsproblem in Zukunft weiter an praktischer Bedeutung gewinnen, da immer größere Instanzen von Graphen verarbeitet werden müssen.

1.1 Beiträge und Ergebnisse dieser Arbeit

In dieser Arbeit werden verschiedene Algorithmen vorgestellt, welche für einen gegebenen Graphen eine Bisektion mit möglichst geringem Cut finden. Diese Verfahren wurden dabei in der Programmiersprache Python implementiert, um diese durch experimentelle Tests zu vergleichen. Dabei werden einerseits exakte Verfahren betrachtet und in ihrer Laufzeit verglichen. Es gibt verschiedene Strategien algorithmisch eine optimale Bisektion zu finden und es werden sowohl ein naiver Brute-Force-Ansatz als auch zwei Formulierungen als lineares Optimierungsproblem betrachtet. Eine dieser beiden Instanzen als Optimierungsproblem ist nur für reguläre Graphen anwendbar und der schnellste der betrachteten exakten Algorithmen. Andererseits werden vier Heuristiken vorgestellt und neben der Laufzeit auch auf ihre Güte untersucht. Im Kern geht es dabei um den Vergleich einer Greedy-Heuristik mit einem Eigenwertansatz. Beide Verfahren werden in verschiedenen Varianten in der Literatur [5] diskutiert und sind von großer Bedeutung in der Praxis. Die in dieser Arbeit durchgeführten Tests sollen die Heuristiken einordnen, um dadurch je nach Anwendungsfall die Wahl der bestmöglichsten Heuristik in der Praxis zu ermöglichen. Für den Vergleich dieser Verfahren werden sowohl zufällige als auch deterministische Graphen betrachtet. Ein Teil dieser Arbeit beschäftigt sich daher mit der Generierung von zufälligen Graphen. Dabei wird ein von Steger und Wormald in [8] zur Generierung von zufälligen regulären Graphen vorgeschlagener Algorithmus betrachtet und für ähnliche Graphklassen abgewandelt. Dieser Algorithmus wurde auch in Python implementiert und dient daher als Grundlage für die experimentellen Tests. In diesen hat sich gezeigt, dass die Performance der Verfahren in hohem Maße von der betrachteten Graphklasse abhängig ist. Kein Verfahren kann als das allgemein beste bezeichnet werden. Die Erkenntnisse dieser Arbeit sind daher umso wichtiger für die Wahl der richtigen Heuristik in einer praktischen Anwendung.

1.2 Stand der Forschung

In der Forschung zum Bisektionsproblem werden zumeist verschiedene Heuristiken untersucht. Viele dieser Heuristiken arbeiten entweder nach dem Prinzip der iterativen Verbesserung, dies sind Greedy-Verfahren, oder nutzen die Eigenwerte bzw. Eigenvektoren von dem Graphen zugehörigen Matrizen.

Dabei sind Greedy-Heuristiken eine klassische praktische Wahl für die Graph-Partitionierung. Der Aufbau dieser Verfahren ist meist ähnlich. Es wird mit einer gewissen balancierten Bisektion gestartet und diese durch den Tausch von Knoten iterativ verbessert. Bereits 1970 stellten Kernighan und Lin in [9] ein Verfahren vor, welches für einen langen Zeitraum im VLSI-Design eingesetzt wurde [6]. Im Jahr 1982 verbesserten Fiduccia und Mattheyses in [10] die Laufzeit dieses Ansatzes auf $\mathcal{O}(|E|)$. Beide Verfahren tauschen in jedem Verbesserungsschritt einzelne Knoten der beiden Partitionen und starten mit einer zufälligen Bisektion. In dem Paper [11] von Jäger wird neben verschiedenen Greedy-Verfahren auch der Einfluss der Wahl der Startbisektion diskutiert. Die dort durchgeführten Tests zeigten eine deutliche Verbesserung der Performance durch die Wahl dieser. In dieser Bachelorarbeit wird daher ein Greedy-Verfahren vorgestellt und untersucht, welches ebenfalls in jeder Iteration einzelne Knoten austauscht. Darüber hinaus wird ausführlicher als in dem Paper von Jäger die Wahl einer Startbisektion diskutiert.

Ein anderer Ansatz für das Finden einer guten Bisektion ist die Verwendung von Eigenwerten und Eigenvektoren. Dabei wird mithilfe des Graphen eine Matrix konstruiert, für die das Eigenwertproblem gelöst wird. Anhand von bestimmten Eigenvektoren werden die Knoten des Graphen partitioniert. Donath und Hoffman waren die ersten die in [12] solche Ansätze vorstellten. Diese wurden im Folgenden weiter untersucht und verbessert, so z.B. in [13]. Dass diese Verfahren auch für simple Graphen schlechte Ergebnisse liefern können, wird in [14] gezeigt.

1.3 Aufbau der Arbeit

In Kapitel 2 werden die Grundlagen für die weitere Arbeit etabliert, indem das Bisektionsproblem und zugehörige Begriffe formal definiert werden. In Kapitel 3 werden drei Ansätze vorgestellt und in ihrer Laufzeit verglichen, welche das Bisektionsproblem exakt lösen. Neben einem naiven Ansatz werden zwei Formulierungen als lineares Optimierungsproblem formuliert. Das Kapitel 4 behandelt verschiedene Heuristiken. Es wird sowohl ein Greedy-Verfahren als auch ein Eigenwertansatz vorgestellt. In Kapitel 5 wird die Generierung verschiedener Klassen von zufälligen regulären Graphen diskutiert. Dazu wird ein Algorithmus von Steger und Wormald vorgestellt. Kapitel 7 beschreibt verschiedene deterministische Graphklassen. Eine Implementierung von Graphgeneratoren für die verschiedenen Graphklassen ermöglicht eine experimentelle Analyse der Heuristiken. In Kapitel 8 werden die vorherigen Ergebnisse zusammengefasst und die Verfahren miteinander verglichen. Darüber hinaus werden offene Fragen angesprochen.

2 Grundlagen

Es werden zunächst einige grundlegende Begriffe und Notationen eingeführt, welche in der Arbeit verwendet werden.

Sei $G = (V, E)$ ein ungerichteter und ungewichteter Graph mit der Knotenmenge V und der Kantenmenge E. In der Arbeit wird die Kardinalität der Knotenmenge als $n := |V|$ und die der Kantenmenge als $m := |E|$ bezeichnet. Eine wichtige Graphklasse, welche in dieser Arbeit von großer Bedeutung ist, sind sogenannte reguläre Graphen.

Definition 1. Ein Graph wird d-regulär genannt, wenn für alle $v \in V$ gilt, dass $|\{w \in V \mid \{v, w\} \in E\}| = d$ ist. Oder anders ausgedrückt, jeder Knoten in G hat genau d Nachbarn. 3-reguläre Graphen werden auch als kubische Graphen bezeichnet.

Der Begriff der Bisektion wurde bereits in der Einleitung intuitiv eingeführt und wird nun mit zugehörigen Begrifflichkeiten definiert.

Definition 2. Eine *Bisektion* von G sei eine Abbildung $\pi : V \to \{0, 1\}$ und ordnet die Knoten den Mengen V_0 und V_1 zu.

Definition 3. Ein *Cut* sei die Menge der Kanten zwischen den Partitionen und ist als $\text{cut}(\pi) := \{\{v, w\} \in E \mid \pi(v) \neq \pi(w)\}$ definiert. Die Größe eines Cuts sei dann die Kardinalität dieser Menge.

Definition 4. Sei eine *balancierte* Bisektion eine solche, bei der $||V_0| - |V_1|| \leq 1$ gilt. Dann ist die *Bisektionsweite* eines Graphen G als $b(G) := \min(|\text{cut}(\pi)| \mid \pi$ ist eine balancierte Bisektion von $G)$ definiert.

Es gibt zwei verschiedene Arten das Bisektionsproblem zu definieren, die bereits in der Einleitung unterschieden wurden, und im Folgenden dargestellt sind. Beide Probleme sind NP-schwer und wenn in dieser Arbeit im Folgenden vom Bisektionsproblem die Rede ist, so ist das SEARCHING Problem gemeint.

BISECTION (OPT)
Eingabe: Ein Graph $G = (V, E)$
Fragestellung: Finde die Bisektionsweite des Graphen G.

BISECTION (SEARCHING)
Eingabe: Ein Graph $G = (V, E)$
Fragestellung: Finde die optimale Bisektion des Graphen G, das heißt die balancierte Bisektion mit dem minimalen Cut.

3 Exakte Verfahren

Die bekannten Verfahren, welche das Bisektionsproblem exakt lösen, haben eine exponentielle Laufzeit. In den folgenden Abschnitten wird diskutiert, inwieweit es dennoch möglich ist in der Praxis mit realisierbarem Zeitaufwand eine optimale Bisektion für kleinere Graphen zu finden. Dazu werden drei verschiedene Strategien zur Lösung des Bisektionsproblems vorgestellt.

3.1 Brute-Force

Ein naiver Ansatz zur Lösung des Bisektionsproblems ist es, für alle möglichen Partitionen die Bisektionsweite zu berechnen und das Minimum auszugeben. Auf diese Art und Weise wird garantiert die optimale Bisektion gefunden. Die Partitionen können durch die Teilmengen von V der Kardinalität $n/2$ beschrieben werden. Eine solche Teilmenge beschreibt eine der beiden Partitionen und die andere ergibt sich aus den übrigen Knoten. Sei S eine Menge, dann bezeichnet $[S]^k$ im Folgenden die Menge der k-elementigen Teilmengen von S, also $[S]^k = \{A \subseteq S \mid |A| = k\}$. Für das Bisektionsproblem beschreibt die Menge $[V]^{n/2}$ folglich alle möglichen Partitionen und es gilt $|[V]^{n/2}| = \binom{n}{\lfloor n/2 \rfloor}$. Das bedeutet, dass ein naiver Brute-Force-Algorithmus $\binom{n}{\lfloor n/2 \rfloor}$ viele Partitionen betrachten muss. Mithilfe der Gleichung, welche zum Beispiel in [15] hergeleitet wird,

$$\binom{n}{k} \leq \frac{en}{k}^k,$$

wobei e die Eulersche Zahl ist, kann eine obere Schranke für diese Anzahl angegeben werden. Damit folgt:

$$\binom{n}{n/2} \leq \frac{en}{n/2}^{n/2} = (2e)^{n/2}$$

In Algorithmus 1 wird der Brute-Force-Ansatz als Pseudocode beschrieben.

Algorithmus 1 : Brute-Force-Algorithmus

input : Ein ungerichteter Graph $G = (V, E)$
output : Eine Teilmenge von V der Kardinalität $n/2$, welche die optimale Bisektion repräsentiert

```
1 min ← ∞
2 best ← NIL
3 foreach subset ∈ [V]^{n/2} do
4 |   if bisectionwidth (G, subset) < min then
5 |   |   min ← bisectionwidth (G, subset)
6 |   |   best ← subset
7 |   end
8 end
9 return subset
```

In Zeile 3 wird über $[V]^{n/2}$ iteriert. Dies ist in der Programmiersprache Python mit der Funktion `itertools.combinations` aus der Standardbibliothek möglich, welche die Menge aller Teilmengen

einer gewissen Kardinalität generiert [16]. Diese Funktion ist in der Programmiersprache C implementiert und bietet eine sehr optimierte Generierung dieser Mengen. Es kann davon ausgegangen werden, dass die Komplexität dieser Funktion in $\mathcal{O}(n/2 \cdot (2e)^{n/2})$ liegt. Für jede dieser Teilmengen wird nun die Bisektionsweite mit einer Funktion `bisectionwidth` berechnet, wobei $\mathcal{O}(m)$ viele Kanten betrachtet werden. Die Implementierung dieser Funktion ist unkompliziert und wird daher nicht weiter diskutiert. In den Variablen `min` und `best` das bisherige Minimum sowie die dazugehörige Teilmenge gespeichert. Letztere wird nach Beendigung der Schleife als optimale Bisektion zurückgegeben. Insgesamt ergibt sich also eine Laufzeit von $\mathcal{O}\left((2e)^{n/2} \cdot (n+m)\right)$.

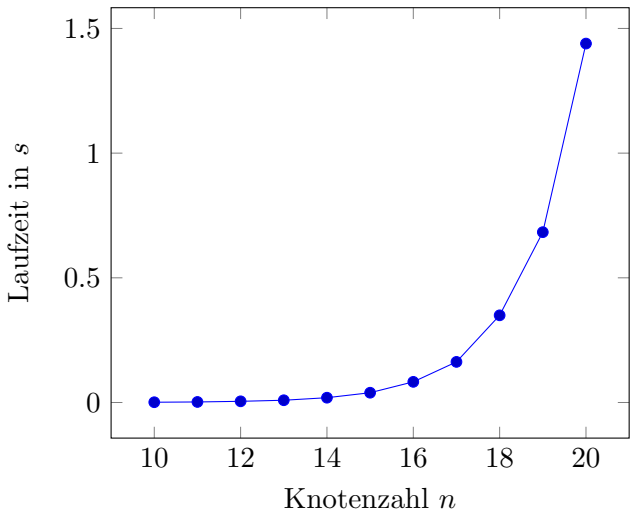

Abbildung 2: Laufzeit des Bruteforce-Ansatzes auf kubischen Graphen

Mithilfe der oben erwähnten Funktion `itertools.combinations` habe ich ein Python-Programm geschrieben, welches den Brute-Force-Ansatz implementiert. In Abbildung 2 ist die Laufzeit dieses Programms dargestellt. Als Testgraphen wurden zufällige kubische Graphen verwendet. Die Generierung solcher Graphen wird in Kapitel 5 diskutiert. Der exponentielle Verlauf der Laufzeit entspricht den theoretisch hergeleiteten Erwartungen. Es ist erkennbar, dass diese Methode nur für eine sehr geringe Anzahl an Knoten durchführbar ist und lediglich bis zu Graphen mit etwa 20 Knoten praktisch nutzbar ist.

3.2 Lineare Programmierung

Das Bisektionsproblem kann auch als Problem der linearen Optimierung (auch lineare Programmierung, abgekürzt LP, genannt) formuliert werden. Instanzen dieser Problemklasse bestehen aus einer linearen Optimierungsfunktion und linearen Bedingungen (engl. constraints). Darüber hinaus können gewisse Bedingungen für die Variablen gelten. Sind diese Variablen reell, so ist lineare Programmierung in Polynomialzeit lösbar. Das in dieser Arbeit behandelte Bisektionsproblem ist jedoch ein diskretes Problem, folglich sind die Variablen in einer Formulierung als Problem der linearen Optimierung ganzzahlig. Diese Probleme sind jedoch NP-schwer. Dennoch sind LP-Solver in

der Praxis oftmals sehr schnell. Daher ist es sinnvoll dieses Verfahren zu betrachten und die Laufzeit mit dem Brute-Force-Verfahren des vorigen Abschnitts zu vergleichen.

Es gibt verschiedene Arten das Bisektionsproblem als lineares Optimierungsproblem zu interpretieren. In dieser Arbeit werden zwei verschiedene Ansätze betrachtet. Der erste ist eine klassische Formulierung und für allgemeine Graphen gültig. Der zweite Ansatz ist lediglich für reguläre Graphen mit gerader Knotenzahl anwendbar und es ist von Interesse, ob die Laufzeit durch diese zusätzliche Beschränkung verbessert werden kann.

Der allgemein anwendbare Ansatz ist in Algorithmus 2 dargestellt. Die Variablen $e_{u,v}$ re-

Algorithmus 2 : Lineares Programm für allgemeine Graphen

input : Ein ungerichteter Graph $G = (V, E)$

output : Ergebnis der Optimierung und Werte der Variablen

$$
\begin{array}{lll}
\text{minimiere:} & \displaystyle\sum_{\{u,v\}\in E} e_{u,v} & \\
\text{unter den Bedingungen:} & e_{u,v} \geq x_u - x_v, & \{u,v\} \in E \\
& e_{u,v} \geq x_v - x_u, & \{u,v\} \in E \\
& \displaystyle\sum_{i=1}^{n} x_i = \lfloor n/2 \rfloor, & i \in \{1,\ldots,n\} \\
& x_i \in \{0,1\}, & i \in \{1,\ldots,n\} \\
& e_{u,v} \in \mathbb{R}, & \{u,v\} \in E
\end{array}
$$

präsentieren Werte, welche den Kanten des Graphen zugeordnet werden. Die Optimierungsfunktion minimiert nun die Summe dieser Werte. Damit diese Formulierung äquivalent zum Bisektionsproblem ist, müssen Kanten zwischen Partitionen den Wert 1 erhalten und Kanten innerhalb einer Partition den Wert 0. Deswegen sind diese Werte abhängig von den Variablen x_u und x_v, welche die Knoten u und v repräsentieren, die durch die Kante $e_{u,v}$ verbunden werden. Dabei repräsentiert die binäre Variable x_i die Partitionszugehörigkeit von Knoten i. Ist diese Variable 0, so gehört der Knoten zur Partition V_0, ist sie 1, so gehört der Knoten zur Partition V_1. Es muss also $\sum_{i=1}^{n} x_i = \lfloor n/2 \rfloor$ verlangt werden, da die Partitionierung balanciert sein soll. Nun werden die Constraints $e_{u,v} \geq x_u - x_v$ und $e_{u,v} \geq x_v - x_u$ aufgestellt, welche die Werte der Kantenvariablen bestimmen. Für eine Kante zwischen den Partitionen gilt $x_u = x_v + 1$ oder $x_u = x_v - 1$ und somit ist eine der angegebenen Differenzen 1 und die andere -1. Folglich gilt für diese Kante, dass der angegebene Wert größer gleich 1 ist. Ist die Kante innerhalb der Partition, so gilt $x_u = x_v$ und beide Differenzen sind 0. Es folgt, dass der Wert der Kante größer gleich 0 ist. Da die Optimierungsfunktion die Summe der Werte dieser Variablen minimiert, trägt eine Kante zwischen den Partitionen folglich den Wert 1 und eine Kante innerhalb einer Partition den Wert 0 zur Summe bei. Damit ist diese Formulierung eines linearen Programms äquivalent zum Bisektionsproblem.

Es gibt jedoch eine weitere Art, das Bisektionsproblem zu formulieren. Dieser Ansatz ist nur für

reguläre Graphen zulässig. Das Ziel ist es, die Anzahl von Kanten in der Partition V_1 zu maximieren. Dieser Ansatz kann folgendermaßen als lineares Optimierungsproblem formuliert werden.

Algorithmus 3 : Lineares Programm für reguläre Graphen

input : Ein ungerichteter Graph $G = (V, E)$
output : Ergebnis der Optimierung und Werte der Variablen

$$
\begin{aligned}
&\text{maximiere:} && \sum_{\{u,v\}\in E} e_{u,v} && \\
&\text{unter den Bedingungen:} && e_{u,v} \leq x_u, && \{u,v\} \in E \\
& && e_{u,v} \leq x_v, && \{u,v\} \in E \\
& && \sum_{i=1}^{n} x_i = \lfloor n/2 \rfloor, && i \in \{1,\ldots,n\} \\
& && x_i \in \{0,1\}, && i \in \{1,\ldots,n\} \\
& && e_{u,v} \in \mathbb{R}, && \{u,v\} \in E
\end{aligned}
$$

Theorem 1. *Das lineare Programm in Algorithmus 3 löst das Bisektionsproblem.*

Beweis. Es wird zunächst gezeigt, dass das lineare Programm die Anzahl der Kanten in Partition V_1 maximiert. Die Optimierungsfunktion maximiert die Summe der Variablen $e_{u,v}$, welche den Kanten des Graphen Werte zuordnen. Dabei soll der Wert dieser Summe nun die Anzahl der Kanten in Partition V_1 sein. Das bedeutet, wenn eine Kante innerhalb von Partition V_1 ist, soll die dazugehörige Variable 1 sein. Liegt die Kante hingegen in Partition V_0 oder zwischen den Partitionen, soll die Variable den Wert 0 annehmen. Diese Werte hängen daher von den binären Variablen x_u und x_v ab, welche den Knoten u und v Werte zuordnen, die die Partitionszugehörigkeit angeben. Diese Knoten werden von der Kante u, v mit der Variable $e_{u,v}$ verbunden. Es muss erneut $\sum_{i=1}^{n} x_i = n/2$ gelten. Die Bedingungen für die Kantenvariablen sind nun $e_{u,v} \leq x_u$ und $e_{u,v} \leq x_v$. Aus dieser Definition folgt, dass der Wert einer Kante kleiner gleich 1 ist, wenn diese in Partition V_1 ist. Liegt die Kante hingegen in Partition V_0 oder zwischen den Partitionen, so lautet mindestens eine der Gleichungen $e_{u,v} \leq 0$. Somit ist $e_{u,v}$ kleiner gleich 1, wenn x_u und x_v in der Partition V_1 sind und sonst kleiner gleich 0. Da die Optimierungsfunktion die Summe dieser Variablen maximiert, tragen diese folglich den Wert 1 zur Summe bei, wenn diese in Partition V_1 liegen, und sonst den Wert 0. Dies entspricht daher der Maximierung der Kanten in Partition V_1.

Es gilt nun zu zeigen, dass eine solche Maximierung äquivalent zur Minimierung der Kanten zwischen den Partitionen und damit äquivalent zum Bisektionsproblem ist. Es wird vorausgesetzt, dass die Anzahl der Knoten gerade ist. Dann folgt aus der Eigenschaft, dass jede Kante zwischen den Partitionen den Grad eines Knoten aus beiden Mengen reduziert, und der Regularität, dass es in beiden Partitionen (unabhängig von der Wahl dieser Partitionen) die gleiche Anzahl von Kanten gibt. Es kann also die Anzahl von Kanten in einer Partition maximiert werden, was ebenfalls

die Anzahl von Kanten in der anderen Partition maximiert und in der Folge die Größe des Cuts minimiert. □

Um die Bisektionsweite zu berechnen, ist nun noch eine Umrechnung nötig. Sei k das Ergebnis der linearen Optimierung, dann ist $b(G) = d \cdot n/2 - 2 \cdot k$. Es gibt insgesamt $d \cdot n/2$ Kanten und $2 \cdot k$ Kanten innerhalb der Partitionen, damit folgt $b(G)$ als Bisektionsweite.

Zur Lösung von Instanzen der Linearen Programmierung gibt es eine Vielzahl an Software. Für die praktische Umsetzung der Ansätze in dieser Arbeit wurde das frei verfügbare Programm `lp_solve` genutzt. Diesem Programm kann die Formulierung eines linearen Optimierungsproblem als Textdatei übergeben werden [17]. Ich habe daher ein Python-Programm geschrieben, welche für einen Graphen eine LP-Formulierung des Bisektionsproblems nach den entsprechenden Vorgaben des Programms in eine Textdatei schreibt. Damit kann das Programm `lp_solve` gestartet und die Laufzeiten der beiden Formulierungen verglichen werden. In Abbildung 3 sind die Laufzeiten der

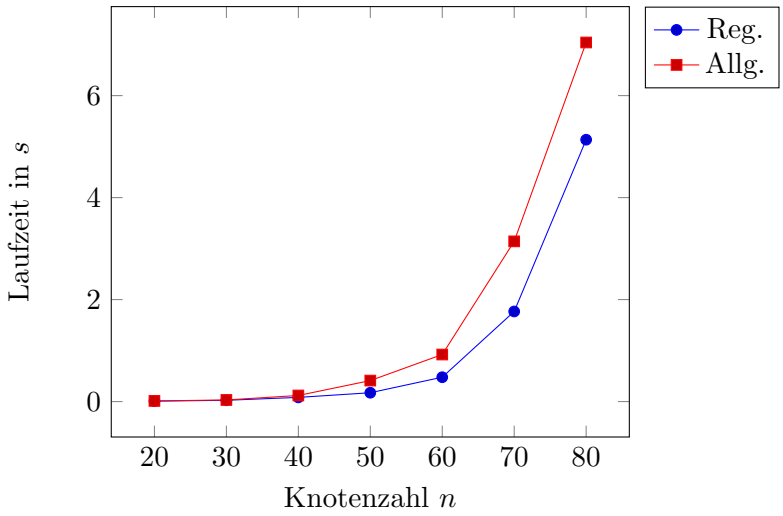

Abbildung 3: Vergleich der Laufzeiten der beiden LP-Ansätze

beiden LP-Ansätze auf kubischen Graphen dargestellt. Die Laufzeit beider Ansätze ist exponentiell, was den Erwartungen entspricht. Es fällt auf, dass die Variante, in der die Regularität des Graphen ausgenutzt wird, schneller ist als die allgemeingültige Formulierung. Diese zusätzliche Information ermöglicht es dem Algorithmus offenbar schneller die Lösung zu finden und der Laufzeitgewinn macht sich für größere Instanzen bemerkbar. Dies ist der Fall, obwohl die Anzahl der Variablen in beiden linearen Programmen identisch ist. Diese Methode ist für kubische Graphen bis zu einer Knotenzahl von ca. 80 noch praktisch nutzbar.

Im Vergleich zu der Laufzeit des naiven Brute-Force-Ansatzes, welche in Abbildung 2 dargestellt ist, ist die Verwendung von linearer Programmierung aufgrund der besseren Laufzeit vorzuziehen. Dennoch ist die Laufzeit weiterhin zu groß, um diese Ansätze in der Praxis zu verwenden und jede anwendbare Heuristik muss praktisch sowie asymptotisch deutlich schneller sein. Damit ist eine Bedingung an die Laufzeit der im folgenden Kapitel vorgestellten Heuristiken gegeben.

4 Heuristiken

Die bereits beschriebenen Ansätze mit exponentieller Laufzeit sind in der Praxis für größere Graphen nicht nutzbar. Daher werden in der Praxis häufig Heuristiken genutzt, welche in Polynomialzeit zwar nicht die optimalen, aber angemessene Ergebnisse liefern sollen. In diesem Kapitel werden vier verschiedene Heuristiken vorgestellt, welche in weiteren Teilen der Arbeit experimentell verglichen werden. Ziel dieser Arbeit ist vor allem der Vergleich von der vorgestellten Greedy-Strategie mit dem Eigenwertansatz.

4.1 Greedy

In diesem Abschnitt wird ein einfaches Greedy-Verfahren vorgestellt, welches die Bisektionsweite minimiert. In der Literatur werden viele verschiedene, jedoch grundsätzlich sehr ähnliche Greedy-Ansätze diskutiert. Das Paper [11] gibt einen Überblick über bekannte Algorithmen. Dabei soll durch iterative, lokale Optimierung eine Bisektion gefunden werden, die zwar nicht optimal ist, aber zumindest lokal minimal. Das bedeutet, dass die Größe des Cuts durch weitere Anwendung des Greedy-Verfahrens nicht mehr verringert werden kann. Der in dieser Arbeit diskutierte Ansatz tauscht im Optimierungsschritt einzelne Knoten der Partitionen aus, wenn diese die Größe des Cuts verringern.

Es wird zunächst mit einer balancierten Bisektion gestartet. Die Wahl dieser Startbisektion wird im weiteren Verlauf der Arbeit noch diskutiert, hier wird zunächst eine zufällige balancierte Bisektion gewählt. Dann beginnt die lokale Optimierung. In Partition V_0 wird der Knoten gesucht, welcher bei einem Transfer in die Partition V_1, die Größe des Cuts maximal verringert, und in diese Menge überführt. Im Anschluss wird in Partition V_1 nach einem Knoten gesucht, welcher den Cut bei einem Transfer nach V_0 maximal verkleinert, und ebenfalls überführt. Dieses Verfahren wird beendet, sobald ein solcher Tausch zu keiner Verbesserung führt. Da mit einer balancierten Bisektion gestartet wurde, ist auch die resultierende Bisektion balanciert. Algorithmus 4 stellt die beschriebene Strategie dar.

Die Zuordnung der Knoten zu den Partitionen wird in der Liste `bis` gespeichert, wobei der i-te Eintrag, welcher entweder 0 oder 1 ist, die Zugehörigkeit von Knoten i angibt. Um wie in Zeile 1 gefordert mit einer zufälligen Bisektion zu starten, muss die Liste zufällig permutiert werden. Dies ist in $\mathcal{O}(n)$ mit dem Fisher-Yates-Shuffle möglich, welcher zum Beispiel von der Programmiersprache Python durch die Funktion `random.shuffle` angeboten wird [18].

Um den Knoten zu finden, welcher bei einem Transfer den Cut um den größten Wert verringert, wird für jeden Knoten die Differenz aus Nachbarn in der fremden und Nachbarn in der eigenen Partition berechnet und in der Liste `diff` gespeichert. Diese wird zunächst in den Zeilen 3-10 initial berechnet. Dazu müssen für jeden Knoten seine Nachbarn betrachtet werden und die Zählervariable `count` je nach Paritionszugehörigkeit der Nachbarn inkrementiert bzw. dekrementiert werden. Es ergibt sich eine Laufzeit von $\mathcal{O}(n + m)$.

Im Anschluss wird eine While-Schleife betreten, welche in jeder Iteration die Größe des Cuts verringern soll. Gelingt dies nicht, so wurde ein lokales Optimum erreicht und es wird abgebrochen. Die maximale Anzahl an Iterationen ist also die Differenz aus maximal und minimal möglicher Größe des Cuts. Im Extremfall sind diese Werte m und 0. Es folgt also, dass die Anzahl der Durchläufe

Algorithmus 4 : Greedy-Algorithmus

input : Ein ungerichteter Graph $G = (V, E)$
output : Eine balancierte Bisektion der Knotenmenge

```
1  Sei bis eine zufällig permutierte Liste mit ⌊n/2⌋ Nullen und ⌈n/2⌉ Einsen
2  Sei diff eine Liste der Größe n
3  foreach u ∈ G do
4  |  count ← 0
5  |  foreach v ∈ G[u] do
6  |  |  if bis[u] = bis[v] then count ← count + 1
7  |  |  else if bis[u] ≠ bis[v] then count ← count - 1
8  |  end
9  |  diff[u] ← count
10 end
11 while true do
12 |  f, u ← getmax (G, diff, bis, 0)
13 |  update (G, diff, bis, u)
14 |  s, v ← getmax (G, diff, bis, 1)
15 |  if f + s < 1 then
16 |  |  bis[u] ← 0
17 |  |  break
18 |  end
19 |  update (G, diff, bis, v)
20 end
21 return bis
```

der While-Schleife in $\mathcal{O}(m)$ liegt.

Innerhalb der Schleife wird zunächst nach dem Knoten aus Partition V_0 gesucht, bei dem der zugehörige Eintrag in `diff` maximal ist. Die Funktion `getmax` findet diesen Knoten sowie das dazugehörige Maximum, welche im Algorithmus in den Variablen u sowie f gespeichert werden. Der Knoten u soll nun in die Partition V_1 transferiert werden. Da der Eintrag in `diff` dieses Knotens maximal ist, verringert dieser Knoten die Bisektionsweite durch einen Transfer maximal. Dieser Transfer führt darüber hinaus dazu, dass sich die Werte in der Liste diff ändern. Diese Liste sowie die Partitionszugehörigkeit von Knoten u wird daher durch die Prozedur `update` aktualisiert. Auf die Implementierung von `getmax` und `update` wird im Folgenden noch genauer eingegangen. Nachdem nun ein Knoten von V_0 nach V_1 überführt wurde, ist der Transfer eines Knoten von V_1 nach V_0 nötig, damit die Partitionen weiterhin balanciert sind. Daher wird nach dem Knoten aus V_1 gesucht, welcher den größten Wert in `diff` besitzt. Dieser kann ebenfalls mit der Funktion `getmax` gefunden werden. In den Variablen s und v wird das Maximum sowie der dazugehörige Knoten gespeichert. Nun wird in Zeile 15 überprüft, ob die Summe der Differenzen f und s kleiner als 1 ist. Ist dies der Fall, so hat sich die Bisektionsweite durch den Tausch nicht verringert. Da bereits durch die Aktualisierung in Zeile 13 die Partitionszugehörigkeit von Knoten u verändert wurde,

obwohl dieser Tausch die Bisektionsweite unter Umständen erhöht hat, wird muss diese Änderung in Zeile 16 rückgängig gemacht werden. Im Anschluss wird die Schleife beendet und die gefundene Bisektion zurückgegeben. In dem Fall, dass die Bedingung der If-Anweisung in Zeile 15 nicht erfüllt ist, wird der Tausch der beiden Knoten durch einen weiteren Aufruf der `update`-Prozedur in Zeile 19 vervollständigt und danach eine neuer Schleifendurchlauf begonnnen.

In Funktion 1 ist der Pseudocode der Funktion `getmax` dargestellt. In der Schleife in Zeile 3

Funktion 1 : getmax(G, diff, bis, p)

```
1 max ← −∞
2 idx ← −1
3 foreach u ∈ G do
4 |   if bis[u] = p and diff[u] > max then
5 |   |   max ← diff[u]
6 |   |   idx ← u
7 |   end
8 end
9 return max, idx
```

wird über alle Knoten des Graphen iteriert und, falls ein solcher in V_p ist, der Wert aus `diff` mit dem Maximum verglichen und letzteres gegebenenfalls aktualisiert. Nach Beendigung dieser Schleife wird das Maximum sowie der dazugehörige Knoten in Zeile 9 zurückgegeben. Diese Implementierung der Maximumssuche hat eine Laufzeit von $\mathcal{O}(n)$. Es sei hier noch angemerkt, dass es insbesondere für die in Kapitel 7 betrachteten deterministischen Graphklassen sinnvoll ist, wenn das gewählte Maximum zufällig unter allen größten Werten gewählt wird. Eine solche Anpassung ist problemlos möglich.

In Prozedur 1 ist die Funktion `update` dargestellt. Diese Prozedur aktualisiert zunächst in Zeile

Prozedur 1 : update(G, diff, bis, u)

```
1 bis[u] ← 1 - bis[u]
2 diff[u] ← -diff[u]
3 foreach v ∈ G[u] do
4 |   if bis[u] = bis[v] then diff[v] ← diff[v] - 2
5 |   else if bis[u] ≠ bis[v] then diff[v] ← diff[v] + 2
6 end
```

1 die Partitionszugehörigkeit des übergebenen Knotens u. Darüber hinaus wird in Zeile 2 der zu Knoten u korrespondierende Eintrag in der Liste `diff` angepasst. Dazu muss der bisherige Eintrag mit -1 multipliziert werden, da alle Nachbarn, welche zuvor in der gleichen Partition waren, nun in der anderen Partition sind und umgekehrt. Im Anschluss werden die Nachbarn von Knoten u betrachtet. Ist ein Nachbar in der gleichen Partition wie der Knoten u gewesen, so verringert sich die Anzahl an Knoten in der eigenen Partition um den Wert 1, während die Anzahl an Knoten in

der anderen Partition um 1 steigt. Hier erhöht sich der Wert der Differenz also um 2. Entsprechend verringert sich diese Differenz um den Wert 2, wenn der Nachbar zuvor in der anderen Partition war. Diese Aktualisierungen benötigen für den Fall, dass der transferierte Knoten $\mathcal{O}(n)$ Nachbarn hat (eine bessere Schranke existiert für allgemeine Graphen nicht), eine Laufzeit von $\mathcal{O}(n)$.

Daraus folgt also für den Greedy-Algorithmus eine asymptotische Laufzeit von $\mathcal{O}(m \cdot n + n + m)$. Für Graphen mit geringer Kantenzahl (zum Beispiel d-reguläre Graphen) ist es möglich die Laufzeit des Algorithmus durch die Wahl geeigneter Datenstrukturen zu verbessern. Das Aktualisieren der Nachbarn in der While-Schleife benötigt in diesem Fall nur $\mathcal{O}(d)$ Zeit und ein besserer Ansatz als die lineare Suche nach dem Maximum verbessert also die Gesamtlaufzeit. Solche Ansätze sind nicht Teil dieser Arbeit, da die Laufzeit der linearen Suche für die experimentelle Analyse des Algorithmus ausreicht.

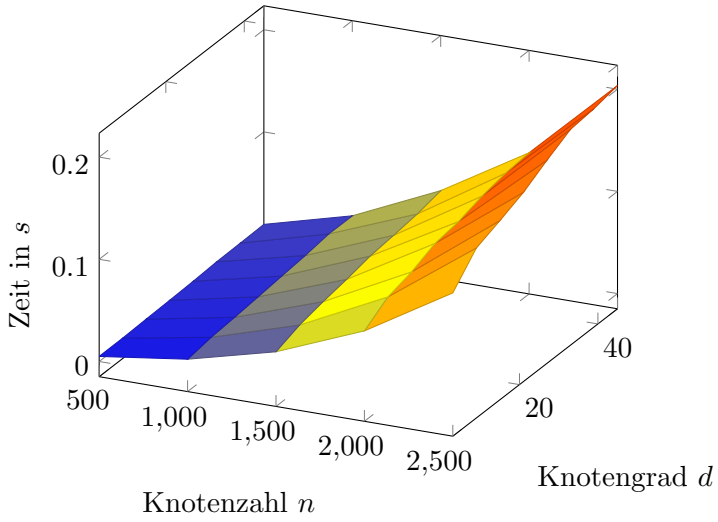

Abbildung 4: Laufzeit der Greedy-Heuristik auf d-regulären Graphen

Abbildung 4 zeigt die Laufzeit des Algorithmus auf d-regulären Graphen. Die Generierung solcher Graphen wird in Kapitel 5 diskutiert. Mit steigender Knotenzahl steigt auch die Kantenzahl für d-reguläre Graphen linear. Daher entspricht der quadratische Anstieg den Erwartungen. Eine Erhöhung des Knotengrads hingegen führt zu einem Anstieg, welcher über ein lineares Wachstum nicht hinausgeht.

Dieser Greedy-Ansatz hat große Ähnlichkeit zu dem in [11] diskutierten *Simple-Greedy-Algorithm*. Eine Generalisierung dieses Verfahrens ist der in [9] vorgestellte Ansatz von Kernighan und Lin aus dem Jahr 1970. Dieser Ansatz ist darüber hinaus von hoher praktischer Relevanz und im Jahr 1989 beschrieben Bui et al. dieses Verfahren in der Anwendung des VLSI-Designs als „recognized champion among the classical approaches to the graph bisection problem“ [19]. Auch in Kapitel 3 von [6] aus dem Jahr 1991 wird der Kernighan-Lin-Algorithmus als häufig gewählter Ansatz für die Graph-Partitionierung im VLSI-Design genannt. Heutzutage werden, wie bereits in der Einleitung erwähnt, analytische Ansätze denen der Graph-Partitionierung vorgezogen [7]. Dennoch arbeiten auch modernere Algorithmen der Graph-Partitionierung im VLSI-Design mit dem Prinzip der ite-

rativen Verbesserung.

Es sei angemerkt, dass im weiteren Verlauf lediglich der hier vorgestellte Greedy-Algorithmus diskutiert wird und ein Vergleich mit dem Ansatz von Kernighan-Lin nicht das Ziel dieser Arbeit ist. Die Ergebnisse eines solchen Vergleichs können in [11] nachgelesen werden.

4.2 Breitensuche

Eine andere Idee, welche nicht lokale Optimierung, sondern das direkte Finden einer Bisektion zum Ziel hat, ist eine klassische Breitensuche. Bei vielen Graphen ist eine gute Bisektion eine solche, in der alle Knoten in den Partitionen zusammenhängen. Damit gibt es für zusammenhängende Graphen $n/2 - 1$ Kanten innerhalb einer Partition, welche die Größe des Cuts folglich nicht erhöhen. Daraus folgt, dass eine solche Bisektion für viele Graphen besser als eine durchschnittliche Bisektion ist. Es gibt sehr viele Bisektionen in zusammenhängende Partitionen mit unterschiedlicher Qualität und die Breitensuche liefert lediglich eine von diesen. Auch wenn dies unter Umständen nicht die beste Bisektion ist, so ist es dennoch ein interessanter Vergleichswert für die Tests in dieser Arbeit. Darüber hinaus ist die erhaltene Bisektion, wie in späteren Abschnitten diskutiert wird, ein guter Ausgangspunkt für weitere Verfahren. Der Ansatz ist, eine Breitensuche zunächst von einem

Algorithmus 5 : Breitensuche zur Bisektionsfindung

input : Ein ungerichteter Graph $G = (V, E)$ und ein Startknoten s
output : Eine balancierte Bisektion der Knotenmenge

```
1  vis ← [false] · n
2  Q ← ∅
3  c ← 1
4  ENQUEUE(Q, s)
5  vis[s] ← true
6  while Q ≠ ∅ and c < n/2 do
7      u ← DEQUEUE(Q)
8      foreach v ∈ G[u] do
9          if not vis[v] then
10             vis[v] ← true
11             c ← c + 1
12             ENQUEUE(Q, v)
13         end
14         if c ≥ n/2 then
15             break
16         end
17     end
18 end
19 return vis
```

bestimmten Knoten zu starten, und die ersten $n/2$ besuchten Knoten als Partition zu wählen. In

Algorithmus 5 ist diese Strategie dargestellt.

Es wird eine Liste gespeichert, in welcher der i-te Eintrag angibt, ob der i-te Knoten bereits besucht wurde. Diese Liste beschreibt am Ende des Verfahrens eine Bisektion, in der die Einträge die Zugehörigkeit der Knoten zu den Partitionen angeben. Kern des Algorithmus ist die Queue Q, welche die besuchten, aber noch nicht bearbeiteten Knoten speichert. Begonnen wird mit dem Startknoten s, der in die Queue eingefügt wird und als besucht markiert wird. Die Variable c zählt die bereits besuchten Knoten und beginnt somit bei 1. Die While-Schleife in Zeile 6 bearbeitet in jeder Iteration einen Knoten. Dazu werden seine Nachbarn besucht und in die Queue eingefügt, falls sie zuvor unbesucht waren. c wird entsprechend hochgezählt und falls $n/2$ Knoten besucht wurden, terminiert der Algorithmus und gibt die Liste `vis` als Bisektion zurück.

In der Praxis kann nicht davon ausgegangen werden, dass alle Graphen zusammenhängend sind, und daher muss der Algorithmus für diesen Fall erweitert werden. Es könnte also passieren, dass die Queue leer ist, bevor $n/2$ Knoten besucht worden sind. Folglich wird ein zufälliger unbesuchter Knoten in Q eingefügt und die While-Schleife weiter ausgeführt. Ein solcher Knoten kann immer in erwarteter Zeit $\mathcal{O}(1)$ gefunden werden, da mindestens $n/2 + 1$ Knoten unbesucht sind. Diese Lösung verhindert zwar fehlerhafte Ausgaben des Algorithmus, sobald der Graph jedoch nicht zusammenhängend ist, ist dieser Algorithmus unter Umständen nicht mehr besonders sinnvoll. Die in dieser Arbeit betrachteten deterministischen Graphklassen sind alle zusammenhängend und für die zufälligen Graphen gilt dies mit hoher Wahrscheinlichkeit. Daher ist das in Algorithmus 5 beschriebene Verfahren ausreichend.

Ein großer Vorteil dieses simplen Verfahrens ist die Laufzeit von $\mathcal{O}(n+m)$. Diese ergibt sich, da durch die Schleife in Zeile 6 in der Summe $\mathcal{O}(n)$ Knoten bearbeitet werden und in der For-Schleife in Zeile 8 insgesamt $\mathcal{O}(m)$ Kanten.

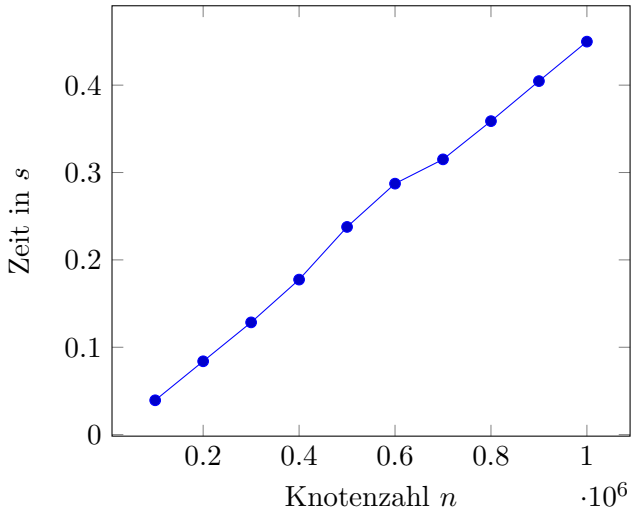

Abbildung 5: Laufzeit der Breitensuche-Heuristik

In Abbildung 5 ist die Laufzeit des Verfahrens auf kubischen Graphen dargestellt. Es ist, wie auch theoretisch hergeleitet, ein linearer Verlauf mit steigender Knotenzahl zu sehen. Darüber hinaus

bleibt die Laufzeit auch für Graphen mit 1 Mio. Knoten unter einer Sekunde. Der Algorithmus ist somit selbst für sehr große Graphen in der Praxis nutzbar.

4.3 Breitensuche und Greedy

Die Breitensuche findet eine sinnvolle, aber nicht weiter optimierte Bisektion und der Greedy-Algorithmus soll eine gegebene Bisektion lokal optimieren. Es liegt nahe die beiden Verfahren zu kombinieren. In [19] wird die Möglichkeit einer Verbesserung des Greedy-Verfahrens durch eine gute Startbisektion erwähnt. In [11] wird dies weiter ausgeführt und es werden ebenfalls verbundene Partitionen als Ausgangspunkt genutzt. In diesem Paper wurden mit einer solchen Strategie hervorragende Ergebnisse erzielt.

Es ist möglich eine Breitensuche von einem beliebigen Knoten zu starten und die erhaltene Bisektion anschließend zu verbessern. Interessanter ist es jedoch von jedem Knoten eine Breitensuche zu starten und anschließend mit dem Greedy-Ansatz die erhaltenen Bisektionen weiter zu optimieren. Dahinter steckt die folgende Idee. Für Graphen mit vergleichsweise kleiner Bisektionsweite sind nur wenige Knoten am Cut der optimalen Bisektion beteiligt. Man stelle sich nun die (unbekannte) Bisektion mit minimaler Weite vor. So gibt es in beiden Partitionen einen Knoten, welcher am weitesten vom Cut entfernt ist. Startet man eine Breitensuche nun gerade von diesem Knoten, so erhält man unter Umständen eine Partition, die viele Knoten einer Partition der optimalen Bisektion enthält. Wenn eine Breitensuche von jedem Knoten gestartet wird, so wird diese also auch bei den eben genannten Knoten begonnen. Das Greedy-Verfahren soll dieses Ergebnis im Anschluss verbessern und, falls die Breitensuche über den Cut hinausgegangen ist, die entsprechenden Knoten zurücktauschen. Es wird am Ende von allen gefundenen Bisektionen die minimale zurückgegeben. Dieser Ansatz ist in Algorithmus 6 dargestellt. In einer Schleife wird eine Breitensuche von jedem

Algorithmus 6 : n-fache Breitensuche mit anschließendem Greedy

```
   input  : Ein ungerichteter Graph G = (V, E)
   output : Eine balancierte Bisektion der Knotenmenge
 1 min = ∞
 2 foreach u ∈ G do
 3 |   bis ← bfs (G, u)
 4 |   bis ← greedy (G, bis)
 5 |   if bisectionwidth(G, bis) < min then
 6 |   |   min ← bisectionwidth(G, bis)
 7 |   end
 8 end
 9 return min
```

Knoten des Graphen G gestartet und durch einen anschließenden Greedy verbessert. Dabei entspricht die Funktion `bfs` der im vorigen Abschnitt vorgestellten Breitensuche und gibt eine Liste der Größe n zurück, bei der der i-te Eintrag die Partitionszugehörigkeit des i-ten Knotens angibt. Die Funktion `greedy` entspricht der zu Beginn dieses Kapitels vorgestellten Greedy-Heuristik, wo-

bei diese hier nicht von einer zufälligen Bisektion gestartet wird, sondern mit der in `bis` gegebenen Partitionierung. Zu der erhaltenen Bisektion kann im Anschluss die Bisektionsweite berechnet und gegebenenfalls das Minimum angepasst werden.

Die Laufzeit dieses Verfahrens ist für jeden Startknoten die Addition der Laufzeiten aus Greedy und Breitensuche und damit also $\mathcal{O}(m \cdot n + n + m)$. Da dieser Ansatz n-mal gestartet wird, folgt eine Laufzeit von $\mathcal{O}(n \cdot (m \cdot n + n + m))$.

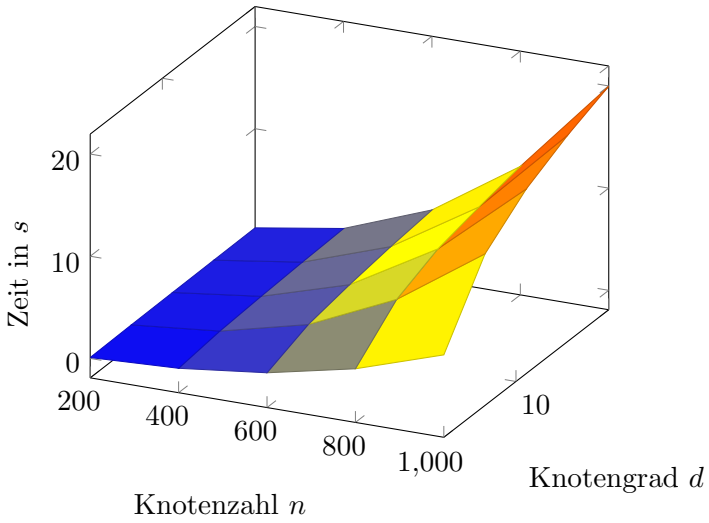

Abbildung 6: Laufzeit der Heuristik bestehend aus Breitensuche und Greedy-Algorithmus auf d-regulären Graphen

Es ist interessant ebenfalls die Laufzeit dieses Verfahrens zu betrachten. Diese ist in Abbildung 6 dargestellt. Es ist zu erkennen, dass die Laufzeit mit steigender Knotenzahl mindestens quadratisch, aber vermutlich kubisch wächst. Bei der Betrachtung des Knotengrads fällt auf, dass der Anstieg für geringe d sehr steil ist und anschließend flacher wird. Dies ist dadurch zu erklären, dass die Breitensuche für einen geringen Knotengrad bereits relativ gute Ergebnisse liefert. Daher durchläuft der zeitaufwändige Greedy-Ansatz hier nur eine geringe Anzahl von Iterationen. Für größere Knotengrade ist die Güte der Bisektion, welche die Breitensuche liefert, jedoch schlechter und relativ ähnlich. Daher ist hier eher ein geringer Anstieg zu sehen. Eine ausführliche Diskussion der Güte der Heuristiken auf d-regulären Graphen ist in Kapitel 6 zu finden.

4.4 Eigenwertansatz

Weitere Heuristiken zur Lösung des Bisektionsproblems basieren auf einer Eigenwertanalyse von Matrizen, welche auf verschiedenen Wegen mithilfe des Graphen konstruiert werden. Diese Ansätze sind sowohl von theoretischer als auch von großer praktischer Relevanz [14] und aktuelles Forschungsthema [5].

Eigenwerte und korrespondierende Eigenvektoren sind die Lösungen des sogenannten Eigenwert-

problems. Dies ist durch die folgende Gleichung definiert:

$$A \cdot \vec{v} = \lambda \cdot \vec{v}$$

Dabei ist $A \in \mathbb{R}^{n \times n}$ eine Matrix, $\lambda \in \mathbb{R}$ ein Skalar, der sogenannte Eigenwert, und $\vec{v} \in \mathbb{R}^n$ ein Vektor, der sogenannte Eigenvektor. Die triviale Lösung $\vec{v} = \vec{0}$ wird hier ausgeschlossen. In dieser Arbeit wird als Matrix A die Adjazenzmatrix des Graphen verwendet, welche folgendermaßen für einen ungerichteten sowie ungewichteten Graphen definiert ist:

$$a_{ij} = \begin{cases} 1 & \text{falls } \{i, j\} \in E, \\ 0 & \text{sonst} \end{cases}$$

Dies ist jedoch nicht die einzige mögliche Wahl und in der Forschung werden viele verschiedene Matrizen diskutiert, welche basierend auf dem Graphen konstruiert werden können [5].

Im Folgenden werden die bis jetzt als Eigenwertansätze bezeichneten Verfahren auch als Spektralansätze bezeichnet. Das Spektrum eines Graphen ist als Menge der Eigenwerte der Adjazenzmatrix eines Graphen mit den dazugehörigen Häufigkeiten definiert [20].

Bevor der Eigenwertansatz, mit dem sich diese Arbeit beschäftigt, betrachtet wird, stelle ich zunächst ein Beispiel vor, dass die Intuition der Nutzung von Eigenwerten in Bezug auf das Bisektionsproblem darstellt. Dazu sind in Abbildung 7 die zu den vier größten Eigenwerten gehörenden

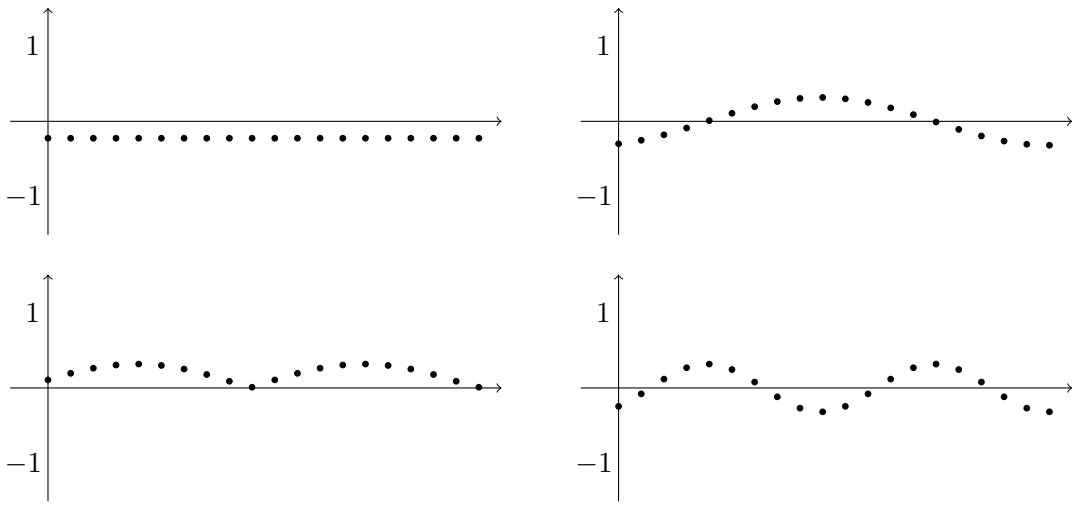

Abbildung 7: Elementweise Darstellung der vier größten Eigenvektoren auf zugehörigem Kreisgraphen. Die y-Koordinate entspricht hierbei dem Wert des Eintrags im Eigenvektor.

Eigenvektoren des Spektrums eines Kreisgraphen elementweise dargestellt. Oben links ist dabei der größte Eigenwert vertreten, rechts daneben der zweitgrößte, links unten der drittgrößte und rechts unten der viertgrößte. Der Kreis besteht dabei aus 20 Knoten, von denen jeder zwei Nachbarn besitzt, der Graph ist also 2-regulär. Dabei ist der Knoten i mit den Knoten $i - 1 \mod 20$ und $i + 1 \mod 20$ benachbart. Die y-Koordinate gibt in der Abbildung den Wert des Eintrags im Eigenvektor an.

Für diesen Graphen ist das Finden der optimalen Bisektion keine besondere Herausforderung und hier auch nicht von Interesse. Stattdessen veranschaulichen die Einträge der Eigenvektoren das Prinzip der Spektralansätze. Um das Bisektionsproblem mithilfe von Eigenwerten und Eigenvektoren zu lösen, werden die Einträge des Eigenvektors den Knoten des Graphen zugeordnet. Der i-te Eintrag korrespondiert also zum i-ten Knoten. Anhand dieser Werte werden die Knoten dann in zwei gleich große Mengen partitioniert. Es ist in Abbildung 7 klar zu erkennen, dass der größte Eigenvektor hier keine Informationen liefert, da alle Werte des Eigenvektors identisch sind. Warum dies der Fall ist, wird im Folgenden noch diskutiert. Gewissermaßen ist dies die *beste* Partitionierung des Graphen in eine Menge, welche aus allen Knoten besteht, und einer leeren Menge. Die Größe des Cuts einer solchen Bisektion ist immer null, da kein Cut existiert. Das Problem wird dadurch jedoch nicht gelöst. Die weiteren Eigenvektoren sind jedoch nicht konstant und der Verlauf ähnelt einer Schwingung. Dabei steigt die Frequenz dieser Schwingungsmoden mit sinkendem Eigenwert. Der zweitgrößte Eigenvektor korrespondiert also zur zweitkleinsten *Frequenz* des Systems. Der Zusammenhang zwischen dem Eigenwertproblem und Schwingungsfrequenzen ist aus der Physik bekannt. Dort können Systeme von schwingenden Massen mithilfe des Eigenwertproblems modelliert werden und verschiedene Eigenwerte korrespondieren zu unterschiedlichen Schwingungsmoden [21]. Dieses Modell findet im Bereich der Graphpartitionierung eine Analogie.

Um das Bisektionsproblem mithilfe der Eigenvektoren zu lösen, werden die Knoten entsprechend des zugehörigen Eintrags im Eigenvektor aufgeteilt. Mithilfe von Abbildung 7 wird deutlich, dass es sinnvoll ist, eine Partition der $n/2$ Knoten mit den kleinsten Einträgen und eine Partition der $n/2$ Knoten mit den größten Einträgen zu erstellen. Dafür sollte der Eigenvektor mit der kleinsten Frequenz gewählt, die dennoch Informationen über eine Partitionierung enthält. Dies ist der Eigenvektor, der zum zweitgrößten Eigenwert korrespondiert. Bei einer solchen Aufteilung sind Nachbarknoten häufig in der gleichen Partition, da die dazugehörigen Einträge im Eigenvektor ähnlich sind.

Anhand dieser ersten intuitiven Überlegungen kann ein Algorithmus zur Lösung des Bisektionsproblems konstruiert werden. Dazu werden zunächst reguläre Graphen betracht. Für d-reguläre Graphen ist der größte Eigenwert d und der dazugehörige Eigenvektor $\vec{1}$. Dass dieser Eigenvektor konstant ist, war bereits in Abbildung 7 zu sehen. Da alle Einträge des Eigenvektors identisch sind, ist es folglich nicht möglich aufgrund dieser eine Einteilung zu bilden, dieser Eigenvektor ist also nutzlos. Die weiteren Eigenvektoren sind jedoch von Interesse. Für symmetrische Matrizen, wie es die hier behandelten Adjazenzmatrizen sind, stehen die Eigenvektoren von verschiedenen Eigenwerten senkrecht aufeinander und damit auch senkrecht auf dem zum größten Eigenwert zugehörigen Eigenvektor [22]. Diese Orthogonalität bedeutet, dass das Skalarprodukt zweier Eigenvektoren 0 ist. Betrachten wir nun das Skalarprodukt eines beliebigen Eigenvektors mit dem zum größten Eigenwert korrespondierenden (hier als $v_{\max}$ bezeichnet), dann folgt:

$$\vec{v}^{\top} \cdot \vec{v}_{\max} = \vec{v}^{\top} \cdot \vec{1} = \sum_{i=0}^{n} v_i = 0$$

Dies bedeutet also, dass die Summe aller Einträge in den weiteren Eigenvektoren 0 ist. Dies ist ein wichtiger Fakt, denn der Eigenvektor $\vec{v} = \vec{0}$ wird ausgeschlossen und damit gibt es im Eigenvektor positive und negative Werte anhand derer die Knoten zu den Partitionen zugeordnet werden können.

Es kann jedoch nicht garantiert werden, dass es tatsächlich $n/2$ negative und $n/2$ positive Werte gibt. In der Praxis muss hier also anders verfahren werden, wenn eine balancierte Partition gefordert ist. Ein häufig verwendeter Ansatz, der bereits aus Abbildung 7 hergeleitet wurde, ist es, den zum zweitgrößten Eigenwert korrespondierenden Eigenvektor nach dem Median aufzuteilen. Das bedeutet alle Einträge größer als der Median bilden eine Partition, alle Einträge kleiner als der Median eine weitere. Dann gilt also

$$V_0 = \{i \in V | x_i \leq \tilde{x}\}$$
$$V_1 = \{i \in V | x_i > \tilde{x}\},$$

wobei x der Eigenvektor zum zweitgrößten Eigenwert ist und $\tilde{x}$ der Median dieses Vektors. Problematisch ist dieser Ansatz, falls mehrere Knoten auf dem Median liegen und die Bisektion nicht mehr balanciert ist. Eine sinnvolle Einteilung ist hier nicht offensichtlich. In den folgenden Analysen wird auf alle Einträge des Eigenvektors ein kleines zufälliges ε addiert. Dies führt dazu, dass es keine identischen Einträge gibt. Die entstandene Aufteilung der nicht zuzuordnenden Knoten ist also zufällig und im Allgemeinen nicht optimal. Der beschriebene Ansatz ist in Algorithmus 7 dargestellt. Die Funktion `eig` gibt einen Vektor $\vec{w}$ mit den Eigenwerten von Matrix A und eine

Algorithmus 7 : Eigenwertansatz

input : Die Adjazenzmatrix A eines Graphen $G = (V, E)$
output : Eine balancierte Bisektion der Knotenmenge

```
1 w⃗, V ← eig(A)
2 x⃗ ← secmax(w⃗, V)
3 x⃗ ← x⃗ + ε⃗
4 Sei bis eine Liste der Größe n
5 foreach i ∈ x⃗ do
6 |   if x_i ≤ x̃ then bis[i] = 0
7 |   else if x_i > x̃ then bis[i] = 1
8 end
9 return bis
```

Matrix V, dessen Spalten die zugehörigen Eigenvektoren sind, zurück. Die Funktion `secmax` gibt den zweitgrößten Eigenvektor zurück. Diese Funktion ist unkompliziert und wird hier nicht weiter diskutiert. Dann kann auf den zweitgrößten Eigenvektor ein Vektor $\vec{\varepsilon}$ addiert werden, dessen Einträge sehr klein und zufällig sind. Dadurch gibt es keine identischen Einträge im Vektor, welche zu Problemen beim Median-Splitting führen könnten. In der Schleife von Zeile 5 bis 8 wird dann die Bisektion anhand der Einträge von $\vec{x}$ gebildet.

Kern einer Implementierung dieses Ansatzes ist die Berechnung der Eigenwerte und Eigenvektoren. Hier wurde die Funktion `eig` der kommerziellen Software Matlab verwendet [23]. Matlab-Funktionen können auch aus einem Python-Programm aufgerufen werden [24], sodass alle in der Entwicklung dieser Arbeit geschriebenen Generatoren und Skripte weiter verwendet werden können. Darüber hinaus habe ich diesen Ansatz ebenfalls ohne die Nutzung kommerzieller Software in Py-

thon mithilfe der Funktion `numpy.linalg.eig` geschrieben [25]. Hierbei wird die Python-Bibliothek `numpy` genutzt. Alle Tests dieser Arbeit wurden jedoch mit der Matlab-Funktion `eig` durchgeführt.

Wir betrachten nun die Komplexität dieses Ansatzes. Für die Laufzeit des Eigenwertansatzes ist nahezu ausschließlich die Laufzeit der Eigenwertanalyse ausschlaggebend. Dies ist eine rechenintensive Operation. Im Allgemeinen sollte die dazugehörige Matlab-Funktion `eig` in $\mathcal{O}(n^3)$ liegen, wobei n die Anzahl der Knoten, also die Länge bzw. Breite der Adjazenzmatrix, ist. Der Hersteller macht jedoch keine offiziellen Angaben zu der Implementierung und der Laufzeit.

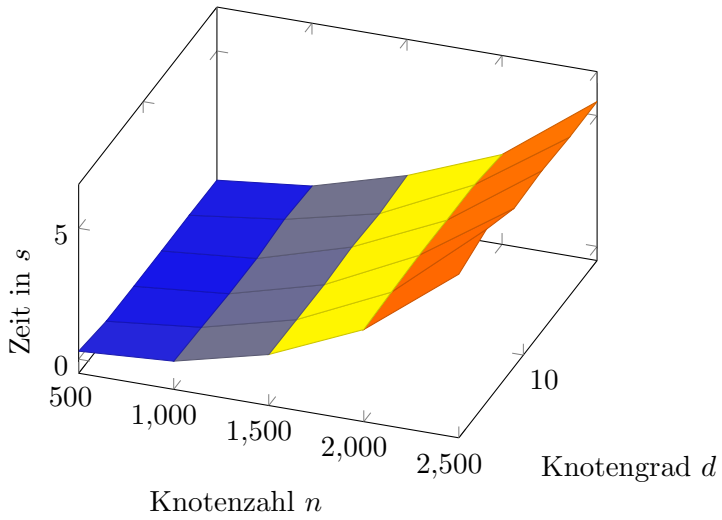

Abbildung 8: Laufzeit der Eigenwert-Heuristik

Aus diesem Grund ist eine Betrachtung der Laufzeit umso interessanter. In Abbildung 8 ist die Laufzeit des Eigenwert-Ansatzes für verschiedene Knotenzahlen n sowie Knotengrade d dargestellt. Die Laufzeiterhöhung mit steigender Knotenzahl entspricht den Erwartungen. Darüber hinaus fällt auf, dass eine steigende Anzahl von Kanten keinen erkennbaren Einfluss auf die Laufzeit hat.

Die bisherigen Betrachtungen des Eigenwert-Ansatzes sind von d-regulären Graphen ausgegangen. Für diese galt, dass die Summe der Einträge des zum zweitgrößten Eigenwert gehörenden Eigenvektors 0 ist. Für allgemeine Graphen gilt dies hingegen nicht. Eine einfache Lösung ist es, mit der Intuition, dass die Heuristik immer noch gut ist, dennoch diesen Eigenvektor zu wählen. Andere Ansätze versuchen, die Matrix A anzupassen.

In dieser Arbeit werden einerseits viele reguläre Graphen diskutiert und als Testgrundlage verwendet. Andererseits liegt das Hauptaugenmerk nicht darauf, der aktuellen Forschung im Bereich der Spektralansätze Genüge zu tun, sondern ganz allgemein die praktische Relevanz verschiedener Verfahren zu beurteilen. Dabei soll es nicht um kleinere Feinheiten zwischen Algorithmen gehen, sondern darum möglichst verschiedene Ansätze zu vergleichen. Aus diesen Gründen wird in dieser Arbeit auch für nicht-reguläre Graphen mit dem zweitgrößten Eigenwert, dem korrespondierenden Eigenvektor sowie der Adjazenzmatrix des Graphen gearbeitet, obwohl für einige Graphklassen unter Umständen weitere Verbesserungen möglich sind.

5 Generierung von zufälligen Graphen

Für eine experimentelle Bewertung von Algorithmen werden Testdaten benötigt. Neben verschiedenen deterministischen Graphklassen, welche in dieser Arbeit beschrieben werden, ist auch die Betrachtung von zufälligen Graphen interessant. Dabei betrachten wir zunächst die Klasse der d-regulären Graphen. Diese sind insbesondere aufgrund der im Abschnitt über die Eigenwertansätze vorgestellten Überlegungen, welche in erster Linie für d-reguläre Graphen gelten, interessant. Es gibt verschiedene Arten, zufällige Graphen zu erzeugen und daher wird in den folgenden Abschnitten auch detailliert auf die praktische Generierung solcher Graphen eingegangen. Der vorgestellte Algorithmus kann abgewandelt werden, um weitere reguläre Graphen zu generieren. Wir betrachten hier reguläre Graphen mit einer geringen Bisektionsweite, bipartite reguläre Graphen, sowie kubische Graphen mit einer Kreisstruktur.

5.1 Zufällige reguläre Graphen

In diesem Abschnitt wird erläutert, wie d-reguläre Graphen effizient generiert werden können. Dabei folgt das beschriebene Verfahren dem Paper [8] von Steger und Wormald. Zunächst wird eine allgemeine Methode zur Generierung von zufälligen regulären Graphen vorgestellt. Diese Methode ist in Algorithmus 8 dargestellt.

Algorithmus 8 : Methode zur Generierung zufälliger d-regulärer Graphen

input : Die Anzahl der Knoten n und der Grad der Knoten d

output : Ein zufällig generierter Graph $G = (V, E)$ mit den gegebenen Eigenschaften

1 Starte mit nd Punkten $\{1, 2, \ldots, nd\}$ (nd gerade) in n Gruppen. Setze $U = \{1, 2, \ldots, nd\}$. (U sei die Menge der ungepaarten Punkte.)

2 Wiederhole den folgenden Schritt bis kein geeignetes Paar mehr gefunden werden kann: Wähle zwei zufällige Punkte i und j in U und, falls diese geeignet sind, paare i mit j und lösche i und j aus U.

3 Erschaffe einen Graph G mit einer Kante von Knoten r zu Knoten s, genau dann wenn es ein Paar zwischen der r-ten und s-ten Gruppe gibt. Falls G d-regulär ist, gib den Graphen aus, kehre sonst zu Schritt 1 zurück.

Es werden in Schritt 1 zunächst nd Punkte erzeugt, wobei jeweils d viele Punkte eine Gruppe bilden. Eine solche Gruppe repräsentiert die Knoten in dem generierten Graphen und jeder der d Punkte einer Gruppe stellt einen Nachbarn dar. In Schritt 2 werden im Anschluss iterativ zwei Punkte gepaart, solange es noch geeignete ungepaarte Punkte gibt. In dem angegebenen Modell sind zwei Punkte *geeignet*, falls sie zu verschiedenen Gruppen gehören und zwischen diesen keine Paarung besteht. Diese Paarungen bilden die Kanten in dem entstehenden Graphen. In Schritt 3 wird der Graph nach den zufällig gewählten Paarungen aufgebaut. Im Falle eines Misserfolges, dass kein d-regulärer Graph entsteht, wird das Verfahren wiederholt.

Theorem 2. *Für $d = o((n/(\log n)^3)^{1/11})$ und $n \to \infty$ entspricht die Verteilung der Graphen, welche durch Algorithmus 8 generiert werden, der Gleichverteilung der d-regulären Graphen der Größe n.*

Darüber hinaus geht die Wahrscheinlichkeit, dass in Schritt 2 des Algorithmus ein regulärer Graph erzeugt wird gegen 1.

Beweis. In Kapitel 3 von [8] werden beide Eigenschaften von Steger und Wormald bewiesen. □

Die obige Methode lässt verschiedene Implementierungen zu. Steger und Wormald schlagen eine effiziente Implementierung von Schritt 2 vor, welche im Folgenden diskutiert wird und für die Generierung der Testgraphen dieser Arbeit in der Programmiersprache Python implementiert wurde. Es sei angemerkt, dass Schritt 1 und 3 auf einfache Art und Weise in $\mathcal{O}(nd)$ implementiert werden können.

Schritt 2 wird in drei Phasen ausgeführt. In der ersten Phase werden zufällig Punkte aus U ausgewählt und gepaart, falls sie geeignet sind. Die Punkte werden dabei in einer Liste L gespeichert. Am Beginn dieser Liste steht U, dahinter in Zweier-Paaren einander bereits zugewiesene Punkte. Darüber hinaus wird eine weitere Liste genutzt, welche an der i-ten Stelle die Position von Punkt i in L speichert. Wir nehmen hier an, dass zwei zufällige Indizes i und j aus U in konstanter Zeit generiert werden können. Im Anschluss können die beiden Punkte in $\mathcal{O}(d)$ auf ihre *Eignung* getestet werden. Dabei müssen $L[i]$ und $L[j]$ in unterschiedlichen Gruppen sein und es muss für jeden weiteren Punkt der Gruppe von $L[i]$ überprüft werden, ob er schon in einer Paarung ist und ob sein Partner zu der Gruppe von $L[j]$ gehört. Es werden solange i und j gezogen, bis geeignete Punkte gefunden worden sind. Schließlich werden diese Punkte mit den letzten beiden in U getauscht und I aktualisiert. U ist nun also zwei Elemente kleiner. Phase 1 endet, sobald U kleiner als $2d^2$ ist.

Da es für jeden Punkt maximal $(d-1)^2$ weitere Punkte gibt, welche keine geeigneten Partner sind und es immer mindestens $2d^2$ Elemente gibt, ist die erwartete Anzahl von Versuchen bis zum Finden eines geeigneten Paares 2. Es werden $\mathcal{O}(nd)$ Paare gebildet und die Überprüfung der Eignung geschieht in $\mathcal{O}(d)$, es folgt eine erwartete Laufzeit von $\mathcal{O}(nd^2)$ für diese Phase.

Phase 2 beginnt also, wenn die Anzahl der Punkte in U kleiner als $2d^2$ ist. Nun werden nicht zwei zufällige Punkte, sondern zwei zufällige Gruppen gewählt, welche noch nicht Grad d haben. Diese Gruppen können mit geringfügigem Zeitaufwand in einer Liste verwaltet werden. Es muss zusätzlich überprüft werden, ob zwischen den Gruppen bereits eine Paarung existiert. Es werden so lange zwei Gruppen gewählt, bis dies nicht der Fall ist. Aus diesen Gruppen werden wiederum zufällig zwei Punkte i und j gewählt und überprüft, ob diese in U liegen. Ist dies der Fall, so werden diese Punkte gepaart. Liegen diese Punkte nicht in U, so werden zwei neue Gruppen und Punkte gewählt. Phase 2 endet, sobald die Anzahl der verfügbaren Gruppen unter $2d$ fällt.

Da es in Phase 2 immer mindestens $2d$ Gruppen gibt und jede maximal $d-1$ Kanten zu anderen Gruppen hat, werden also mit einer Wahrscheinlichkeit von mindestens $1/2$ zwei geeignete Gruppen gewählt. Mit einer Wahrscheinlichkeit größer als $1/d^2$ sind die gefundenen Paare in U. Da $\mathcal{O}(d^2)$ Punkte gepaart werden, ist die erwartete Laufzeit also $\mathcal{O}(d^4)$.

Sind weniger als $2d$ Gruppen verfügbar, so beginnt Phase 3. Wir betrachten nun eine Liste E von allen Paarungen, welche mit den gegebenen Gruppen möglich sind. Die Konstruktion einer solchen Liste ist in $\mathcal{O}(d^2)$ möglich und verbraucht $\mathcal{O}(d^2)$ Platz. Dann wird eine zufällige Paarung dieser Liste gewählt und mit Wahrscheinlichkeit x_{uv}/d^2 akzeptiert. x_{uv} sei dabei das Produkt der Anzahl von geeigneten Punkten in u multipliziert mit der Anzahl von geeigneten Punkten in v. Eine Paarung kann damit in $\mathcal{O}(d^2)$ gefunden werden. Das Updaten der Liste E ist ebenfalls in $\mathcal{O}(d^2)$

möglich. Solange E nicht leer ist, werden Paare gezogen. Da maximal $\mathcal{O}(d^2)$ Paare gebildet werden, liegt die erwartete Laufzeit also in $\mathcal{O}(d^4)$. Paare werden gebildet, solange E nicht leer ist.

In jeder Phase werden zwei zufällige Punkte mit einer Gleichverteilung ausgewählt, damit folgt diese Implementierung den Vorgaben von Algorithmus 8. Damit kann folglich Schritt 2 von Algorithmus 8 für alle $d = \mathcal{O}(n^{1/2})$ mit erwarteter Laufzeit $\mathcal{O}(nd^2)$ und Platzverbrauch $\mathcal{O}(nd)$ implementiert werden.

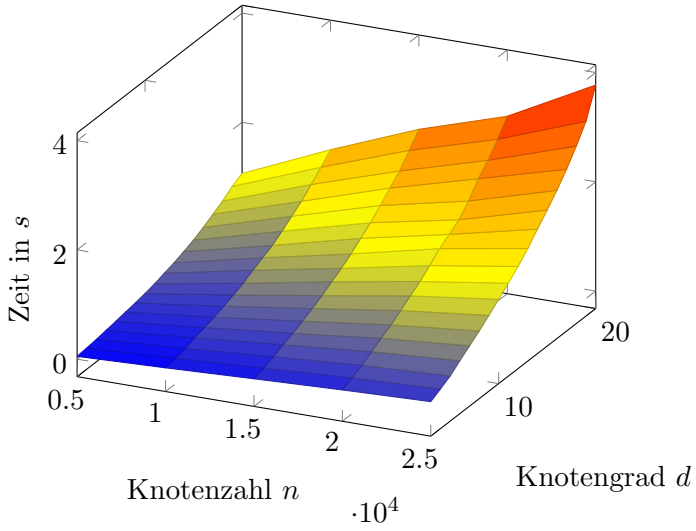

Abbildung 9: Laufzeit der Generierung von zufälligen regulären Graphen

Die Laufzeit des von mir geschriebenen Python-Programms ist in Abbildung 9 dargestellt. Diese entspricht der theoretischen Betrachtung des Algorithmus. Die Laufzeit steigt linear mit der Knotenzahl und quadratisch mit dem Knotengrad.

Es sei angemerkt, dass nach Abschluss von Schritt 2 unter Umständen noch ungepaarte Gruppen vorliegen. In diesem Fall entsteht also kein d-regulärer Graph und entsprechend dem Schema in Algorithmus 8 wird also wieder in Schritt 1 gestartet. Für $d = o((n/\log n)^3)^{1/11}$ geht die Wahrscheinlichkeit eines Scheiterns für $n \to \infty$ gegen 0 und so kann für diese Fälle also ein d-regulärer Graph in erwarteter Zeit $\mathcal{O}(nd^2)$ generiert werden. Darüber hinaus vermuten Steger und Wormald, dass der Algorithmus bis zu einem d von $n/2$ noch eine praktikable Erfolgswahrscheinlichkeit hat und lediglich für $d \gg n^{1/2}$ die Verteilung nicht mehr annähernd uniform ist.

Mit diesem Algorithmus ist es zum Beispiel problemlos möglich kubische Graphen mit 10^6 Knoten zu generieren, wie sie zum Beispiel bei der Laufzeitanalyse der Breitensuche in Abbildung 5 genutzt wurden. Auch größere kantenreiche Graphen (zum Beispiel Graphen mit 5000 Knoten und einem Knotengrad von 50) können in der Praxis schnell generiert werden. Damit ist dieses Verfahren gut geeignet, um Testdaten für die vorgestellten Verfahren zu liefern.

5.2 Zufällige reguläre Graphen mit geringer Bisektionsweite

Für die im vorigen Abschnitt generierten zufälligen d-regulären Graphen ist die optimale Bisektion nicht aus der Konstruktion heraus bekannt, was für einige Analysen von Nachteil ist. Darüber hinaus liegt die Bisektionsweite in $\Theta(m)$ und für die Bewertung der vorgestellten Heuristiken sind auch Graphen mit einer geringeren Bisektionsweite interessant. Die Hoffnung ist, dass die Algorithmen diese kleinen Bisektionen finden und für diese Klasse von Graphen folglich gute Ergebnisse liefern. Aus diesem Grund werden diese Graphen auch in der Forschung genutzt, zum Beispiel in [13] und [2]. Daher wird im Folgenden eine Graphklasse eingeführt, welche eine konstruierte Bisektion enthält, und somit eine Bisektion mit einer gewählten Weite und bekannten dazugehörigen Partitionen besitzt. Jedoch kann nicht garantiert werden, dass dies tatsächlich die optimale Bisektion ist.

Die konstruierte Bisektion besteht aus den Partitionen $\{1, 2, \ldots, n/2\}$ sowie $\{n/2 + 1, n/2 + 2, \ldots, n\}$. Die grundlegende Idee der Generierung solcher Graphen ist es, eine gewisse Anzahl von Kanten zwischen diesen Partitionen einzufügen und im Anschluss die beiden Partitionen durch weitere Kanten bis zur d-Regularität aufzufüllen.

Um diese Idee umzusetzen, wird sich an dem oben beschriebenen Ansatz von Steger und Wormald für zufällige Graphen orientiert. Es werden also $n \cdot d$ Punkte betrachtet, wobei diese in n Gruppen aufgeteilt sind, welche die Knoten des Graphen repräsentieren. Eine Paarung von zwei Punkten ist eine Kante im generierten Graphen. Die Methode zur Generierung solcher Graphen ist in Algorithmus 9 dargestellt.

Algorithmus 9 : Methode zur Generierung zufälliger d-regulärer Graphen mit geringer Bisektionsweite

---

input : Die Anzahl der Knoten n, der Grad der Knoten d und die Größe der konstruierten Bisektion k

output : Ein zufällig generierter Graph $G = (V, E)$ mit den gegebenen Eigenschaften

1 Starte mit nd Punkten $\{1, 2, \ldots, nd\}$ (nd gerade und $n \cdot d/2 - k \equiv 0 \mod 2$) in n Gruppen. Setze $U = \{1, 2, \ldots, nd/2\}$ und $V = \{nd/2 + 1, nd/2 + 2, \ldots, nd\}$. ($U$ sei die Menge der ungepaarten Punkte der ersten, V die Menge der ungepaarten Punkte der zweiten Partition der konstruierten Bisektion.)

2 Wiederhole den folgenden Schritt bis k Paare gefunden wurden: Wähle einen zufälligen Punkt i aus U und einen zufälligen Punkt j aus V und, falls diese geeignet sind, paare i mit j und lösche i aus U und j aus V

3 Wiederhole den folgenden Schritt bis kein geeignetes Paar mehr gefunden werden kann: Wähle zwei zufällige Punkte i und j in U und, falls diese geeignet sind, paare i mit j und lösche i und j aus U.

4 Wiederhole den folgenden Schritt bis kein geeignetes Paar mehr gefunden werden kann: Wähle zwei zufällige Punkte i und j aus V und, falls diese geeignet sind, paare i mit j und lösche i und j aus V.

5 Erschaffe einen Graph G mit einer Kante von Knoten r zu Knoten s, genau dann wenn es ein Paar zwischen der r-ten und s-ten Gruppe gibt. Falls G d-regulär ist, gib den Graphen aus, kehre sonst zu Schritt 1 zurück.

Bei diesem Ansatz werden in Schritt 1 zwei Punktmengen U und V erstellt, welche die beiden Partitionen der konstruierten Bisektion repräsentieren. Dann wird in Schritt 2 damit begonnen, Kanten zwischen den Partitionen einzufügen, indem zwei zufällige geeignete Punkte gewählt werden. Einer dieser Punkte stammt dabei aus der Punktmenge U, der andere aus der Menge V. Die Anzahl der eingefügten Kanten ist damit die Größe des Cuts der konstruierten Bisektion, welche als Parameter k übergeben wurde. Sind dies nun relativ wenige, im Folgenden wird hier als Größe maximal $\sqrt{n}$ verwendet, so ist diese Bisektion mit hoher Wahrscheinlichkeit die minimale. Dabei muss beachtet werden, dass nicht für alle k ein Graph mit den gegebenen Eigenschaften existiert. Jeder Knoten des Graphen soll schließlich d Nachbarn haben und wenn die Summe der Restkapazitäten aller Knoten einer Partition nach dem Einfügen der k Kanten zwischen den Partitionen ungerade ist, so ist die Konstruktion eines solchen d-regulären Graphen unmöglich. Dies liegt daran, dass die Summe durch jede eingefügte Kante um 2 verringert wird, da zwei Knoten verbunden werden, und diese Summe somit niemals 0 werden kann. Dies kann auch mit den Parametern n, d und k ausgedrückt werden, wobei k die Größe des Cuts der konstruierten Bisektion ist. Eine Graphgenerierung ist unmöglich, wenn

$$n \cdot d/2 - k \equiv 1 \mod 2$$

gilt. Es muss daher sichergestellt werden, dass k bei der Generierung dieser Graphen korrekt gewählt wird. In Schritt 3 und 4 werden die restlichen Kanten durch die Wahl von zwei Punkten innerhalb der Partitionen U und V eingefügt. Im Anschluss wird der erstellte Graph ausgegeben, wenn er d-regulär ist, und das Verfahren sonst wiederholt.

Bei der Implementierung dieser Methode ist in erster Linie die Umsetzung von Schritt 3 und 4 von großer Bedeutung. Dieser ist jedoch identisch zu Schritt 2 von Algorithmus 8 und für diesen schlagen Steger und Wormald eine effiziente Implementierung vor, welche im vorigen Abschnitt diskutiert wurde. Diese Implementierung kann hier folglich übernommen werden. In Schritt 2 von Algorithmus 9 werden zwei Punkte aus verschiedenen Mengen gepaart. Da in dieser Arbeit (und bei jeder sinnvollen Anwendung dieses Algorithmus) nur wenige Kanten zwischen den Partitionen eingefügt werden, reicht für diesen Schritt eine naive Implementierung aus, bei der jeweils ein zufälliger Knoten aus U bzw. V gewählt und gepaart werden. Ist die Größe der konstruierten Bisektion zum Beispiel $\sqrt{n}$, so findet man bei der Wahl des ersten Punktes mit einer Wahrscheinlichkeit von mindestens $(n-\sqrt{n})/n$ einen solchen, der mit allen Punkten der anderen Partition gepaart werden kann, da es von der eigenen Gruppe noch keine Paarungen gibt. Für $n \geq 4$ ist diese Wahrscheinlichkeit mindestens $1/2$. Die zufällige Wahl in Schritt 2 ist also in $\mathcal{O}(1)$ möglich. Für konstruierte Bisektionen mit einem größeren Cut kann anstelle einer naiven Implementierung auch die von Steger und Wormald vorgeschlagene Implementierung angepasst werden. Dies wird im folgenden Abschnitt für die Generierung von bipartiten Graphen diskutiert.

Es kann nun die Laufzeit von Algorithmus 9 analysiert werden. Eine Iteration dieses Algorithmus hat die von Algorithmus 8 bekannte asymptotische Laufzeit von $\mathcal{O}(n \cdot d^2)$ für $d = \mathcal{O}(n^{1/2})$. Die Gesamtlaufzeit hängt jedoch von der Erfolgswahrscheinlichkeit ab. Weitere theoretische Fragen sind daher, ob auch für diese Art der Graphgenerierung die Erfolgswahrscheinlichkeit für $n \to \infty$ gegen 1 geht und ob die erhaltenen Graphen gleichverteilt sind.

Für den weiteren Verlauf dieser Arbeit ist vor allem eine praktische Beantwortung dieser Fragen interessant. Ist das angegebene Verfahren geeignet, um zufällige reguläre Graphen mit geringer

Bisektionsweite zu generieren? Eine Antwort kann durch die Betrachtung der Laufzeit gegeben werden.

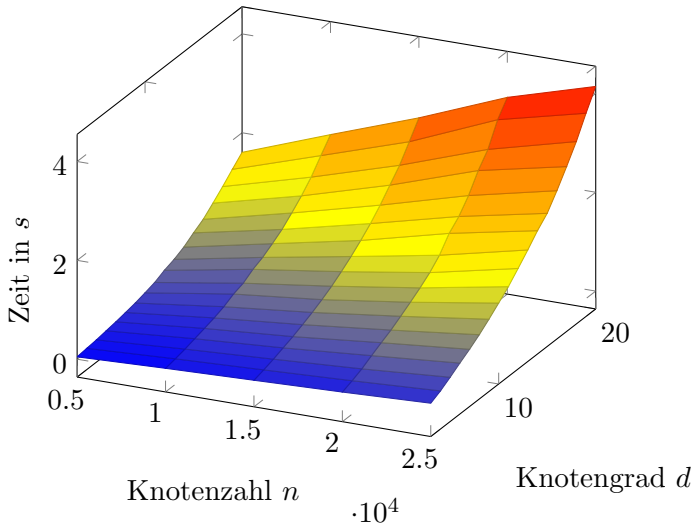

Abbildung 10: Laufzeit der Generierung von Graphen mit geringer Bisektionsweite

Dieses Verfahren habe ich in der Programmiersprache Python implementiert. Die Laufzeit des Verfahrens ist in Abbildung 10 dargestellt und nur leicht höher als die des grundlegenden von Steger und Wormald beschriebenen Verfahrens. Bui et al. verwenden in ihrer Arbeit [2] ein analoges Verfahren, um reguläre Graphen mit geringer Bisektionsweite zu betrachten. Es wird daher davon ausgegangen, dass auch die Verteilung der Graphen angemessen ist.

5.3 Bipartite Graphen

Ein bipartiter Graph besteht aus zwei disjunkten Knotenmengen U und V, sodass jede Kante einen Knoten aus U mit einem Knoten aus V verbindet. Ein 2-regulärer bipartiter Graph mit 5 Knoten in U und V ist in Abbildung 5.3 zu sehen. Sei die Maximierung der Größe des Cuts das inverse Problem zu dem in dieser Arbeit diskutierten Bisektionsproblem. Dann ist offensichtlich, dass dieses Problem, also die Maximierung des Cuts, für bipartite Graphen mit $|U| = |V|$ trivial ist.

Interessanterweise stellen bipartite Graphen dennoch keine Vereinfachung für das Bisektionsproblem dar und das Problem ist auch für diese Instanzen NP-schwer [3]. Es gibt verschiedene Ansätze bipartite Graphen zufällig zu erstellen, von denen einer im Folgenden diskutiert wird.

Der in dieser Arbeit betrachtete Eigenwertansatz ist aus Überlegungen bezüglich regulärer Graphen hergeleitet worden. Darüber hinaus geht es in diesem Kapitel um die Generierung zufälliger regulärer Graphen. Daher werden wir hier *reguläre* bipartite Graphen betrachten. So ist ein Vergleich mit den zufälligen regulären Graphen des vorigen Abschnitts möglich. Für reguläre bipartite Graphen muss $|U| = |V|$ gelten. Um diese Graphen zu generieren, kann das in Abschnitt 5.1 vorgestellte Verfahren von Steger und Wormald angepasst werden.

Im Gegensatz zu der in Algorithmus 8 vorgestellten Methode werden nun zwei Listen mit $n/2 \cdot d$

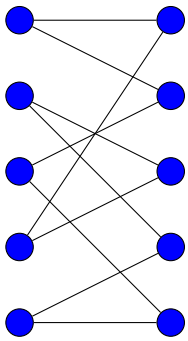

Abbildung 11: Ein 2-regulärer bipartiter Graph mit 10 Knoten

Algorithmus 10 : Methode zur Generierung zufälliger d-regulärer bipartiter Graphen

input : Die Anzahl der Knoten n und der Grad der Knoten d

output : Ein zufällig generierter Graph $G = (V, E)$ mit den gegebenen Eigenschaften

1 Starte mit nd Punkten $\{1, 2, \ldots, nd\}$ (nd gerade) in n Gruppen. Setze $U = \{1, 2, \ldots, nd/2\}$ und $V = \{nd/2 + 1, nd/2 + 2, \ldots, nd\}$. ($U$ und V seien die Mengen der ungepaarten Punkte der beiden Knotenmengen des bipartiten Graphen.)

2 Wiederhole den folgenden Schritt bis kein geeignetes Paar mehr gefunden werden kann: Wähle zwei zufällige Punkte i in U und j in V und, falls diese geeignet sind, paare i mit j und lösche i und j aus U.

3 Erschaffe einen Graph G mit einer Kante von Knoten r zu Knoten s, genau dann wenn es ein Paar zwischen der r-ten und s-ten Gruppe gibt. Falls G d-regulär ist, gib den Graphen aus, kehre sonst zu Schritt 1 zurück.

Punkten verwaltet. Erneut bilden d Punkte eine Gruppe, welche einen Knoten des generierten Graphen repräsentiert. Diese Punktmengen repräsentieren daher die beiden Partitionen U und V. Zur Konstruktion eines bipartiten Graphen wird folglich ein Punkt der ersten Menge mit einem der zweiten gepaart. Diese Strategie ist in Algorithmus 10 dargestellt. Dabei ist vor allem die Implementierung von Schritt 2 interessant. Diese kann analog zu der in Abschnitt 5.1 diskutierten durchgeführt werden. Die drei von Steger und Wormald vorgestellten Phasen werden nun jeweils beendet, sobald die Bedingungen für eine der beiden Punktmengen ungültig werden. Die Laufzeit einer solchen Generierung entspricht also der Laufzeit des Verfahrens von Steger und Wormald, vorausgesetzt, dass die Erfolgswahrscheinlichkeit in Schritt 2 angemessen ist.

Mithilfe dieser Überlegungen habe ich ein Python-Programm geschrieben, welches zufällige bipartite Graphen generiert. In Abbildung 12 ist die Laufzeit dieser Implementierung zu sehen. Die Laufzeiten sind vergleichbar mit denen des Algorithmus von Steger und Wormald und somit aus praktischer Sicht sehr gut anwendbar. In dieser Arbeit wird dieser Generierungsalgorithmus nicht weiter theoretisch auf eine vorliegende Gleichverteilung oder die Erfolgswahrscheinlichkeit untersucht. Es wird davon ausgegangen, dass diese Parameter für eine praktische Nutzung ausreichend sind.

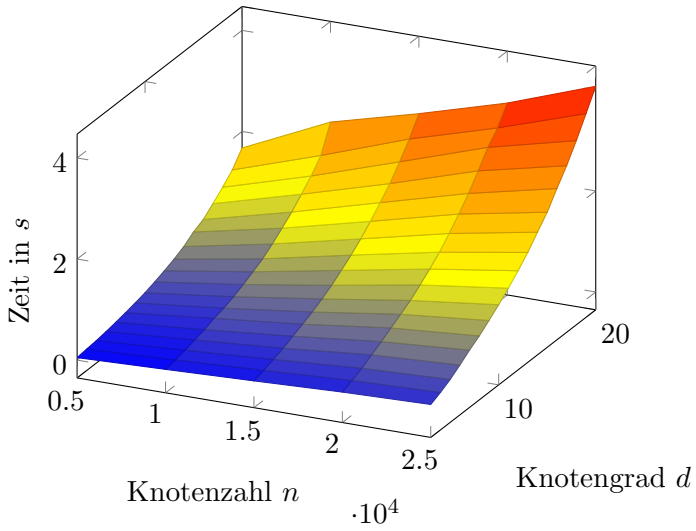

Abbildung 12: Laufzeit der Generierung von d-regulären bipartiten Graphen

5.4 Kubische Kreisgraphen

2-reguläre Graphen bestehen aus Kreisen und sind für das Bisektionsproblem nicht von großem Interesse. Ein großer Teil dieser Arbeit beschäftigt sich jedoch mit kubischen Graphen und daher ist es interessant, diesen zusätzlichen Knotengrad zu nutzen, um innerhalb eines Kreises weitere Querverbindungen einzubauen. In Abbildung 13 ist ein beispielhafter Kreis mit 6 Knoten zu sehen. Im Folgenden werden solche Graphen auch als kubische Kreisgraphen bezeichnet. Es ist eine offene

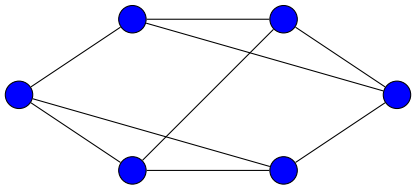

Abbildung 13: Kubischer Kreis mit 6 Knoten

Fragestellung, ob das Bisektionsproblem auch für solche Graphen NP-vollständig ist.

Solche Graphen können ebenfalls mit dem Verfahren von Steger und Wormald, welches in Abschnitt 5.1 vorgestellt wurde, erzeugt werden. Es muss lediglich mit einem Kreis gestartet werden. Diese Strategie ist in Algorithmus 11 dargestellt. Dazu wird Schritt 1 der in Algorithmus 8 vorgestellten Methode erweitert: Nachdem eine Menge mit $3n$ Punkten gebildet wurde, werden Nachbargruppen verbunden. Jede Gruppe besteht aus 3 Punkten. Es wird der erste Punkt jeder Gruppe mit dem zweiten Punkt der nachfolgenden Gruppe verbunden. Nachfolger der letzten Gruppe ist dabei gerade die erste Gruppe. Auf diese Weise entsteht die grundlegende Kreisstruktur des Graphen. Im Anschluss wird das bekannte Verfahren ausgeführt und die weiteren Kanten eingefügt. Die Laufzeit entspricht also auch hier der des Algorithmus von Steger und Wormald, welche für ein festes d linear

Algorithmus 11 : Methode zur Generierung zufälliger kubischer Kreisgraphen

input : Die Anzahl der Knoten n

output : Ein zufällig generierter Graph $G = (V, E)$ mit den gegebenen Eigenschaften

1 Starte mit $3n$ Punkten $\{1, 2, \ldots, 3n\}$ (n gerade) in n Gruppen. Setze $U = \{1, 2, \ldots, 3n\}$. (U sei die Menge der ungepaarten Punkte.)

2 Verbinde den ersten Punkt jeder Gruppe mit zweiten Punkt der nachfolgenden Gruppe. Der Nachfolger der letzten Gruppe ist dabei die erste Gruppe.

3 Wiederhole den folgenden Schritt bis kein geeignetes Paar mehr gefunden werden kann: Wähle zwei zufällige Punkte i und j in U und, falls diese geeignet sind, paare i mit j und lösche i und j aus U.

4 Erschaffe einen Graph G mit einer Kante von Knoten r zu Knoten s, genau dann wenn es ein Paar zwischen der r-ten und s-ten Gruppe gibt. Falls G d-regulär ist, gib den Graphen aus, kehre sonst zu Schritt 1 zurück.

in n ist unter der Voraussetzung, dass in $\mathcal{O}(1)$ Versuchen erfolgreich ein kubischer Graph gefunden werden kann.

Auf analoge Weise können auch Graphen mit Kreisstruktur und einem höheren Knotengrad erzeugt werden. Die Bisektionsweite sowie die Performance der Algorithmen unterscheidet sich für eine steigenden Knotengrad jedoch nicht mehr von zufälligen d-regulären Graphen. Dies ist dadurch zu erklären, dass die erzeugten Kreisgraphen aus einem $(d-2)$-regulären Graphen mit zusätzlichem Kreis bestehen. Die beiden Nachbarn pro Knoten, welche den Kreis bilden, haben für große d einen äußerst geringen Einfluss auf die Bisektionsweite. Daher werden in dieser Arbeit kubische Kreise untersucht.

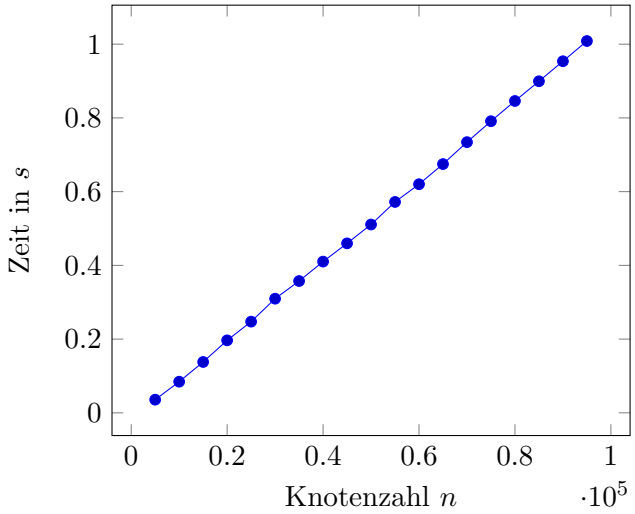

Abbildung 14: Laufzeit der Generierung von kubischen Kreisen

Ich habe diesen Ansatz mit der Programmiersprache Python implementiert. In Abbildung 14

ist die Laufzeit dieses Programms für die Generierung von kubischen Kreisen zu sehen. Die lineare Laufzeit entspricht den Erwartungen. Auch bei diesem Verfahren ist die Erfolgswahrscheinlichkeit hoch. Daher ist dieses Verfahren praktisch gut nutzbar. Wie in der Abbildung 14 zu sehen ist, können auch Graphen mit 10^5 Knoten innerhalb einer Sekunde generiert werden.

6 Resultate auf zufälligen Graphen

In diesem Abschnitt werden die in Kapitel 4 vorgestellten Heuristiken auf den im vorigen Kapitel vorgestellten Graphklassen getestet. Dabei werden zunächst zufällige d-reguläre Graphen für verschiedene d betrachtet.

6.1 Zufällige reguläre Graphen

Es werden zunächst die Ergebnisse auf zufälligen regulären Graphen betrachtet. Bevor diese Resultat, welche in Tabelle 1 dargestellt sind, diskutiert werden, kann zunächst der Erwartungswert der Größe eines zufälligen Cuts auf solchen Graphen betrachtet werden. Diese Größe kann als grober Vergleichswert für die Verfahren genutzt werden, da jede sinnvolle Heuristik bessere Werte liefern sollte. In einem d-regulären Graphen mit n Knoten gibt es $n \cdot d/2$ Kanten. Sei Z die Zufallsvariable, welche angibt, ob eine Kante zwischen den Partitionen liegt. Diese Zufallsvariable ist damit 1, falls dies der Fall ist, und 0 sonst. Jede Kante verbindet zwei Knoten i und j. Die Zufallsvariablen X und Y geben nun an, ob die Knoten i bzw. j in Partition 0 oder 1 sind. Somit kann Z auch durch X und Y ausgedrückt werden. Für die Wahrscheinlichkeit, dass diese in unterschiedlichen Partitionen sind, gilt also:

$$\begin{aligned} P(Z=1) &= P((X=0 \cap Y=1) \cup (X=1 \cap Y=0)) \\ &= \frac{n/2}{n} \cdot \frac{n/2}{n-1} + \frac{n/2}{n} \cdot \frac{n/2}{n-1} \\ &= \frac{n/2}{n-1} \end{aligned}$$

Es folgt für den Erwartungswert $E(n/2 \cdot d \cdot Z)$, welchen wir im Folgenden mit $A(n,d)$ bezeichnen:

$$\begin{aligned} E(n/2 \cdot d \cdot Z) &= n/2 \cdot d \cdot E(Z) \\ &= n/2 \cdot d \cdot \frac{n/2}{n-1} \\ &= \frac{n^2 \cdot d}{4n-4} := A(n,d) \end{aligned}$$

Dieser Wert entspricht für größere n ungefähr $d \cdot n/4$. In Tabelle 1 sind die Ergebnisse von den in Kapitel 4 vorgestellten Ansätzen dargestellt: Ein von einer zufälligen Bisektion gestarteter Greedy, eine Breitensuche von einem zufälligen Knoten gestartet, eine Kombination aus einer n-fachen Breitensuche und anschließendem Greedy sowie ein Eigenwert-Ansatz. Darüber hinaus wurden die Erwartungswerte einer zufälligen Bisektion, nach obiger Formel berechnet, angegeben. Die angegebenen Werte sind der Mittelwert aus zehn Messungen. Die besten Resultate sind dabei hervorgehoben, diese Konvention wird bei der Präsentation von allen weiteren Ergebnissen beibehalten.

Zunächst kann man als erste Einstufung sagen, dass eine einfache Breitensuche bei diesen Graphen am schlechtesten abschneidet. Etwas besser ist ein Greedy, welcher mit einer zufälligen Bisek-

Tabelle 1: Ergebnisse der Verfahren auf zufälligen d-regulären Graphen. Für die Generierung der Daten wurden die Algorithmen auf zehn Graphen ausgeführt und der Mittelwert berechnet. Die besten Ergebnisse sind hervorgehoben.

Graph		Resultate				
d	n	$A(n,d)$	Greedy	BFS	BFS + Greedy	Eigenwert
3	500	375	118	154	**80**	85
3	1000	750	235	313	168	**166**
3	1500	1125	346	467	255	**251**
3	2000	1500	458	626	348	**326**
6	500	751	428	555	**351**	352
6	1000	1501	835	1123	711	**710**
6	1500	2251	1268	1679	1079	**1049**
6	2000	3001	1665	2225	1448	**1403**
10	500	1252	801	1052	**729**	750
10	1000	2502	1589	2147	**1469**	1476
10	1500	3752	2399	3219	**2213**	2233
10	2000	5002	3161	4242	**2956**	2970
30	500	3757	2905	3595	**2824**	2902
30	1000	7507	5822	7154	**5644**	5798
30	1500	11257	8713	10774	**8486**	8651
30	2000	15007	11582	14328	**11331**	11550

tion beginnt. Es sei angemerkt, dass auch eine mehrfache Ausführung des Greedy oder der Breitensuche die Resultate nicht wesentlich verbessert, daher wurden hier die Ergebnisse einer einfachen Ausführung dargestellt. Die Kombination aus Greedy und Breitensuche sowie der Spektralansatz sind am besten und ungefähr gleich gut. Dabei liegen die Stärken von ersterem Verfahren eher bei kleineren Graphen sowie solchen mit höherem Knotengrad. Darüber hinaus ist der einfache Greedy mit größerem d näher an den beiden besten Verfahren und für $d = 30$ schon nahezu gleich gut. Alle Verfahren liefern bessere Bisektionen als der Erwartungswert einer zufälligen Bisektion $A(n, d)$. Hier fällt jedoch auf, dass das Verhältnis von $A(n, d)$ und den praktisch ermittelten Werten mit steigendem Knotengrad immer geringer wird. Dies kann daran liegen, dass die Bisektionsweite der Graphen immer näher am Durchschnittswert liegt oder die Verfahren mit höherem Knotengrad an Effektivität verlieren. Auf diese Beobachtung wird am Ende des Kapitels genauer eingegangen.

6.2 Zufällige reguläre Graphen mit geringer Bisektionsweite

Weitere Tests können auf regulären Graphen mit konstruierter Bisektion durchgeführt werden. In der Tabelle 2 ist die Größe des Cuts k der konstruierten Bisektion der Graphen angegeben. Dies ist eine obere Schranke für die Bisektionsweite b, da bei der Konstruktion auch bessere Bisektionen entstehen können. Für diese Graphklasse ist es interessant zu betrachten, welche Verfahren die konstruierte Bisektion (oder bessere Bisektionen) finden.

Tabelle 2: Ergebnisse der Verfahren auf zufälligen d-regulären Graphen mit konstruierter Bisektion. Die Erfolgswahrscheinlichkeit des Greedy wird durch 1000 Wiederholungen ermittelt. Die Resultate sind der Mittelwert aus 10 Messungen.

Graph			Resultate				
d	n	k	Erfolg Greedy	100-facher Greedy	n-fache BFS	BFS + Greedy	Eigenwert
3	500	22	0,0	94	55	24	**22**
3	1000	32	0,0	204	106	**34**	**33**
3	1500	38	0,0	308	140	41	**38**
3	2000	44	0,0	426	173	49	**43**
3	500	12	0,0	94	30	**12**	**12**
3	1000	16	0,0	206	51	**16**	**16**
3	1500	20	0,0	306	74	**20**	**20**
3	2000	22	0,0	421	88	**22**	**22**
5	500	22	0,3218	**22**	63	**22**	**22**
5	1000	32	0,2292	**32**	113	**32**	**32**
5	1500	38	0,1778	**38**	150	**38**	**38**
5	2000	44	0,1329	**44**	194	**44**	**44**
5	500	0	0,4158	**0**	**0**	**0**	**0**
5	1000	0	0,2964	**0**	**0**	**0**	**0**
5	1500	0	0,2286	**0**	**0**	**0**	**0**
5	2000	0	0,1704	**0**	**0**	**0**	**0**

In Tabelle 2 sind die Ergebnisse von solchen Tests dargestellt. Dabei werden Knotenzahl, Knotengrad und Weite der konstruierten Bisektion variiert. Diese Weite wird im Folgenden auf $\sqrt{n}$, $\sqrt{n}/2$ oder 0 gesetzt. Es ist möglich, dass für diese Anzahl von Kanten zwischen den Partitionen keine d-regulären Graphen erzeugt werden können. Dieser Fall wird bei der Diskussion der Konstruktion dieser Graphen in Abschnitt 5.2 angesprochen. Wenn sich solche Werte für $\sqrt{n}$, $\sqrt{n}/2$ oder 0 ergeben, wird k stattdessen als $\sqrt{n}$, $\sqrt{n}/2$ oder 1 gewählt.

Betrachten wir zunächst die Ergebnisse des Greedy-Verfahrens. Dieses Verfahren wird von einer zufälligen Bisektion gestartet und findet mit einer bestimmten Wahrscheinlichkeit die optimale Bisektion. Wird nicht die optimale Bisektion gefunden, so können die Resultate auch deutlich schlechter sein. Die Erfolgswahrscheinlichkeit, welche durch 1000-malige Ausführung ermittelt wurde, ist ebenfalls in der Tabelle 2 zu finden. Für $d = 3$ sowie auch $d = 4$ (letztere Werte sind in der Tabelle nicht angegeben) findet der Greedy die optimale Bisektion mit hoher Wahrscheinlichkeit nicht. Für $d \geq 5$ hingegen wird mit einer annehmbaren Wahrscheinlichkeit die beste Bisektion gefunden. Diese Wahrscheinlichkeit nimmt mit zunehmender Knotenzahl exponentiell ab. Um die Erfolgswahrscheinlichkeit zu erhöhen, wird das Verfahren mehrfach gestartet. Würde man dieses Verfahren in der Praxis für große Graphen anwenden, so müsste die Anzahl der Wiederholungen exponentiell mit der Graphgröße steigen. Für die Tests auf Graphgrößen bis 2000 Knoten ist jedoch eine Anzahl von 100 Wiederholung aus praktischer Sicht ausreichend und wird hier dementsprechend gewählt. Für

diese Parameter findet der Algorithmus mit hoher Wahrscheinlichkeit die konstruierte Bisektion.

Eine n-fache Breitensuche, diese wird von jedem Knoten gestartet, schneidet für $d = 3$ besser ab als der Greedy-Ansatz und liefert passable Ergebnisse. Für einen höheren Knotengrad wird dieser Ansatz jedoch immer schlechter. Einen positiven Einfluss hat eine geringere Bisektionsweite, sodass der Ansatz solche mit Größe 0 findet. Die Kombination aus Breitensuche und Greedy zeigt nur bei einer hohen Bisektionsweite von $\sqrt{n}$ sowie $d = 3$ kleinere Schwächen. Für große d ist der Greedy im Allgemeinen gut genug, dass das Verfahren nahezu optimal ist. Für dieses Verfahren reicht im Gegensatz zum Greedy bereits ein Knotengrad von 4 zur Findung der optimalen Ergebnisse aus. Es wird deutlich, dass das Verfahren für kleinere Knotengrade von den Stärken der Breitensuche und für größere Knotengrade von denen des Greedys profitiert. Der Eigenwertansatz findet nahezu immer die optimale Bisektion und schneidet bei diesen Tests am besten ab.

6.3 Bipartite Graphen

Es werden nun die Testergebnisse von bipartiten Graphen betrachtet. Für diese Graphklasse ist ein Vergleich zu zufälligen regulären Graphen interessant. Es stellt sich die Frage inwieweit die Eigenschaft, dass der Graph bipartit ist, zu einer Verbesserung oder Verschlechterung der Ergebnisse der einzelnen Verfahren führt. In Tabelle 3 sind die Resultate auf bipartiten Graphen zu sehen. Auf

Tabelle 3: Resultate der Verfahren auf d-regulären bipartiten Graphen. Die Ergebnisse sind der Mittelwert aus 10 Messungen.

Graph		Resultate			
d	n	Greedy	BFS	BFS + Greedy	Eigenwerte
3	500	117	170	**79**	83
3	1000	239	336	**166**	171
3	1500	351	468	256	**246**
3	2000	462	674	333	**332**
6	500	417	562	**349**	356
6	1000	828	1438	**699**	714
6	1500	1260	1858	1067	**1051**
6	2000	1679	2242	1453	**1415**
10	500	792	1252	**725**	751
10	1000	1583	3159	**1448**	1482
10	1500	2412	3692	**2203**	2228
10	2000	3210	4344	**2967**	2978
30	500	2945	5796	**2842**	2940
30	1000	5855	10787	**5666**	5815
30	1500	8735	12461	**8505**	8704
30	2000	11619	14840	**11346**	11548

diesen Graphen schneidet die Breitensuche am schlechtesten ab. Der Greedy liefert etwas bessere

Ergebnisse. Während diese für Graphen mit einer geringen Kantenzahl noch nicht besonders gut sind, steigt die Güte dieses Verfahrens mit der Kantenzahl. Am besten sind die Kombination aus Breitensuche und Greedy sowie der Spektralansatz. Vergleicht man diese Resultate mit denen auf zufälligen d-regulären Graphen in Tabelle 1, so fällt auf, dass die Ergebnisse nahezu identisch sind. Dabei sind sowohl die Größen der gelieferten Bisektionen sehr ähnlich als auch das Verhältnis der einzelnen Algorithmen. Die Eigenschaft, dass ein Graph bipartit ist, scheint also keinen Einfluss auf dessen Bisektionsweite oder auf die Erfolgschancen der hier diskutierten Algorithmen zu haben.

6.4 Kubische Kreisgraphen

Die letzte Graphklasse, welche zufällig generiert wurde, ist die der kubischen Kreise. In Tabelle 4

Tabelle 4: Resultate der Verfahren auf kubischen Graphen mit Kreisstruktur. Die Ergebnisse sind der Mittelwert aus 10 Messungen.

Graph	Resultate			
n	Greedy	BFS	BFS + Greedy	Eigenwerte
250	64	80	**39**	44
500	120	158	**79**	84
750	175	238	**119**	123
1000	234	305	**160**	164
1250	289	380	**200**	207
1500	346	464	**243**	247
1750	406	544	**285**	**285**
2000	461	622	**324**	329

sind die Resultate der Tests dargestellt. Bei der folgenden Analyse ist erneut ein Vergleich mit den Resultaten der zufälligen kubischen Graphen interessant, wie sie in Tabelle 1 zu sehen sind. Auf den kubischen Kreisgraphen liefern der Greedy sowie die Breitensuche die schlechtesten Ergebnisse. Die Resultate sind wiederum vergleichbar mit denen in Tabelle 1. Der Eigenwertansatz liefert ebenfalls sehr ähnliche Resultate. Bei der Kombination aus Greedy-Verfahren und Breitensuche ist hingegen eine leichte Verbesserung der Ergebnisse erkennbar. Dadurch verändert sich hier die Rangordnung der Heuristiken und dieses Verfahren schneidet auf der Graphklasse am besten ab.

6.5 Zusammenfassung

Für eine Bewertung der Verfahren muss erwähnt werden, dass diese Tests auf zufälligen Graphen durchgeführt wurde. Daher sollten vor allem kleinere Unterschiede nur mit Vorsicht zur Kenntnis genommen werden. Diese Ergebnisse geben vielmehr eine erste Einschätzung über die Stärken und Schwächen sowie die Güte der Algorithmen. Dennoch haben die mit dem hier genutzten Verfahren generierten zufälligen Graphen sehr stabile Eigenschaften und verhalten sich sehr ähnlich.

In den Tests hat sich gezeigt, dass sich die verschiedenen Klassen von zufälligen regulären Graphen in Bezug auf die hier behandelten Heuristiken sehr ähnlich verhalten. Dabei gibt es zwischen

den bipartiten und den zufälligen regulären Graphen keine wahrnehmbaren Unterschiede. Bei kubischen Kreisgraphen war die Heuristik aus Breitensuche und Greedy im Gegensatz zu den weiteren zufälligen regulären Graphen durchgehend etwas besser als der Eigenwertansatz, auch wenn die Unterschiede gering waren.

Die Graphklasse mit einer bekannten geringen Bisektionsweite wird auch häufig in theoretischen Betrachtungen von Heuristiken zur Bisektionsfindung genutzt, wie in [13] und in [2]. In letzterem Paper gehen Bui et al. dabei auf die Gründe ein. Für die meisten d-regulären Graphen liegen sowohl die beste als auch die schlechteste Bisektion in $\Theta(m)$. Folglich ist der Faktor zwischen diesen Größen konstant. Wird nun der Knotengrad d erhöht, so nähert sich dieser Faktor dem Wert 1. Damit sind folglich gute und schlechte Bisektionen nicht mehr unterscheidbar. Dieser Effekt kann auch in den in Tabelle 1 vorgestellten Resultaten beobachtet werden. Mit steigendem Knotengrad nähern sich die Bisektionen, welche die Verfahren liefern, dem Wert einer zufälligen Bisektion an. Für Graphen mit einer Bisektionsweite, welche nicht in $\Theta(m)$ liegt, gelten diese Überlegungen nicht. Daher sind Graphklassen mit einer geringen Bisektionsweite unter Umständen besser geeignet, um gute und schlechte Heuristiken zu unterscheiden.

Die Resultate auf solchen Graphen in Tabelle 2 zeigen, dass das Bisektionsproblem für Graphen mit einem Knotengrad größer gleich 5 von drei verschiedenen Heuristiken gelöst werden kann. Eine Erhöhung des Knotengrads führt hier anscheinend zu einer Vereinfachung des Problems aus Sicht dieser Heuristiken. Für kubische Graphen hingegen ist nur der Eigenwertansatz in der Lage die beste Bisektion zu finden. Die Kombination aus Breitensuche und Greedy liefert hier passable, aber nicht die optimalen Ergebnisse. Die weiteren Ansätze schneiden deutlich schlechter ab. Während es korrekt ist, dass diese Graphklasse dabei hilft, gute und schlechte Heuristiken zu unterscheiden, so gilt dies für einen Vergleich von guten Heuristiken nur eingeschränkt. Für kubische Graphen sind Unterschiede auszumachen, aber für einen größeren Knotengrad sind die Probleminstanzen für einen sinnvollen praktischen Vergleich zu einfach.

Aus den Tests auf zufälligen d-regulären Graphen können aus praktischer Sicht weitere Erkenntnisse gewonnen werden. Während die Breitensuche mit ansteigendem Knotengrad immer schlechter wird, verbessern sich die Ergebnisse des Greedys für ein größeres d. Diese Ergänzung der unterschiedlichen Stärken ist eine erste Erklärung, warum die Kombination aus Breitensuche und Greedy so gut abschneidet. Zwischen diesem Verfahren und dem Spektralansatz sind nur wenige Unterschiede auszumachen.

Wir werden daher in der weiteren Arbeit eine größere Anzahl von Graphklassen betrachten, um ein detaillierteres Bild der Performance der angesprochenen Algorithmen zu liefern.

7 Deterministische Graphen

Die nun betrachteten Graphen können entsprechend ihrer Klasse deterministisch konstruiert werden. Im Gegensatz zu dem vorherigen Abschnitt wird die Implementierung der Graphgenerierung für die folgenden Graphklassen nicht diskutiert. Mit den angegebenen Konstruktionsvorschriften ist es ohne Schwierigkeit möglich diese Graphen in $\mathcal{O}(n+m)$ zu generieren.

Eine besonders wichtige Eigenschaft solcher Graphklassen ist, dass die Bisektionsweite bekannt ist. Dies ist für alle hier behandelten Graphen der Fall.

Neben verschiedenen Arten von Gittern werden auch Netzwerktopologien betrachtet. Für diese entspricht die Bisektionsweite dem Bottleneck des Netzwerks und ist daher von Interesse. Weitere Graphen sind *bad inputs* für die behandelten Heuristiken.

Das Greedy-Verfahren sowie die Breitensuche können sowohl einmal von einer zufälligen Startbisektion bzw. einem zufälligen Startknoten gestartet werden als auch mehrfach, wobei in diesem Fall die minimale Bisektion zurückgegeben wird. Dabei wird die Breitensuche, falls dies von Vorteil ist, von allen Knoten gestartet. Das Greedy-Verfahren findet für manche Graphklassen mit einer gewissen Wahrscheinlichkeit die optimale Bisektion und sonst deutlich schlechtere. Das Verfahren wird in diesen Fällen mehrfach gestartet, damit die optimale Bisektion mit einer hohen Wahrscheinlichkeit gefunden werden kann. Die Anzahl der Wiederholungen wird dabei abhängig von der Erfolgswahrscheinlichkeit gewählt.

7.1 Leiter

Wir führen zunächst einen Graphen ein, welcher im Folgenden als Leiter bezeichnet wird. L_k mit $k > 1$ beschreibt dabei den entsprechenden Graphen mit $n = 2k$ vielen Knoten. Eine Verbindung besteht zwischen den Knoten i und $i+1$ für $i \in \{1, 2, \ldots, k-1\} \cup \{k+1, k+2, \ldots, 2k-1\}$. Zusätzlich existieren die Kanten $\{1, k\}$ sowie $\{k+1, 2k\}$. Darüber hinaus gibt es Verbindungen zwischen dem Knoten i und $n+i$ für alle $i \in \{1, 2, \ldots, n\}$. Der Graph besteht also aus zwei verbundenen Reihen von Knoten. In Abbildung 15 sind die Graphen L_4 und L_5 dargestellt.

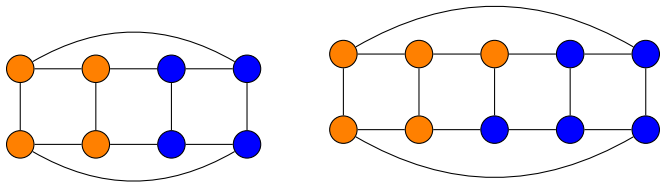

Abbildung 15: Leiter-Graphen L_4 und L_5 mit optimaler Bisektion

Leiter-Graphen sind 3-regulär, jeder Knoten hat zwei Nachbarn in der eigenen Reihe und einen Nachbarn in der gegenüberliegenden Reihe. Für eine optimale Bisektion, welche auch beispielhaft in der Abbildung 15 farbig eingezeichnet ist, gilt:

$$b(n) = \begin{cases} 4 & \text{für } n \text{ gerade} \\ 5 & \text{sonst} \end{cases}$$

Dass dies die tatsächliche Bisektionsweite ist, ist anhand der graphischen Darstellung zu erkennen. So gilt für $k = 2$ und $k = 3$ (oder auch $k = 4$ bzw. $k = 5$) offensichtlich $b = 4$ bzw. $b = 5$. Würde man nun auf der linken und rechten Seite ein weiteres Paar hinzufügen, so wird die Bisektionsweite dadurch nicht geringer und die bisherige Bisektion ist weiterhin optimal. Die Bisektionsweite ist also konstant. Es stellt sich daher die Frage, welche Verfahren in der Lage sind, die optimale oder zumindest eine konstante Bisektion zu finden.

Tabelle 5: Resultate der verschiedenen Algorithmen auf Leiter-Graphen. Die Werte des Greedy sind dabei der Mittelwert von 10 Messungen.

		Resultate			
Graph	b	Greedy	BFS	BFS + Greedy	Eigenwert
L_{125}	5	42	**5**	**5**	**5**
L_{250}	4	78	**4**	**4**	**4**
L_{375}	5	116	**5**	**5**	**5**
L_{500}	4	154	6	**4**	6
L_{625}	5	189	**5**	**5**	**5**
L_{750}	4	232	6	**4**	**4**
L_{875}	5	268	**5**	**5**	**5**
L_{1000}	4	306	6	**4**	6

Die Ergebnisse von Tests auf dieser Graphklasse sind in Tabelle 5 zu sehen. Bei den gegebenen Graphen findet der Greedy-Ansatz Bisektionen mit einer Weite in $\Theta(n)$. Die anderen Ansätze finden konstante Bisektionsweiten, wobei lediglich die Kombination aus Breitensuche und Greedy tatsächlich immer die optimale Bisektion findet. Da dieser Graph regulär ist und eine geringe Bisektionsweite besitzt, ist ein Vergleich mit den Resultaten der zufälligen regulären Graphen mit geringer Bisektionsweite möglich. Die Ergebnisse ähneln sich sehr, bei konstanten Bisektionsweiten scheint eine einfache Breitensuche meist gute Ergebnisse zu liefern. In Abschnitt 7.10 wird hierfür jedoch ein Gegenbeispiel gegeben.

Der Greedy schafft es auf Graphen mit geringem Knotengrad nicht, die optimale Bisektion zu finden. Das liegt daran, dass dieser Algorithmus mit einer zufälligen Bisektion startet. Die Knoten einer Partition sind also über den gesamten Graphen verteilt. Hier ist es nicht möglich sich mit einfachen Vertauschungen dem Optimum anzunähern. Die schlechte Performance des Greedy-Verfahrens auf Graphen mit geringer Bisektionsweite war bereits in vorigen Resultaten im Abschnitt 6 zu beobachten und wird hier bestätigt.

Es kommt vor, dass die Breitensuche und der Eigenwertansatz eine Bisektion der Größe 6 ausgeben, obwohl die Bisektionsweite 4 ist. Die erhaltenen Bisektionen beider Verfahren entsprechen der in Abbildung 16 dargestellten Form. Dabei entsteht ein *verzahnter* Übergang in die andere Partition. Dadurch entstehen zwei weitere Kanten zwischen den Partitionen und der Cut wird größer. Bei der Breitensuche ist es sowohl möglich, dass die dargestellte Bisektion zurückgegeben wird, als auch, dass die optimale Bisektion gefunden wird. Dies hängt von der Reihenfolge ab, in der die Knoten und deren Nachbarn durchlaufen werden.

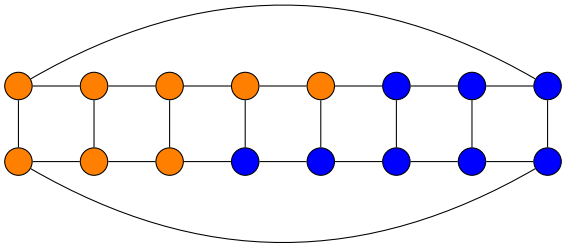

Abbildung 16: Von der Breitensuche sowie dem Eigenwertansatz gefundene Bisektion.

7.2 2D-Gitter

Ein 2D-Gitter-Graph G_k kann man sich als Quadrat mit $k \times k$ Feldern vorstellen. Dabei sind Nachbarn (Nachbarn seien die Felder, welche direkt links, rechts, unten und oben angrenzen) mit einer Kante verbunden. Folglich haben Eckknoten 2 Nachbarn, Knoten am Rand 3 und solche in der Mitte 4 Nachbarn. In Abbildung 17 ist ein 2D-Gitter dargestellt. Ist k ungerade, so ist dies auch die Knotenzahl n und daher ist keine Partitionierung in zwei gleich große Knotenmengen möglich. Hier wird also eine Partition ein Element mehr enthalten.

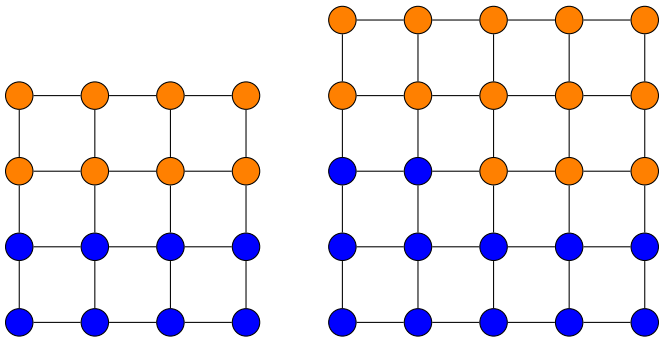

Abbildung 17: Gitter G_4 sowie G_5 mit farbig markierter Bisektion

Ein Gitter G_k besitzt die folgende Bisektionsweite:

$$b_{\text{opt}}(k) = \begin{cases} k & \text{für } k \text{ gerade} \\ k+1 & \text{sonst} \end{cases}$$

In Abbildung 17 sind Bisektionen eingezeichnet, welche die entsprechende Formel erfüllen. Bei gerader Knotenzahl kann immer ein vertikaler oder horizontaler Cut in die Graphmitte eingefügt werden. Somit werden k Verbindungen getrennt. Ist die Knotenzahl ungerade, so ist die Trennung einer weiteren Kante notwendig. Eine kleinere Bisektionsweite existiert nicht, hier kann analog zum 2D-Torus im folgenden Abschnitt argumentiert werden. Die Bisektionsweite steigt also linear mit k.

Wir können nun die Verfahren auf 2D-Gitter-Graphen testen. Die Resultate sind in Tabelle 6 zu sehen.

Tabelle 6: Resultate auf 2D-Gitter-Graphen

		Resultate			
Graph	b	Greedy	n-fache BFS	BFS + Greedy	Eigenwerte
G_{20}	20	121	36	**32**	38
G_{25}	26	199	46	**41**	48
G_{30}	30	307	56	**51**	58
G_{35}	36	412	66	**61**	68
G_{40}	40	541	76	**71**	78

Es ist zu erkennen, dass kein Verfahren die optimale Bisektion findet. Der Greedy schneidet mit Abstand am schlechtesten ab und liefert eine Bisektion in $\Theta(n)$. Die anderen Verfahren liefern ähnliche Bisektionen, wobei die Kombination aus Breitensuche und Greedy die besten Resultate liefert. Dies liegt hauptsächlich an der Breitensuche, welche hier etwas besser als der Spektralansatz ist. Alle Verfahren finden Bisektionen in $\Theta(\sqrt{n}) = \Theta(k)$. Dennoch sind die gefundenen Bisektionen fast doppelt so groß wie die optimale. Dies ist für eine simple Graphklasse aus praktischer Sicht kein besonders gutes Ergebnis. In Abbildung 18 sind die resultierenden Bisektionen auf einem 40×40

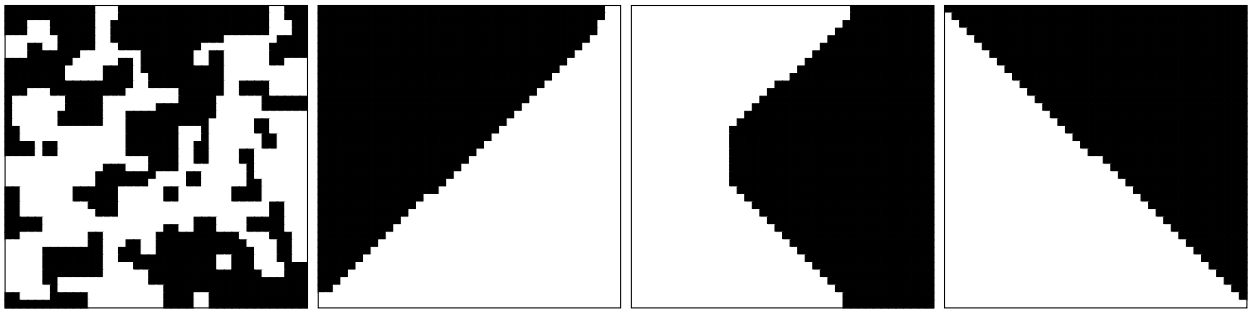

Abbildung 18: Gitter der Größe 40×40 als Schwarz-Weiß-Pixelbild mit Bisektion. Reihenfolge von links nach rechts: Greedy, BFS, BFS+Greedy, Eigenwerte.

Gitter als Schwarz-Weiß-Pixelbild dargestellt. Jedes Pixel entspricht dabei einem Knoten und die Farbe gibt die Partitionszugehörigkeit an. Die Reihenfolge der Bilder entspricht der in Tabelle 6 und lautet von links nach rechts: Greedy, BFS, BFS+Greedy, Eigenwerte. Es ist zu erkennen, warum der Greedy hier am schlechtesten abschneidet. Es bilden sich einzelne Flecken, welcher einer Partition zugehörig sind, diese können jedoch nicht zusammengeführt werden. Gute Partitionen sind auf dieser Graphklasse jedoch zusammenhängend. Es ist interessant, dass die Breitensuche und der Eigenwertansatz ganz ähnliche Ergebnisse liefern. Hier wird das Gitter diagonal durchtrennt. Die Kombination aus Breitensuche und Greedy schafft es besser als die anderen Verfahren einfarbige Spalten zu bilden, welche die Größe des Cuts nicht erhöhen.

7.3 2D-Torus

Ein 2D-Torus ist ähnlich dem 2D-Gitter aufgebaut. Hier existieren jedoch weitere Verbindungen. In jeder Zeile und Spalte sind das erste und letzte Element miteinander verbunden. Jeder Knoten hat hier also vier Nachbarn, damit ist der Graph 4-regulär. T_k sei der 2D-Torus mit $k^2 = n$ Knoten. Wie beim 2D-Gitter ist auch hier eine ungerade Knotenzahl möglich. Für diese Graphklasse ist aufgrund des ähnlichen Aufbaus vor allem ein Vergleich zum 2D-Gitter interessant.

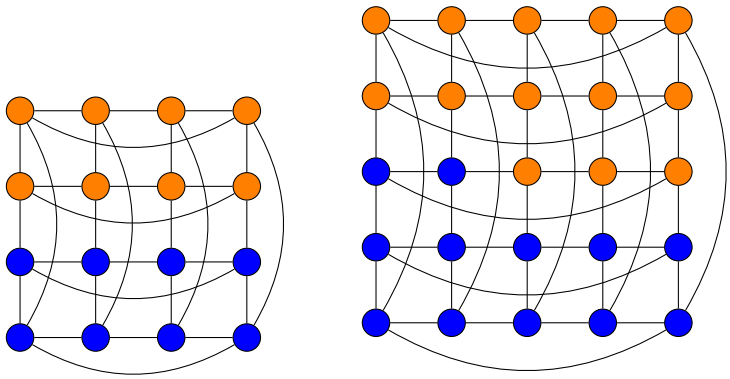

Abbildung 19: Torus T_4 sowie T_5 mit farbig markierter Bisektion

Zunächst kann jedoch die Bisektionsweite dieser Graphklasse hergeleitet werden. Um eine untere Schranke für die Weite der optimalen Bisektion zu finden, können folgende Überlegungen für gerade k getroffen werden. Man stelle sich vor, die Knoten werden zu gleichen Teilen schwarz und weiß gefärbt, wobei die Farben die beiden Partitionen der Bisektion darstellen. Dann erhöht jede Zeile oder Spalte, welche nicht einfarbig ist, die Bisektion mindestens um den Wert zwei. Angenommen es gäbe eine Bisektion mit einer Weite kleiner als $2k$. Dann wären also mehr als k Zeilen und Spalten einfarbig, da weniger als k Zeilen und Spalten zweifarbig sind. Daraus folgt, dass sowohl Zeilen als auch Spalten einfarbig sein müssen und folglich alle einfarbigen Zeilen bzw. Spalten die gleiche Farbe haben. Somit ist für diese Partition die Anzahl der Spalten oder die der Zeilen größer als $k/2$ und die Fläche somit größer als $k^2/2$, dies ist ein Widerspruch. Folglich ist jede Bisektionsweite größer gleich $2k$ und Abbildung 19 zeigt, dass dieser Wert auch erreicht werden kann, indem für die Partitionen jeweils $k/2$ benachbarte Zeilen bzw. Spalten gewählt werden. Der Fall für ungerade k verläuft analog, es muss lediglich eine weitere Zeile bzw. Spalte aufgeteilt werden. Es gilt also:

$$b(k) = \begin{cases} 2k & \text{für } k \text{ gerade} \\ 2(k+1) & \text{sonst} \end{cases}$$

Die Bisektionsweite von Tori steigt also linear mit k und da ein Graph k^2 Knoten hat, liegt das Wachstum in $\mathcal{O}(\sqrt{n})$. In Tabelle 7 sind die Ergebnisse auf Tori verschiedener Größen dargestellt.

Es sind auf den ersten Blick mehrere Unterschiede im Vergleich zum ähnlich aufgebauten 2D-Gitter. Der Eigenwertansatz ist hier mit Abstand am besten und deutlich besser als bei einem 2D-Gitter, obwohl dieser auch nicht die optimale Bisektion findet. Dies lässt sich unter Umständen

Tabelle 7: Resultate auf 2D-Tori

		Resultate			
Graph	b	Greedy	BFS	BFS + Greedy	Eigenwerte
G_{20}	40	151	**76**	**76**	**76**
G_{25}	52	233	96	94	**60**
G_{30}	60	337	116	116	**76**
G_{35}	72	457	136	134	**92**
G_{40}	80	588	156	156	**100**

dadurch erklären, dass der 2D-Torus ein regulärer Graph ist und der hier betrachtete Eigenwertansatz für diese Graphen entwickelt wurde.

Eine Breitensuche liefert im Gegensatz zum 2D-Gitter unabhängig vom Startknoten immer das gleiche Ergebnis. Daher wird hier die Breitensuche einmal von einem zufälligen Knoten gestartet. Darüber hinaus liefern die Breitensuche und die Kombination mit dem Greedy nahezu identische Ergebnisse. Folglich kann die von der Breitensuche erhaltene Bisektion nicht weiter lokal optimiert werden. Die Ergebnisse dieser Verfahren sind wie schon beim 2D-Gitter ungefähr doppelt so groß wie die optimale Bisektion. Ein einfacher Greedy schneidet wiederum am schlechtesten ab und liefert Ergebnisse in $\Theta(n)$.

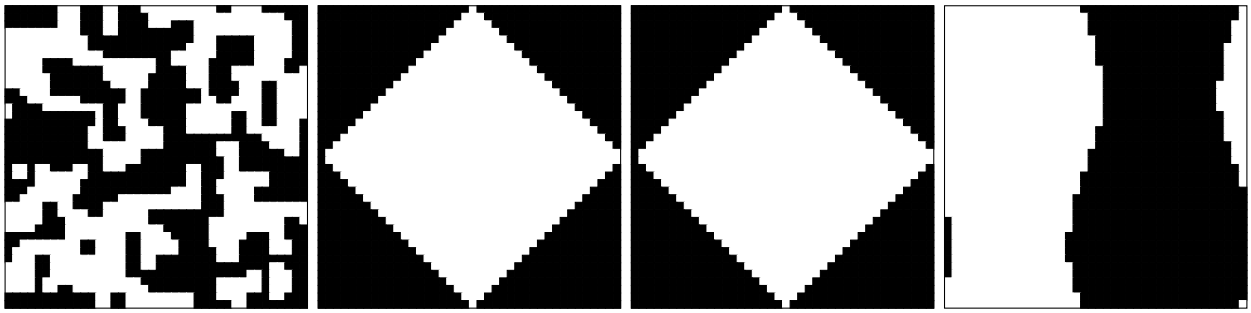

Abbildung 20: Torus der Größe 40×40 als Schwarz-Weiß-Pixelbild mit Bisektion. Reihenfolge von links nach rechts: Greedy, BFS, BFS+Greedy, Eigenwerte.

In Abbildung 20 sind die Resultate der Verfahren auf 2D-Tori als Pixelbild dargestellt. Eine Partition wird dabei durch weiße Pixel, die andere durch schwarze Pixel repräsentiert. Das Ergebnis des Greedy-Verfahrens ist ähnlich dem auf 2D-Gittern und daher nicht überraschend. Die iterative Verbesserung durch den Austausch einzelner Knoten ist nicht in der Lage verschiedene Komponenten einer Partition zusammenzuführen. Das Ergebnis der Breitensuche hingegen überrascht nicht, es entstehen zwei verbundene Partitionen. Diese können anschließend nicht durch einen Greedy verbessert werden. Das Resultat des Eigenwertansatzes hat eine Form, die an eine Schwingung erinnert. Der Zusammenhang von Eigenwerten und Schwingungen wurde bereits in 4.4 diskutiert und ist daher nicht überraschend. Es ist darüber hinaus offensichtlich, dass dieses Resultat das beste der vier abgebildeten ist.

7.4 Rook-Graph

Als Rook-Graph werden Graphen ähnlich dem 2D-Gitter bezeichnet, bei denen alle Knoten in einer Zeile bzw. Spalte benachbart sind. Der Graph beschreibt somit also die legalen Züge eines Turms (engl. rook) im Schachspiel, wenn die Knoten als Felder interpretiert werden, und erhält dadurch seinen Namen. Es sei R_k der entsprechende Graph mit einem $k \times k$-Gitter und somit $n = k^2$ vielen Knoten. Jeder Knoten hat $k - 1$ Nachbarn in der gleichen Zeile und $k - 1$ viele Nachbarn in der gleichen Spalte. Der Graph R_k ist damit also $2(k - 1)$-regulär. Die Graphen R_3 und R_4 sind in Abbildung 21 dargestellt.

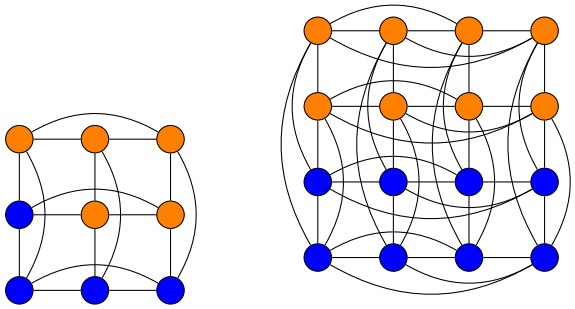

Abbildung 21: Rook-Graph R_3 und R_4 mit farbig markierter optimaler Bisektion

Es wird deutlich, dass bereits kleine Graphen aufgrund der hohen Kantenzahl recht unübersichtlich werden. Es ist außerdem eine Bisektion eingezeichnet, welche das Gitter in der Mitte halbiert. Aufgrund der Symmetrie spielt es keine Rolle, ob dieser Schnitt vertikal oder horizontal ist. Für gerade n schneidet eine optimale Bisektion also n Zeilen in der Mitte, damit gibt es pro Zeile $(n/2)^2$ Kanten zwischen den Partitionen. Für ungerade n wird noch eine weitere Spalte geschnitten, es folgt:

$$b(k) = \begin{cases} k^3/4 & \text{für } k \text{ gerade} \\ (k+1) \cdot \lfloor k/2 \rfloor \cdot \lceil k/2 \rceil & \text{sonst} \end{cases}$$

Wie auch die Resultate zeigen, kann vermutet werden, dass dies tatsächlich die optimale Bisektion ist.

In Tabelle 8 sind die Ergebnisse der Verfahren dargestellt. Der Greedy liefert hier mit einer gewissen Wahrscheinlichkeit die optimalen Ergebnisse. Wird nicht die optimale Bisektion gefunden, sind die Ergebnisse teilweise deutlich schlechter. Die Wahrscheinlichkeit, dass der Greedy die optimale Bisektion liefert, ist in der Tabelle angegeben und wurde durch eine 1000-fache Ausführung berechnet. Es ist zu erkennen, dass die Wahrscheinlichkeit für ungerade k höher ist und im Allgemeinen abnimmt. Damit der Greedy mit hoher Wahrscheinlichkeit die optimalen Ergebnisse liefert, ist eine mehrfache Ausführung nötig. Für eine Nutzung dieses Verfahrens für größere Graphen, müsste die Anzahl der Wiederholungen des Greedy von n abhängig sein. Für die Graphgrößen, welche bei diesen Experimenten betrachtet werden, ist eine 100-fache Ausführung jedoch ausreichend. In diesem Fall liefert der Greedy die besten und auch die optimalen Ergebnisse. Dieses Ergebnis ist nicht überraschend, da bereits in Kapitel 6 beobachtet wurde, dass der Greedy mit ansteigender

Tabelle 8: Resultate auf Rook-Graphen. Die Erfolgswahrscheinlichkeit des Greedy-Verfahrens wurde durch 1000-fache Ausführung berechnet.

		Resultate				
Graph	b	Erfolg Greedy	100-facher Greedy	n-fache BFS	BFS + Greedy	Eigenwerte
R_{20}	2000	0,297	**2000**	2180	2180	2628
R_{25}	4056	0,389	**4056**	4078	4056	5147
R_{30}	6750	0,218	**6750**	7170	7170	8944
R_{35}	11016	0,294	**11016**	11048	11016	14158
R_{40}	16000	0,130	**16000**	16760	16760	21174

Kantenzahl besser wird. Wie auch auf den bereits betrachteten Gitter-Graphen liefert eine Breitensuche, welche von jedem Knoten gestartet wird, bereits gute Ergebnisse. Darüber hinaus kann ein anschließender Greedy die Ergebnisse nicht wesentlich verbessern. Interessant ist bei der Betrachtung des Greedy-Verfahrens, dass die von der Breitensuche gelieferten Startbisektionen nicht für weitere Verbesserungen geeignet sind, zufällige Startbisektionen jedoch durchaus zur optimalen Bisektion führen. Die Wahl der Startbisektion durch eine Breitensuche kann also auch negative Auswirkungen auf die Güte des Greedy-Verfahrens haben. Der Spektralansatz ist für diese Graphklasse mit Abstand am schlechtesten und damit schlechter als das Greedy-Verfahren und die n-fache Breitensuche.

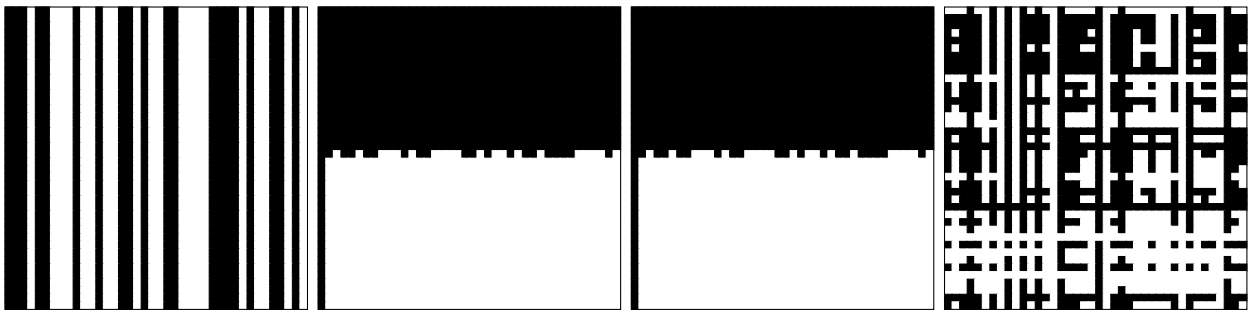

Abbildung 22: Rook-Graph mit 1600 Knoten als Schwarz-Weiß-Pixelbild mit Bisektion. Reihenfolge von links nach rechts: Greedy, BFS, BFS+Greedy, Eigenwerte.

Abbildung 22 zeigt die Bisektionen, welche die Verfahren auf Rook-Graphen liefern, als Pixelbild. Das Greedy-Verfahren liefert optimale Ergebnisse, da hier alle Knoten einer Spalte der gleichen Partition zugeordnet sind. Die Breitensuche liefert nahezu optimale Ergebnisse, die durch den Greedy nicht mehr verbessert werden können. Dabei ist zu beachten, dass am linken Bildrand eine Spalte mit schwarzen Pixeln ist. Diese wird dann durch die Breitensuche bis zur Mitte besucht, wobei auch die weiteren Knoten der entsprechenden Zeile entdeckt werden. Der Eigenwertansatz liefert eine Bisektion, bei der sowohl horizontale als auch vertikale Linien entstehen. Eine Begründung hierfür ist, dass der zweitgrößten Eigenwert mit einer gewissen Häufigkeit (hier 78-fach) auftritt und ein

Unterraum entsteht. Wird nun eine Linearkombination der Eigenvektoren gebildet, so entsteht ein solches Muster. Dieses ist jedoch keine besonders gute Bisektion, weshalb dieses Verfahren hier schlechte Ergebnisse liefert.

7.5 King-Graph

Ein King-Graph K_k basiert auf einem $k \times k$ großen Gitter bei dem zwei Knoten verbunden sind, falls sie horizontal, vertikal oder diagonal benachbart sind. Der Name hat seinen Ursprung in den legalen Zügen eines Königs (engl. king) im Schachspiel, welche analog zu den Kanten im Graphen sind. Der Graph ist nicht regulär, da Eckknoten 3 Nachbarn besitzen, Randknoten jedoch 5 und innere Knoten 8. In Abbildung 23 sind die Graphen K_3 und K_4 mit optimaler Bisektion eingezeichnet.

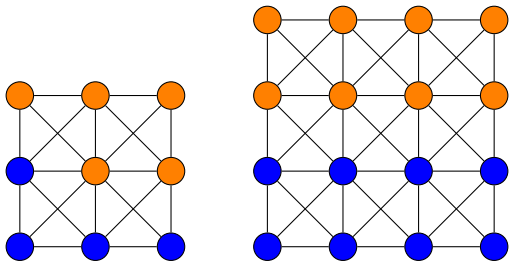

Abbildung 23: King-Graph K_3 und K_4 mit farbig markierten Bisektionen

Es kann vermutet werden, dass eine Bisektion dieser Art auch für größere Graphen optimal ist. Dann werden also k vertikale und $2(k-1)$ diagonale Kanten durchschnitten. Ist k ungerade, so wird eine zusätzliche horizontale Kante durchtrennt. Es folgt also:

$$b(k) = \begin{cases} k + 2(k-1) & \text{für } k \text{ gerade} \\ k + 2(k-1) + 1 & \text{sonst} \end{cases}$$

Wir betrachten nun in Tabelle 9 die experimentellen Resultate der Verfahren.

Tabelle 9: Resultate auf King-Graphen

		Resultate			
Graph	b	Greedy	n-fache BFS	BFS + Greedy	Eigenwerte
K_{20}	58	146	**58**	**58**	84
K_{25}	74	258	101	**74**	96
K_{30}	88	410	**88**	**88**	130
K_{35}	104	499	**104**	**104**	149
K_{40}	118	625	**118**	**118**	173

Der Greedy-Ansatz schneidet für diese Graphklasse am schlechtesten ab. Eine Breitensuche, welche von jedem Knoten gestartet wird, findet für gerade k die optimale Bisektion. Wird im Anschluss noch ein Greedy zur lokalen Optimierung gestartet, so liefert das Verfahren immer optimale

Ergebnisse. Der Eigenwertansatz hingegen schneidet schlechter ab. Hier muss noch erwähnt werden, dass diese Resultate in hohem Maße von der zufälligen Verteilung von Werten auf dem Median des Eigenvektors abhängen. Das bedeutet, dass eine große Anzahl von Einträgen des Eigenvektors direkt auf dem Median liegen. Das hier verwendete Verfahren ist nicht in der Lage diese optimal zu verteilen.

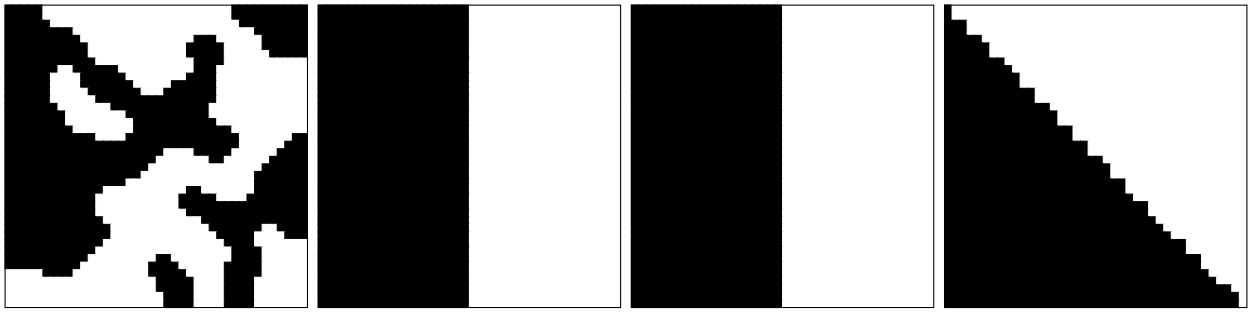

Abbildung 24: King-Graph mit 1600 Knoten als Schwarz-Weiß-Pixelbild mit Bisektion. Reihenfolge von links nach rechts: Greedy, BFS, BFS+Greedy, Eigenwerte.

In Abbildung 24 sind die Resultate der Verfahren auf King-Graphen mit 1600 Knoten als Pixelbild dargestellt. Das Greedy-Verfahren ist nicht in der Lage zwei zusammenhängende Partitionen zu bilden. Dennoch entstehen hier deutlich weniger *Flecken*, als dass beim 2D-Gitter und 2D-Torus der Fall war. Die größere Anzahl von Kanten scheint hier dazu zu führen, dass sich einzelne Komponenten besser verbinden können. Die Breitensuche findet für diese Graphen die optimale Bisektion. Der Eigenwertansatz schneidet den Graph diagonal in zwei Hälften. Eine ähnliche Bisektion ist bereits vom 2D-Gitter bekannt. Im Pixelbild ist darüber hinaus gut zu erkennen, dass viele Werte direkt auf dem Median liegen und zufällig verteilt werden. Dies sind die Knoten auf der Diagonale des Gitters, auf der ein unregelmäßiges Zackenmuster entsteht.

7.6 Hypercube

Beim Hypercube Graph H_k werden die Knoten als Elemente des Alphabets $\{0,1\}^k$ interpretiert. Es gibt also $n = |\{0,1\}^k| = 2^k$ viele Knoten. Dabei sind zwei Knoten genau dann benachbart, wenn sie sich an genau einer Stelle unterscheiden oder anders formuliert ihr Hamming-Abstand 1 ist. Jeder Knoten hat somit k Nachbarn und der Graph H_k ist k-regulär. Die optimale Bisektionsweite beträgt $b = 2^{k-1} = n/2$. Dies wird von Stacho und Vrt'o in [26] gezeigt. Der Hypercube-Graph $\mathcal{H}_3$ ist in Abbildung 25 veranschaulicht. Es ist erkennbar, dass die Bisektionsweite $2^{3-1} = 4$ ist und es neben der durch eine rote Linie eingezeichneten Bisektion zwei weitere mit der gleichen Weite gibt. Tatsächlich besitzt ein Hypercube immer k-viele optimale Bisektionen, welche folgendermaßen beschrieben werden können. Bei der i-ten Bisektion sind alle Knoten mit einer 0 an der i-ten Stelle in Partition V_0 und alle Knoten mit einer 1 an der i-ten Stelle in Partition V_1. Es können nun die Ergebnisse der Verfahren auf Hypercubes untersucht werden.

In Tabelle 10 sind die Ergebnisse der verschiedenen Verfahren zu sehen. Bei diesen Graphen findet das Greedy-Verfahren die optimale Bisektion nur bei einigen Startbisektionen. Die Erfolgs-

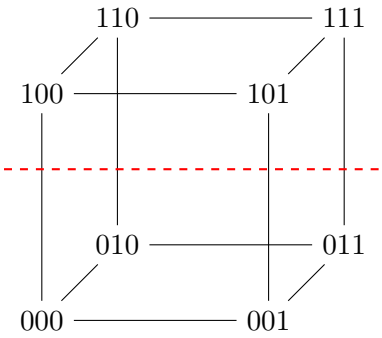

Abbildung 25: Hypercube-Graph $\mathcal{H}_3$ mit beispielhaftem Cut

Tabelle 10: Resultate auf Hypercubes

		Resultate				
Graph	b	Erfolg Greedy	1000-facher Greedy	n-fache BFS	BFS + Greedy	Eigenwerte
H_4	8	0,467	**8**	12	12	12
H_5	16	0,48	**16**	30	**16**	26
H_6	32	0,113	**32**	60	60	60
H_7	64	0,102	**64**	140	96	104
H_8	128	0,016	**128**	280	280	252
H_9	256	0,023	**256**	630	448	570
H_{10}	512	0,002	**512**	1260	1260	1164
H_{11}	1024	0,001	**1024**	2772	1976	2226

wahrscheinlichkeit ist dabei in der Tabelle zu sehen. Für größere Graphen geht diese sehr schnell gegen 0. Daher wurde dieses Verfahren 1000-fach gestartet. Für größere Graphen kann jedoch nicht davon ausgegangen werden, dass die optimale Bisektion auf diesem Wege gefunden wird. Die n-fach gestartete Breitensuche liefert die schlechtesten Resultate, liegt aber zumindest für gerade k in der Nähe der anderen Verfahren. Für diese Werte ist auch ein anschließender Greedy nicht in der Lage, die Bisektion zu verbessern. Dies könnte daran liegen, dass für gerade k jeder Knoten auch eine gerade Anzahl von Nachbarn hat. Es ist also möglich, dass ein Knoten genauso viele Verbindungen in die andere Partition wie in die eigene hat. In diesen Fällen kann der Greedy keine Verbesserung durch einen einfachen Austausch vornehmen. Im ungeraden Fall hingegen schneidet dieser Ansatz am zweitbesten ab. Der Spektralansatz liefert ähnlich schlechte bzw. etwas schlechtere Ergebnisse als letzterer Ansatz. Der Grund liegt darin, dass es mehrere Eigenvektoren zum gleichen zweitgrößten Eigenwert gibt, welche die k optimalen Bisektionen repräsentieren. Der Grund hierfür liegt daran, dass es k-viele optimale Bisektionen für den Graph H_k gibt. Folglich entsteht ein k-dimensionaler Unterraum und durch die Spektralanalyse werden nicht unbedingt die gewünschten Vektoren gefunden, sondern andere Linearkombinationen aus diesem Unterraum. Diese liefern jedoch nicht unbedingt gute Bisektionen.

7.7 Butterfly-Netzwerk

Das $(\log k)$-dimensionale Butterfly-Netzwerk B_k besteht aus $n = k(\log k + 1)$ Knoten, welche in $\log k + 1$ Ebenen aus n Knoten angeordnet sind. Jeder Knoten kann durch ein Tupel $\langle w, i\rangle$ beschrieben werden, wobei $i \in \{0, 1, \ldots, \log k\}$ die Ebene und die Binärzahl $w \in \{0, 1\}^{\log k}$ die Spalte des Knoten angibt. Zwei Knoten $\langle w, i\rangle$ und $\langle w', i'\rangle$ sind verbunden, falls $i' = i + 1$ gilt und w und w' identisch sind oder sich nur durch das Bit an der Stelle i' (an der Stelle 1 sei das höchste Bit, an der Stelle $\log n$ das niedrigste) unterscheiden. In Abbildung 26 ist das Butterfly-Netzwerk B_8 dargestellt [27].

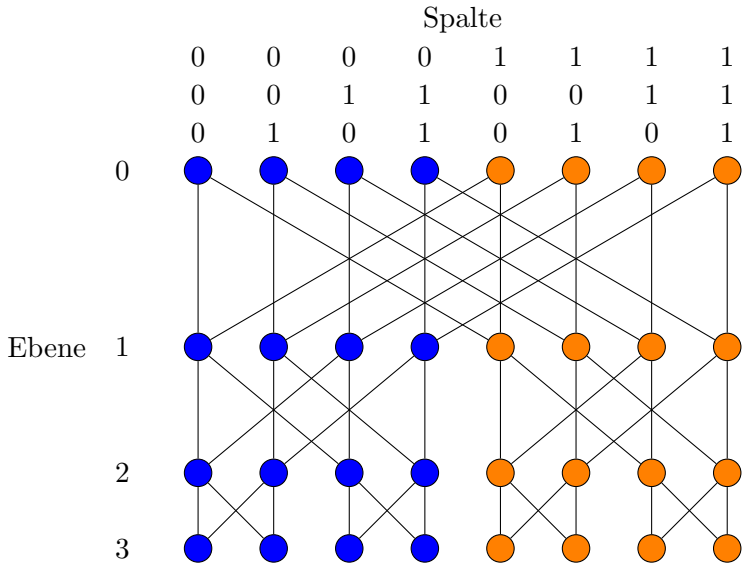

Abbildung 26: Butterfly-Netzwerk B_8 mit 32 Knoten und optimaler Bisektion

Ein Spezialfall des Butterfly-Netzwerks liegt vor, wenn die Knoten auf Ebene 0 und $\log k$ als gleich angenommen werden. Der Graph hat dann lediglich $k \log k$ Knoten und wird als *Wrap-around*-Butterfly-Netzwerk bezeichnet. Dieses Netzwerk ist 4-regulär.

Bornstein et al haben in [27] gezeigt, dass die Bisektionsweite des klassischen Butterfly-Netzwerkes $2(\sqrt{2} - 1)k + o(k) \approx 0.82k$ ist und nicht, wie häufig angenommen, k. Das Wrap-Around-Butterfly-Netzwerk hingegen hat eine Bisektionsweite von k.

In Tabelle 11 sind zunächst die Ergebnisse auf normalen Butterfly-Netzwerken zu sehen. Da die optimale Bisektion nicht bekannt ist, wurde in der Tabelle k als obere Schranke für die Bisektionsweite angegeben. Der Eigenwert-Ansatz ist hier tatsächlich in der Lage diese Bisektion zu finden. Die anderen Verfahren schneiden schlechter ab, wobei sich die Kombination aus Breitensuche und Greedy noch am besten präsentiert. Die von jedem Knoten gestartete Breitensuche liefert etwas schlechtere, jedoch immer noch passable Ergebnisse. Der Greedy ist hingegen nicht besonders gut. Dies war zu vermuten, da der maximale Knotengrad 4 beträgt und der Greedy oftmals erst ab einem Grad von 5 gute Ergebnisse liefert, wie zum Beispiel in Abschnitt 6 zu beobachten war.

Die Ergebnisse auf Wrap-Around-Butterfly-Netzwerken sind in Abbildung 12 dargestellt. Zunächst

Tabelle 11: Resultate der Verfahren auf Butterfly-Netzwerken

		Resultate			
Graph	k	z-facher Greedy	n-fache BFS	BFS + Greedy	Eigenwerte
B_4	4	5	**4**	**4**	**4**
B_8	8	10	10	**8**	**8**
B_{16}	16	27	20	**16**	**16**
B_{32}	32	59	40	**32**	**32**
B_{64}	64	155	92	**64**	**64**
B_{128}	128	371	224	**128**	**128**
B_{256}	256	866	448	272	**256**
B_{512}	512	1937	896	778	**512**

Tabelle 12: Resultate der Verfahren auf Wrap-Around-Butterfly-Netzwerken

		Resultate			
Graph	b	z-facher Greedy	n-fache BFS	BFS + Greedy	Eigenwerte
B_4^w	4	**4**	6	**4**	6
B_8^w	8	12	14	**8**	12
B_{16}^w	16	29	48	24	28
B_{32}^w	32	74	94	**50**	52
B_{64}^w	64	166	220	102	**88**
B_{128}^w	128	397	482	**208**	224
B_{256}^w	256	912	1060	482	**432**
B_{512}^w	512	2046	2354	1328	**964**

fällt auf, dass alle Verfahren hier schlechtere Ergebnisse liefern als auf normalen Butterfly-Netzwerken. Dies fällt vor allem bei der Breitensuche auf, welche nun katastrophale Bisektionen findet. Dies führt höchstwahrscheinlich auch zu der schlechteren Performance bei dem Verfahren mit anschließendem Greedy. Ein Grund dafür könnte sein, dass die Breitensuche jetzt nicht mehr vorteilhaft von einem der beiden Enden des Graphen gestartet werden kann, da diese Knoten zusammengeführt wurden. Die Reihenfolge der betrachteten Algorithmen bleibt jedoch unverändert, da die Spektralanalyse immer noch am besten abschneidet. Jedoch sind auch die Ergebnisse dieses Verfahrens deutlich schlechter. Dies ist interessant, da die Wrap-Around-Butterfly-Graphen reguläre Graphen sind und bisher angenommen wurde, dass dies die Performance des Eigenwertansatzes verbessert. Diese Graphklasse ist ein gutes Gegenbeispiel zu dieser Vermutung.

7.8 Cockroach-Graph

Der Cockroach-Graph C_k ist ein von Guattery und Miller in [14] vorgestellter Graph mit $n = 2k$ Knoten, wobei k gerade ist. Dieser besteht aus zwei Reihen von Knoten der Länge k. In der

zweiten Hälfte der Reihen werden Querverbindungen zwischen den Knoten eingefügt. Der Graph C_{10} ist in Abbildung 27 mit 20 Knoten dargestellt. Diese Graphklasse wurde von Guattery und Miller als *bad input* Graph für einen anderen Spektralansatz eingeführt und sollte zeigen, dass solche Verfahren auch auf einfach aufgebauten Graphen schlechte Ergebnisse liefern können. Es ist interessant zu sehen, inwieweit die in dieser Arbeit diskutierte Eigenwert-Heuristik auf diesem Graphen abschneidet.

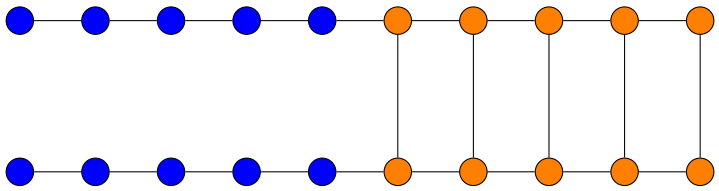

Abbildung 27: Der Cockroach-Graph C_{10} mit 20 Knoten

Farbig wurde die optimale Bisektion in die Abbildung 27 eingezeichnet. Diese hat immer die Weite 2 und liegt damit in $\mathcal{O}(1)$. Inwieweit diese Bisektion von den Verfahren gefunden wird, ist in Tabelle 13 dargestellt.

Tabelle 13: Resultate der Verfahren auf Cockroach-Graphen

		Resultate			
Graph	b	Greedy	n-fache BFS	BFS + Greedy	Eigenwerte
C_{100}	2	31	**2**	**2**	4
C_{200}	2	66	**2**	**2**	16
C_{300}	2	102	**2**	**2**	51
C_{400}	2	131	**2**	**2**	94
C_{500}	2	165	**2**	**2**	111
C_{600}	2	198	**2**	**2**	129

Eine n-fache Breitensuche reicht für diese Graphen aus, um die optimale Bisektion zu finden. Dies ist der Fall, wenn die Breitensuche vom Knoten rechts oben bzw. unten in der Abbildung 27 gestartet wird. Es sei angemerkt, dass je nach Reihenfolge der in der Breitensuche besuchten Knoten auch eine höhere Bisektionsweite möglich wäre. Diese wäre aber dennoch in $\mathcal{O}(1)$. Interessanterweise ist bei diesen Graphen sogar ein einfach ausgeführter Greedy, welcher im Allgemeinen sehr schlecht bei Graphen mit geringem Grad ist, besser als der Eigenwertansatz. Beide Verfahren liefern Bisektionsweiten in $\mathcal{O}(n)$. Dieses Resultat entspricht den Erwartungen, da dieser Graph explizit als *bad input* für Eigenwertansätze eingeführt wurde.

7.9 Verbundene Binärbäume

Im Folgenden betrachten wir zwei Binärbäume der Tiefe k. Die Wurzeln der Binärbäume werden durch eine Kante verbunden. Die Bisektionsweite ist damit also 1. Wir werden diese Graphklasse

im Folgenden mit V_k bezeichnen. Dabei besteht V_k aus $n = 2 \cdot (2^{k+1} - 1) = 2^{k+2} - 2$ Knoten und der maximale Knotengrad ist 3.

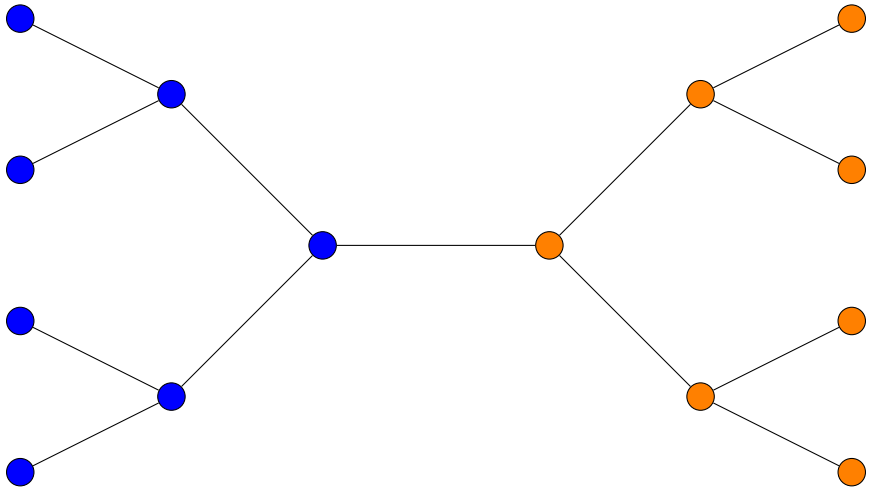

Abbildung 28: Zwei verbundene Binärbäume der Tiefe 2

Der Graph, bestehend aus zwei Binärbäumen mit Tiefe 2, ist in Abbildung 28 abgebildet. Die optimale Bisektion ist farbig hervorgehoben.

Tabelle 14: Resultate der Verfahren auf verbundenen Binärbäumen

		Resultate			
Graph	b	Greedy	n-fache BFS	BFS + Greedy	Eigenwerte
V_3	1	4	9	5	**1**
V_4	1	7	17	5	**1**
V_5	1	13	33	7	**1**
V_6	1	24	65	9	**1**
V_7	1	48	129	9	**1**
V_8	1	92	257	11	**1**

In Tabelle 14 sind die Resultate der Verfahren dargestellt. Diese Graphklasse ist interessant, weil sie ein *bad input* für eine Breitensuche von jedem Knoten ist. Dies liegt daran, dass es unmöglich für die Breitensuche ist, lediglich einen der beiden Bäume zu erkunden. Es muss die Wurzel besucht werden, bevor eine weitere Hälfte des Baumes besucht werden kann. Dann wird die Breitensuche jedoch auch den anderen Baum besuchen, es folgt eine schlechte Bisektion. Der anschließende Greedy schafft es jedoch die Bisektionsweite enorm zu verringern. Dies zeigt, dass der Greedy oftmals für kubische Graphen dann besonders effizient ist, wenn er von einer zusammenhängenden Komponente gestartet wird. Im Gegensatz dazu liefert ein Greedy, der von zufälligen Bisektionen gestartet wird, für Graphen mit kleinem Knotengrad, wie zum Beispiel in Abschnitt 6 gezeigt, schlechte Resultate. Auch hier ist dieser daher nicht in der Lage, einen Cut geringerer Größe zu finden. Das bedeutet,

dass bei der angegeben Graphklasse der Greedy und die Breitensuche einzeln scheitern, aber in der Kombination sehr effektiv sind. Dennoch ist die erhaltene Bisektion nicht optimal und sie scheint linear mit k anzusteigen. Der Eigenwertansatz hingegen findet die optimale Bisektion.

7.10 Comb-Graph

Der Comb-Graph $\mathcal{C}_k$, wobei k gerade ist, besteht aus einem Gitter von k^2 Knoten. Es sind alle Knoten in einer Reihe durch Kanten verbunden. Zusätzlich gibt es vertikale Verbindungen in der ersten Spalte des Gitters. Der Comb-Graph $\mathcal{C}_k$ mit $k = 4$ ist in Abbildung 29 zu sehen.

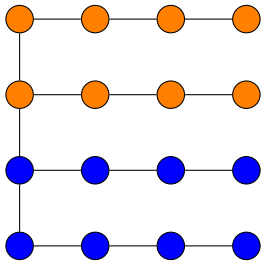

Abbildung 29: Comb-Graph mit 16 Knoten

Die optimale Bisektion ist eingezeichnet und hat eine Weite von 1. Diese ist unabhängig von der Größe des Graphen und liegt somit in $\mathcal{O}(1)$.

Tabelle 15: Resultate der Verfahren auf Comb-Graphen

		Resultate			
Graph	b	Greedy	n-fache BFS	BFS + Greedy	Eigenwerte
$\mathcal{C}_{10}$	1	17	9	7	**1**
$\mathcal{C}_{20}$	1	68	19	17	**1**
$\mathcal{C}_{30}$	1	158	29	27	**1**
$\mathcal{C}_{40}$	1	278	39	37	**1**

Wie in den Daten in Tabelle 15 zu sehen, findet das Verfahren aus Greedy und Breitensuche eine Bisektion von $\Theta(\sqrt{n}) = \Theta(k)$. Dies liegt daran, dass zunächst die Breitensuche keine bessere Bisektion findet. Diese erkundet das gesamte Rückgrat, besucht jedoch nur einen Teil der Knoten in den davon ausgehenden Reihen, bevor die Breitensuche nach dem Finden von $n/2$ Knoten terminiert. Im Anschluss kann der Greedy diese Bisektion nicht verbessern. Das liegt daran, dass dieser bei 2-regulären, verbundenen Teilstücken wirkungslos ist und hier keine Knoten vertauschen kann. Die langen Reihen nehmen hier also die Möglichkeit einer Verbesserung. Diese Graphklasse wurde in diese Arbeit aufgenommen, weil sie einen *bad input* für die Kombination aus Greedy und Breitensuche liefert. Obwohl die optimale Bisektionsweite konstant 1 ist, findet dieser Ansatz eine Bisektion in $\Theta(k)$. Die Spektralanalyse hingegen findet hier die optimale Bisektion. Es muss jedoch angemerkt werden, dass die Einträge des Eigenvektors in den langen Ketten sehr nah am Median

Graphklasse	Greedy	BFS	BFS + Greedy	Eigenwerte
Leiter	−−	+	++	+
2D-Gitter	−−	−	−	−
2D-Torus	−−	−	−	+
Rook-Graph	++	+	+	−
King-Graph	−−	+	++	+
Hypercube	++	−−	−	−
Butterfly	−−	−	+	++
WA-Butterfly	−−	−−	−	−
Cockroach	−−	++	++	−−
Binärbäume	−	−−	+	++
Comb-Graph	−−	−	−	++

Tabelle 16: Überblick über die durchgeführten Tests. Die Verfahren werden mithilfe der Skala $++ / + / - / --$ bewertet. Dabei wird die Bewertung ++ nur vergeben, wenn immer die optimale Bisektion gefunden wird.

liegen. Zum Beispiel liegt für den Graphen $\mathcal{C}_{40}$ mit 1600 Knoten ein Eintrag im Eigenvektor nur $3 \cdot 10^{-14}$ vom Median entfernt. Für sehr große Graphen könnten hier numerische Probleme auftreten.

7.11 Zusammenfassung

In diesem Kapitel wurden die Verfahren auf 10 verschiedenen Graphklassen getestet. Mithilfe dieser Tests können diese Algorithmen nun bewertet und genau miteinander verglichen werden.

In Tabelle 16 sind die Ergebnisse aller in diesem Kapitel durchgeführten Tests zu sehen. Dabei wurden die Verfahren jeweils auf der Skala $++ / + / - / --$ eingestuft. Die Bewertung ++ wurde dann zugewiesen, wenn der Algorithmus immer die optimale Bisektion findet. Ein Verfahren wurde mit einem + bewertet, wenn eine Bisektion nahe der optimalen gefunden wurde. Die Bewertungen − erhält ein Algorithmus, wenn die gelieferten Bisektionen deutlich schlechter als die optimale Bisektion sind. Ein −− wird vergeben, wenn von der praktischen Nutzung dieses Algorithmus auf dieser Graphklasse abzuraten ist, weil die Ergebnisse kaum von zufälligen Bisektionen zu unterscheiden sind.

Betrachten wir die Ergebnisse der einzelnen Verfahren. Das Greedy-Verfahren liefert sehr unterschiedliche Ergebnisse. In den angegebenen Graphklassen gibt es nur zwei Möglichkeiten: Entweder das Greedy-Verfahren ist optimal oder es liefert katastrophale Ergebnisse. Interessant ist dabei, dass das Greedy-Verfahren, wenn es optimale Bisektionen findet, auch das einzige der hier betrachteten Verfahren ist, welches dazu in der Lage ist. Es kann also nicht davon die Rede sein, dass diese Probleminstanzen generell sehr einfach sind. Die Graphklassen, bei denen der Greedy-Ansatz optimal ist, sind Rook-Graphen und Hypercubes. Die Gemeinsamkeit dieser Graphklassen ist die große Kantenzahl. In einer praktischen Anwendung ist das Greedy-Verfahren kein Alleskönner und sollte nur für spezielle Graphklassen mit einer hohen Kantenzahl verwendet werden.

Die betrachtete Breitensuche hat bei den hier verglichenen Algorithmen den Nachteil, dass

sie nie besser als die Kombination aus Breitensuche und Greedy sein kann. Bei den Ergebnissen dieses Verfahrens sollte jedoch die Laufzeit des Verfahrens nicht außer Acht gelassen werden. Diese ist deutlich besser als die der anderen Algorithmen und liegt in $\mathcal{O}(n + m)$ bei einer einfachen Ausführung und in $\mathcal{O}(n \cdot (n + m))$ bei einer n-fachen Ausführung. Auf dem Leitergraph sowie verschiedenen Gittergraphen ist dieses Verfahren nicht deutlich schlechter als der Eigenwertansatz sowie die Breitensuche mit anschließendem Greedy. Auf dem Cockroach-Graph liefert dieser Ansatz sogar optimale Ergebnisse. Steigt jedoch die Kantenzahl, so ist dieser Ansatz in der Praxis nicht besonders gut. Für praktische Probleme mit einer geringen Kantenzahl bzw. einer Gitterstruktur und sehr großen Graphen ist dieser Ansatz durchaus geeignet.

Die Kombination aus Breitensuche und Greedy ist der Allrounder der angegebenen Algorithmen. Es wurde bereits diskutiert, dass der Greedy für große Kantenzahlen gute Ergebnisse liefert und die Breitensuche eher für geringe. Dies führt dazu, dass die Kombination dieser Verfahren selten katastrophale Resultate liefert, sondern bei sehr verschiedenen Graphklassen gut abschneidet. Daher sind wirkliche Schwächen nicht auszumachen. Dieses Verfahren kann in der Praxis genutzt werden, wenn die Probleminstanzen unbekannt oder sehr verschieden sind.

Der Eigenwertansatz schneidet auf vielen aus der Praxis bekannten Graphklassen, wie dem 2D-Torus oder dem Butterfly-Netzwerk, gut ab. Bei letzterer Graphklasse sind die Ergebnisse sogar optimal. Dennoch lassen sich einfach aufgebaute Graphen konstruieren, wie der von Guattery und Miller in [14] vorgestellte Cockroach-Graph, auf denen dieses Verfahren extrem schlechte Ergebnisse liefert. Für eine praktische Nutzung sollte also ausgeschlossen werden, dass dieser Ansatz das Bisektionsproblem für solche Graphen lösen muss.

Es lässt sich zusammenfassen, dass es kein Verfahren gibt, welches klar besser als alle anderen ist und somit immer genutzt werden sollte. Vielmehr hängt die Performance der Algorithmen in hohem Maß von der betrachteten Graphklasse ab.

8 Fazit und Ausblick

In dieser Arbeit wurden drei verschiedene Themen behandelt.

1. Es wurden drei Ansätze vorgestellt, welche das Bisektionsproblem exakt lösen, und deren Laufzeiten verglichen.

2. Es wurde die Generierung von zufälligen regulären Graphen mit dem Verfahren von Steger und Wormald diskutiert, welches in abgewandelter Form auch die Generierung von weiteren Graphklassen ermöglicht.

3. Es wurden verschiedene Heuristiken auf ihre praktische Performance untersucht. Hier stand vor allem der Vergleich eines Eigenwertansatzes mit einem Greedy-Verfahren im Vordergrund.

Alle bekannten Algorithmen, welche das Bisektionsproblem exakt lösen, besitzen eine exponentielle Laufzeit. Daher werden diese Verfahren in der Praxis nur sehr selten genutzt. Dennoch hat sich gezeigt, dass eine geschickt gewählte Formulierung als lineares Optimierungsproblem eine minimale Bisektion regulärer Graphen von bis zu 100 Knoten in angemessener Zeit finden kann. Dies war eine deutliche Verbesserung im Vergleich zu naiven Brute-Force-Implementierungen.

Da solche Graphgrößen jedoch für viele Anwendungen deutlich zu gering sind, werden in der Praxis Heuristiken verwendet. Diese auf ihre Güte zu untersuchen, war folglich das zentrale Ziel der Arbeit. Dafür war eine Generierung von Testgraphen notwendig. Für die Generierung von zufälligen regulären Graphen wurde das Verfahren von Steger und Wormald genutzt. Eine Implementierung dieses Algorithmus erfüllte die Erwartungen und es konnten auch große Graphen mit einem hohen Knotengrad generiert werden. Dieser Algorithmus war die Grundlage für eine Reihe von Tests auf zufälligen Graphen. Mithilfe von kleineren Änderungen konnten mit dem Verfahren auch bipartite Graphen, kubische Kreise und Graphen mit geringer Bisektionsweite generiert werden. Auch diese Generierungen waren in angemessener Zeit möglich.

Die Heuristiken, welche in der Praxis genutzt werden, gehören oftmals zu der Klasse der Greedy-Algorithmen oder der Spektralanalysen [5]. Daher wurde ein praktisch relevanter Algorithmus aus beiden Klassen betrachtet und implementiert. Für den Greedy-Ansatz wurde ebenfalls die Wahl der Startbisektion diskutiert.

Die Tests auf zufälligen Graphen waren jedoch nicht ausreichend. Diese Graphen konnten nur erste Einschätzungen über die Güte der Verfahren liefern, aber nicht die Stärken und Schwächen dieser herausarbeiten. Das Greedy-Verfahren mit einer Startbisektion, welche durch eine Breitensuche generiert wurde, sowie der Eigenwertansatz waren hier ungefähr gleich gut. Darüber hinaus konnte beobachtet werden, dass der Greedy mit zunehmender Kantenzahl besser, die Breitensuche hingegen schlechter wird.

Daher wurde eine Vielzahl von weiteren Graphen betrachtet. In diesen ausführlichen Tests zeigte sich, dass die Verfahren je nach betrachteter Graphklasse sehr unterschiedliche Ergebnisse liefern. Es gab kein Verfahren, das allgemein besser als die anderen abgeschnitten hat. Das bedeutet, dass die spezifische Wahl eines Algorithmus für eine praktische Anwendung von entscheidender Bedeutung ist. Es ist nicht zu empfehlen ein Standardverfahren für alle möglichen Anwendungen zu nutzen.

Falls jedoch nur ein Verfahren gewählt werden kann, so hat sich in den Tests dieser Arbeit gezeigt, dass die Kombination aus Breitensuche und Greedy am ehesten ein Allrounder ist. Dieser Ansatz hat wenige Schwächen und schneidet aus praktischer Sicht selten deutlich schlechter als die anderen Verfahren ab. Eine Anwendung des Greedy-Verfahrens ist nur für eine sehr spezifische Anwendung bei Graphen mit großer Kantenzahl empfehlenswert.

Der Spektralansatz liefert im Allgemeinen auch gute Ergebnisse. Es gibt jedoch einige einfache Graphen auf denen dieses Verfahren sehr schlecht abschneidet. Daher sollte dieses Verfahren genutzt werden, wenn die Probleminstanzen bekannt sind. Für einige Graphklassen wie die Butterfly-Netzwerke liefert der Eigenwertansatz sogar optimale Ergebnisse.

Eine Breitensuche hat gegenüber den anderen Verfahren Vorteile in der Laufzeit. Dieses Verfahren schneidet allerdings im Allgemeinen eher schlecht ab. Die Resultate für Graphen mit geringer Kantenzahl sind jedoch teilweise gut. Für einige Graphklassen ist dieser einfache Ansatz überraschenderweise sogar besser als die Spektralanalyse.

Durch diese Erkenntnisse kann in der Praxis folglich eine sehr präzise Wahl eines geeigneten Algorithmus getroffen werden.

In dieser Arbeit wurden auch einige Fragen aufgeworfen, welche unbeantwortet bleiben und weiterer Forschung bedürfen.

Das Generierungsverfahren von Steger und Wormald wurde für bipartite Graphen, Graphen mit geringer Bisektionsweite sowie Graphen mit einer Kreisstruktur angepasst. Eine praktische Generierung war dadurch gut möglich und das Verfahren konnte für die betrachteten Tests genutzt werden. Weil diese nicht direkt mit dem Ziel dieser Arbeit zusammenhängen, wurden jedoch einige theoretische Fragen zu diesem Ansatz der Generierung außer Acht gelassen. Es bleibt eine offene Frage, ob die generierten Graphen gleichverteilt sind und ob die Generierung für $n \to \infty$ mit hoher Wahrscheinlichkeit erfolgreich ist. Aus praktischer Sicht kann zumindest letztere Frage positiv beantwortet werden, da alle Generierungen in angemessener Zeit erfolgten.

Für die Graphklasse der kubischen Kreise konnte in dieser Arbeit nicht beantwortet werden, ob das Bisektionsproblem für diese NP-schwer ist, während dies sowohl für reguläre als auch bipartite Graphen bekannt ist. Die Beantwortung dieser Frage würde auch bei der Bewertung der Performance der Heuristiken auf dieser Graphklasse helfen. Diese war nicht wesentlich besser als bei den weiteren betrachteten zufälligen Graphen.

In dieser Arbeit wurde ein Ansatz aus der Klasse der Greedy-Verfahren mit verschiedenen Startbisektionen sowie ein Ansatz der Spektralansätze implementiert und praktisch getestet. In [11] werden verschiedene Greedy-Verfahren auf wenigen Graphen miteinander verglichen. Das dortige Ergebnis ist, dass die Güte der Greedy-Verfahren in hohem Maße von der Startbisektion abhängt. Dies konnte in dieser Arbeit bestätigt werden. Der Vergleich von verschiedenen Greedy-Verfahren führt jedoch zu sehr ähnlichen Ergebnissen. Ein ausführlicher, praktischer Vergleich von verschiedenen Spektralansätzen wäre hochinteressant und eine sinnvolle Fortführung dieser Arbeit.

Literatur

[1] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[2] T.N. Bui, S. Chaudhuri, F.T. Leighton, and M. Sipser. Graph bisection algorithms with good average case behavior. *Combinatorica*, 7(2):171–191, 1987.

[3] M. Onsjö and O. Watanabe. Bisection problem on bipartite graphs. Joint Workshop "New Horizons in Computing" and "Statistical Mechanical Approach to Probabilistic Information Processing", July 2005.

[4] U. Feige and R. Krauthgamer. A polylogarithmic approximation of the minimum bisection. *SIAM J. Comput.*, 31(4):1090–1118, 2002.

[5] A. Buluç, H. Meyerhenke, S. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. In *Algorithm Engineering: Selected Results and Surveys, LNCS 9220*. Springer-Verlag, 2015.

[6] K. Shahookar and P. Mazumder. VLSI cell placement techniques. *ACM Comput. Surv.*, 23(2):143–220, June 1991.

[7] I.L. Markov, J. Hu, and M.C. Kim. Progress and challenges in VLSI placement research. *Proceedings of the IEEE*, 103(11):1985–2003, Nov 2015.

[8] A. Steger and N. C. Wormald. Generating random regular graphs quickly. *Comb. Probab. Comput.*, 8(4):377–396, July 1999.

[9] B.W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell Systems Technical Journal*, 49(2), 1970.

[10] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181, 1982.

[11] G. Jäger. An efficient algorithm for graph bisections of triangularizations. *Applied Mathematical Sciences*, 25(1):1203–1215, 2007.

[12] W. E. Donath and A. J. Hoffman. Lower bounds for the partitioning of graphs. *IBM J. Res. Dev.*, 17(5):420–425, 1973.

[13] R.B. Boppana. Eigenvalues and graph bisection: An average-case analysis (extended abstract). In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, pages 280–285, 1987.

[14] S. Guattery and G.M. Miller. On the quality of spectral separators. *SIAM Journal on Matrix Analysis and Applications*, 19(3):701–719, July 1998.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[16] Python documentation, `itertools.combinations`. `https://docs.python.org/2/library/itertools.html#itertools.combinations`. Zugriff: 13. September 2016.

[17] Lp file format. `http://lpsolve.sourceforge.net/5.5/`. Zugriff: 13. September 2016.

[18] Python documentation, `random.shuffle`. `https://docs.python.org/2/library/random.html#random.shuffle`. Zugriff: 13. September 2016.

[19] T. Bui, C. Heigham, C. Jones, and T. Leighton. Improving the performance of the kernighan-lin and simulated annealing graph bisection algorithms. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, DAC '89, pages 775–778. ACM, 1989.

[20] A. E. Brouwer and W.H. Haemers. *Spectra of Graphs*. Springer-Verlag New York, 2012.

[21] G.M.L. Gladwell. On isospectral spring-mass systems. *Inverse Problems*, 11(3):591, 1995.

[22] B.N. Parlett. *The Symmetric Eigenvalue Problem*. Prentice-Hall Series in Computational Mathematics. Pearson Education Canada, 1980.

[23] Matlab documentation, `eig`. `http://de.mathworks.com/help/matlab/ref/eig.html`. Zugriff: 13. September 2016.

[24] Matlab documentation, call matlab functions from python. `http://de.mathworks.com/help/matlab/matlab_external/call-matlab-functions-from-python.html`. Zugriff: 13. September 2016.

[25] numpy documentation, `numpy.linalg.eig`. `http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.eig.html`. Zugriff: 13. September 2016.

[26] L. Stacho and Vrt'o I. Bisection width of transposition graphs. *Discrete Applied Mathematics*, 84:221–235, 1998.

[27] C. Bornstein, A Litman, B.M. Maggs, R.K. Sitaraman, and T. Yatzkar. On the bisection width and expansion of butterfly networks. *Theory of Computing Systems*, 34:491–518, 2001.