UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Diplomarbeit

von

Christian Bachmaier

Vergleich und Implementierung von Partitionierungsverfahren auf Graphen

---

im März 2000
vorgelegt bei

Prof. Dr. F. J. Brandenburg

Lehrstuhl für theoretische Informatik

Zusammenfassung

In dieser Arbeit wird einleitend eine Auswahl an derzeit existierenden Partitonierungsverfahren für Graphen kurz vorgestellt, um dem Leser einen kleinen Überblick in dieser Thematik zu schaffen. Danach werden die beiden heuristischen Partitionierungsalgorithmen von Fiduccia und Mattheyses und *Ratio Cut* nach Wei und Cheng detailiert beschrieben. Die Laufzeit beider Heuristiken ist linear in der Anzahl der Kanten des Graphen.

Zusätzlich sind dabei beide Algorithmen in der Programmiersprache C++ im Rahmen von GTL vollständig implementiert worden. In diesem Zusammenhang werden Verbesserungen und notwendige Abweichungen in dieser Implementierung von der jeweiligen Originalbeschreibung aufgezeigt und erläutert.

Anschließend wird noch kurz angedeutet, wie diese Verfahren in Hinblick auf MULTIWAY PARTITION erweitert werden können. Mit anderen Worten, hier wird beschrieben, wie man mit diesen Zweiteilungsalgorithmen einen Graphen in mehrere Teile zerlegen kann.

Im Anhang wird schließlich die reale Laufzeit und die Qualität der gelieferten Ergebnisse beider Verfahren in verschiedenen Testsituationen ermittelt und durch Gegenüberstellung miteinander verglichen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

Der zunehmende Einsatz der Rechnertechnik in der Gesellschaft bringt immer mehr und komplexere Anforderungen an Mathematik, Elektrotechnik und Informatik mit sich. Immer schnellere und leistungsfähigere Systeme sollen Menschen Aufgaben abnehmen oder erst eine sinnvolle Bearbeitung großer Probleme, wie z. B. die genaue Wettervorhersage über mehrere Tage und Wochen, ermöglichen.

Die gleiche Tendenz setzt sich heutzutage auch mit der global zunehmenden Vernetzung fort. Hier sind nicht nur Computer und das Internet gemeint, sondern z. B. auch die normalen Telefon- oder Stromnetze.

Die große Komplexität eines solchen Systems kann ein einzelner Mensch bei weitem nicht mehr überblicken. Millionen von Komponenten müssen miteinander korrekt verbunden werden, um derartigen Anforderungen gerecht zu werden und um solche Systeme bauen zu können. Dies geht derzeit sogar schon so weit, daß auch leistungsfähige Computer und deren Softwaretools damit überfordert sind. Es ist daher notwendig, das zu realisierende System zuerst in kleine, überschaubare Subsysteme aufzuteilen, die später wieder miteinander verknüpft werden, um sie nach außen hin als eine Einheit erscheinen zu lassen.

Selbstverständlich will man Subsysteme einer passenden Größe haben und z. B. aus Kostengründen oder physikalischen Limitierungen die Verbindungen zwischen den Subsystemen minimieren. Zum besseren Verständnis stelle man sich das Telefonnetz vor. Innerhalb eines Subnetzes zahlt man günstige Ortsgebühr, außerhalb eine etwas höhere Ferngebühr. Die Ortsnetzbereiche stellen also Subsysteme dar, innerhalb denen i. a. mehr Telefongespräche geführt werden als zwischen den Orten.

Doch auch schon die Aufteilung eines großen Gesamtsystems in geeignete Untersysteme nach solchen Gesichtspunkten bereitet Schwierigkeiten. Um in

vertretbarer Zeit eine gute Lösung zu berechnen, bedient sich die theoretische Informatik der diskreten Mathematik und verwendet dazu Heuristiken. Leider kann man so, im Gegensatz zu Approximationen, weder absolute Voraussagen über die Qualität des Ergebnisses treffen noch eine Garantie über die maximale prozentuale Abweichung vom optimalen Ergebnis geben.

Natürlich spielen solche Zerlegungsansätze auch massiv in den algorithmischen Bereich hinein. Bestimmte Probleme können erst durch Aufteilung in kleinere Unterprobleme effizient berechnet werden. Als Stichworte hierzu seien z. B. „Divide & Conquer“ oder „Parallel Computing“ genannt.

In den nachfolgenden Kapiteln werden die „gängigsten“ solcher sog. Partitionierungsverfahren vorgestellt, wovon zwei Bi-Partitonierungsverfahren anschließend im Detail erläutert werden. Dabei geht es sowohl um den Entwurf effizienter Datenstrukturen als auch um Analyse ihres Verhaltens. Die erzielten Fortschritte und die Praxisrelevanz der behandelten Probleme lassen erwarten, daß derartige Algorithmen und deren dazugehörenden Datenstrukturen noch lange Zeit Gegenstand intensiver Forschung bleiben werden.

Übrigens arbeiten alle anschließend behandelten Heuristiken nicht direkt mit Schaltplänen, Algorithmusspezifikationen oder ähnlichem, sondern auf etwas abstrakteren Datenstrukturen, nämlich auf Graphen bzw. Hypergraphen.

Kapitel 2

Überblick

Dieses Kapitel gibt einführende Definitionen und spiegelt den Gegenstand dieser Arbeit wider. Nach anfänglicher Klärung einiger Begriffe und Verdeutlichung der Problemstellung werden aktuelle Lösungsansätze vorgestellt. Exemplarisch werden anschließend typische Anwendungsgebiete der Graphpartitionierung in der Praxis aufgezeigt.

2.1 Begriffsklärung und Notation

Begriffe aus der Graphentheorie, welche evtl. nachfolgend nicht oder im Rahmen dieser Arbeit zu kompakt erklärt werden, können in [ClaHol94] nachgelesen werden.

2.1.1 Definition eines Graphen

Ein Graph G = (V, E) besteht aus zwei endlichen Mengen:

V, der nicht leeren Knotenmenge des Graphen.

$E \subseteq V \times V$, der möglicherweise auch leeren Kantenmenge des Graphen. Jede Kante $e \in E$ wird durch ein Paar von Knoten (u, v), die als Endknoten bezeichnet werden, dargestellt.

Werden die Endpunkte einer Kante als Start- und Zielknoten betrachtet, d. h. die Reihenfolge spielt eine Rolle, so wird diese als gerichtet bezeichnet. Falls nur die

Verbindung zweier Knoten u und v im Vordergrund steht, wird die Kante als ungerichtet bezeichnet. Die Kantenmenge E ist dann als $\{\{u, v\} \mid u, v \in V\}$ definiert. Der Einfachheit halber schreibt man für eine Kante aber dennoch (u, v) und nicht {u, v}.

Ein Graph G = (V, E) wird als gerichtet angesehen, falls er nur gerichtete Kanten enthält, als ungerichtet, falls er nur ungerichtete Kanten enthält. Eine Mischform davon wird i. a. nicht betrachtet.

Zwei Knoten werden als adjazent bezeichnet, falls es eine Kante zwischen ihnen gibt. Eine Kante heißt inzident mit ihren beiden Endknoten.

Der Grad deg(v) eines Knotens $v \in V$ ist im ungerichteten Fall die Anzahl zugehöriger Adjazenzkanten. Bei gerichteten Graphen ist deg(v) die Anzahl auslaufender Kanten.

2.1.2 Definition eines Hypergraphen

Ein Hypergraph HG = (V, H) ist ein verallgemeinerter Graph, bei dem die Beschränkung der Endknotenanzahl einer Kante auf zwei aufgehoben wird. D. h. eine Hyperkante $h \in H$ ist im ungeordneten Fall eine Menge von Knoten $\{v_1, ..,v_n\} \subseteq V$. Sie kann also nun mehr als zwei Knoten miteinander verbinden. Die Anzahl dieser Knoten bezeichnet den Grad dieser Kante, deg(h) = |h| = n, wobei deg: $E \rightarrow \mathbf{N}$ eine Abbildung ist.

Eine Kante eines geordneten Hypergraphen hingegen ist ein Vektor $(v_1, ..,v_n) \in V^n$, die Reihenfolge der Knoten spielt hier also eine Rolle.

2.1.3 Seperatoren

Die drei Definitionen in diesem Abschnitt entsprechen sinngemäß denen aus [Bran99].

Ein Knotenseperator ist eine Menge $VS \subset V$, die einen Graphen G = (V, E) beim Löschen der Knoten VS aus V, samt deren adjazenten Kanten aus E, in zwei disjunkte Teilgraphen G' = (V', E') und G'' = (V'', E'') zerlegt. Es gilt also $V = V' \cup VS \cup V''$, $V' \cap VS = V' \cap V'' = VS \cap V'' = \varnothing$ und $(u, v) \in E \Rightarrow \neg(u \in V' \wedge v \in V'')$.

Ein Kantenseperator ist eine Menge ES $\subset$ E, die einen Graphen G = (V, E) beim Löschen der Kanten ES aus E in zwei disjunkte Teilgraphen G' = (V', E') und G'' = (V'', E'') zerfallen läßt. Es gilt also $u \in V' \wedge v \in V'' \Rightarrow (u, v) \in ES$.

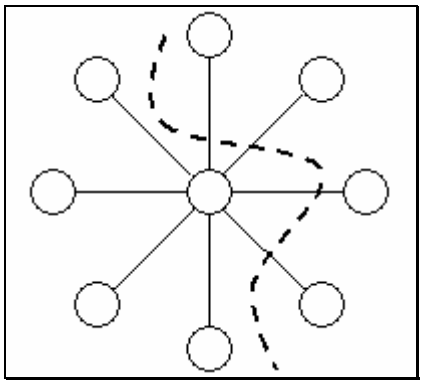

Abbildung 2-1: Beispiel Stern

Bei obigem Graph, einem sog. Stern, gilt |VS| = 1 (Knoten in der Mitte) und |ES| $\approx$ ½ |V|.

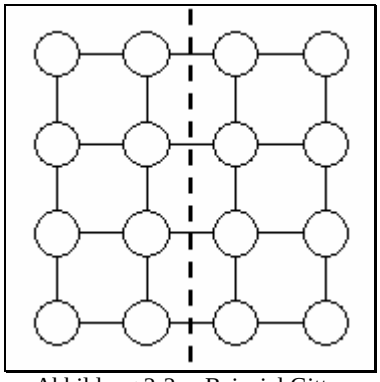

Abbildung 2-2: Beispiel Gitter

Bei Seperatoren, die einen annähernd quadratischen „Gittergraphen" in zwei gleich große Teile zerlegen gilt: $|VS| \approx \sqrt{|V|}$, $|ES| \approx \sqrt{|V|}$.

Ein Bifurkator ist ein spezieller Kantenseperator, bei dem die Eigenschaft $|V'| \leq |V''| \leq |V'| + 1$ gilt.

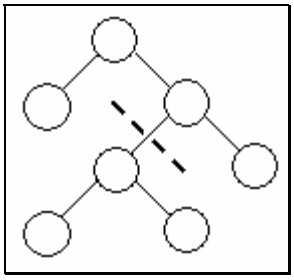

Abbildung 2-3: Beispiel Binärbaum

Durch geschicktes Vorgehen kann bei einem Binärbaum immer in Linearzeit O(|V| + |E|) ein Kanten- oder Knotenseperator gefunden werden, bei dem die Teilbäume immer mindestens $\frac{1}{3}$ und höchstens $\frac{2}{3}$ von |V| viele Knoten aufweisen. Um einen solchen Seperator effizient zu finden, startet man bei der Wurzel und prüft den kleineren Unterbaum, ob er bereits $\frac{1}{3}$ * |V| viele Knoten hat. Falls ja schneidet man ihn ab, falls nein steigt man in den größeren Unterbaum eine Stufe ab und vergleicht erneut. Natürlich muß man in einem vorherigen Schritt, etwa durch Tiefensuche, bei jedem Knoten die Größen der jeweiligen Unterbäume abspeichern. Es gilt dann |VS| = 1 und |ES| = 1.

Bei Bäumen mit maximalem Knotengrad k kann auf ähnliche Art eine Aufteilung in den beweisbaren Grenzen $\frac{1}{k+1}$ und $\frac{k}{k+1}$ gefunden werden.

In den folgenden Kapiteln wird nur noch die Rede von Kantenseperatoren sein.

2.2 Problemstellung

Unter Zuhilfenahme der soeben erfolgten Begriffserklärungen kann nun das sog. MULTIWAY PARTITION Problem wie folgt definiert werden:

Sei HG = (V, H) ein Hypergraph. Sei `node_weight`: $V \rightarrow \mathbf{N}$ eine Funktion, die jedem Knoten ein ganzzahliges Gewicht zuordnet und `edge_weight`: $H \rightarrow \mathbf{N}$ dementsprechend jeder Kante.

V wird in eine k-Wege Partitionierung $\Pi = \{V_1, ..., V_k\}$ so zerteilt, daß für $1 \leq i \leq k$ gilt $V_i \subset V$, wobei V_i paarweise disjunkt sind und $\bigcup_{i=1}^{k} V_i = V$.

Für alle Partitionen V_i muß $b(i) \leq$ `node_weight[`V_i`]` $\leq B(i)$ gelten, wobei b(i) eine minimale und B(i) eine maximale Größe der jeweiligen Partition darstellt. Dabei gilt `node_weight[`V_i`]` := $\sum_{v_j \in V_i}$ `node_weight[`v_j`]`.

Ziel ist es nun,

$$\texttt{cutsize} := \texttt{edge\_weight}[\Pi] := \tfrac{1}{2} \sum_{i=1}^{k} \sum_{h \in H_{ext,\,i}} \texttt{edge\_weight[h]}$$

auch Schnittgröße oder einfach Schnitt genannt, zu minimieren. Dabei ist

$$H_{ext,\,i} := \{h \in H \mid h \cap V_i \neq \emptyset, h \setminus V_i \neq \emptyset\}$$

die Menge der sog. externen Kanten von V_i.

Den minimalen Schnitt eines beliebig großen Graphen mit Größenbeschränkung der einzelnen Teile und $k \geq 2$ zu berechnen, ist jedoch, wie in [GarJoh79] beschrieben, im allgemeinen Fall NP-vollständig. Dort wird auf eine Arbeit von Hyafil und Rivest aus dem Jahre 1973 verwiesen, in der das ebenfalls NP-vollständige Problem PARTITION INTO TRIANGLES[1] in einer typischen Art und Weise auf MULTIWAY PARTITION reduziert worden ist. Das Prinzip solcher Reduktionen ist in [HopUll94] ausführlich erklärt. Um in einigermaßen vernünftiger Zeit ein brauchbares Ergebnis zu erhalten, ist es also notwendig, eine gute und schnelle Heuristik anzuwenden.

Natürlich wird es nicht einfacher, wenn k von vornherein nicht bekannt ist (vgl. [Leng90]). Diese allgemeinere Variante nennt man das FREE PARTITION Problem. Dabei können die Größenbegrenzungen b(i) bzw. B(i) von Partition zu Partition nicht variieren. Gewöhnlich wird hier nur eine einheitliche obere Grenze $B \in \mathbf{N}$ vorgeschrieben. Handelt es sich bei dem zu zerlegenden Graphen G allerdings um einen Baum mit einheitlichen Knoten- und Kantengewichten, dann ist, ebenfalls laut [Leng90], FREE PARTITION in polynomialer Zeit lösbar, liegt also dabei in der Komplexitätsklasse P.

Wenn man nur daran interessiert ist, dichte Teilgraphen voneinander zu trennen, reicht es aus `node_weight[v]` und `edge_weight[h]` für alle $v \in V$ und alle $h \in H$ jeweils auf 1 zu setzen, was jedoch laut [GarJoh79] auch nichts an der exponentiellen Laufzeit von MULTIWAY PARTITION bzw. FREE PARTITION ändern kann.

---

[1] Gegeben sei ein Graph G = (V, E) mit |V| = 3k, wobei $k \in \mathbf{N}$. Das Problem bei PARTITION INTO TRIANGLES lautet folgendermaßen (vgl. [GarJoh79]): Können die Knoten von G in k disjunkte, genau 3elementige Teilmengen V_1, ..., V_k aufgeteilt werden, so daß für jede Menge $V_i = \{u_i, v_i, w_i\}$, $1 \leq i \leq k$, alle der drei Kanten $\{u_i, v_i\}$, $\{u_i, w_i\}$ und $\{v_i, w_i\}$ in E enthalten sind?

2.3 Lösungsansätze

Da in den letzten 20 Jahren intensive Forschungen auf diesem Gebiet betrieben worden sind, gibt es heutzutage viele Möglichkeiten, an die oben genannten Problemstellungen algorithmisch heranzutreten. Eine einigermaßen vollständige Übersicht der etablierten Verfahren und Verweise auf eine Auswahl der mittlerweile unübersichtlich vielen Veröffentlichungen zu dem ausgedehnten Themengebiet Partitionierung werden in [AlpKah95] gegeben.

2.3.1 Iterative Verbesserung

Die erste große Gruppe, die hier genannt wird, sind die auf Knotentransfer basierenden Ansätze. Dazu gehören *Greedy*-Methoden, *iterativer Austausch* [KerLin70] oder [FidMat82] (vgl. Kapitel 3), *Tabu Search* [AlpKah95], *Simulated Annealing* [JAMcGS89] und Evolutionsalgorithmen, wie z. B. „Genetic Algorithm and Graph Partitioning“ in [BuiMoo96]. Sie dominieren sowohl die Literatur wie auch den praktischen Einsatz in der Industrie aus diversen Gründen. Erstens, weil sie sehr leicht verständlich sind. Die Technik, eine vorher gegebene Lösung durch kleine Schritte, Umlagerung einzelner Knoten in eine andere Partition, immer besser zu machen, ist sehr intuitiv. Ein zweiter Grund dafür mag vielleicht sein, daß man iterative Algorithmen verhältnismäßig leicht beschreiben und implementieren kann. Darum haben sich wahrscheinlich der Partitionierungsalgorithmus von Fiduccia und Mattheyses, wie er in Kapitel 3 beschrieben wird, und die Mehrwege-Partitionsmethode von Sanchis - siehe [Sanchis89] - zu Standardverfahren gegenüber fast allen anderen Heuristiken entwickelt.

Eine allgemeine Beschreibung der grundsätzlichen Vorgehensweisen bei *Simulated Annealing, Tabu Search* oder bei den *Evolutionsverfahren,* nicht speziell auf Graphpartitionierung festgelegt, findet sich u. a. in [Bran99].

2.3.1.1 Simulated Annealing

Simulated Annealing ist ein anderes erfolgreiches Beispiel der iterativen Verbesserungsmethoden. Der eigentliche Kerngedanke bei *Simulated Annealing* ist analog zur Energie in einem physikalischen System und jede Knotenbewegung entspricht Veränderungen des Energiestatus dieses Systems. Um zu entscheiden, ob eine Knotenbewegung akzeptabel ist, wird hier die sog. *Metropolis Monte Carlo Methode* verwendet. Bei einer gegebenen, gültigen Lösung wird ein

Knotentransfer in eine andere Partition durchgeführt. Ergibt sich daraus eine bessere Lösung, so wird sie angenommen. Falls nicht, wird sie nur mit einer Wahrscheinlichkeit von $e^{-\frac{\Delta}{T}}$ angenommen, ansonsten wird zur ursprünglichen Lösung zurückgekehrt. Dies soll helfen, aus lokalen Minima herauszufinden. Der Parameter Δ ist gleich den Kosten der neuen Lösung (z. B. Schnittgröße oder Balancierung) minus den Kosten der vorherigen Lösung. T ist der aktuelle, sog. Temperaturwert. Um die Konvergenzrate des Algorithmus und die Strategie für das Durchsuchen des Lösungsraums zu kontrollieren, wird die Temperatur T meistens mit zunehmender Anzahl an durchgeführten Knotenbewegungen verkleinert, man sagt in diesem Zusammenhang auch abgekühlt. *Simulated Annealing* produziert bei guter Parameterwahl gewöhnlich sehr gute Ergebnisse, die aber mit einer immens langen Programmlaufzeit bezahlt werden müssen. Man kann sogar beweisen, daß *Simulated Annealing* mit Erlaubnis unendlich vieler Knotenbewegungen - also mit einer unendlich langen Laufzeit - und einer Temperaturplanung, die das Verfahren genügend langsam abkühlt, in einem globalen Optimum konvergiert.

2.3.1.2 Tabu Search

Tabu Search ist den Methoden von Kernighan und Lin und Fiduccia und Mattheyses sehr ähnlich und kann als Alternative zu dem dort eingesetzten Lockingmechanismus gesehen werden. *Tabu Search* unterhält eine sog. Tabuliste, die jeweils die letzten r Knotentransfers beinhaltet (r ist dabei eine vorher festzusetzende Konstante). Diese Liste soll ein „Kreisen“ um ein lokales Minimum verhindern, denn Transfers, die in der Tabuliste stehen, dürfen normalerweise nicht ausgeführt werden. Gelegentlich darf trotzdem ein Tabutransfer vollzogen werden, wenn es die sog. Aspiration Function erlaubt. Das kann z. B. der Fall sein, wenn man mit einem Tabuknotentransfer eine bis jetzt noch nicht dagewesen kleine Schnittgröße erreichen würde. *Tabu Search* ist so gestaltet, daß es relativ schnell ein lokales Minimum findet, dann aus dem umgebenden „Tal“ herausklettert und zum nächsten lokalen Minimum wandert. Auf diese Weise kann es sein, daß *Tabu Search* den Lösungsraum effizienter durchsucht als z. B. *Simulated Annealing*.

2.3.1.3 Evolutionsverfahren

Genetic Algorithms sind von der Evolutionstheorie von Charles Darwin inspiriert. Dort produzieren „höher gestellte“ Mitglieder mehr Nachkommen in nachfolgenden Generationen als „niedere“ Mitglieder. Ein genetischer Algorithmus startet mit einer initialen Population von Lösungen, hier also mit

mehreren zulässigen Schnitten. Diese Population entwickelt sich über mehrere Generationen. Die Lösungen der aktuellen Generation werden dabei durch eine Menge aus Nachfolgelösungen der nächsten Generation ersetzt. *Genetic Algorithms* haben typischerweise einen sog. Crossover- und einen sog. Mutation-Operator. Der Crossover-Operator wählt aufgrund einer bestimmten Wahrscheinlichkeitsverteilung zwei Lösungen der aktuellen Population aus. Mit ihren teilweise vermischten Eigenschaften (Schnitte, Balancierungen) werden Nachkommen produziert. Der Mutation-Operator ermöglicht wenige Knotentransfers auf einer einzelnen Lösung. Gute Lösungen werden dabei gegenüber schlechten Lösungen als dominant betrachtet. Ein Ersetzungsschema entscheidet schließlich, welche Nachkommen welche Mitglieder der aktuellen Population ersetzen.

Die beste zwischendurch erhaltene Partitionierung wird als Ergebnis zurückgegeben.

Die kritischen Abschnitte sind bei *Genetic Algorithms* also die Crossover- bzw. Mutation-Operatoren, das Auswahl- und das Ersetzungsschema. Ihre Implementierung trägt entscheidend zum Erfolg oder Mißerfolg eines Evolutionsverfahrens für Graphpartitionierungen bei.

2.3.2 Kombinatorische Methoden

Kombinatorische Methoden transferieren das Paritionierungsproblem dagegen in einen anderen Bereich der Optimierung, z. B. auf *Netzwerkfluß*- oder auf mathematischer Programmierung basierende Ansätze. Das Problem (vgl. Abschnitt 4.1.2) bei Verfahren, die Netzwerkflußalgorithmen verwenden, ist, daß man nicht im voraus weiß, wo die Quelle s und die Senke t liegen sollen. Man kann zwar den Schnitt durch das Max-Flow-Min-Cut Theorem[2] wirklich klein halten, was aber dann zu größenmäßig recht unbalancierten Partitionen führt. Weitere Informationen dazu finden sich in [AlpKah95]. Ein anderes Beispiel für ein stochastisches Verfahren findet man in [HagKah91]. Dort wird anhand einer Eigenvektor-Dekomposition der Adjazenzmatrix[3] eines Graphen eine Lösung errechnet.

---

[2] Sei G ein Graph. Der Wert des maximalen Fluß von s nach t ist gleich den Kosten eines minimalen s-t Schnitts. Außerdem sind alle „durchgeschnittenen“ Kanten gesättigt (ausführliche Darstellung und Beweis siehe [Bran99] und [ClaHol94]).

[3] Eine |V| x |V| Matrix ist genau dann eine Adjazenzmatrix A(G), wenn ihre Einträge $a_{vw} = 1$, falls $(v, w) \in E$ und $a_{vw} = 0$ sonst. Falls G gewichtete Kanten besitzt, ist a_{vw} gleich dem Gewicht von (u, w).

2.3.3 Clustering

Schließlich gibt es noch *Clustering*-Algorithmen, wie [BaChFa], die jeweils mehrere Knoten in viele kleine Cluster zusammenfügen und auf diese Weise die Partitionierung sozusagen von unten nach oben aufbauen.

2.4 Anwendungen

Die in Abschnitt 2.2 vorgestellten Probleme sind bei weitem nicht nur von theoretischer Bedeutung, sondern auch aus ökonomischen Gründen in der Praxis interessant. Das dürfte auch die Hauptursache sein, warum auf diesem Gebiet sehr viel Forschungsarbeit bereits geleistet worden ist und auch heute noch wird.

Es gibt viele Gründe, warum Partitionierung in letzter Zeit zu einer kritischen Phase in der Optimierung von VLSI[4] Systemzusammensetzung und Layout avancierte. Der schwerwiegendste ist wahrscheinlich die immer mehr zunehmende Komplexität von Schaltungen. Systeme mit mehreren Millionen Transistoren sind an der Tagesordnung. Sie stellen eine Komplexität dar, die für existierende Designwerkzeuge nicht mehr ordentlich handhabbar sind. Partitionierung zerteilt das System in mehrere überschaubare Komponenten. Eine gute Partitionierung kann die Performance eines Schaltkreises merklich verbessern und dessen Layoutkosten senken.

In diesem Zusammenhang kann man das Knotengewicht `node_weight` als Blockgröße einer logischen Einheit, etwa eines Chips, sehen. Das Kantengewicht `edge_weight` kann Auskunft darüber geben, wieviele Informationskanäle sich eine Leitung zwischen Knoten teilen, wie breit ein Bus ist, d. h. wieviele Bits darüber parallel gesendet werden können, usw.

Auch zur algorithmischen Problemlösung sind Partitionierungsverfahren oft unabdingbar. Als Schlagwort sei hier „Divide & Conquer“ genannt. Dabei wird ein Problem rekursiv in kleinere Instanzen zerlegt, die sich dann leichter lösen lassen als insgesamt. Man ist jedoch auch hier bemüht, den Informationsfluß zwischen den kleineren Teilen in Grenzen zu halten. Denn um eine Antwort auf das große Problem zu erhalten, müssen die kleinen selbstverständlich untereinander „synchronisiert“ werden. In der Graphentheorie z. B. gibt es oft Algorithmen, die nur auf zusammenhängenden Graphen vernünftig laufen. Bei allgemeinen Graphen muß man daher zuerst alle Zusammenhangskomponenten

[4] Abkürzung für Very Large-Scale Integration

herausfinden, bevor man sich der eigentlichen Hauptaufgabe auf diesen Komponenten einzeln widmen kann.

Weitere Anwendungsgebiete der Partitionierung sind auch parallel arbeitende Programme. Dort ist es notwendig, eine große Anzahl an Berechnungen auf eine feste und meistens auch kleinere Zahl an Prozessoren zu verteilen. Jede Kante repräsentiert dabei eine Dateneinheit, die sich die Berechnungen, d. h. die zur Kante inzidenten Knoten, teilen.

Kapitel 3

Die Fiduccia Mattheyses Heuristik

Ein Spezialfall des MULTIWAY PARTITION Problem ist das sog. BIPARTITION Problem. Dabei wird der Graph in nur zwei Teile zerlegt, die man in diesem Zusammenhang auch Seiten nennt. Im folgenden werden sie mit Seite A und Seite B bezeichnet. Auch hier sollten sie einigermaßen gleich groß, also balanciert, und der Schnitt minimal sein. Zur Erinnerung, der Schnitt ist die Summe über alle Gewichte von Kanten, die „durchgeschnitten“ werden.

Als Instanz des MULTIWAY PARTITION Problems ist das BIPARTITION Problem, wie auch in [Leng90] oder [GarJoh79] beschrieben, auf allgemeinen Graphen NP-vollständig. Welcher Komplexitätsklasse jedoch BIPARTITION auf planaren Graphen angehört, ist bis heute noch offen (vgl. [Leng90]). Falls (auch auf allgemeinen Graphen) die Größenbeschränkungen der Seiten trivial sind, also wenn jede Seite ohne „Strafe“ alle Knoten aufnehmen darf, kann durch Anwendung von Netzwerkflußalgorithmen (Max-Flow = Min-Cut) in linearer Zeit eine Lösung gefunden werden. Offenbar macht die Anforderung balancierte Schnitte zu generieren BIPARTITION zu einem schwierigen Problem.

Es ist also in der Tat auch hier notwendig, eine gute Heuristik anzuwenden, um einen in der Praxis verwendbaren Algorithmus zu erhalten. Mit anderen Worten, hier wird ein „Trade-Off“ zwischen Laufzeitverhalten und Qualität der Lösung vorgenommen.

Die erste brauchbare Heuristik dazu war der Kernighan Lin Ansatz, der 1970 erstmals in [KerLin70] vorgestellt wurde. Bei dem Verfahren von Fiduccia und Mattheyses handelt es sich um einen im Jahre 1982 vorgestellten Lösungsansatz, der zwar auf dem Kernighan Lin Verfahren basiert, aber dennoch, wie in Absatz 3.1 diskutiert, zu bevorzugen ist. Wie bei Kernighan und Lin handelt es sich dabei um ein iteratives Verbesserungsverfahren. Der Ansatz von Fiduccia und Mattheyses ist ebenfalls eine Heuristik, so daß i. a. nur eine annähernd optimale Lösung gefunden werden kann.

Die in diesem Kapitel beschriebene Bi-Partitonierungs-Heuristik ist noch immer die in der Praxis meist benutzte Methode (vgl. Aussagen von [Leng90] oder [AlpKah95]), um Graphen bzw. Hypergraphen in zwei Teile zu zerlegen. Sie ist robust und errechnet schnell eine zufriedenstellende Lösung für nahezu alle Anwendungsbereiche.

3.1 Vorteile gegenüber der Kernighan Lin Heuristik

Kernighan und Lin lösen eigentlich das sog. BISECTION und nicht das allgemeinere BIPARTITION Problem. Dort müssen sich auf jeder Seite genau gleich viele Knoten mit einheitlichen Gewichten befinden und nur mit dieser Einschränkung wird der Schnitt minimiert bzw. zu minimieren versucht. Beim Fiduccia und Mattheyses Verfahren wird dagegen bei einem einzelnen Knotentransfer über die Schnittlinie nur ein einzelner Knoten bewegt und nicht zwei Knoten gegeneinander ausgetauscht. Man erhält zwar dadurch nicht immer exakt balancierte Seiten, kann aber aus diesem Grund auch uneinheitliche Knotengewichte berücksichtigen. Eine wirklich exakte Balancierung ist bei den meisten Anwendungen aber auch gar nicht erforderlich. Zusätzlich können hier die Kanten unterschiedliche Gewichte haben. Außerdem darf bei dieser Methode die Knotenanzahl |V| auch ungerade sein, ohne daß man zusätzliche Maßnahmen, wie z. B. das Einfügen von „Dummy-Knoten“, ergreifen müßte.

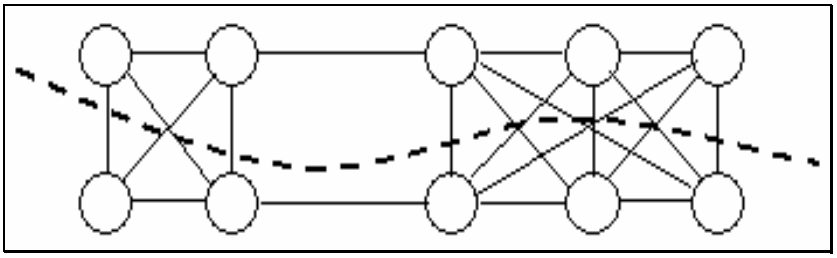

Abbildung 3-1: BISECTION produziert von Kernighan Lin

Wie man sieht, hat die Schnittgröße in Abbildung 3-1 einen Wert von 13, in Abbildung 3-2 nur 2.

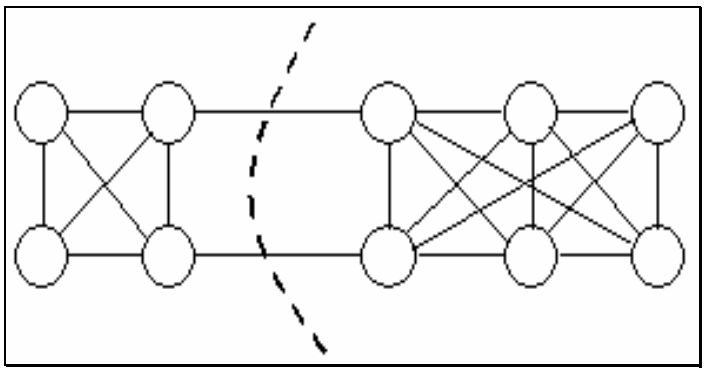

Abbildung 3-2: BIPARTITION produziert von Fiduccia Mattheyses

Die Fiduccia Mattheyses Heuristik kann, wie in [Leng90] beschrieben, auch auf Hypergraphen angewendet werden. Da die im Rahmen dieser Arbeit angefertigte Implementierung der Fiduccia Mattheyses Methode für GTL[5] erfolgt ist und diese Bibliothek in der aktuellen Version keinerlei Unterstützung für Hypergraphen bietet, beschränkt sich hier die Diskussion hauptsächlich auf normale Graphen. Diese Einschränkung hat jedoch nur kleine Auswirkungen auf Implementierungsdetails.

Eine spezielle in Abschnitt 3.2.3 vorgestellte Datenstruktur spart soviel Laufzeit ein, daß eine Komplexität von O(|E|), also linear in der Anzahl der Kanten, erreicht werden kann. Die Heuristik von Kernighan und Lin läuft dagegen in $O(|V|^3)$, weil sie im Grunde aus drei ineinandergeschachtelten Schleifen jeweils über alle Knoten besteht.

3.2 Grundlagen

Man startet mit einer gewichtsmäßig in etwa ausbalancierten und ansonsten willkürlichen Zweiteilung der Knotenmenge. Es gibt viele Methoden, eine solche initiale Aufteilung von V zu generieren. Einige davon werden in [HauBor97] vorgestellt. Hier wird vorgeschlagen, das in Abschnitt 3.2.2 beschriebene Verfahren zu verwenden.

[5] Die GRAPH TEMPLATE LIBRARY (GTL) ist eine C++ Bibliothek, die Elemente und Algorithmen für den Umgang mit Graphen bereitstellt. Da sie auf der STANDARD TEMPLATE LIBRARY (STL) basiert, ist sie mit dieser voll kompatibel. Sie ist sowohl unter MS-Windows als auch unter den meisten Unix-Varianten lauffähig. Entwickelt wurde GTL als Grundlage für GRAPHLET am Lehrstuhl von Prof. Dr. Brandenburg an der Universität Passau. Für weitere Informationen siehe [FoPiRa99].

Wie beim Kernighan Lin Algorithmus wird ein bestimmter Knoten ausgewählt, testweise auf die andere Seite bewegt und anschließend gesperrt. Das Sperren bedeutet, daß der Knoten in eine Liste `lockedA` bzw. `lockedB` geschrieben wird, um nicht noch einmal innerhalb eines Durchlaufes ausgewählt zu werden, da ansonsten Endlosschleifen entstehen würden.

Für alle nachfolgenden, testweisen Knotenbewegungen liegt dann dieser Knoten auf der anderen Seite. Dies kann natürlich die Schnittgröße nicht nur verkleinern, sondern temporär auch vergrößern. Durch dieses Vorgehen erhofft man sich laut [FidMat82], daß der Algorithmus aus einem lokalen Minimum herausklettern kann.

Um dies zu erreichen, wird dabei eine Art Protokoll geführt, welche Knotenbewegungen welche Schnittgröße verursacht haben. Wenn alle Knoten schließlich gesperrt sind, also genau einmal versuchsweise bewegt worden sind, wird genau die Sequenz an Knotenbewegungen, die erste bis einschließlich die mit minimaler Schnittgröße, wirklich realisiert. Ist die anfängliche Schnittgröße ohnehin nirgends unterschritten worden, wird natürlich auch kein Knoten transferiert.

Jetzt werden zusätzliche Durchläufe der Heuristik mit der jeweils vorher erhaltenen Partitionierung gestartet. Dies wird so lange fortgeführt, bis sich die Schnittgröße nicht mehr verbessern kann.

```
1:  initialisiere die Datenstruktur;
2:  while (Knotentransfers möglich)
3:  {
4:     wähle einen ungesperrten Knoten v;
5:     update Datenstruktur, um die versuchsweise Umlagerung von v
       zu manifestieren;
6:     sperre Knoten v;
7:     führe Protokoll über diesen versuchsweisen Transfer;
8:  }
9:  führe den ersten bis zum besten Knotentransfer, d. h. bei dem
    die kleinste Schnittgröße erreicht wurde, wirklich aus;
```

Abbildung 3-3: Grobgliederung der Fiduccia Mattheyses Heuristik

Die Abbildung 3-3 zeigt die Grobgliederung eines Durchlaufes der Fiduccia Mattheyses Heuristik in „Pseudo-Code“. Nachfolgend wird diskutiert, wie diese

Heuristik in Linearzeit implementiert werden kann. Zeile 1 wird in Abschnitt 3.2.3 genauer erläutert, die Zeilen 2 – 6 werden in 3.2.4 abgehandelt. Die Führung einer Protokoll-Tabelle in Zeile 7 und die letztendliche Bestimmung der Ergebnis-Partitionierung erfolgt im Prinzip wie bei Kernighan und Lin in [KerLin70]. Die genaue Vorgehensweise hierzu wird auch im Beispiel 3.2.6 klar.

3.2.1 Balancekriterium

Natürlich kann man nicht einfach alle Knoten auf eine Seite bewegen. Die Schnittgröße wäre zwar dann auf alle Fälle minimal, jedoch möchte man im Endeffekt aber auch einigermaßen ausbalancierte Partitionen erhalten. D. h. die Knoten sollten so verteilt sein, daß auf jeder Seite in etwa die gleiche Summe an Knotengewichten liegt.

Hier wird, wie in [Len90] beschrieben, folgendes Balancekriterium, das vor jedem Knotentransfer auf Gültigkeit überprüft wird, verwendet.

$$\text{node\_weight\_on\_sideA} := \sum_{v_i \in A} \text{node\_weight}[v_i]$$

$$\text{node\_weight\_on\_sideB} := \sum_{v_i \in B} \text{node\_weight}[v_i]$$

$$\text{total\_node\_weight} := \sum_{v_i \in V} \text{node\_weight}[v_i] =$$

$$= \text{node\_weight\_on\_sideA} + \text{node\_weight\_on\_sideB}$$

$$\text{max\_node\_weight} := \max\{\text{node\_weight}[v_i] \mid v_i \in V\}$$

Abbildung 3-4: Definition von Gewichten

Es gelten die Definitionen aus Abbildung 3-4.

Nun muß für Seite A gelten (für Seite B analog):

$$\text{node\_weight\_on\_sideA} \leq \frac{\text{total\_node\_weight}}{2} + \text{max\_node\_weight}$$

Abbildung 3-5: Balanceformel

Selbstverständlich könnten hier auch andere Balancekriterien, die z.B. ein Knotengewicht von $\frac{2}{3}$ `* total_node_weight` auf einer Seite erlauben, verwendet werden. Aber auch für diese muß wenigstens die Eigenschaft gelten, daß zu jedem Zeitpunkt mindestens ein Knoten transferiert werden kann und zwar bis jeder genau einmal bewegt worden ist (vgl. Abschnitt 3.2.4). Durch lockerere Größenbeschränkungen können i. a. besserere, d. h. kleinere Schnitte gefunden werden. Die Balancierung kann darunter aber je nach Relaxation leiden.

3.2.2 Initialpartitionierung

Da weder in [FidMat82] noch in [Leng90] eine Anleitung gegeben wird, wie die Initialpartitionierung generiert werden soll, wird hier die *Random Initialization* Methode aus [HauBor97] vorgeschlagen (in [PreDie96] werden ebenfalls einige Methoden zur Generierung einer initialen Aufteilung vorgestellt).

Um eine einigermaßen ausgeglichene Start-Zweiteilung der Knotenmenge zu erreichen, werden zuerst alle Knoten in einer zufälligen Reihenfolge angeordnet. Danach wird der „Trennknoten“ gefunden, der am besten die Summe der Knotengewichte vor und nach ihm balanciert. Dazu enthält anfangs weder Seite A noch B einen Knoten. Dann werden Knoten von links nach rechts zu A hinzugefügt, und zwar nur so lange das Balancekriterium aus Abschnitt 3.2.1 noch nicht verletzt wird. Der Knoten, der es dann schließlich verletzt, ist der Trennknoten; er und die restlichen werden der Seite B zugeordnet.

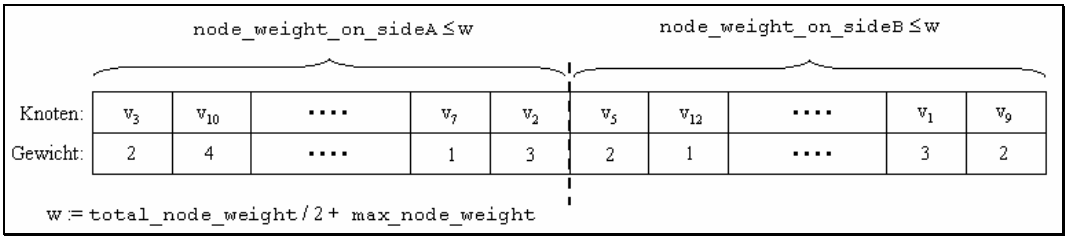

Abbildung 3-6: Generierung initialer Partitionen

Das ganze Verfahren benötigt O(|V|) Zeit.

Diese einfache Methode ist, wie es die Autoren von [HauBor97] durch umfangreiche Testreihen begründen, eine der besten der dort vorgestellten. Dafür gibt es ebenfalls laut [HauBor97] zwei Ursachen. Erstens, um so zufälliger die initialen Partitionen sind, um so leichter fällt es später davon abzuweichen. Mit anderen Worten, man ist nicht von vornherein in einer schlechten Partitionierung

gefangen und kann bessere Ergebnisse erzielen. Zweitens, die „intelligenteren“ Ansätze tendieren dazu, weniger Variationen in den Initialpartitionen zu produzieren, was zu weniger Variation in den Ergebnissen führt. Wenn man nun den besten aus mehreren Heuristikläufen herausnimmt, bekommt man bessere Resultate, indem man mehr Variation pro Lauf hat.

3.2.3 Die Datenstruktur und ihre Initialisierung

Für die nun folgenden Definitionen sei $v_i \in A$ beliebig, für alle $u_i \in B$ gelten sie analog.

Die externen Kantenkosten eines Knotens v_i werden wie folgt definiert:

$$E(v_i) := \sum_{e \in E_{ext,\,i}} \texttt{edge\_weight[e]}$$

wobei $E_{ext,\,i} = \{e \in E \mid v_i$ ist einziger Endknoten auf Seite A$\}$. Anders ausgedrückt, $E_{ext,\,i}$ ist die Menge der Kanten, bei denen v_i ein Endpunkt ist und der andere Endpunkt auf der Seite B liegt.

Die internen Kantenkosten eines Knotens v_i werden wie folgt definiert:

$$I(v_i) := \sum_{e \in E_{int,\,i}} \texttt{edge\_weight[e]}$$

wobei $E_{int,\,i} = \{e \in E \mid v_i$ ist ein Endknoten und kein Endknoten liegt auf Seite B$\}$. Mit anderen Worten, $E_{int,\,i}$ ist die Menge der Kanten, bei denen v_i ein Endpunkt ist und der andere Endpunkt ebenfalls auf der Seite A liegt.

In diesem Zusammenhang werden $E_{ext,\,i} \cup E_{int,\,i}$ die *kritischen Kanten* für v_i genannt. Im Gegensatz zu Hypergraphen sind bei normalen Graphen die *kritischen Kanten* eines Knotens genau alle dazu inzidenten Kanten.

Mit diesen Vorbereitungen wird der Beitrag eines Knotenstransfers zum Schnitt, das sog. *Gain* eines Knotens definiert:

$$\texttt{gain\_value[}v_i\texttt{]} := E(v_i) - I(v_i)$$

Dieser Wert kann effizient für Knoten der Seiten A und B, wie in der vorhergehenden Abbildung 3-7 exemplarisch für Seite A gezeigt, berechnet werden ($O(|V|+|E|)$).

```
1:  for (alle Knoten v_i ∈ A)
2:  {
3:     gain_value[v_i] := 0;
4:     for (jede Kante e_j mit v_i als einen Endknoten)
5:     {
6:        if (aside[e_j] = 1)
7:        {
8:           gain_value[v_i] += edge_weight[e_j]
9:        }
10:       if (bside[e_j] = 0)
11:       {
12:          gain_value[v_i] -= edge_weight[e_j]
13:       }
14:    }
15: }
```

Abbildung 3-7: Effiziente Berechnung der *Gain*-Werte

`aside[e]` bzw. `bside[e]` werden dabei schon in O(|E|) in einer vorhergehenden Schleife über alle Kanten e ∈ E bestimmt und enthalten die Anzahl der Endknoten von e aus Seite A bzw. B. In der gleichen Schleife, die `aside[e]` und `bside[e]` berechnet, werden auch zwei Knotenlisten `unlockedA[e]` und `unlockedB[e]` initialisiert (O(|E|)). Sie enthalten dadurch für jede Kante e eine Liste ihrer noch nicht gesperrten Endknoten, bei normalen Graphen können es maximal zwei sein.

Der *Gain*-Wert zeigt den Wert an, den ein Knoten bei einem Seitenwechsel zur Schnittgröße beiträgt. Positive *Gains* verkleinern den Schnitt um ihren Absolutwert und negative vergrößern ihn entsprechend.

Für jede Seite wird ein sog. *Bucket Array*, also `bucketA` für Seite A und `bucketB` für Seite B, wie es in Abbildung 3-8 dargestellt ist, aufgebaut. Hierbei handelt es sich um eine Art offenes Hashing. Die einzelnen *Buckets* werden durch doppelt verkettete Listen verwaltet.

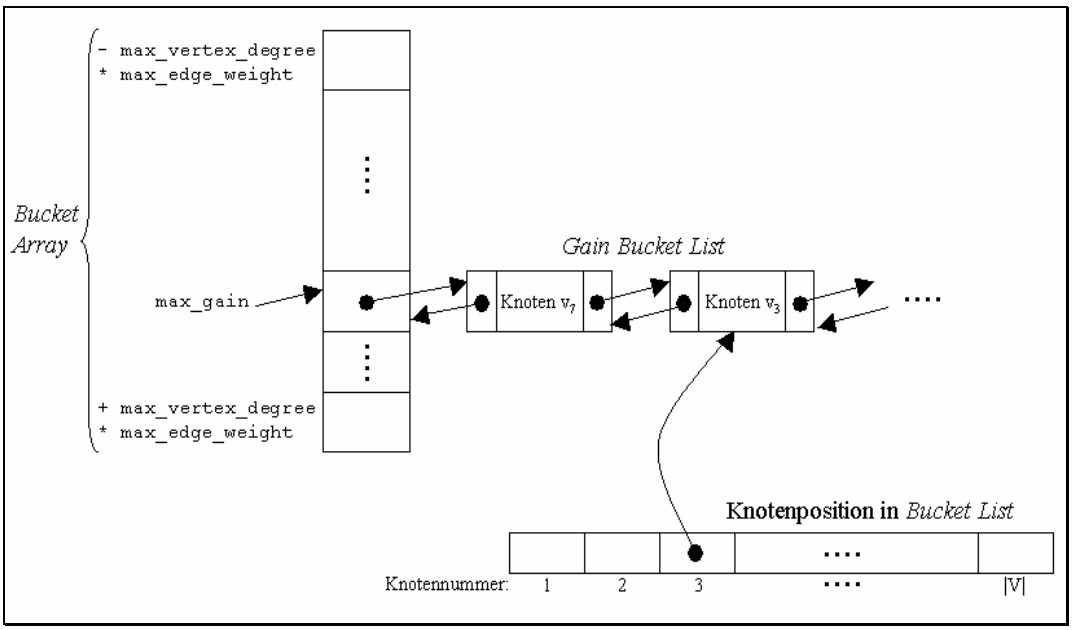

Abbildung 3-8: Datenstruktur für die Knotenauswahl

Die *Bucket Arrays* werden von dem in Abbildung 3-9 dargestellten Wertebereich indiziert.

```
[-max_vertex_degree * max_edge_weight ..
  max_vertex_degree * max_edge_weight]
```

Abbildung 3-9: Größe eines *Bucket Arrays*

Jeder Pointer in diesem Array zeigt also auf eine Liste von Knoten, die ein dem Array-Index entsprechendes *Gain* haben. Wie man leicht sieht, reicht die Größe des Arrays aus, um alle Knoten in sich aufzunehmen (siehe Definition von `gain_value[]` in diesem Abschnitt).

Die Laufzeit, um ein solches *Bucket Array* zu generieren und alle Knoten darin einzutragen, beträgt O(|V|).

3.2.4 Auswahl des nächsten Transferknoten

Der Knoten mit dem höchsten *Gain* kommt nun in Frage, auf die andere Seite bewegt zu werden. Um ihn in Linearzeit in |E| bestimmen zu können, wird die in

Abschnitt 3.2.3 beschriebene Datenstruktur benutzt. Zusätzlich besitzt jedes *Bucket Array*, `bucketA` bzw. `bucketB`, einen Zeiger, `max_gainA` bzw. `max_gainB`, der auf den jeweils höchsten Wert zeigt, für den gilt: `bucketA[max_gainA]` $\neq$ `nil` bzw. `bucketB[max_gainB]` $\neq$ `nil`.

Um nun den nächsten Knoten auszuwählen, kommen ein $v_A \in$ `bucketA[max_gainA]` und ein $v_B \in$ `bucketB[max_gainB]` in die engere Wahl. Beide Knoten existieren, außer eine Seite hat keine ungesperrten Knoten mehr und das dazugehörende *Bucket Array* ist daher schon leer. Sind beide leer, ist man mit dem aktuellen Durchlauf fast fertig, d. h. man kann das Ergebnis bestimmen.

Aufgrund des Balancekriteriums aus Abbildung 3-5 kann immer mindestens ein Knoten von beiden auf die andere Seite bewegt werden. Falls es genau einen solchen Knoten gibt, wird er auf die andere Seite transferiert, gesperrt und aus seinem *Bucket Array* gelöscht. Sperren heißt dabei, einen Knoten in seine zugehörige Liste `lockedA` bzw. `lockedB` zu kopieren, um ihn nicht nochmals innerhalb eines Durchlaufes auszuwählen.

Wenn beide Knoten v_A und v_B in Frage kommen, wird derjenige mit dem höheren *Gain* bevorzugt ausgewählt. Falls nun beide den gleichen *Gain*-Wert haben, fällt die Entscheidung zugunsten der Knotenbewegung, aus der die besser ausbalancierte Partitionierung resultiert. Kann auf diese Art und Weise noch immer nicht für einen der beiden Knoten entschieden werden, wird in [FidMat82] erlaubt, einen beliebigen auszuwählen. Hier wird vorgeschlagen, dann einfach den Knoten v_A auf die Seite B zu transferieren.

Der ganze Auswahlvorgang eines Knoten dauert O(1).

Danach muß ein Update der Datenstruktur, wie in Abschnitt 3.2.5 beschrieben, erfolgen.

Kann kein Knoten mehr bewegt werden, ist dieser Durchlauf fertig und die beste Zweiteilung daraus kann als Durchlaufergebnis zurückgegeben werden.

3.2.5 Update der Datenstruktur

Nachdem ein Knoten v_i auf die andere Seite bewegt worden ist, muß er aus seinem alten *Bucket Array* gelöscht werden. Das geschieht mit Hilfe des `position_in_bucket[`v_i`]` Arrays rechts unten in Abbildung 3-8. Hier wird die Postition des Knotens in seiner Knotenliste, auch *Bucket List* genannt, gespeichert. D. h. die Liste `bucketX[gain_value[`v_i`]]` muß nicht vollständig durchsucht werden, sondern v_i kann in O(1) gelöscht werden.

Desweiteren müssen nun alle Werte von Kanten e_j mit $v_i \in e_j$ neu berechnet werden. Updates erfolgen nur für diese Kanten, da sich sonst kein Wert in `aside[]` oder `bside[]` ändert.

Knoten, deren *Gain* sich dadurch ändert, müssen neu in ihr jeweiliges *Bucket Array* einsortiert werden, was natürlich wieder mit Hilfe von `position_in_bucket[`v_i`]` in je O(1) erreicht wird.

In Abbildung 3-10 wird der gesamte Ablauf exemplarisch für $v_i \in A$ zusammengefaßt, für $v_i \in B$ funktioniert das Ganze analog.

```
1:  for (jede zu v_i inzidente Kante e_j)
2:  {
3:     lösche v_i aus unlockedA[e_j];
4:     aside[e_j] -= 1;
5:     if (aside[e_j] = 0)
6:     {
7:        for (alle v_k ∈ unlockedB[e_j])
8:        {
9:           gain_value[v_k] –= edge_weight[e_j];
10:          update_bucketB(v_k);    // v_k neu einsortieren
11:       }
12:    }
13:    else if (aside[e_j] = 1)
14:    {
15:       for (alle v_k ∈ unlockedA[e_j])
16:       {
17:          gain_value[v_k] += edge_weight[e_j];
18:          update_bucketA(v_k);    // v_k neu einsortieren
19:       }
20:    }
21:    bside[e_j] += 1;
22:    if (bside[e_j] = 1)
23:    {
24:       for (alle v_k ∈ unlockedA[e_j])
25:       {
26:          gain_value[v_k] += edge_weight[e_j];
```

```
27:          update_bucketA(v_k);    // v_k neu einsortieren
28:       }
29:    }
30:    else if (bside[e_j] = 2)
31:    {
32:       for (alle v_k ∈ unlockedB[e_j])
33:       {
34:          gain_value[v_k] -= edge_weight[e_j];
35:          update_bucketB(v_k);    // v_k neu einsortieren
36:       }
37:    }
38: }
```

Abbildung 3-10: Neuberechnung der *Gain*-Werte

Nicht vergessen werden darf allerdings eine Neuberechnung von `max_gainA` und `max_gainB`, die aber teilweise gleich „on the fly“ in den Prozeduren `update_bucketA(`v_k`)` bzw. `update_bucketB(`v_k`)` erfolgt, wenn sich ihr Wert erhöht. Wenn sie jedoch kleiner werden, müssen sie anschließend auf den nächst kleineren Wert i mit `gain_valueA[i]` ≠ `nil` bzw. `gain_valueB[i]` ≠ `nil` gesetzt werden.

Verständlicherweise erhält man hier insgesamt eine Abschätzung der Laufzeit von O(|E| * `max_edge_weight`), vgl. auch [Leng90]. Der Faktor `max_edge_weight` erklärt sich folgendermaßen: Da `max_gainX` nach oben höchstens einmal per Update von `gain_value` um maximal `max_edge_weight` zunehmen kann, ergibt das insgesamt eine Komplexität von O(|E| * `max_edge_weight`). Andererseits kann die Anzahl der Erniedrigungen von `max_gainX` nicht die Größe des *Bucket Arrays* plus die Summe aller Erhöhungen übersteigen. Insgesamt erhält man also O(`max_vertex_degree` * `max_edge_weight` + |E| * `max_edge_weight`) und kann somit nach oben auch durch O(|E| * `max_edge_weight`) abgeschätzt werden.

3.2.6 Beispiel

Gegeben sei ein Graph G = (V, E), wie in Abbildung 3-11 gezeigt. Die Zahlen mit vorausstehendem Kreuz stellen die Knotenbezeichnungen dar. Alle anderen Zahlen sind Knoten- bzw. Kantengewichte, die hier, um das Beispiel nicht unnötig zu komplizieren, alle auf 1 gesetzt werden.

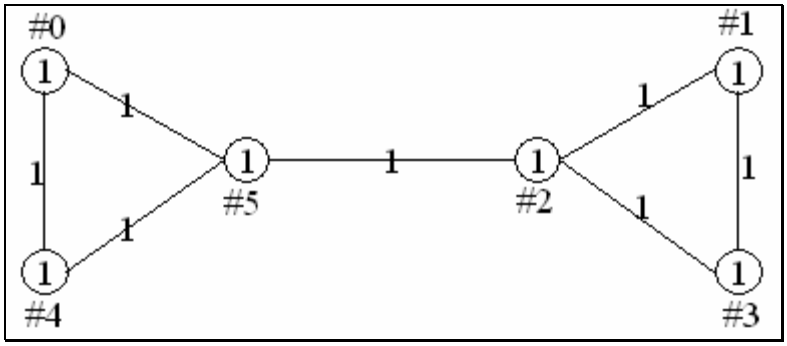

Abbildung 3-11: Beispiel Ausgangsgraph G

Als erstes wird V in zwei gleich große Hälften zufällig aufgeteilt. Eine mögliche Aufteilung befindet sich in Abbildung 3-12. Die dunkle Färbung der Knoten soll andeuten, daß sie zur Seite A gehören. Die weißen werden der Seite B zugerechnet.

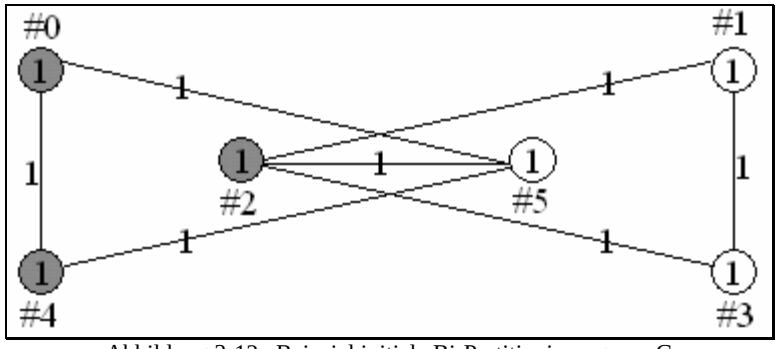

Abbildung 3-12: Beispiel initiale Bi-Partitionierung von G

Die Schnittgröße beträgt jetzt 5 * 1 = 5. Jetzt wird der erste Durchlauf der Heuristik gestartet. In die beiden *Bucket Arrays* mit einem Wertebereich von jeweils [-3, .., 3] werden alle Knoten entsprechend ihrem *Gain* aus den beiden nachfolgenden Tabellen einsortiert.

Knoten	gain_value
#0	1 – 1 = 0
#2	3 – 0 = 3
#4	1 – 1 = 0

Tabelle 3-1: Beispiel `gain_value[A]` vor erster Knotenbewegung

Knoten	gain_value
#1	1 – 1 = 0
#3	1 – 1 = 0
#5	3 – 0 = 3

Tabelle 3-2: Beispiel `gain_value[B]` vor erster Knotenbewegung

Nun wird, wie oben beschrieben, der Knoten #2 ausgewählt und von Seite A auf Seite B transferiert.

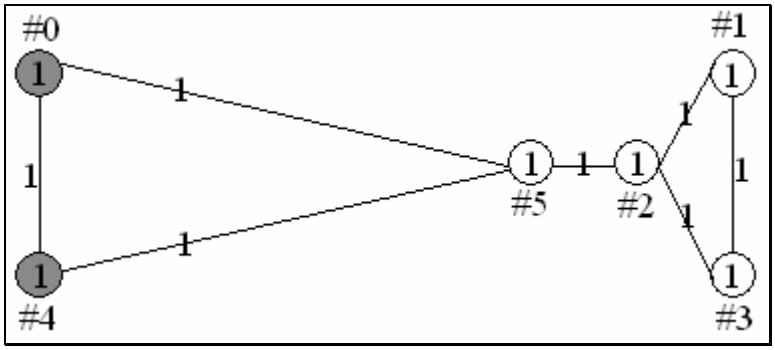

Abbildung 3-13: Beispiel G nach erstem Knotentransfer von #2

Die so erhaltene Schnittgröße beträgt 2 * 1 = 2. Nach dem Update der Datenstruktur sehen die *Knotengains* folgendermaßen aus:

Knoten	gain_value
#0	1 – 1 = 0
#4	1 – 1 = 0

Tabelle 3-3: Beispiel `gain_value[A]` nach dem Transfer von Knoten #2

In `gain_values[B]` tritt Knoten #2 nicht mehr in Erscheinung. Er wurde schon gesperrt und aus seiner Hash-Tabelle `bucketA` gelöscht.

Knoten	gain_value
#1	0 – 2 = -2
#3	0 – 2 = -2
#5	2 – 1 = 1

Tabelle 3-4: Beispiel `gain_value[B]` nach dem Transfer von Knoten #2

Da der Knoten #5 jetzt das höchste *Gain* hat, wird er ausgewählt und auf Seite A transferiert.

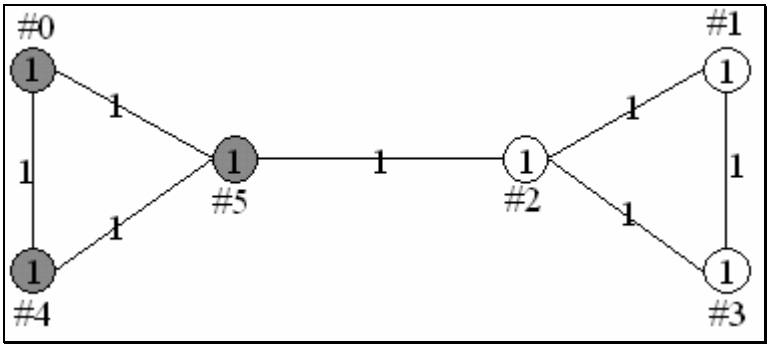

Abbildung 3-14: Beispiel G nach dem Transfer von Knoten #5

Die aktuelle Schnittgröße beträgt jetzt 1 * 1 = 1. Die so erhaltenen *Gains* betragen jetzt (zufälligerweise) alle –2.

Knoten	gain_value
#0	0 – 2 = -2
#4	0 – 2 = -2

Tabelle 3-5: Beispiel `gain_value[A]` nach dem Transfer von Knoten #5

Knoten	gain_value
#1	0 – 2 = -2
#3	0 – 2 = -2

Tabelle 3-6: Beispiel `gain_value[B]` nach dem Transfer von Knoten #5

Als nächstes kommt Knoten #0 an die Reihe. An dieser Stelle hätte man natürlich genauso einen anderen aus den beiden Tabellen nehmen können.

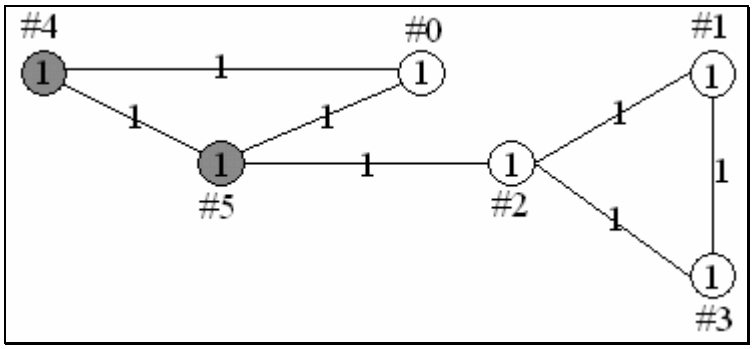

Abbildung 3-15: Beispiel G nach dem Transfer von Knoten #0

Damit hat man nun eine Schnittgröße von 3 * 1 = 3. Die neuen *Gain*-Werte sehen folgendermaßen aus:

Knoten	gain_value
#4	1 – 1 = 0

Tabelle 3-7: Beispiel `gain_value[A]` nach dem Transfer von Knoten #0

Knoten	gain_value
#1	1 – 1 = -2
#3	1 – 1 = -2

Tabelle 3-8: Beispiel `gain_value[B]` nach dem Transfer von Knoten #0

Der Knoten #4 hat zwar jetzt das höchste *Gain*, kann aber aus Balancegründen nicht auf Seite B bewegt werden. Auf jeder Seite darf sich ein Knotengewicht von maximal 6 / 2 + 1 = 4 befinden. Da die Summe über alle Knotengewichte auf der

Seite B schon 4 beträgt, wird Knoten #1 auf der B-Seite ausgewählt, um ihn auf Seite A zu transferieren.

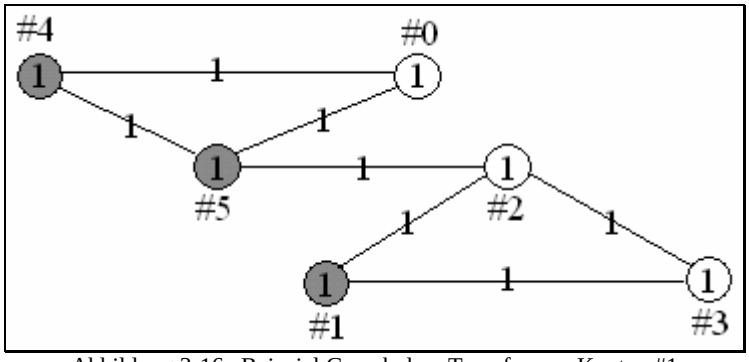

Abbildung 3-16: Beispiel G nach dem Transfer von Knoten #1

Die Schnittgröße beträgt nun 5 * 1 = 5.

Knoten	gain_value
#4	1 – 1 = 0

Tabelle 3-9: Beispiel `gain_value[A]` nach dem Transfer von Knoten #1

Knoten	gain_value
#3	1 – 1 = 0

Tabelle 3-10: Beispiel `gain_value[B]` nach dem Transfer von Knoten #1

Den nächsten Zuschlag bekommt Knoten #4.

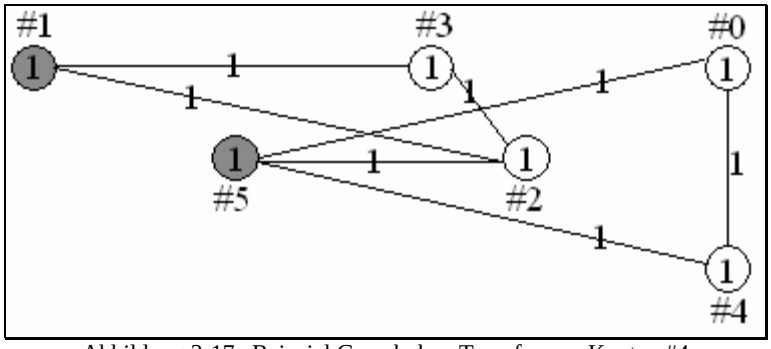

Abbildung 3-17: Beispiel G nach dem Transfer von Knoten #4

Aufgrund der gleichen *Gain*-Werte hätte natürlich ebenso Knoten #3 genommen werden können.

Die Schnittgröße bleibt dadurch 5. `bucketA` ist aber somit leer und es bleibt nur noch Knoten #3 auf der Seite B übrig.

Knoten	**gain_value**
#3	1 – 1 = 0

Tabelle 3-11: Beispiel `gain_value[B]` nach dem Transfer von Knoten #4

Nachdem dieser auf die Seite A bewegt worden ist, bekommt man schließlich folgende Zweiteilung:

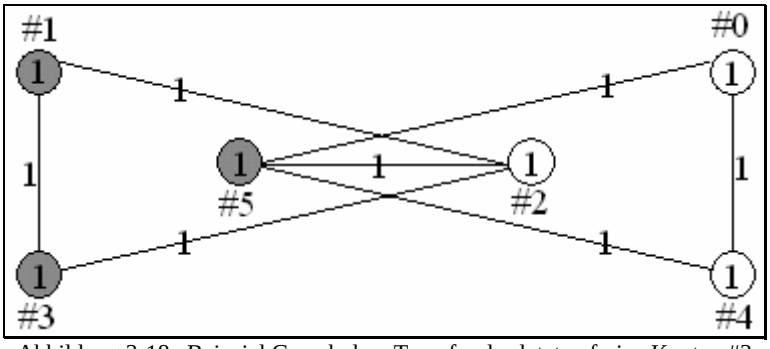

Abbildung 3-18: Beispiel G nach dem Transfer des letzten freien Knoten #3

Auch hier bleibt die Schnittgröße unverändert 5. Wie man sieht, erhält man am Schluß die gleiche Partitionierung wie am Anfang in Abbildung 3-12, nur mit dem Unterschied, daß alle Knoten genau einmal die Seite getauscht haben. Da kein weiterer freier Knoten mehr existiert, ist somit der Durchlauf beendet. Als Durchlaufergebnis wird die Zweiteilung mit der kleinsten Schnittgröße 1 aus Abbildung 3-14 festgesetzt.

Diese Aufteilung wird der Heuristik nun statt der Initialen aus Abbildung 3-12 im nächsten Durchlauf übergeben. Da es sich hier aber nicht nur um ein lokales Optimum, sondern bereits schon um ein globales Optimum handelt, wird der Algorithmus feststellen, daß er die Schnittgröße nicht mehr verbessern kann und terminiert somit.

Das Endergebnis ist also bereits in Abbildung 3-14 dargestellt.

3.3 Implementierungsbedingte Entscheidungen

Die im Rahmen dieser Arbeit angefertigte GTL-Implementierung namens `fm_partition` des Bipartitionierungsverfahren von Fiduccia und Mattheyses stützt sich im wesentlichen auf [FidMat82] und [Leng90]. Im großen und ganzen wurde auch die in [CaKaMa99] vorgeschlagene Architektur des Algorithmus eingehalten. Dies hat dank Objektorientierung den Vorteil, daß man die C++ Klasse des Algorithmus durch Vererbung ableiten und die in [CaKaMa99] beschriebene Komponenten, wie z. B. den `move_manager(...)`, an eigene Bedürfnisse anpassen und überschreiben kann. Auf diese Weise lassen sich selbstverständlich auch die Balancekriterien aus Abbildung 3-5 je nach Anforderung komfortabel verändern.

Wie schon in Abschnitt 3.1 erwähnt, werden hier nur normale Graphen und keine Hypergraphen unterstützt.

Programmierungsdetails können aus dem Source-Code bzw. aus den darin enthaltenen Kommentaren im KDOC–Format[6] abgelesen werden. Außerdem wird eine Anleitung für die praktische Nutzung dieses Algorithmus in der GTL-Dokumentation aufgenommen und daher online auf http://www.infosun.fmi.uni-passau.de/GTL/#doku bereitgestellt.

3.3.1 Vorverarbeitung

In einem Vorverarbeitungsschritt werden Schleifen[7] aus dem Graphen entfernt und damit vor dem Algorithmus versteckt. Natürlich hat das keinerlei Auswirkung auf die Partitionierung, da Schleifen sowieso nicht „zerteilt“ werden können.

3.3.2 Abweichungen

In diesem Abschnitt werden kleine Abänderungen des Originalalgorithmus aus [FidMat82] diskutiert.

---

[6] KDOC ist ein C++ Interface Dokumentationswerkzeug. Weitere Informationen dazu findet man z. B. unter http://www.ph.unimelb.edu.au/~ssk/kde/kdoc/ oder http://www.kde.org.

[7] Kanten $(u, v) \in E$ mit $u = v$

3.3.2.1 Initialisierung

Da die Qualität des Schnitts oft stark von der automatisch generierten initialen Zweiteilung der Knotenmenge abhängig ist, wird hier dem Benutzer zusätzlich die Möglichkeit eröffnet, eine solche initiale Aufteilung selbst zu erstellen und sie dem Algorithmus dann einfach als Parameter zu übergeben. Dies funktioniert durch Angabe der initialen Seite für jeden einzelnen Knoten.

3.3.2.2 Ungerichtete Graphen

Diese Fiduccia Mattheyses Implementierung läuft nur auf ungerichteten Graphen, da die Richtung einer Kante bei der Partitionierung keine Rolle spielt. Falls es dennoch gewünscht wird, einen gerichteten Graphen `G` zu partitionieren, kann man ihn vorher in O(1) mit einer Methode von GTL, `G.make_undirected()`, problemlos in einen ungerichteten verwandeln.

3.3.2.3 *Mehrere Schnitte mit minimaler Größe*

In den ersten Zeilen des Abschnitts 3.2 wurde erklärt, wie man aus dem Protokoll über die versuchsweisen Knotenbewegungen die Ergebnis-Partitionierung eines Durchlaufes berechnet. Und zwar wird dort, wie auch in [FidMat82] und [Leng90] angegeben, genau die Sequenz an Knotenbewegungen, die erste bis einschließlich die, die zu einer in diesem Durchlauf minimalen Schnittgröße führt, wirklich realisiert. Gibt es jedoch zufälligerweise mehrere Schnitte mit minimaler Größe in diesem Protokoll, scheint es sinnvoll, darunter denjenigen auszuwählen, aus dem die beste Balancierung resultiert.

3.3.2.4 Locked Lists

Hier werden keine Listen `lockedA` bzw. `lockedB` verwaltet. Statt dessen exisitiert eine Knoten-Map, die zu jedem Knoten seine momentane Seite speichert. Die doppelte Auswahl innerhalb eines Durchlaufes ist sowieso unmöglich, da ein schon bearbeiteter Knoten bereits aus den *Bucket Arrays* gelöscht ist. Ein weiterer Grund dafür ist, daß man die *Locked List* nicht, wie in [FidMat82] und [Leng90] vorgeschlagen, zum Aufbau der Datenstruktur des nächsten Durchlaufes verwenden kann. Bei einem Durchlauf werden alle Knoten jeweils einmal bewegt und dadurch auf der gegenüberliegenden Seite gesperrt, also in der *Locked List* der neuen Seite gespeichert. Dadurch erhält man am Ende des Durchlaufes in `lockedA` alle Knoten, die vorher auf Seite B waren und in `lockedB` alle von Seite A. Wenn man nun diese Listen bei einem erneuten

Durchlauf zum Aufbau der *Bucket Arrays* verwenden würde, startet man mit der gleichen alten Zweiteilung der Knotenmenge wie beim vorhergehenden Durchlauf und nicht mit der neuen, die man soeben ausgerechnet hat.

3.3.2.5 LIFO Organisation der Bucket Lists

Eine weitere Abweichung von der Beschreibung in [Leng90] und [FidMat82] ist die Verwaltung der *Bucket Lists* durch eine Last-In-First-Out (LIFO) Struktur. Dies soll laut [HaHuKa95] einen großen Performancegewinn mit sich bringen. Eine mögliche, ebenfalls dort angegebene Erklärung dafür ist, daß, wenn erst kürzlich besuchte Knoten bei der Auswahl bevorzugt werden, deren Nachbarschaft oder vielleicht sogar Cluster implizit zusammengefügt werden können.

Mit dieser Änderung ist jedenfalls kein großer Implementierungsaufwand verbunden.

3.3.2.6 Feste Knoten

Der Benutzer kann vor dem Durchlauf des Algorithmus optional festlegen, auf welcher Seite bestimmte Knoten im Ergebnis liegen müssen. Es liegt natürlich auf der Hand, daß darunter die Schnittgröße und die Balancierung, die natürlich auch Gewichte dieser festen Knoten berücksichtigt, extrem leiden können.

Wenn, bei einer ebenfalls durch den Nutzer vorgegebenen Initialpartitionierung, Knoten auf der anderen Seite liegen, als mit der Fixierung angegeben, bestimmt im Endeffekt die Fixierung.

3.3.3 Nachbearbeitung

Die Nachbearbeitung beschränkt sich darauf, Schleifen, die man bei der Vorverarbeitung 3.3.1 versteckt hat, wieder sichtbar zu machen. Damit wird der Graph, den der Benutzer dem Algorithmus übergibt, nicht verändert.

3.3.4 Ergebnisabfrage

Zusätzlich zu der Seite, auf der sich ein Knoten im Ergebnis befindet, kann auch die Größe des Schnitts, die Anzahl der benötigten Durchläufe der Heuristik (vgl.

Abschnitt 3.2) und die jeweilige Summe über alle Gewichte von Knoten auf der Seite A bzw. B abgefragt werden.

Diese Möglichkeiten erleichtern dem Benutzer, die Qualität der vom Algorithmus gelieferten Partitonierung zu beurteilen.

Falls der Benutzer es wünscht und er den entsprechenden Parameter setzt, werden auch Iteratoren bereitgestellt, die zum Durchlauf aller zum Schnitt gehörenden Kanten notwendig sind. Das Gleiche gilt für Iteratoren, mit denen man alle Knoten auf einer Seite durchlaufen kann.

3.4 Komplexität und Laufzeitverhalten

Die theoretische lineare Laufzeit O(|E|) der oben vorgestellten Bi-Partitionierungs-Heuristik erreicht natürlich in etwa auch die Implementierung. Die Entscheidungen, die in Abschnitt 3.3 getroffen worden sind, beeinträchtigen auf keinen Fall das Laufzeitverhalten. Eine genaue experimentelle Laufzeitbetrachtung findet sich in Anhang A.

Selbstverständlich muß dabei `max_edge_weight` - über alle Eingabegraphen gesehen - in der Größenordnung von O(1) liegen. Ansonsten käme eine Laufzeit von O(|E| * `max_edge_weight`) zustande. Das ist glücklicherweise bei den meisten Anwendungsgebieten, wie z. B. beim VLSI-Layout, der Fall. Denn im Gegensatz zu großen Knotengewichten liegt das maximale Gewicht einer Kante im Bereich von Wortlängen, die auf kleine Konstanten beschränkt sind.

Neben Schleifen, die alle Knoten durchlaufen (O(|V|)), brauchen Schleifen, die alle Kanten abwandern (O(|E|)), einen Großteil der CPU-Zeit innerhalb eines Durchlaufes. Eine genaue Laufzeitanalyse der einzelnen Stufen findet sich direkt bei den jeweiligen Beschreibungen im Abschnitt 3.2 oder in [Leng90].

Die maximale Anzahl der notwendigen Durchläufe ist ein wenig größer als beim Kernighan Lin Algorithmus mit experimentell ermittelten 4, fällt aber mit ebenfalls experimentell ermittelten 7 dennoch nicht aus dem Rahmen (vgl. [Leng90]). Es wird jeweils davon ausgegangen, daß diese nur mit Testläufen ermittelten Werte in etwa richtig sind, d. h. daß es sich dabei nur um kleine Konstanten handelt. Beweisen konnte dies bis jetzt noch niemand, das Gegenteil aber auch nicht.

Für Hypergraphen würde die Laufzeit übrigens O(p * `max_edge_weight`) betragen, wobei $p := \sum_{h \in H} |h|$ die Summe über die Anzahl von Elementen jeder Hyperkante ist.

Wie erwartet, läuft die Fiduccia Mattheyses Heuristik um einiges schneller als die Kernighan Lin Heuristik. Die Ergebnisse sind zwar von ähnlicher Qualität, aber laut [Leng90] nicht in allen Fällen ebenbürtig.

3.5 Graphische Oberfläche

GRAPHLET[8] eignet sich als Frontend für GTL hervorragend, um diese Partitionierungs-Heuristik zu visualisieren. Die Integration ist im Rahmen dieser Arbeit geschehen, um die Implementierung einigermaßen komfortabel testen und sie auch ordentlich präsentieren zu können.

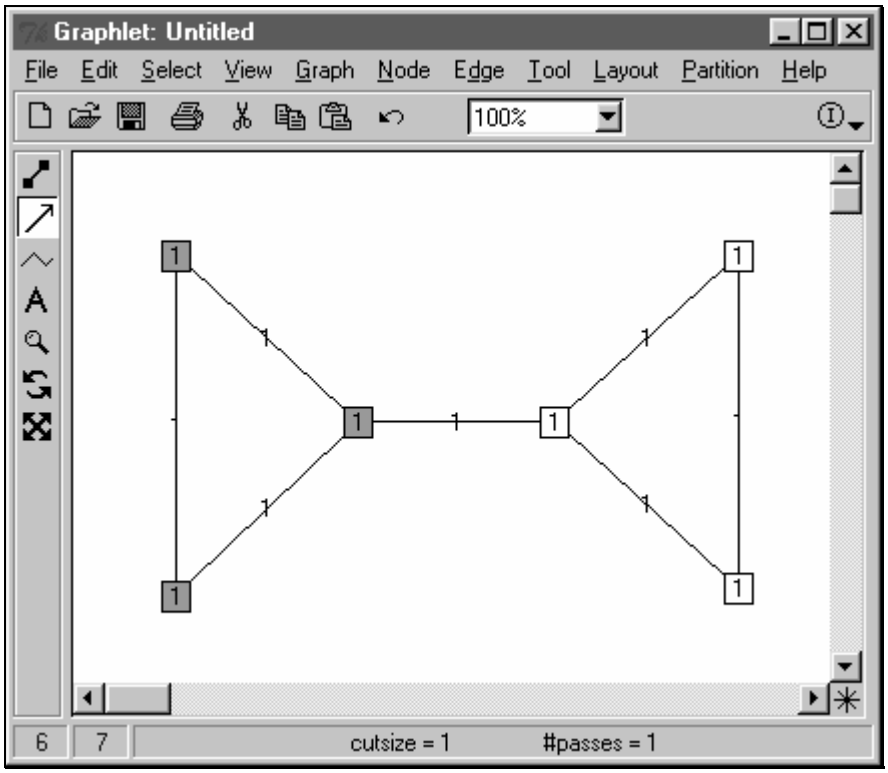

Abbildung 3-19: Graphlet 5.0.0 für MS-Windows mit Fiduccia Mattheyses Heuristik

[8] GRAPHLET ist ein Toolkit für Grapheneditoren und Graphenalgorithmen. Es wurde für die Betriebssysteme Unix und MS-Windows am Lehrstuhl von Prof. Dr. Brandenburg an der Universität Passau entwickelt. Weitere Informationen sind unter http://www.infosun.fmi.uni-passau.de/Graphlet/ zu finden.

Unter dem Menü „Partition“ versteckt sich der Aufrufpunkt für Fiduccia Mattheyses.

Eine initiale Partitionierung wird hierbei zufällig errechnet. Kein Knoten wird hier auf eine bestimmte Seite fix festgelegt. Die Knotenbeschriftung wird als `node_weight` und die Kantenbeschriftung als `edge_weight` benutzt. Falls diese leer sind, werden sie standardmäßig auf 1 gesetzt.

Unten in der Statuszeile befinden sich Angaben über die Größe des berechneten Schnitts, sowie die Anzahl an dafür benötigten Heuristik-Durchläufen.

3.6 Mögliche Erweiterung

Da sich der Fiduccia Mattheyses Algorithmus in den letzten zehn Jahren zum wahrscheinlich meist benutzten Partionierungsverfahren etabliert hat, gibt es sehr viele Arbeiten (z. B. [HaHuKa95] oder [AlHuKa98]), die sich mit der Verbesserung dieser Heuristik beschäftigen. Eine kleine Übersicht findet sich in [HauBor97]. In diesem Zusammenhang ist ein am meisten zitiertes Verfahren die Erweiterung von Krishnamurthy [Krishn84], welches in Abschnitt 3.6.2 kurz erklärt wird.

3.6.1 Clustering

Eine übliche Optimierung für das Fiduccia Mattheyses Verfahren, wie sie auch in [HauBor97] beschrieben wird, ist das sog. *Clustering*. Darunter versteht man das Zusammenfassen einzelner Knoten des zu partitionierenden Graphen. Knoten die gruppiert worden sind, werden aus dem Graphen gelöscht und ein Cluster-Knoten nimmt stellvertretend ihren Platz ein. Kanten, die vorher zu einem Knoten dieser Gruppe inzident waren, werden nun mit dem Knoten, der die Gruppe repräsentiert, verbunden.

Clustering Algorithmen werden vor der Anwendung des Fiduccia Mattheyses Algorithmus auf dem Graphen abschnittsweise eingesetzt, um anschließend eine bessere Performance in der Laufzeit zu erhalten. Denkbar wäre auch, durch solche manuelle *Clustering*-Eingriffe eine bessere Qualität des errechneten Ergebnisses zu erreichen, wenn die gruppierten Knoten sowieso nicht voneinander getrennt werden sollen oder dürfen.

Im Zuge dieser Arbeit ist natürlich auch ein effizienter *Clustering* Algorithmus namens `clustering` im Umfeld von GTL realisiert worden.

3.6.2 Gains höherer Ordnung

Krishnamurthy zeigt in [Krishn84] wie man die Fiduccia Mattheyses Heuristik effizient dahingehend erweitern kann, mehr „Look Ahead“ in die Auswahl aus mehreren Knoten des höchsten *Gain Bucket* zu bringen. Das originale Fiduccia Mattheyses Verfahren wählt zwischen Knoten mit gleichem *Gain* mehr oder weniger zufällig aus. Krishnamurthy versucht also, ein wenig von dem „verbleibenden Nichtdeterminismus“ aufzulösen. Dies geschieht durch Knotenauswahl in Hinblick auf damit zukünftig neu entstehende *Gain*-Werte.

In der folgenden Abbildung 3-20 stimmen `gain_value[`v_1`]` = 1 – 0 = 1 und `gain_value[`v_2`]` = 1 – 0 = 1 überein. Trotzdem sollte Knoten v_2 auf die Seite B bevorzugt transferiert werden, da dadurch `gain_value[`v_3`]` von 0 auf 1 erhöht wird. Der Transfer von Knoten v_1 hingegen verändert die *Gains* der verbleibenden Knoten von Seite A überhaupt nicht.

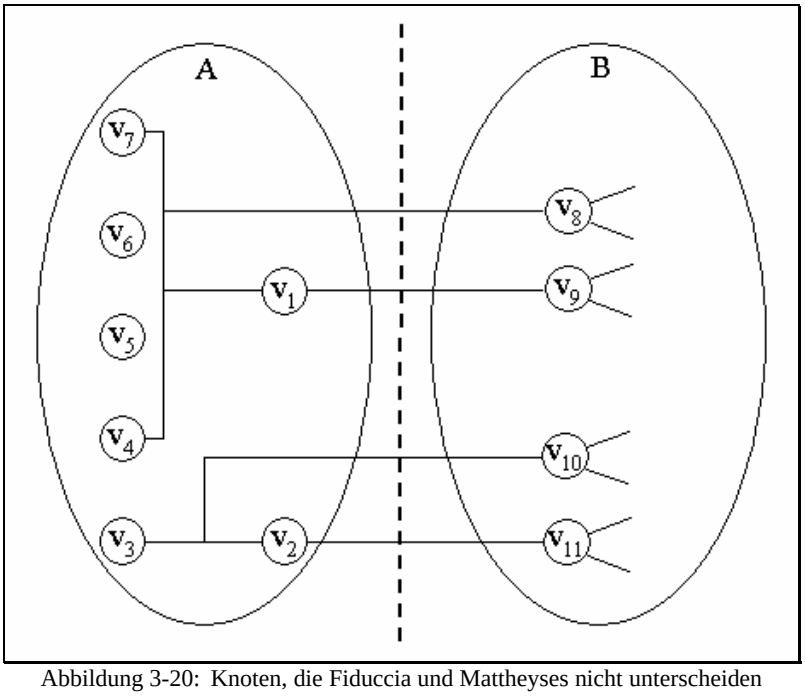

Abbildung 3-20: Knoten, die Fiduccia und Mattheyses nicht unterscheiden

Tatsächlich erweitert Krishnamurthy den *Gain*-Wert eines Knotens zu einem sog. *Gain Vektor*, welcher eine Serie von möglichen *Gains* für zukünftige Knotentransfers darstellt.

Sei HG = (V, H) ein Hypergraph. Um die Einträge des Vektors zu spezifizieren, wird als erstes die sog. *Binding Number* β definiert: $\beta_A(h)$ einer Kante $h \in H$ auf der Seite A ist die Anzahl der noch nicht gesperrten Endknoten von h auf der Seite A. Falls es einen gesperrten Endknoten von h auf der Seite A gibt, wird $\beta_A(h) := \infty$ gesetzt. $\beta_B(h)$ wird analog definiert.

Der k-te Eintrag `gain_vector[v_i][k]` eines *Gain Vektors* für einen Knoten v_i auf Seite A ist folgendermaßen definiert:

$$\texttt{gain\_vector}[v_i][k] := \sum_{h \in H_{pos,\,k}(v_i)} \texttt{edge\_weight}[h] - \sum_{h \in H_{neg,\,k}(v_i)} \texttt{edge\_weight}[h]$$

wobei: $1 \le k \le$ `number_of_levels`

$$H_{pos,\,k}(v_i) := \{h \in H \mid v_i \in h,\ \beta_A(h) = k,\ \beta_B(h) > 0\}$$

$$H_{neg,\,k}(v_i) := \{h \in H \mid v_i \in h,\ \beta_A(h) > 0,\ \beta_B(h) = k - 1\}.$$

Abbildung 3-21: Definition von `gain_vector[][]`

Die Auswahl eines Knotens geschieht jetzt durch Vergleich der jeweils zugehörigen *Gain Vektoren* bezüglich ihrer lexikographischen Reihenfolge. Bei jedem Knotenauswahlschritt wird ein bis jetzt noch nicht gesperrter Knoten mit dem lexikographisch größten *Gain Vektor* ausgewählt.

Im Beispiel von Abbildung 3-20 ist `gain_vector[v1]` = (1 – 0, 0 - 2) = (1, -2) und `gain_vector[v2]` = (1 – 0, 1 - 2) = (1, -1). Darum wird hier korrekterweise erst der Knoten v_2 auf die Seite B gebracht.

Je höher die Anzahl der Levels, `number_of_levels`, gewählt wird, desto mehr Knoten kann die Heurisik auf diese Art und Weise unterscheiden. Der erste Eintrag in diesem Vektor ist offenbar der ganz normale *Gain*–Wert, wie er anfangs in diesem Kapitel definiert wurde. Es macht offensichtlich nur Sinn, `number_of_levels` höchstens auf die maximale Anzahl inzidenter Knoten zu setzen, die eine Kante aus dem Hypergraphen hat.

3.6.2.1 Komplexität

Solange `number_of_levels` eine Konstante ist und damit O(`number_of_levels` * p) = O(p) mit p := $\sum_{h \in H} |h|$ gilt, kann das Fiduccia Mattheyses Verfahren so abgeändert werden, daß die Laufzeit erhalten bleibt, wie sie schon in Abschnitt 3.4 angegeben wird. Für normale Graphen G = (V, E) kann also wieder eine lineare Laufzeit von O(|E|) erreicht werden.

3.6.2.2 Datenstruktur

Eine Datenstruktur, die dies ermöglicht, ist eine `number_of_levels`-dimensionale Hash-Tabelle. Jeder Eintrag in dieser Tabelle zeigt auf eine doppelt verkettete Liste von Knoten, deren *Gain Vektor* mit den Koordinaten in der Tabelle übereinstimmt. Auch hier wird für Seite A und B jeweils eine solche Datenstruktur benötigt. In der folgenden Abbildung 3-22 ist sie exemplarisch für `number_of_levels` = 2 aufgezeichnet:

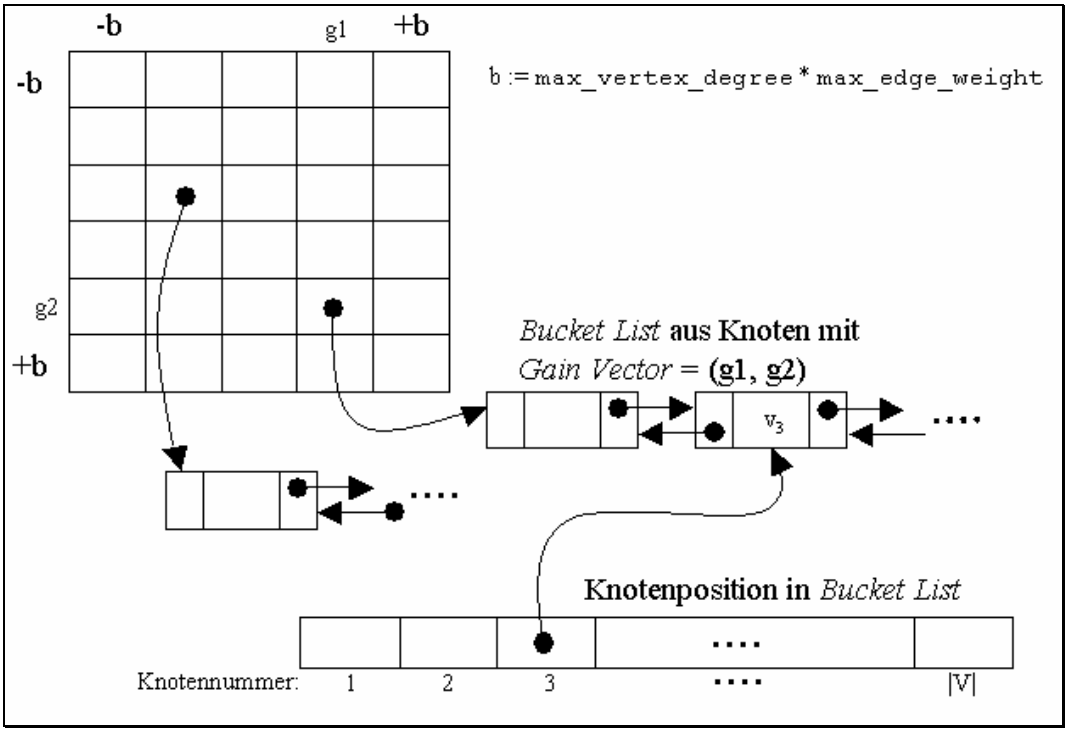

Abbildung 3-22: Krishnamurthys Datenstruktur

Der dazu benötigte Speicher wächst exponential mit `number_of_levels`.

Auch diese Methode kann, wie in [HaHuKa95] diskutiert, durch die Verwaltung der *Bucket Lists* als LIFO Struktur noch weiter verbessert werden. Falls dies nicht geschieht - so wird zumindest in [HaHuKa95] behauptet - läuft der normale Algorithmus ohne *Gains* höherer Ordnung, wie er vorher ebenfalls mit LIFO *Bucket List* Organisation vorgestellt worden ist, in der Praxis schneller.

Kapitel 4

Die Ratio Cut Heuristik

Das *Ratio Cut* Partitionierungsverfahren wurde 1991 erstmals vollständig in [WeiChe91] von Y. Wei und C. Cheng vorgestellt. Im großen und ganzen basiert darauf auch dieses Kapitel. Weitere Literatur dazu findet sich u. a. auch in [CheWei91] und [RoxSen97]. Wie schon bei Fiduccia und Mattheyses wird auch hier versucht, eine heuristische Annäherung an die optimale Lösung des NP-vollständigen BIPARTITION Problems zu finden.

Dieser Algorithmus baut im wesentlichen auf der Idee mit den *Bucket Arrays* von Fiduccia und Mattheyses auf. Dies geschieht hauptsächlich aus drei Gründen: Die Datenstruktur ist effizient aufgrund linearer Laufzeit, generiert relativ vielversprechende Ergebnisse und bietet die Möglichkeit, Knoten auf eine bestimmte Seite fix festzusetzen.

Ratio Cut kann ebenfalls auf Hypergraphen angewendet werden. Hier bleibt die Diskussion jedoch wieder auf normale Graphen beschränkt, da die GTL-Implementierung des Algorithmus, wie auch schon in Abschnitt 3.1 mit der Fiduccia Mattheyses Heuristik erwähnt, auf normalen Graphen arbeitet.

4.1 Motivation

Bei *Ratio Cut* handelt es sich um eine Methode, die die natürliche Clusterung eines Graphen bestimmt. Dabei werden bewußt keine festen Größenbeschränkungen der Partitionen eingesetzt, wie bei [KerLin70] oder in Abbildung 3-5 angegeben, da diese eventuell von vornherein gar nicht genau bekannt sind und daher eine künstliche Einschränkung darstellen können.

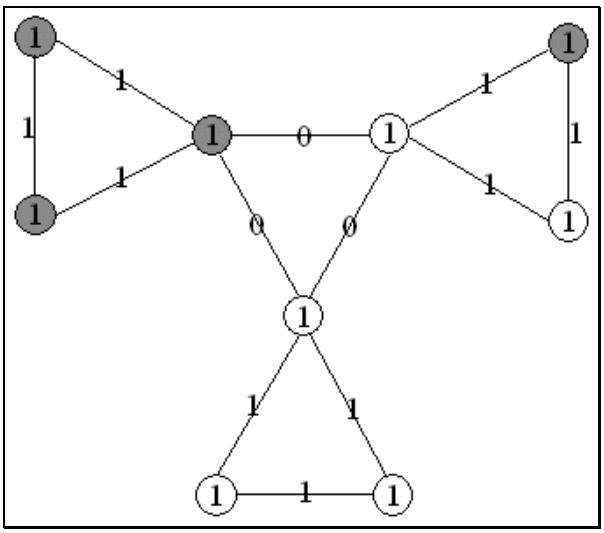

Abbildung 4-1: Partitionierung mit Fiduccia Mattheyses

Die Zahlen an den Knoten und Kanten stellen die jeweiligen Gewichte dar.

Aufgrund des Balancekriteriums aus 3.2.1 darf sich auf jeder Seite ein Knotengewicht von maximal (9 / 2) + 1 = 5,5 befinden. D. h. hier dürfen auf jeder Seite maximal 5 Knoten sein, was in einer nicht optimalen Schnittgröße von 2 resultiert. Dieser Schnitt entspricht also einem Bifurkator.

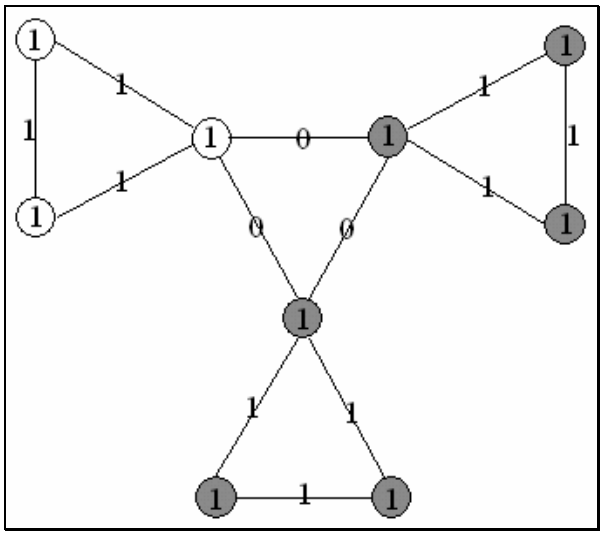

Abbildung 4-2: Partitionierung mit *Ratio Cut*

Hier wird auf Kosten einer etwas schlechteren Balancierung ein viel natürlicherer Schnitt der Größe 0 gefunden.

Falls ein Knoten des obigen Graphen ein Gewicht von mindestens 8 und alle anderen 1 haben, faßt der Fiduccia Mattheyses Algorithmus alles in eine Partition zusammen, da sich nun auf einer Seite ein Gewicht von (16 / 2) + 8 = 16 befinden darf. Die Schnittgröße ist mit 0 offensichtlich minimal.

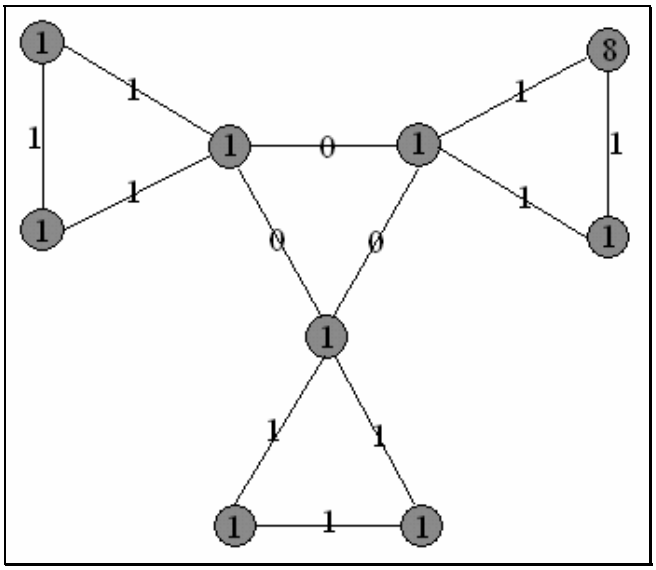

Abbildung 4-3: Fiduccia Mattheyses mit zu großen Knotengewichten

Ratio Cut liefert beim gleichen Graphen folgendes Ergebnis:

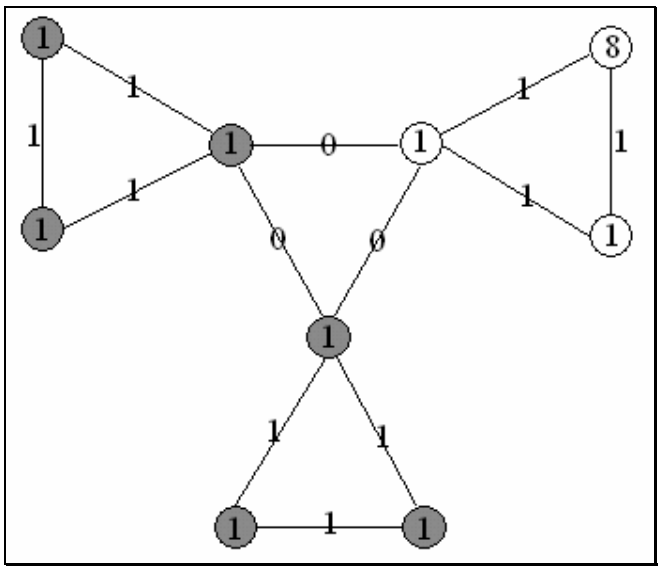

Abbildung 4-4: *Ratio Cut* mit großen Knotengewichten

4.1.1 Ziel

Das Ziel dieses Verfahrens ist es, das zukünftig mit *Ratio* bezeichnete Verhältnis eines Schnitts, `cut_ratio`, wie es nachfolgend definiert wird, zu minimieren.

$$\texttt{cut\_ratio} := \frac{\texttt{cutsize}}{\texttt{node\_weight\_on\_sideA} * \texttt{node\_weight\_on\_sideB}}$$

wobei:

$$\texttt{node\_weight\_on\_sideA} := \sum_{v_i \in A} \texttt{node\_weight}[v_i] \neq 0$$

$$\texttt{node\_weight\_on\_sideB} := \sum_{v_i \in B} \texttt{node\_weight}[v_i] \neq 0$$

Abbildung 4-5: Definition: *Ratio* eines Schnitts

Falls alle Knotengewichte einheitlich 1 betragen, entspricht `node_weight_on_sideA` bzw. `node_weight_on_sideB` klarerweise der Anzahl der Knoten, die sich auf der Seite A bzw. B befinden.

Einen Schnitt mit einem optimalen *Ratio*, auch *Ratio Cut* genannt, zu finden, ist laut [WeiChe91] NP-vollständig. Daher ist es auch hier notwendig, eine gute und schnelle Heuristik zu verwenden, um einen auch in der Praxis brauchbaren Algorithmus, wie er in Absatz 4.2 beschrieben wird, zu erhalten.

4.1.2 Clustering Eigenschaft von *Ratio Cut*

Wieso soll ausgerechnet die Minimierung des in Abbildung 4-5 definierten *Ratio* eine zufriedenstellende Annäherung an die optimale Lösung des BIPARTITION Problems liefern?

Um diese Frage zu beantworten, stelle man sich einen zufälligen Graphen G = (V, E) mit gleichmäßig durch p verteilten Kanten vor. Sei p dabei die Wahrscheinlichkeit, mit der sich eine Kante e = (v_1 ,v_2) $\in$ E zwischen zwei disjunkten Knoten v_1 und v_2 aus V befindet. Die Knoten- und Kantengewichte stelle man sich der Einfachheit halber auf 1 gesetzt vor. Sei CUT_1 nun ein Schnitt, der G in die Teilmengen (Seiten) A und B partitioniert. A hat dabei eine Größe von α * |V| und B von (1 - α) * |V| Knoten, wobei $0 < \alpha < 1$ gilt.

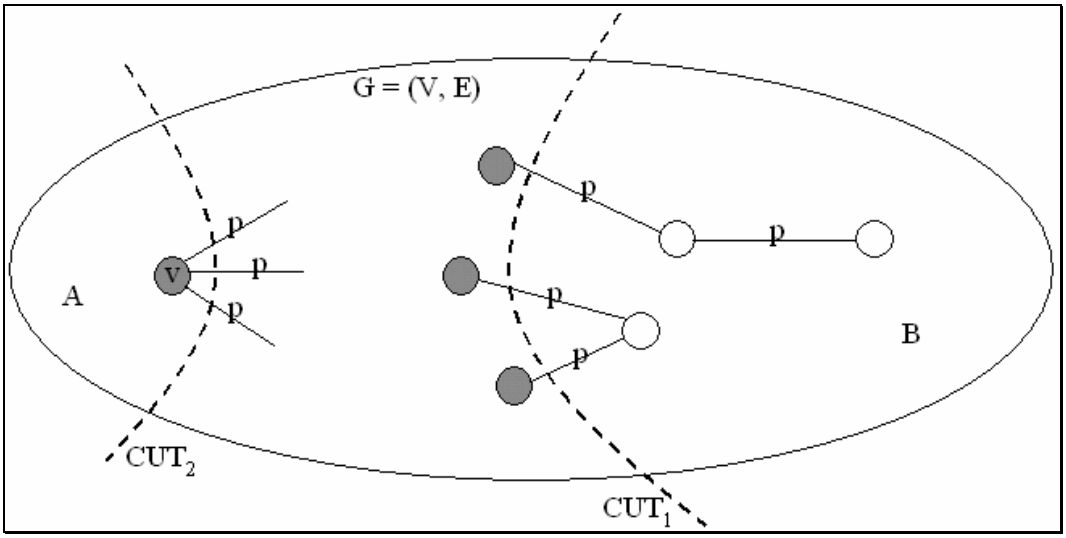

Abbildung 4-6: Zufallsgraph mit |V| Knoten und Kantenwahrscheinlichkeit p

Die zu erwartende Schnittgröße `cutsize` von CUT_1 ist p multipliziert mit der Anzahl an möglichen Kanten zwischen der Seite A und B:

$$EW(cutsize) = p * |A| * |B| = p * \alpha * (1 - \alpha) * |V|^2$$

Abbildung 4-7: Erwartungsgemäße Schnittgröße

Angenommen ein anderer Schnitt CUT_2 trennt nur einen Knoten $v \in V$ von den übrigen ab, d. h. $A = \{v\}$ und $B = V - \{v\}$, dann sieht die zu erwartende Schnittgröße folgendermaßen aus:

$$EW(cutsize) = p * (|V| - 1)$$

Abbildung 4-8: Schnittgröße bei Abtrennung nur eines Knotens

Bei $|V| \rightarrow \infty$ wird der Wert von Abbildung 4-7 natürlich viel größer als der von Abbildung 4-8. Dies erklärt jetzt auch, wie in der Einleitung schon erwähnt, warum Netzwerkfluß-Algorithmen i. a. sehr unbalancierte Lösungen errechnen. Sehr unterschiedlich große Teilmengen haben natürlicherweise die kleinste Schnittgröße. Darum wird ein *Ratio*-Wert von $\frac{cutsize}{|A|*|B|}$ vorgeschlagen.

Bei gleicher Schnittgröße ist eine halb-halb Aufteilung $\frac{1}{2} * \frac{1}{2} = \frac{1}{4}$ 3mal besser als eine mit dreiviertel-einviertel $\frac{1}{4} * \frac{3}{4} = \frac{3}{16}$.

Es ergibt sich, daß es sich bei dem zu erwartenden *Ratio*, in Hinblick auf verschiedene Schnitte, um einen konstanten Wert handelt:

$$EW(cut\_ratio) = EW\left(\frac{cutsize}{|A| * |B|}\right) = EW\left(\frac{p * |A| * |B|}{|A| * |B|}\right) = p$$

Abbildung 4-9: Konstanter *Ratio*-Wert

Aus diesem Grund haben bei Graphen mit gleichverteilten Kanten erwartungsgemäß alle Schnitte das gleiche *Ratio*. Mit anderen Worten, bei diesen Spezialgraphen ist die Auswahl des Schnitts gleichgültig.

Handelt es sich bei der Verteilung der Kanten nicht etwa um eine Gleichverteilung, generieren verschiedene Schnitte natürlich auch verschiedene *Ratios*. Schnitte, die schwächer zusammenhängende Knotengruppen trennen, haben ein kleineres *Ratio* als Schnitte durch stärker zusammenhängende Komponenten.

Eine weitere Rechtfertigung des *Ratio Cut* über Netzwerkfluß-Formulierungen wird in [WeiChe91] vorgestellt.

4.2 Grundlagen

Im großen und ganzen besteht der *Ratio Cut* Algorithmus aus den drei Hauptabschnitten Initialisierung, Iteratives Shifting und Gruppenaustausch.

Diese Phasen werden nachfolgend detailiert beschrieben.

4.2.1 Initialisierung

Anfangs wird ein Knoten `source_node` $\in$ V zufällig als Startknoten ausgewählt. Ein weiterer Knoten `target_node` $\in$ V wird auf das Ende eines

graphentheoretisch längsten Pfades gesetzt, der durch Breitensuche ausgehend von `source_node` gefunden wird. Der Startknoten gehört fest zur Seite A, der Endknoten fest zur Seite B.

Danach wird A := {`source_node`} und B := V – {`source_node`} gesetzt.

Anschließend wird, genau wie bei Fiduccia und Mattheyses, für Seite B ein *Bucket Array* aufgebaut (vgl. 3.2.3) und die Knoten entsprechend ihrem *Gain*-Wert einsortiert.

Außer `target_node` werden jetzt die Knoten der Reihe nach mit dem jeweils höchsten `ratio_gain` von der Seite B auf die Seite A gebracht. Je größer das `ratio_gain` eines Knotens ist, um so kleiner wird `cut_ratio`, wenn dieser Knoten auf die andere Seite transferiert wird. Natürlich kann `ratio_gain` auch negativ sein, d. h. die Variable `cut_ratio` wird bei dieser Knotenauswahl größer. Eine exakte Definition folgt in der nächsten Abbildung:

Für $v_i \in A$ gilt:

```
ratio_gain[vi] := gain_value[vi] /
((node_weight_on_sideA + node_weight[vi]) *
 (node_weight_on_sideB – node_weight[vi]))
```

Für $v_i \in B$ gilt:

```
ratio_gain[vi] := gain_value[vi] /
((node_weight_on_sideA - node_weight[vi]) *
 (node_weight_on_sideB + node_weight[vi]))
```

Abbildung 4-10: *Ratio Gain* eines Knotens

Um nicht alle Knoten aus V durchsuchen zu müssen, um denjenigen mit dem größten `ratio_gain` zu finden, wird nur für jeden Knoten v_k aus der Liste `bucketB[max_gainB]` das zugehörige `ratio_gain[`v_k`]` berechnet. Der Grund, warum nur Knoten mit dem höchsten *Gain*-Wert betrachtet werden, ist folgender: Wenn ein Knotengewicht nicht ungewöhnlich groß ist, trägt es, im Nenner der Formel aus Abbildung 4-10 stehend, bei weitem nicht so viel zu `ratio_gain` bei, wie die Variable `gain_value` im Zähler. Anschließend ist selbstverständlich auch hier ein Update der Datenstruktur nötig, wie z. B. `max_gainB` neu bestimmen, adjazente Knoten mit verändertem *Gain* neu einsortieren, Werte für inzidente Kanten neu berechnen, etc (vgl. Abschnitt 3.2.5).

Wie oben schon beschrieben, kann sich so `cut_ratio` auch verschlechtern, also vergrößern. Dies soll dem Algorithmus die Möglichkeit bieten, aus etwaigen lokalen Minima herausklettern zu können.

Jeder Schritt wird samt daraus resultierendem `cut_ratio` protokolliert. Der Schnitt mit einem minimalen `cut_ratio` aus diesem Protokoll wird festgehalten. Den ganzen Vorgang nennt man in diesem Zusammenhang *Left Shifting Operation*.

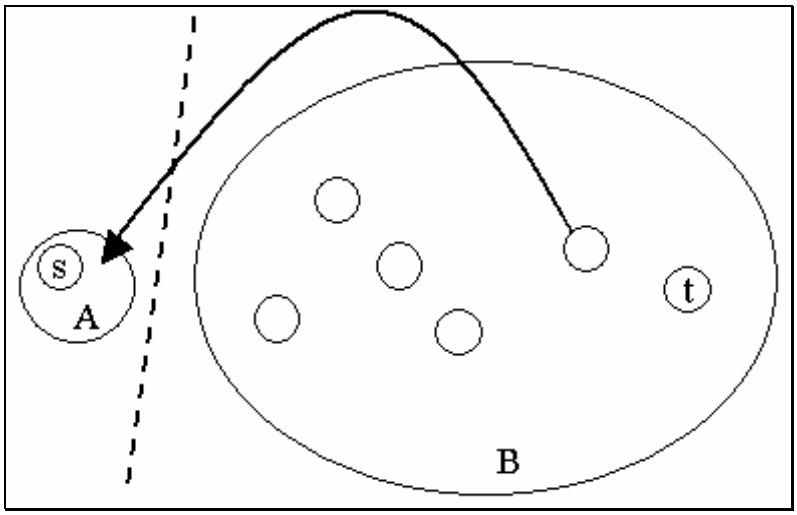

Abbildung 4-11: *Left Shifting Operation*

Als nächstes wird jetzt die ganze Prozedur mit A := V – {`target_node`} und B := {`target_node`} wiederholt. Das *Bucket Array* `bucketA` wird dementsprechend aufgebaut. Alle Knoten von Seite A (außer `source_node`) werden der Reihe nach dem höchsten `ratio_gain` auf die Seite B transferiert und der daraus resultierende Schnitt mit dem kleinsten `cut_ratio` wird anschließend aus den zugehörigen Protokollangaben ermittelt. Diesen Vorgang nennt man *Right Shifting Operation*.

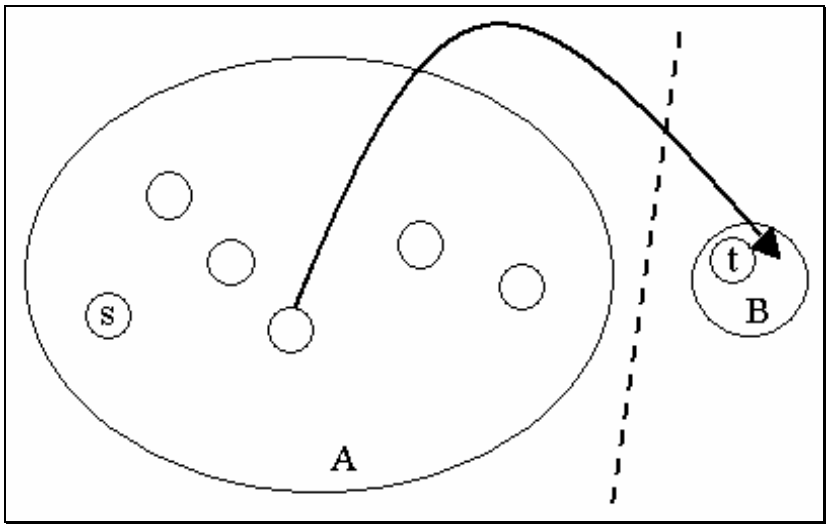

Abbildung 4-12: *Right Shifting Operation*

Zum Schluß wird festgestellt, welche Operation, *Left Shifting* oder *Right Shifting*, den besseren Schnitt, d. h. den mit dem niedrigsten *Ratio*, erzeugt hat. Genau diese Zweiteilung der Knotenmenge wird dann als initiale Partitionierung der nächsten Phase der Heuristik, dem iterativen Knoten-Shifting, übergeben.

```
1:  Wähle zufällig einen Knoten source_node ∈ V aus;
2:  Finde einen Knoten target_node ∈ V am Ende eines
    graphentheoretisch längsten Pfades durch Breitensuche;
3:  Baue Datenstruktur aus A := {source_node} und
    B := V - {source_node, target_node} auf;
4:  repeat
    {
       Wähle Transferknoten v_i mit höchstem ratio_gain aus der
       Liste bucketB[max_gainB] aus;
       Transferiere v_i auf Seite A → A := A ∪ {v_i} und
       B := B – {v_i};
       update bucketB und max_gainB;
    }
    until (B = ∅);
5:  Baue Datenstruktur aus A := V - {source_node, target_node} und
    B := {target_node} auf;
```

```
6:  repeat
    {
       Wähle Transferknoten v_i mit höchstem ratio_gain aus der
       Liste bucketA[max_gainA] aus;
       Transferiere v_i auf Seite B → A := A - {v_i} und
       B := B ∪ {v_i};
       update bucketA und max_gainA;
    }
    until (A = ∅);
7:  return (Partitionierung mit kleinstem hier gefundenen
    cut_ratio);
```

Abbildung 4-13: Kurzbeschreibung der Initialisierung

4.2.2 Iteratives Shifting

Auf einer derartig generierten, initialen Knotenaufteilung wiederholt man die *Shifting Operation*, diesmal in die andere Richtung. Kann sich dabei das *Ratio* des Schnitts verbessern, startet man die *Shifting Operation* mit der so erhaltenen Partitionierung erneut, aber wieder in die entgegengesetzte Richtung als vorher. Stellt sich jedoch keine Verbesserung ein, bleibt alles bei der Ausgangssituation.

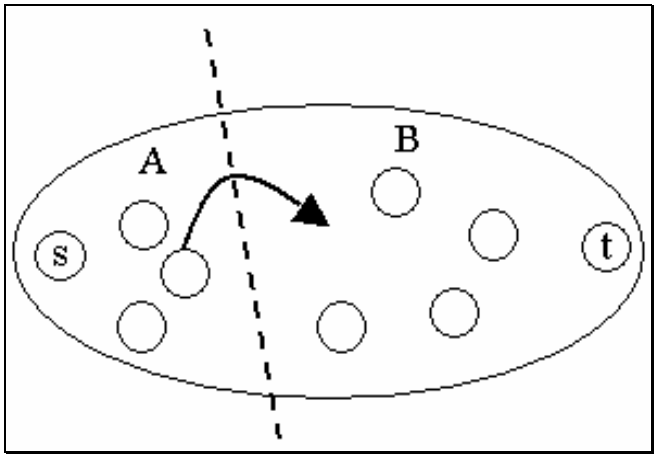

Abbildung 4-14: Iteratives *Right Shifting*

Angenommen der initiale Schnitt ist in Abschnitt 4.2.1 durch eine *Left Shifting Operation* gefunden worden, so werden nun die Knoten der Seite A nach B transferiert (bei initialem *Right Shifting* analog von Seite B nach A).

Diese Phase hat also folgenden Ablauf:

```
1:  Baue Datenstruktur entsprechend Startpartitionierung auf;
2:  repeat
    {
       right_shifting_operation();
    }
    until (A besteht nur noch aus source_node);
3:  wähle Schnitt mit minimalem Ratio aus Schritt 2 aus;
    if (cut_ratio wurde durch Schritt 2 verbessert)
    {
       Schnitt mit minimalem cut_ratio ist neue
       Startpartitionierung;
    }
    else
    {
       return Startpartitionierung;
       exit;
    }
4:  wiederhole Schritt 1 bis 3 mit left_shifting_operation();
5:  wiederhole Schritt 1 bis 4;
```

Abbildung 4-15: Kurzbeschreibung des iterativen *Shiftings*

An diesem Punkt ist jetzt ein lokales Minimum erreicht und zwar in dem Sinn, daß ein Transfer eines einzelnen Knotens von einer Seite auf die andere keine Verbesserung von `cut_ratio` mehr bewirken kann.

4.2.3 Gruppenaustausch

Aus diesem Grund wird nun für jede Seite ein *Bucket Array* aufgebaut. Der Reihe nach werden erneut alle Knoten auf ihre gegenüberliegende Seite gebracht. Der

Unterschied zu den beiden vorherigen Phasen ist hier die Auswahl des nächsten Transferknotens. Nun werden nämlich beide Listen `bucketA[max_gainA]` und `bucketB[max_gainB]` nach dem Knoten mit dem höchsten `ratio_gain` durchsucht. Dies wird so lange fortgeführt, bis alle Knoten genau einmal bewegt worden sind.

Hier können erstmals auch `source_node` und `target_node` auf die gegenüberliegende Seite wandern.

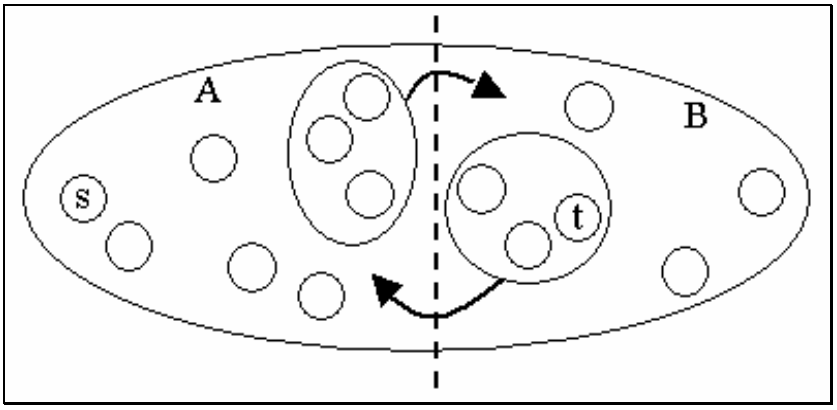

Abbildung 4-16: Knotengruppenaustausch

Es ist dabei allerdings zu beachten, daß keine Seite komplett leer an Gewicht wird, da ansonsten `node_weight_on_sideA` oder `node_weight_on_sideB` den Wert 0 hat! Dies hätte nämlich in der Formel von `cut_ratio` aus Abbildung 4-5 eine Division durch 0 zufolge.

```
1:  Baue Datenstruktur aus Startpartitionierung auf;
2:  while (∃ mindestens ein ungesperrter Knoten)
    {
       Wähle einen Knoten v mit dem höchsten ratio_gain aus
       bucketA[max_gainA] oder bucketB[max_gainB] aus;
       Transferiere Knoten v auf die andere Seite und sperre ihn;
       update bucketA, bucketB, max_gainA und max_gainB;
    }
```

```
3:  if (cut_ratio wurde dadurch verbessert)
    {
       Schnitt mit diesem cut_ratio ist neue Startpartitionierung;
       Starte Gruppenaustausch erneut;
    }
    else
    {
       return Startpartitionierung;
       exit;
    }
```

Abbildung 4-17: Kurzbeschreibung des Gruppenaustausches

Auch hier wird wieder jeder Schritt mit daraus resultierendem `cut_ratio` protokolliert. Falls der in dieser Prozedur errechnete Schnitt mit dem minimalen `cut_ratio` eine Verbesserung gegenüber vorher ist, wird mit diesem der Gruppenaustausch erneut gestartet. Andernfalls wird einfach die ursprüngliche Lösung ausgegeben.

4.3 Implementierungsbedingte Entscheidungen

Die GTL-Implementierung von *Ratio Cut* heißt `ratio_cut_partition`.

Wie schon bei der Fiduccia Mattheyses Heuristik hängt die Qualität des Ergebnisses von der Qualität der Initialpartitionierung ab. Aus diesem Grund kann der Benutzer auch hier eine solche Zweiteilung der Knotenmenge als Parameter optional vorgeben, d. h. auf Wunsch kann für jeden einzelnen Knoten eine Initialseite vorgegeben werden. Unabhängig davon kann man selbstverständlich auch hier den Graphen mit `clustering` (vgl. Abschnitt 3.6.1) vorab bearbeiten, um insgesamt ein besseres Ergebnis zu erhalten.

Desweiteren sind auch der Start- und der Zielknoten bei Bedarf vorher festlegbar. Dieser Algorithmus läuft ebenfalls nur auf ungerichteten Graphen.

Programmierungsdetails können aus dem Source-Code bzw. aus den darin enthaltenen Kommentaren im KDOC–Format abgelesen werden. Die

Beschreibung für die praktische Nutzung findet sich in der GTL-Dokumentation, also online unter http://www.infosun.fmi.uni-passau.de/GTL/#doku.

Variablen und Methoden sind aufgrund der besseren Übersicht, soweit sie in der Fiduccia Mattheyses Implementierung identische Bedeutung haben, auch gleich benannt worden.

Auch hier kann die C++ Klasse `ratio_cut_partition` problemlos abgeleitet und dadurch Methoden, wie z. B. `cutratio()`, überschrieben und nach eigenen Bedürfnissen modifiziert werden.

In den folgenden Abschnitten werden weitere, in dieser Implementierung realisierte Abweichungen und Verbesserungen vorgestellt.

4.3.1 Vorverarbeitung

In einem Vorverarbeitungsschritt wird der Graph temporär zusammenhängend gemacht. Dies geschieht durch eine Tiefensuche, bei der alle Startknoten, auch Wurzeln genannt, in einer Liste gespeichert werden. Diese werden dann durch Kanten mit einem Gewicht von 0 verbunden. Auf diese Weise tragen sie nicht zur Schnittbildung bei.

Das Ganze geschieht nicht zuletzt, um am Anfang durch Breitensuche bei automatisch generiertem Startknoten einen guten Endknoten finden zu können (siehe Abschnitt 4.2.1 Initialisierung).

4.3.2 Verbesserte *Ratio* Formel

Der Algorithmus berechnet bei einer Schnittgröße von 0, unabhängig von der momentanen Balancierung, immer ein `cut_ratio` von 0. Der Nenner `node_weight_on_sideA * node_weight_on_sideB` wird einfach nicht berücksichtigt. Bei nicht zusammenhängenden Graphen kann es nun passieren, daß mehrere Schnitte mit Wert 0 existieren, deren Balancierung der Algorithmus, zumindest so wie er in [WeiChe91] vorgestellt wird, am *Ratio* nicht unterscheiden kann. Wie man sich an einem Extrembeispiel, wie einem Graphen mit leerer Kantenmenge (nur isolierte Knoten), leicht vorstellen kann, werden auf diese Weise i. a. sehr unbalancierte Schnitte berechnet. Im Klartext heißt dies dann, daß evtl. ein Knoten auf der einen Seite liegen würde und die restlichen auf der anderen.

Das gleiche Phänomen tritt übrigens auch in Erscheinung, wenn ein Graph Kanten mit Gewicht 0 enthält und dadurch auch hier mehrere Schnitte mit `cutsize` = 0 auftreten.

Aus diesen Gründen wird eine leicht modifizierte Berechnungsformel für das *Ratio* vorgeschlagen:

```
                    cut_ratio :=
                  (cutsize + 1) /
(node_weight_on_sideA * node_weight_on_sideB)
```

Abbildung 4-18: Modifizierte *Ratio* Berechnungsformel

Durch die Addition von 1 im Zähler wird verhindert, daß im Zähler eine Null erscheint. Die Ordnung der Schnitte, wie sie durch *Ratio*-Berechnung mit der Formel aus Abbildung 4-5 und anschließende, aufsteigende Sortierung entsteht, wird durch diese modifizierte Version nicht zerstört. Man ist ja nicht direkt an den berechneten *Ratio*-Werten interessiert, sondern an dem Schnitt, den das minimale *Ratio* beschreibt.

Weil *Ratio-Cut* auf Graphen mit gleichverteilten Kanten mit der originalen Berechnungsformel aus Gründen, die in Abschnitt 4.1.2, insbesondere Abbildung 4-9 dargelegt sind, Schnitte mit sehr schlechter Balancierung errechnet (vgl. Anhang A.2.1), wird in dieser Implementierung tatsächlich folgende Formel verwendet:

```
                    cut_ratio :=
                (cutsize + number_of_nodes) /
(node_weight_on_sideA * node_weight_on_sideB)
```

Abbildung 4-19: Verbesserte *Ratio* Berechnungsformel

Neben verbesserter Balancierung (vgl. Anhang A.2.1) braucht man hier im Zähler auch nicht mehr 1 zu addieren, da `number_of_nodes` - außer beim leeren Graphen - immer größer als 0 ist.

Nicht zuletzt begründet sich diese Methode der Verbesserung durch praktische Tests u. a. im Anhang.

4.3.3 Mehrere Schnitte mit minimalem *Ratio*

Da auch hier in jeder Phase Protokoll über die jeweiligen *Ratios* bei versuchsweisen Knotentransfers geführt wird, um anschließend den besten Schnitt auswählen zu können, wird eine ähnliche Verbesserung, wie in Abschnitt 3.3.2.3 für den Fiduccia Mattheyses Algorithmus, eingeführt.

Der Unterschied dazu ist nur, daß hier nicht bei gleicher Schnittgröße, sondern bei gleichem `cut_ratio` der Schnitt mit der besseren Balancierung den Vortritt erhält.

4.3.4 Nachbearbeitung

Die bei der Vorverarbeitung 4.3.1 eingefügten künstlichen Kanten müssen nach dem Durchlauf des Algorithmus wieder entfernt werden. Dadurch wird der Originalgraph wieder exakt so hergestellt, wie ihn der Benutzer dem Algorithmus übergeben hat.

4.3.5 Ergebnisabfrage

Zusätzlich zu der Seite, auf der sich ein Knoten im Ergebnis befindet, kann auch die Größe des Schnitts, das *Ratio* dieses Schnitts und die jeweilige Summe über alle Gewichte von Knoten auf der Seite A bzw. B abgefragt werden.

Damit wird die qualitative Beurteilung der vom Algorithmus generierten Partitionierung etwas erleichtert.

Falls der Benutzer es wünscht, d. h. wenn er den entsprechenden Aufrufparameter setzt, werden auch Iteratoren bereitgestellt, um alle zum Schnitt gehörigen Kanten in einer Schleife durchlaufen zu können. Ähnliches gilt für Iteratoren, mit denen man jeweils alle Knoten auf einer Seite der Reihe nach abarbeiten kann.

4.4 Komplexität

Die Initialisierung, Abschnitt 4.2.1, und das Iterative Shiften, Abschnitt 4.2.2, können als Spezialfall des Gruppenaustausch aus Abschnitt 4.2.3 implementiert

werden. Diese beiden Phasen bewegen bei einem Durchlauf Knoten nur in eine Richtung, während der Gruppenaustausch flexibel genug ist, Knoten auf Seite A und B zu transferieren.
Der Gruppenaustausch basiert auf der gleichen Datenstruktur und im Prinzip auf den gleichen Mechanismen wie bei [FidMat82] und im obigen Kapitel 3 beschrieben. Die daraus resultierende Gesamtkomplexität mit Graphen ist O(|E|).
Um jeweils den nächsten Transferknoten auszuwählen, müssen hier jedoch immer beide *Bucket Lists* nach dem höchsten Knoten-*Ratio* durchsucht werden. Die Komplexität O(|E|) kann so mit $O\left(\sum_{i=1}^{|V|} i\right) = O\left(\frac{|V| * (|V| + 1)}{2}\right)$ leicht verletzt werden, wenn sich in diesen Listen die meisten Knoten des Graphen befinden, da insgesamt |V| viele Knoten zu transferieren sind. Theoretisch könnte man für diese Listen zwar eine *Bucketsort*-Strategie nach den *Ratio*-Werten verwenden, mit der das Ganze in O(|V|) realisierbar wäre. Tatsächlich wird das aber in dieser Implementierung, wie übrigens auch im Originalpaper [WeiChe91], offen gelassen und aus praktischen Gründen - „Overkill“ - nicht weiter verfolgt.
Die in der Initialisierung 4.2.1 notwendige Breitensuche für die automatische Bestimmung des Zielknotens läuft bekanntlich in O(|V| + |E|) und kann im Normalfall, etwa bei zusammenhängenden Graphen, auch durch O(|E|) abgeschätzt werden. Wenn `source_node` und `target_node` durch den Benutzer schon vorgegeben werden, fällt diese Traversierung des Graphen natürlich weg.
Die Tiefensuche, die unter Umständen gebraucht wird, um einen Graphen zusammenhängend zu machen, fällt auch aus dem gleichen Grund mit O(|V| + |E|) nicht aus dem O(|E|) Rahmen.
Wie schon in Abschnitt 3.4, wird auch hier davon ausgegangen, daß es sich bei `max_edge_weight` - gesehen über alle Eingabegraphen - um eine kleine Konstante handelt. Auch hier muß nämlich O(|E| * `max_edge_weight`) = O(|E|) gelten.

Die theoretische, lineare Laufzeit O(|E|) der eben vorgestellten Bipartitionierungs-Heuristik erreicht in etwa auch die Implementierung. Die Entscheidungen, die in Abschnitt 4.3 getroffen worden sind, beeinträchtigen auf keinen Fall das theoretische Laufzeitverhalten. Eine genaue experimentelle Analyse findet sich in Anhang A.

Für Hypergraphen würde die Laufzeit übrigens, wie schon in Abschnitt 3.4 für das Fiduccia Mattheyses Verfahren beschrieben, auch O(p) betragen. Dabei ist p wieder als Summe über die Anzahl der Elemente jeder Hyperkante aus H definiert.

4.5 Graphische Oberfläche

Wie schon das Verfahren von Fiduccia und Mattheyses ist auch die *Ratio Cut* Heuristik in GRAPHLET eingebunden worden, um auch diese Implementierung testen und graphisch präsentieren zu können.

Der Aufrufpunkt befindet sich wieder unter dem Menü „Partition“.

Eine initiale Partitionierung wird hier genauso automatisch berechnet wie Start- und Endknoten. Kein Knoten wird auf eine bestimmte Seite festgelegt. Auch hier wird die Knotenbeschriftung als `node_weight` und die Kantenbeschriftung als `edge_weight` benutzt. Bei leeren Beschriftungen werden sie standardmäßig mit 1 attributiert.

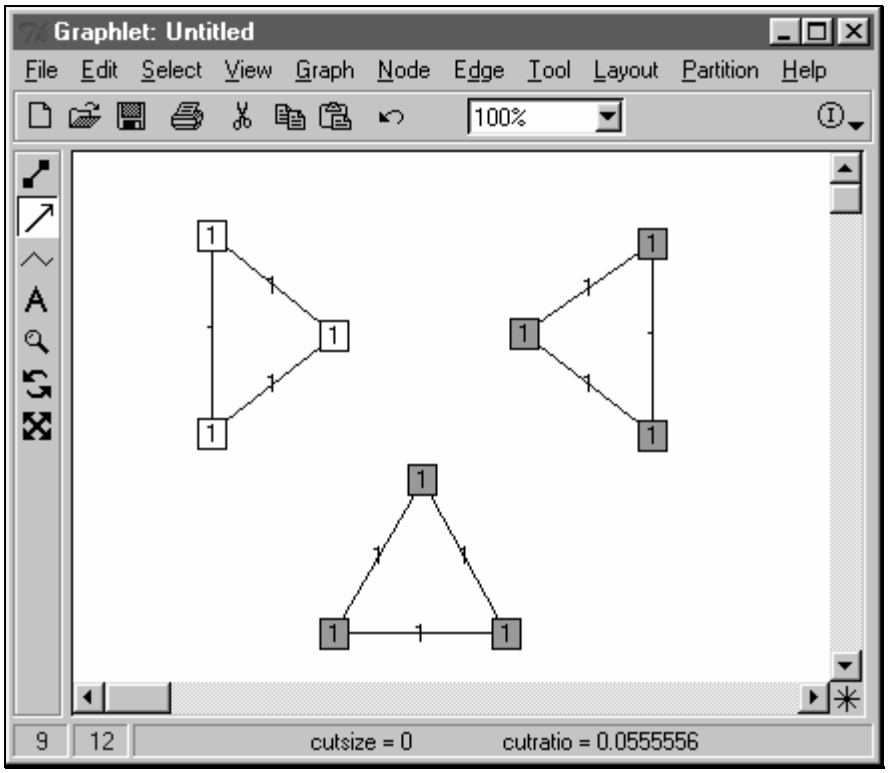

Abbildung 4-20: Graphlet 5.0.0 für MS-Windows mit integrierter *Ratio Cut* Heuristik

In der Statuszeile von GRAPHLET befinden sich Angaben über die Größe des berechneten Schnitts sowie das zugehörige *Ratio* (vgl. Abschnitt 4.3.2 Verbesserte *Ratio* Formel).

4.6 Mögliche Erweiterungen

Die Philosophie von *Ratio Cut* ist es, die natürlichen Cluster in einem Graphen zu finden. Der Algorithmus hat in Hinsicht auf die Größe der Ergebnisteilmengen freie Hand. Dennoch ist es manchmal notwenig, eine Größenbeschränkung vorzugeben, z. B. beim VLSI-Design aufgrund physikalischer Einschränkungen oder Kostenfaktoren.

Wie in [WeiChe91] vorgeschlagen, würde der Benutzer durch Eingabespezifikationen vorgeben, wie groß eine Teilmenge maximal sein darf.

Jedesmal wenn der originale Algorithmus angewendet wird, erhält man als Ergebnis zwei Teile A und B. Falls ein Teil größer sein sollte als die obere vom Benutzer gesetzte Grenze, werden die Knoten und Kanten der kleinen Seite temporär entfernt und der Algorithmus erneut auf den Rest angewendet. Dies wird so lange fortgeführt, bis alle Teilmengen kleiner oder gleich ihren Größenbegrenzungen sind.

Zum Schluß werden dann alle Teilmengen wieder zu zwei richtigen Seiten A und B zusammengesetzt. Dazu setzt man die größere Seite des letzten Partitionierungsvorganges als Seite A. Alles verbleibende wird zu Seite B zusammengefügt. Anschließend wird ein Durchlauf an *Left- / Right Shifting Operatoren* angewendet, um eine bestimmte Anzahl an Knoten von der größeren auf die kleinere Seite zu schaffen. Dabei wird unter Beachtung der Größenbeschränkungen das beste, d. h. das minimale, *Ratio* produziert.

Da so die Möglichkeiten, Knoten auf die andere Seite zu transferieren, durch die Größenvorgaben erheblich eingeschränkt werden, kann verständlicherweise die Qualität der Partitionierung darunter sehr leiden.

Bei d Algorithmusdurchläufen, um den Graphen in kleine Teile zu zerlegen, erhält man eine Zeitkomplexität von $O((d + 1) * |E|) = O(d * |E|)$. Glücklicherweise wurde ebenfalls in [WeiChe91] experimentell beobachtet, daß d dabei maximal 5 ist. Man kann also auch bei Größenbeschränkung der Partitionen von einer linearen $O(|E|)$ Komplexität ausgehen.

Kapitel 5

Erweiterungen zur Multiway Partitionierung

Es gibt mehrere heuristische Wege, um das MULTIWAY PARTITION Problem auf das BIPARTITION Problem zu reduzieren. Beide oben vorgestellten Algorithmen können dabei verwendet werden.

Ein in [Leng90] gemachter Vorschlag funktioniert wie folgt: Man startet mit einer anfänglich willkürlichen Aufteilung der Knotenmenge V in r disjunkte, annähernd gleich große Partitionen. Dann wird eine in dieser Arbeit vorgestellte Bi-Partitionierungs-Heuristik auf Paare von Partitionen angewendet. Jedes Paar, bei dem sich mindestens eine Partition während der zuletzt durchgeführten Bi-Partitionierung geändert hat, qualifiziert sich für eine erneute Anwendung des Verfahrens. Dies wird so lange fortgesetzt, bis Stabilität eintritt. Selbst wenn man den Vorgang bereits früher anhält, wird brauchbares Material geliefert. Die größten Kostenverbesserungen des Schnitts erhält man nämlich während den ersten Schritten. Dieses Verfahren funktioniert prinzipiell auch mit dem Kernighan Lin Algorithmus [KerLin70], allerdings mit der Einschränkung, daß alle Partitionen genau gleich groß sein müssen.

Eine etwas andere Vorgehensweise ist, zuerst den ganzen Graphen in zwei Teile und dann jeweils rekursiv Seite A und Seite B zu partitionieren. Damit wird so lange fortgefahren, bis entweder jede einzelne Partition ihre vorgegebene maximale Größe unterschritten hat oder eine vorher festgelegte Anzahl an Partitionen erreicht worden ist. Diesen Algorithmus nennt man *hierarchische Partitionierung*. Läßt man das Verfahren so lange laufen, bis die einzelnen Partitionen eine Größe im Bereich eines oder zumindest weniger Knoten erreichen, beträgt die Zeitkomplexität $O(|E| \log |E|)$.

Denkbar wäre auch, die so erhaltenen Blöcke zu einem Cluster zusammenzufügen, d. h. jede Partition stellt in einem Supergraphen einen

Superknoten dar. Das Gewicht dieses Knotens könnte die Summe aller Knotengewichte in diesem Cluster sein. Zwischen den Superknoten sind Kanten, die je nach dem, wieviele Kanten mit wieviel Gewicht zwischen den Clustern verlaufen, ein angepaßtes Kantengewicht erhalten. Auf dem so erhaltenen Supergraphen kann man dann, je nach Belieben, dieses Verfahren erneut anwenden. Diese Technik kann u. a. auch dazu benutzt werden, um die visuelle Komplexität von großen Graphen mit vielen Knoten stufenweise zu reduzieren.

Anzumerken wäre jedoch, daß aufgrund experimenteller Ergebnisse aus [Sanchis89] der uniforme Partitionierungsalgorithmus [Sanchis99] ohne Zeitbeschränkung meistens in der Lage ist, bessere Schnitte zu finden. Dies liegt offensichtlich daran, daß zwar die erste Bi-Partitionierung die Anzahl der Verbindungen zwischen den Seiten minimiert, gleichzeitig aber dazu tendiert, die Verbindungen innerhalb der beiden Seiten zu maximieren. Das macht es für nachfolgende, rekursive Partitionierungsaufrufe schwerer, darin wiederum einen guten Schnitt zu finden. Bei dem uniformen Partitionierungsalgorithmus [Sanchis99] werden hingegen bei jedem Iterationsschritt mögliche Transfers eines in Frage kommenden Knoten von seinem Herkunftsblock in alle anderen Blöcke global in Betracht gezogen und anschließend der Beste daraus ausgewählt.

Die Laufzeit von Sanchis' Algorithmus auf normalen Graphen beträgt O(l * m * b * (log b + m * l)) bzw. O(l * m * b * (log b + p + l)) in einer speichersparenden Version. Die Variable l bezeichnet dabei die Anzahl der Levels, also die Größe eines *Gain Vektors* (ähnlich `number_of_levels` in Abschnitt 3.6.2), m und p sind bei normalen Graphen gleich |E| zu setzen und b ist die Anzahl der Partitionen.

Wenn nur begrenzte Rechenzeit zur Verfügung steht, sind laut [Sanchis89] die hierarchischen Algorithmen jedoch meistens in der Lage, bessere Schnitte zu produzieren.

Kapitel 6

Bewertung und Ausblick

Ein Schwachpunkt von Fiduccia Mattheyses ist wohl der, daß die Qualität des Ergebnisses, d. h. die Größe des Schnitts und die resultierende Balancierung, sehr stark von der Qualität der initialen Zweiteilung abhängt. Deshalb erscheint es sinnvoll, den Algorithmus mehrmals auf jeweils verschiedenen initialen Knotenaufteilungen eines Graphen zu starten und anschließend das beste Ergebnis daraus auszuwählen.

Die maximale Anzahl an notwendigen Heuristik-Durchläufen, nämlich 7 (vgl. Abschnitt 3.4), konnte nicht bestätigt werden. Bei den umfangreichen Tests in Anhang A beträgt sie 13. Dabei handelt es sich aber erfreulicherweise immer noch um eine kleine Konstante, so daß die theoretische Linearlaufzeit O(|E|) dadurch nicht beeinträchtigt wird.

Ratio Cut findet in allen in A.2.3.1 durchgeführten Testläufen eine künstlich in den Graphen eingebaute „Sollbruchstelle“. Die Fiduccia Mattheyses Heuristik hingegegen wird dazu einfach zu stark von den maximalen Partitionsgrößen eingeengt. Außerdem liefert die *Ratio Cut* Heuristik in fast allen anderen Tests Schnitte einer kleineren Größe. Leider hängt auch hier die Qualität des Ergebnisses, vor allem die Balancierung, sehr stark von der initalen Wahl des Start- und Endknotens, also von der initialen Zweiteilung ab. Deshalb wird auch hier vorgeschlagen, den Algorithmus mehrmals auf jeweils verschiedene Knotenaufteilungen anzuwenden.

Normalerweise liegen die Laufzeiten beider Algorithmen etwa in der selben Größenordung. Für signifikant bessere Ergebnisse braucht aber *Ratio Cut* oft auch signifikant mehr an CPU-Zeit (vgl. z. B. Anhang A.2.2).

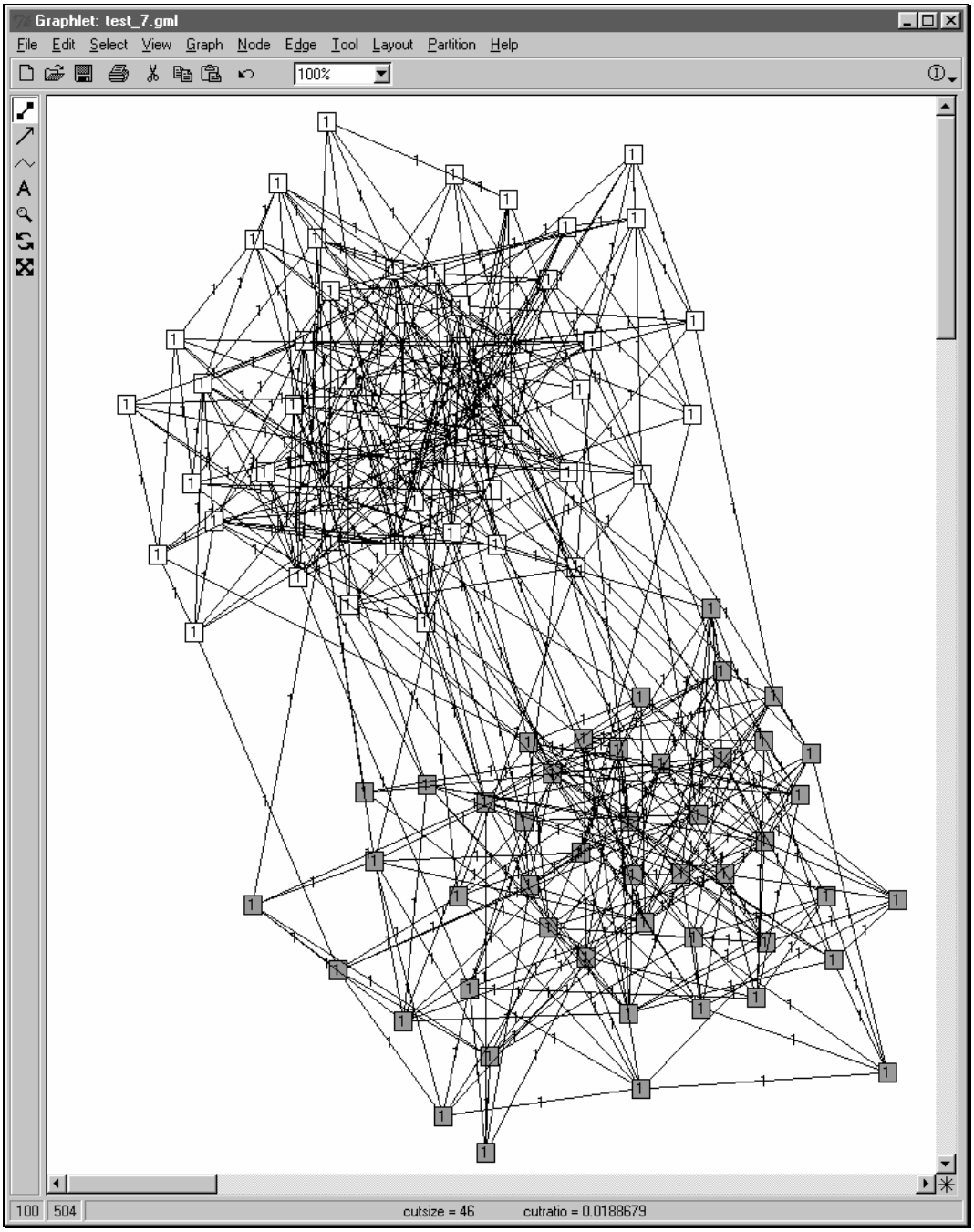

Abbildung 6-1: *Ratio Cut* auf einem Graphen mit 100 Knoten und 504 Kanten

Bei Graphen mit gleichmäßig verteilten Kanten stellt sich bei *Ratio Cut* mit original *Ratio* Formel allerdings generell eine Art Hilflosigkeit ein. Dort wird zwar im allgemeinen ein sehr guter Schnittwert ermittelt, was aber wiederum mit einer sehr schlechten Balancierung bezahlt werden muß. Meistens werden hier

sogar nur ein paar Knoten vom Rest getrennt. Bei diesen Graphen wären also in der Tat Größenbeschränkungen der einzelnen Partitionen notwendig. Doch wie groß soll man diese wählen? Der ganze Vorteil gegenüber Fiduccia und Mattheyses in der Freiheit der „uneingeschränkten“ Schnittwahl wäre so wieder verschwunden. Glücklicherweise kommen Graphen mit gleichverteilten Kanten bei den großen Anwendungsgebieten von *Ratio Cut* in der Praxis eigentlich nicht vor. Bei VLSI-Design z. B. stammen die zu bearbeitenden Graphen von einem Schaltplan ab und Schaltkreise bzw. Schaltnetze sind daher alles andere als gleichverteilt. Da die GTL-Implementierung mit einer verbesserten Berechnungsformel ausgestattet ist, wird sie davon nicht betroffen!

Ein großer Vorteil beider Algorithmen und auch ihrer hierarchischen Erweiterung zur MULTIWAY Partitionierung liegt in der Tatsache, daß man sie sehr leicht parallelisieren kann. Dabei können z. B. verschiedene Durchläufe auf verschiedenen Prozessoren eines Parallelrechners ausgeführt werden und anschließend kann dann aus diesen das beste Resultat ermittelt werden.

Die große Praxisrelevanz, gerade bei VLSI-Layout, läßt vermuten, daß sich künftig noch viele Forscher über Netzwerk- bzw. Graphenpartitionierung den Kopf zerbrechen werden. Noch nicht ausreichend beleuchtete Bereiche hierbei wären zum Beispiel Mechanismen, um vorher die Qualität des Ergebnisses festlegen oder zumindest voraussagen zu können. Auch Algorithmen, die den partitionierten Graphen anschließend graphisch irgendwie sinnvoll darstellen, wären von großem Nutzen. Alle Knoten, die zu einer Partition gehören, sollen dabei zeichnerisch so gruppiert werden, daß man ihre Zusammengehörigkeit visuell auch sofort erkennen kann.

Ein Denkanstoß für uniformes MULTIWAY bzw. FREE PARTITION wäre etwa die Minimierung einer Kostenfunktion wie $\frac{\texttt{cutsize} + \texttt{number\_of\_nodes}}{\prod_{i=1}^{k} V_i}$.

Dadurch könnte auch hier die Notwendigkeit fester Größenbeschränkungen der einzelnen Teilmengen aufgeweicht werden und eine etwas natürlichere Aufteilung gefunden werden.

Anhang A

Performance der Implementierung

Da die in Kapitel 3 und Kapitel 4 vorgestellten Algorithmen für GTL implementiert worden sind, werden umfangreiche Tests damit durchgeführt, um ihre Effizienz aufzuzeigen und nicht zuletzt, um möglichst viele Fehlerquellen zu beseitigen.

Als Testplattform dient ein Intel Pentium PC mit 200MHz Prozessor und 128MB RAM Speicher unter dem Betriebssystem Linux mit Kernel 2.2.5.

A.1 Laufzeit in CPU-Sekunden

Für die Tests werden sog. einfache Graphen verwendet, die mit dem eigens im Rahmen dieser Arbeit für GTL erstellten Graphengenerator, namens `graph_generator`, erzeugt worden sind. Es handelt sich dabei um ungerichtete Graphen, die keine Schleifen und keine parallelen Kanten[9] enthalten.

Ein derartiger Graph mit |V| Knoten kann daher maximal $\frac{|V|*(|V|-1)}{2}$ viele Kanten aufweisen. Man nennt ihn dann einen vollständigen Graphen. Umgekehrt beinhaltet dieser vollständige Graph mit |E| Kanten $\frac{1}{2}+\sqrt{\frac{1}{4}+2|E|}$ Knoten.

[9] Kanten nennt man parallel, wenn deren Endknoten übereinstimmen. Manchmal werden sie auch als Mehrfachkanten bezeichnet. Bei gerichteten Graphen muß außerdem die Richtung übereinstimmen; falls dies nicht zutrifft, heißen sie antiparallel.

Zur Ermittlung der Laufzeit werden vorher beschriebene Testgraphen mit unterschiedlicher Größe verwendet. Da die Komplexität der zu testenden Algorithmen linear in der Anzahl der Kanten steigt, werden Testgraphen der Reihe nach mit linear steigender Kantenanzahl generiert. Dabei wurde mit 0 Kanten begonnen und bis 10.000 um jeweils 50 Kanten erhöht.

Um nicht nur vollständige Graphen zu betrachten und gleichzeitig zu verhindern, daß |V| durch Erhöhung von |E| quadratisch wächst, wird die Anzahl der Knoten eines Testgraphens auf $3 * \left\lceil \frac{1}{2} + \sqrt{\frac{1}{4} + 2|E|} \right\rceil$ festgesetzt. Knoten- und Kantengewichte sind dabei alle mit 1 parameterisiert.

Die Kanten werden dabei im Graphen nach einer Gleichverteilung gesetzt. Es werden 2 disjunkte Knoten zufällig aus V ausgewählt. Existiert zwischen beiden noch keine Kante, wird eine dementsprechende erzeugt. Dies wird so lange fortgeführt, bis |E| viele Kanten generiert worden sind.

In der nachfolgenden Abbildung wird das Laufzeitverhalten der beiden Algorithmen graphisch dargestellt, auf der x-Achse ist die Kantenzahl und auf der y-Achse die benötigte Durchlaufzeit in Sekunden aufgetragen.

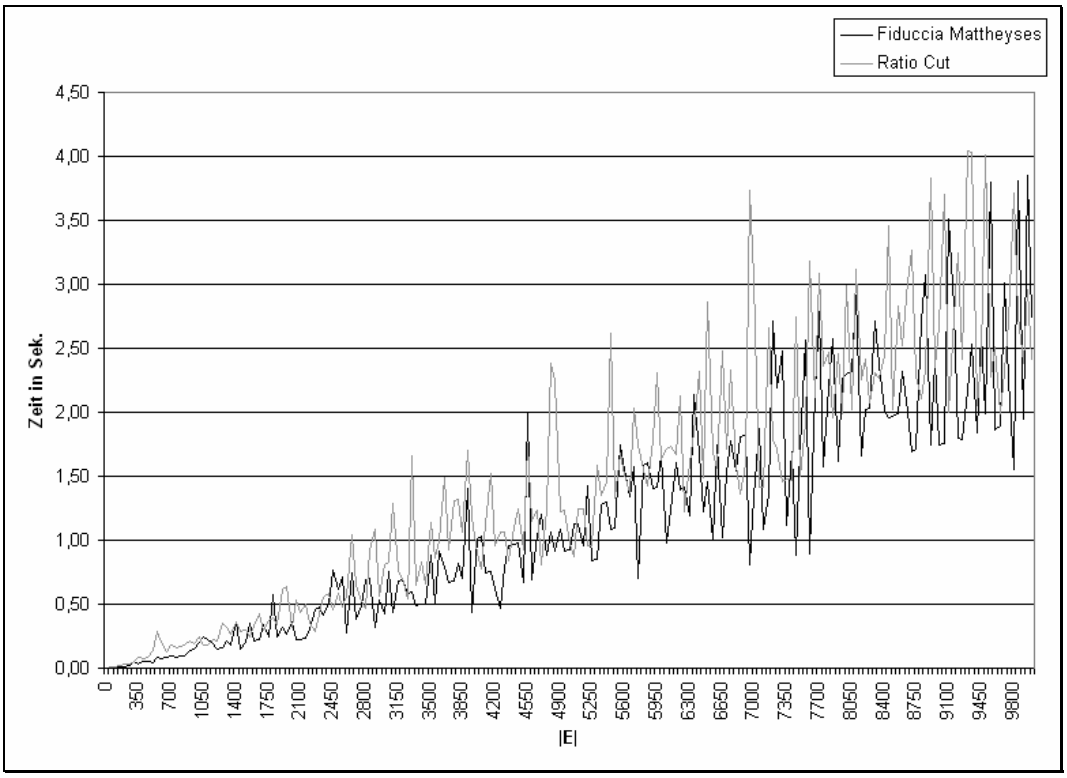

Abbildung A-1: Laufzeitverhalten

Auffallend dabei ist die Tatsache, daß die Laufzeiten von *Ratio Cut* fast immer etwas länger als bei Fiduccia und Mattheyses sind. Das mag vielleicht daran liegen, daß *Ratio Cut* 3 Phasen während eines Durchganges abläuft und der Algorithmus von Fiduccia und Mattheyses eigentlich nur 1 Phase. Außerdem ist der zu durchsuchende Lösungsraum bei *Ratio Cut* aufgrund fehlender, fester Größenbeschränkungen normalerweise größer als der von Fiduccia und Mattheyses. Insgesamt ist aber bei beiden Algorithmen ein klarer Trend zur linearen Laufzeit O(|E|) erkennbar.

A.2 Vergleich

Wenn schon die Laufzeiten in ihrer Größenordnung nicht ganz übereinstimmen, gibt es wohl auch Unterschiede in der Qualität der generierten Schnitte.

In allen Tests werden Initialpartitionierungen automatisch durch den jeweiligen Algorithmus ausgerechnet. Kein Knoten wird von vornherein auf eine bestimmte Seite festgesetzt. Bei *Ratio Cut* wird `source_node`, ebenso wie `target_node`, automatisch bestimmt.

A.2.1 Testgraphen mit gleichverteilten Kanten

Folgende Tabelle A-1 zeigt Testläufe von Fiduccia Mattheyses auf Graphen mit gleichverteilten Kanten, wie in Abschnitt A.1 definiert. Dabei wurde mit der leeren Knotenmenge begonnen und bis einschließlich 1.000 Knoten schrittweise um 50 erhöht. Bis 5.000 folgen dann 500er Schritte. Die Kantenzahl |E| ist dabei jeweils das dreifache der Knotenzahl |V|. Knoten- und Kantengewichte wurden hier ebenfalls auf 1 gesetzt.

Für jeden so erhaltenen Graphen wird der Fiduccia Mattheyses Algorithmus jeweils 5mal gestartet. Die nachfolgende Tabelle A-1 zeigt davon das beste Ergebnis mit der kleinsten Schnittgröße. Unter Variation findet sich jeweils die maximale Abweichung der Schnittgröße innerhalb der 5 Durchläufe, d. h. addiert man sie zur kleinsten, so erhält man dabei die größte, gefundene Schnittgröße. Die Spalte „Durchläufe“ bezeichnet dabei die Anzahl, wie oft sich die Heuristik von selbst neu startet, um die Schnittgröße abermals zu verkleinern.

|V|	|E| = 3|V|	Schnittgr.	Var.	Durchläufe	Gewichtsverteilung	Rechenzeit
0	0	0	0	0	0:0	0,00
50	150	37	0	2	26:24	0,01
100	300	72	0	3	50:50	0,04
150	450	109	8	5	76:74	0,09
200	600	140	0	3	100:100	0,08
250	750	168	11	4	124:126	0,13
300	900	206	7	4	150:150	0,16
350	1050	243	19	3	175:175	0,15
400	1200	262	14	5	200:200	0,26
450	1350	295	21	5	225:225	0,30
500	1500	332	15	4	250:250	0,28
550	1650	366	28	5	275:275	0,38
600	1800	400	19	6	300:300	0,48
650	1950	418	21	3	325:325	0,30
700	2100	458	31	5	350:350	0,50
750	2250	494	39	3	375:375	0,36
800	2400	530	27	4	400:400	0,48
850	2550	553	31	6	425:425	0,72
900	2700	588	37	7	450:450	0,87
950	2850	624	35	5	474:476	0,70
1000	3000	638	31	8	500:500	1,09
1500	4500	1005	51	7	750:750	1,41
2000	6000	1338	22	5	1000:1000	1,53
2500	7500	1658	41	7	1250:1250	2,61
3000	9000	2008	42	11	1500:1500	4,77
3500	10500	2321	49	12	1750:1750	6,12
4000	12000	2669	88	8	2000:2000	5,19
4500	13500	2993	39	6	2250:2250	4,38
5000	15000	3330	137	13	2500:2500	9,64

Tabelle A-1: Testläufe mit Fiduccia Mattheyses auf uniformen Graphen, wobei |E| = 3* |V|

Auf derartigen Graphen generiert dieses Verfahren sehr große Schnitte, da es aufgrund des Balancekriteriums aus Abbildung 3-5 höchstens um das maximale Knotengewicht, also 1, von der exakten Zweiteilung abweichen darf.

Auf den gleichen Graphen wurde das *Ratio Cut* Verfahren ebenfalls 5mal gestartet und wieder das beste Ergebnis, d. h. das mit dem kleinsten `cut_ratio`, aufgelistet.

Folgende Tabelle zeigt Läufe mit der etwas modifizierten *Ratio* Berechnungsformel aus Abbildung 4-18.

|V|	|E| = 3|V|	Ratio	Schnittgr.	Var.	Gewichtsverteilung	Rechenzeit
0	0	0,00000000	0	0	0:0	0,00
50	150	0,04964539	6	0	3:47	0,03
100	300	0,01374570	3	0	97:3	0,04
150	450	0,00671141	0	0	149:1	0,07
200	600	0,00757576	2	0	198:2	0,09
250	750	0,00401606	0	0	249:1	0,13
300	900	0,00422297	4	0	296:4	0,15
350	1050	0,00143678	0	0	348:2	0,16
400	1200	0,00251256	1	0	398:2	0,21
450	1350	0,00298285	3	0	447:3	0,25
500	1500	0,00100402	0	0	498:2	0,23
550	1650	0,00201363	12	4	538:12	0,32
600	1800	0,00083612	0	0	598:2	0,30
650	1950	0,00077160	0	0	648:2	0,41
700	2100	0,00191296	3	0	697:3	0,43
750	2250	0,00133511	0	0	749:1	0,40
800	2400	0,00125156	0	0	799:1	0,52
850	2550	0,00019747	0	0	844:6	0,46
900	2700	0,00037161	0	0	897:3	0,49
950	2850	0,00052743	0	0	948:2	0,65
1000	3000	0,00020101	0	0	995:5	0,56
1500	4500	0,00033378	0	0	1498:2	0,90
2000	6000	0,00010025	0	0	1995:5	1,33
2500	7500	0,00008016	0	0	2495:5	1,78
3000	9000	0,00003344	0	0	2990:10	2,52
3500	10500	0,00002049	0	0	3486:14	3,04
4000	12000	0,00002784	0	0	3991:9	4,12
4500	13500	0,00002474	0	0	4491:9	4,49
5000	15000	0,00001822	0	0	4989:11	5,36

Tabelle A-2: Testläufe mit modifiziertem *Ratio Cut* auf uniformen Graphen, wobei |V| = 3 * |E|

Die hier verwendeten Graphen werden mit steigender Knotenzahl immer dünner, weshalb man auch erklären kann, daß ab |V| = 750 nur noch Schnitte der Größe 0 am Rand berechnet werden.

Es fällt ebenfalls auf, daß *Ratio Cut* ohne Größenbeschränkungen der einzelnen Partitionen oder verbesserte *Ratio* Formel sehr unbalancierte Schnitte auf Graphen mit einer gleichverteilten Kantenmenge generiert. Für diese Art von Graphen könnte man entweder Größenbeschränkungen einführen (die Frage hierbei ist nur

wann und wie restriktiv das wirklich nötig ist) oder die *Ratio* Formel, wie hier geschehen, nochmals etwas verbessern (vgl. Abbildung 4-19):

|V|	|E| = 3|V|	Ratio	Schnittgr.	Var.	Gewichtsverteilung	Rechenzeit
0	0	0,00000000	0	0	0:0	0,00
50	150	0,13760000	36	0	25:25	0,03
100	300	0,07058344	73	0	57:43	0,05
150	450	0,04480000	102	0	75:75	0,14
200	600	0,03345649	126	2	116:84	0,12
250	750	0,02718777	166	6	143:107	0,17
300	900	0,02214286	196	4	140:160	0,26
350	1050	0,01933287	229	4	201:149	0,35
400	1200	0,01662404	250	8	230:170	0,42
450	1350	0,01492747	294	29	253:197	0,26
500	1500	0,01332340	323	11	277:223	0,62
550	1650	0,01226729	360	20	237:313	0,59
600	1800	0,01108204	391	30	324:276	0,70
650	1950	0,01033861	422	28	281:369	1,15
700	2100	0,00952663	459	21	321:379	0,67
750	2250	0,00887597	481	27	419:331	0,74
800	2400	0,00836957	509	24	460:340	1,03
850	2550	0,00781423	558	10	446:404	1,74
900	2700	0,00733129	574	24	488:412	1,43
950	2850	0,00696842	622	8	469:481	2,42
1000	3000	0,00668508	648	40	441:559	2,14
1500	4500	0,00444784	972	49	668:832	3,99
2000	6000	0,00335459	1319	28	897:1103	5,76
2500	7500	0,00267949	1593	0	1063:1437	5,41
3000	9000	0,00222820	1995	62	1409:1591	12,65
3500	10500	0,00189938	2256	51	1571:1929	13,80
4000	12000	0,00167636	2635	63	2205:1795	11,85
4500	13500	0,00147728	2834	51	1937:2563	16,29
5000	15000	0,00134694	3233	37	2871:2129	12,52

Tabelle A-3: Testläufe mit verbessertem *Ratio Cut* auf uniformen Graphen, wobei |V| = 3 * |E|

Die Schnittgrößen hier entsprechen fast denen von Fiduccia und Mattheyses, da bei diesen Testgraphen einfach keine natürlichen Einschnitte vorhanden sind, die der Algorithmus wahrnehmen könnte. Die Rechenzeiten sind jetzt höher, da der zu durchsuchende Lösungsraum größer geworden ist. Dies spiegelt sich in einer größeren Variation der Schnittgröße während mehrerer Durchläufe auf einem Graphen wider.

Aufgrund dieser wirklich einschneidenden Verbesserung in der Balancierung, wird nachfolgend nur noch diese verbesserte Formel verwendet.

A.2.2 Benchmark Testgraphen

Die folgenden Testläufe werden auf Graphen aus der Benchmarkbibliothek [Alpert99] durchgeführt. Darunter befinden sich auch einige Graphen, mit denen Leistungsmessungen in [WeiChe91] erfolgt sind. Da es sich dabei im Original um Hypergraphen HG = (U, H) handelt, müssen sie vorher in normale Graphen G = (V, E) im GML-Format[10] konvertiert werden.

Die Knoten werden dabei eins zu eins übernommen. Für jede Hyperkante wird ein neuer Knoten, ein sog. Hyperknoten, eingefügt. Dieser wird dann mit allen zu der Hyperkante inzidenten Knoten durch ungerichtete Kanten direkt verbunden. Aus den Endknoten jeder Hyperkante wird also eine sog. Clique[11] geformt.

Für alle Knoten aus V und Kanten aus E wird ein einheitliches Gewicht von 1 verwendet.

Es läßt sich leider nicht vermeiden, daß die natürlichen Schnitte, die es in HG gibt, eventuell in dem transformierten G etwas verwischt sind. Man erhält ein anderes Kostenmodell. Der Schnitt durch eine Hyperkante $h \in H$ kostet einmalig 1 und trennt dabei $|h|$ Knoten auf. Um diese Knoten im transferierten Graphen aufzutrennen, müssen dafür $|h|$ Kanten aus E mit deren Kostenbeitrag $|h| * 1$ durchgeschnitten werden. Außerdem gilt $|V| = |U| + |H|$.

Die nachfolgende Tabelle zeigt das beste Ergebnis von jeweils 5 Fiduccia Mattheyses Durchläufen.

Benchm.	**|V|**	**|E|**	**Schnittgr.**	**Durchl.**	**Gewichtsverteilung**	**Rechenzeit**
qgr8	35	46	3	2	18:17	0,00
3k5	48	66	8	1	24:24	0,01
IC67	205	474	36	3	103:102	0,07
fract	296	462	26	6	148:148	0,12
IC116	444	876	29	4	222:222	0,17
IC151	570	987	62	5	285:285	0,21
balu	1536	2697	184	4	768:768	0,56

[10] Die GRAPH MODELLING LANGUAGE (GML) ist ein portables Dateiformat für Graphen. Es wurde am Lehrstuhl Prof. Dr. Brandenburg an der Universität Passau als Teil von GRAPHLET entwickelt. GTL ist natürlich ebenfalls mit GML kompatibel. Für weitere Informationen siehe [Hims96].

[11] In einer Clique sind alle Knoten adjazent.

primary1	1735	2908	118	4	868:867	0,58
bm1	1785	2910	91	7	893:892	0,90
test04	3173	5975	409	7	1587:1586	1,98
test03	3225	5807	293	4	1613:1612	1,22
test02	3383	6134	411	11	1936:1936	3,01
struct	3872	5471	163	13	1936:1936	3,20
primary2	6043	11219	259	6	3022:3021	3,24
19ks	6126	10547	402	12	3063:3063	5,97
biomed	12256	21040	1378	7	6128:6128	7,70

Tabelle A-4: Fiduccia und Mattheyses auf Benchmark Graphen

Die besten Ergebnisse von jeweils 5 *Ratio Cut* Durchläufen sind anschließend tabellarisch aufgeführt.

Benchm.	**|V|**	**|E|**	**Ratio**	**Schnittgr.**	**Gewichtsvert.**	**Rechenzeit**
qgr8	35	46	0,12418301	3	18:17	0,01
3k5	48	66	0,09391304	6	23:25	0,02
IC67	205	474	0,02293927	36	103:102	0,09
fract	296	462	0,01517465	32	165:131	0,11
IC116	444	876	0,00959744	29	222:222	0,24
IC151	570	987	0,00771848	56	296:274	0,42
balu	1536	2697	0,00289128	164	725:811	3,30
primary1	1735	2908	0,00243924	100	851:884	2,82
bm1	1785	2910	0,00236769	101	892:893	3,52
test04	3173	5975	0,00143055	360	1804:1369	9,08
test03	3225	5807	0,00132877	230	1613:1612	16,10
test02	3383	6134	0,00133688	428	1589:1794	14,04
struct	3872	5471	0,00107127	142	1970:1902	6,65
primary2	6043	11219	0,00069775	327	3014:3029	35,19
19ks	6126	10547	0,00071353	567	3020:3106	48,20
biomed	12256	21040	0,00036099	1300	6128:6128	107,23

Tabelle A-5: *Ratio Cut* auf Benchmark Graphen

Da die Testgraphen von realen Schaltplänen abstammen und diese alles andere als gleichverteilte Kanten enthalten, werden jetzt auch durchaus brauchbare Ergebnisse geliefert.

Auffallend ist die Tatsache, daß *Ratio Cut* bei den letzten Testgraphen signifikant mehr Rechenzeit verbraucht wie Fiduccia Mattheyses. Die Schnittgrößen und die Balancierung beider Verfahren sind außer bei 19ks in etwa äquivalent. Dieses Phänomen läßt sich auf Graphen dieser Größe leider nur schwer nachvollziehen.

A.2.3 Testgraphen mit „Perforationen“

Um zu testen, wie gut die Algorithmen in den Graphen vorhandene „Perforationen“ als Schnittmöglichkeit wahrnehmen, wird jetzt bei den Testgraphen eine künstliche Clusterung eingebaut.

A.2.3.1 Zwei Cluster

Der Reihe nach werden Graphen mit einer Knotenanzahl |V| von 0 bis 750 in 50er Schritten generiert. Die Knoten sind dabei zufällig in zwei Mengen, sog. Blöcke bzw. Cluster, aufgeteilt. Zufällig heißt hier, daß die Blockgröße 60% von |V| nicht übersteigen darf. Zwischen disjunkten Knoten innerhalb dieser dichteren Cluster werden Kanten mit einer inneren Kantenwahrscheinlichkeit von 20% gezogen, zwischen Knoten aus verschiedenen Blöcken mit einer äußeren Kantenwahrscheinlichkeit von 2%. So werden also wieder Graphen ohne Schleifen und parallele Kanten mit dem GTL-Tool `graph_generator` erzeugt, die beiden Verfahren die Möglichkeit bieten, einen relativ guten Schnitt in der „Mitte“ des Graphen zu finden.

Die Knoten und Kanten sind wieder einheitlich mit Gewicht 1 attributiert.

Auch hier werden die Algorithmen auf dem gleichen Graphen jeweils 5mal gestartet und anschließend die besten Ergebnisse abgedruckt.

|V|	|E|	|OE|	Schnittgr.	Durchl.	Gewichtsverteilung	Rechenzeit
0	0	0	0	0	0:0	0,00
50	138	16	15	1	24:26	0,01
100	530	44	53	1	51:49	0,03
150	1273	94	208	3	74:76	0,16
200	2242	213	332	2	99:101	0,22
250	3349	298	396	3	124:126	0,43
300	5011	405	737	4	149:151	0,83
350	6704	600	639	1	176:174	0,48
400	8699	766	807	1	201:199	0,63
450	11254	1024	1341	2	224:226	1,23
500	14112	1241	2785	3	249:251	2,10
550	16408	1541	1885	2	276:274	1,84
600	20054	1742	3410	2	299:301	2,31
650	23800	2066	4429	7	326:324	7,10
700	27034	2464	2925	2	349:351	3,11
750	32073	2805	6863	5	374:376	7,45

Tabelle A-6: Fiduccia Mattheyses auf Graphen mit zwei Clustern

Die „äußeren Kanten“ werden dabei in der Menge OE zusammengefaßt.

|V|	|E|	|OE|	Ratio	Schnittgr.	Gewichtsverteilung	Rechenzeit
0	0	0	0,00000000	0	0:0	0,00
50	138	16	0,10305958	14	27:23	0,02
100	530	44	0,05797101	44	54:46	0,08
150	1273	94	0,04494382	94	61:89	0,15
200	2242	213	0,04190341	213	112:88	0,27
250	3349	298	0,03521625	298	133:117	0,43
300	5011	405	0,03184426	405	169:131	0,65
350	6704	600	0,03102953	600	178:172	1,05
400	8699	766	0,02915656	766	197:203	1,41
450	11254	1024	0,02919910	1024	237:213	1,86
500	14112	1241	0,02878876	1241	205:295	1,97
550	16408	1541	0,02769390	1541	286:264	2,35
600	20054	1742	0,02651751	1742	341:259	2,86
650	23800	2066	0,02636279	2066	37:274	3,50
700	27034	2464	0,02585411	2464	339:361	4,69
750	32073	2805	0,02627572	2805	448:302	4,62

Tabelle A-7: *Ratio Cut* auf Graphen mit zwei Clustern

Wie man leicht ablesen kann, stellt sich in diesem Szenario *Ratio Cut* eindeutig als das bessere Verfahren heraus, da es trotz einer guten Balancierung und ähnlicher Laufzeit viel kleinere Schnitte findet. In allen Fällen wird die mit den Graphen angebotene Schnittmöglichkeit voll wahrgenommen, |OE| ist jeweils gleich der Schnittgröße (einmal ist sie sogar kleiner).

Für die nächsten Testläufe werden ebenfalls oben beschriebene Graphen generiert. Der Unterschied ist nur, daß jeder Knoten nun ein zufälliges Gewicht von mindestens 1 und höchstens 10% von |V| hat. Weiterhin hat jede Kante ein zufälliges Gewicht aus {1, ..., 5}.

|V|	|E|	OE-Ge.	Schnittgr.	Durchl.	Gewichtsverteilung	Rechenzeit
0	0	0	0	0	0:0	0,00
50	113	21	30	3	84:77	0,01
100	506	114	198	2	273:256	0,05
150	1256	353	422	2	561:579	0,12
200	2189	581	949	2	1092:1055	0,21
250	3410	906	1060	2	1650:1700	0,34
300	4952	1378	2063	2	2260:2320	0,52
350	6725	1741	2398	2	3235:3170	0,73

400	9145	2384	5110	4	4176:4099	1,64
450	10897	2834	5058	2	5282:5209	1,22
500	13649	3753	3753	1	6554:6588	1,04
550	17153	4618	10265	4	7890:7787	3,25
600	20011	5169	8395	1	9254:9163	1,57
650	23058	6313	7441	2	10836:10708	2,67
700	26784	7300	7525	2	12362:12223	3,11
750	30875	8222	10137	1	13497:13646	2,49

Tabelle A-8: Fiduccia Mattheyses auf Graphen mit zwei Clustern und unterschiedlicher Gewichtung

Die Spalte „OE-Gewicht“ enthält jeweils die Summe über die Gewichte aller „äußeren Kanten“.

|V|	|E|	OE-Ge.	Ratio	Schnittgr.	Gewichtsvert.	Rechenzeit
0	0	0	0,00000000	0	0:0	0,00
50	113	21	0,01132376	21	95:66	0,02
100	506	114	0,00313995	114	307:222	0,06
150	1256	353	0,00155832	353	616:524	0,15
200	2189	581	0,00068936	581	1213:934	0,27
250	3410	906	0,00041314	906	1762:1588	0,53
300	4952	1378	0,00032595	1378	2600:1980	0,66
350	6725	1741	0,00020572	1741	3505:2900	1,12
400	9145	2384	0,00017007	2384	3272:5003	1,23
450	10897	2834	0,00012167	2834	5969:4522	1,52
500	13649	3753	0,00009850	3753	6554:6588	2,30
550	17153	4618	0,00008806	4618	6178:9499	2,43
600	20011	5169	0,00006864	5169	10074:8343	3,46
650	23058	6313	0,00006013	6313	11263:10281	3,37
700	26784	7300	0,00005295	7300	12154:12431	4,69
750	30875	8222	0,00004883	8222	14236:12907	5,42

Tabelle A-9: *Ratio Cut* auf Graphen mit zwei Clustern und unterschiedlicher Gewichtung

Im Gegensatz zum Fiduccia Mattheyses Algorithmus findet *Ratio Cut* auch unter diesen Testbedingungen die vorgegebene, künstliche „Perforation“. Die Summen über die Gewichte der „äußeren Kanten“ entsprechen in allen Fällen genau den Schnittgrößen der Ergebnispartitionierungen.

Dennoch liefert das Verfahren von Fiduccia Mattheyses ebenfalls gute Ergebnisse. Es darf jetzt schließlich mit jeder Partitionsgröße um 0,1 * |V| von $\frac{|V|}{2}$ abweichen. Die so erhaltenen, größeren Variationsmöglichkeiten spiegeln

sich gegenüber den Testläufen mit einheitlichen Knotengewichten in einer verhältnismäßig kleineren Schnittgröße wider. Es bietet sich daher für Graphen mit einheitlichen Gewichten an, ein lockereres Balancekriterium als in [Leng90] oder [FidMat82] vorgeschlagen wird zu verwenden.

A.2.3.2 Fünf Cluster

Die nachfolgend in Tabelle A-10 bzw. Tabelle A-11 verwendeten, erneut mit `graph_generator` erzeugten Graphen enthalten jetzt fünf ungefähr gleich große Cluster. Die innere Kantenwahrscheinlichkeit beträgt wieder 20%, die äußere 2%. Sie werden mit einem Knoten- und Kantengewicht von einheitlich 1 jeweils wieder 5mal durchlaufen.

|V|	|E|	Schnittgröße	Durchläufe	Gewichtsverteilung	Rechenzeit
0	0	0	0	0:0	0,00
50	67	9	2	25:25	0,01
100	280	37	2	51:49	0,03
150	626	116	2	76:74	0,06
200	1102	223	2	100:100	0,11
250	1717	335	6	126:124	0,38
300	2489	535	3	149:151	0,32
350	3405	708	3	176:174	0,45
400	4507	947	5	199:201	0,92
450	5687	1242	4	225:225	1,01
500	6985	1487	2	249:251	0,76
550	8392	1826	5	274:276	1,81
600	10109	2226	2	300:300	1,13
650	11796	2618	3	325:325	1,78
700	13664	2993	4	349:351	2,72
750	15690	3431	4	375:375	3,19

Tabelle A-10: Fiduccia Mattheyses auf Graphen mit fünf Clustern

Auch hier stellt sich *Ratio Cut* eindeutig als die bessere Alternative heraus.

|V|	|E|	Ratio	Schnittgröße	Gewichtsverteilung	Rechenzeit
0	0	0,00000000	0	0:0	0,00
50	67	0,08800000	5	25:25	0,02
100	280	0,05434783	35	54:46	0,05
150	626	0,04747067	109	88:62	0,11
200	1102	0,03931943	179	119:81	0,20

250	1717	0,03493333	274	150:100	0,31
300	2489	0,03351955	426	121:179	0,38
350	3405	0,03135498	574	209:141	0,73
400	4507	0,02994798	750	160:240	0,72
450	5687	0,02973251	995	180:270	0,94
500	6985	0,02801667	1181	300:200	1,15
550	8392	0,02705234	1414	220:330	1,71
600	10109	0,02730324	1759	240:360	2,10
650	11796	0,02687377	2075	390:260	2,45
700	13664	0,02585884	2341	420:280	2,85
750	15690	0,02508889	2637	300:450	3,45

Tabelle A-11: *Ratio Cut* auf Graphen mit fünf Clustern

Die *Ratio Cut* Heuristik hat einfach die Möglichkeit zwischen den Blöcken durchzuschneiden, hingegen Fiduccia und Mattheyses aufgrund der vorgegebenen Balancekriterien nicht. Dadurch findet *Ratio Cut* in der Regel die Schnitte, die in die Testgraphen künstlich eingebaut worden sind. So werden fast immer 3 Blöcke auf einer Seite angesiedelt, die verbleibenden 2 auf der anderen.

Für das letzte Testszenario werden oben beschriebene Graphen erneut generiert. Der Unterschied ist nur, daß jeder Knoten nun ein zufälliges Gewicht von mindestens 1 und höchstens 10% von |V| hat. Zusätzlich hat jede Kante ein zufälliges Gewicht aus dem Bereich von 1 bis 5.

|V|	**|E|**	**max K-Gew.**	**Schnittgr.**	**Durchl.**	**Gewichtsvert.**	**Rechenzeit**
0	0	0	0	0	0:0	0,00
50	70	5	23	1	78:88	0,00
100	269	10	138	4	276:296	0,05
150	645	15	310	5	654:645	0,13
200	1181	20	717	4	992:1014	0,18
250	1654	25	972	3	1467:1507	0,22
300	2515	30	1493	2	2308:2272	0,25
350	3424	35	2113	4	3177:3243	0,59
400	4423	40	2768	4	4143:4092	0,78
450	5565	45	3492	5	5312:5236	1,20
500	6967	50	4375	7	6470:6529	2,00
550	8214	55	5271	2	8036:7960	0,91
600	10006	60	6301	2	8884:8995	1,14
650	11814	65	7752	5	10536:10595	2,63
700	13729	70	8915	3	12300:12440	2,10
750	15903	75	10454	3	14141:14028	2,45

Tabelle A-12: Fiduccia Mattheyses auf Graphen mit fünf Clustern und unterschiedlichen Gewichtsattributen

Fiduccia und Mattheyses schneiden mindestens einen Block in der Mitte durch. Die Schnittgröße ist aber hier mit unterschiedlicher Knotengewichtung durchaus brauchbar.

|V|	|E|	max K-Gew.	Ratio	Schnittgr.	Gewichtsvert.	Rechenzeit
0	0	0	0,00000000	0	0:0	0,00
50	70	5	0,00915033	13	81:85	0,02
100	269	10	0,00292027	128	347:225	0,05
150	645	15	0,00100237	265	738:561	0,11
200	1181	20	0,00086966	617	1261:745	0,24
250	1654	25	0,00052513	874	1221:1753	0,30
300	2515	30	0,00030309	1237	1874:2706	0,52
350	3424	35	0,00021379	1767	2576:3844	0,55
400	4423	40	0,00016068	2230	4883:3352	0,71
450	5565	45	0,00012184	2817	4273:6275	1,12
500	6967	50	0,00009787	3532	7522:5477	1,16
550	8214	55	0,00007888	4280	9651:6345	1,41
600	10006	60	0,00007322	5120	10279:7600	1,74
650	11814	65	0,00006546	6298	8224:12907	2,04
700	13729	70	0,00005094	4700	19227:5513	2,37
750	15903	75	0,00004688	8138	11121:17048	2,79

Tabelle A-13: *Ratio Cut* auf Graphen mit fünf Clustern und unterschiedlichen Gewichtsattributen

Ratio Cut findet auch hier in etwa der gleichen Zeit eindeutig die besseren Ergebnisse.

Literaturverzeichnis

[AlpKah95] C. J. Alpert, A. B. Kahng, Recent directions in netlist partitioning: a survey, INTEGRATION, the VLSI journal 19, Elsevier Science B. V. 1995, pp. 1 - 81

[Alpert96] C. J. Alpert, Fiduccia-Mattheyses Code in C, http://vlsicad.cs.ucla.edu/~cheese/codes.html#fmcode, 1996

[AlHuKa98] C. J. Alpert, J. H. Huang, A. B. Kahng, Multilevel Circuit Partitioning, IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN VOL. 17 NO. 8, 1998, pp. 655 - 667

[Alpert99] C. J. Alpert, Benchmarks, http://vlsicad.cs.ucla.edu/~cheese/benchmarks.html, 1999

[BaChFa] S. Basagni, I. Chlamtac, A. Faragó, A Generalized Clustering Algorithm for Peer-to-Peer Networks, The University of Texas at Dallas, Submitted to Discrete Applied Mathematics

[BerGol99] J. W. Berry, M. K. Goldberg, Path optimization for graph partitioning problems, Discrete Applied Mathematics 90, Elsevier Science B. V. 1999, pp. 27 - 50

[Bran99] F. J. Brandenburg, Vorlesungsskript Effiziente Algorithmen, Universität Passau, Wintersemester 1999/2000

[Budd98] T. A. Budd, Data Structures in C++ Using the Standard Template Library, Addison Wesley, Reading Massachusetts [e. a]., 1998

[BuiMoo96] T. N. Bui and B. R. Moon, Genetic Algorithm and Graph Partitioning, IEEE TRANSACTIONS ON COMPUTERS VOL. 45 NO. 7, 1996, pp. 841 - 855

[CaKaMa99] A. E. Caldwell, A. B. Kahng and I. L. Markov, Design and Implementation of the Fiduccia-Mattheyses Heuristic for VLSI Netlist Partitioning, Algorithm Engineering and Experimentation, Lecture Notes in Computer Science 1619, 1999, pp. 177 - 193

[CheWei91] C. K. Cheng and Y. C. Wei, An Improved Two-Way Partitioning Algorithm With Stable Performance, IEEE Transactions on Computer-Aided Design Vol. 10 No. 12, 1991, pp. 1502 - 1511

[ClaHol94] J. Clark, D. A. Holton, Graphentheorie Grundlagen und Anwendungen, Spektrum Verlag, Heidelberg 1994

[ENRS99] G. Even, J. Noar, S. Rao and B.Schieber, Fast Approximate Graph Partitioning Algorithms, SIAM JOURNAL ON. COMPUTING, Vol. 28, No. 6, 1999, pp. 2187 - 2214

[FidMat82] C. M. Fiduccia, R. M. Mattheyses, A Linear-Time Heuristic for Improving Network Partitions, ACM/IEEE Proceedings of the 19th Design Automation Conference, 1982, pp. 175 - 181

[FoPiRa99] M. Forster, A. Pick, M. Raitner, The Graph Template Library (GTL) Manual, http://www.infosun.fmi.uni-passau.de/GTL/, Passau 1999

[GarJoh79] M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman and Company, New York 1979

[HagKah91] L. W. Hagen and A. B. Kahng, Fast Spectral Methods for Ratio Cut Partitioning and Clustering, Proceedings International Conference on Computer-Aided Design, Santa Clara 1991, pp. 10 - 13

[HaHuKa95] L. W. Hagen, D. J. Huang and A. B. Kahng, On Implementation Choices for Iterative Improvement Partitioning Algorithms, Proc. European Design Automation Conference, 1995, pp. 144 - 149

[HauBor97] S. Hauck and G. Borriello, An Evaluation of Bipartitioning Techniques, IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS VOL. 16 NO. 8, 1997, pp. 849 - 866

[HopUll94] J. E. Hopcroft, J. D. Ullman, Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie, 3. Auflage, Addison Wesley, Bonn [e. a.] 1994

[Hims96] M. Himsolt, GML: Graph Modelling Language, http://www.fmi.uni-passau.de/archive/archive.theory/ftp/graphlet/GML.ps.gz, 1996

[JAMcGS89] D. Johnson, C. Aragon, L. McGeoch, C. Schevon, Optimization By Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning, Perations Research Vol. 37 No. 6, 1989, pp. 865 - 892

[KerLin70] B. W. Kernighan and S. Lin, An Efficient Heuristic Procedure for Partitioning Graphs, Bell Systems Technical Journal, 49(2), 1997, pp. 291 - 307

[Krishn84] B. Krishnamurthy, An Improved Min-Cut Algorithm For Partitioning VLSI Networks, IEEE Transactions on Computers, C-33(5), 1984, pp. 438 - 446

[Leng90] T. Lengauer, Combinatorial Algorithms for Integrated Curcuit Layout, B. G. Teubner, Stuttgart 1990, pp. 251 - 301

[Lipp91] S. B. Lippman, C++ Einführung und Leitfaden – 2. erw. Auflage, Addison Wesley, Bonn [e. a.] 1991

[MehNäh99] K. Mehlhorn and S. Näher, The LEDA Platform of Combinatorial and Geometric Computing, http://www.mpi-sb.mpg.de/~mehlhorn/LEDAbook.html, Cambridge University Press, 1999

[Moni98] B. Monien, Graph Partitioning, Universität Paderborn, http://www.uni-paderborn.de/fachbereich/AG/monien/RESEARCH/PART/, 1998

[MNSU99] K. Mehlhorn, S. Näher, M. Seel, C. Uhrig, The LEDA User Manual, Version 3.8, Max-Planck-Institut für Informatik, http://www.mpi-sb.mpg.de/LEDA, 1999

[NaKaIb99] H. Nagamochi, S. Katayama and T. Ibaraki, A Faster Algorithm for Computing Minimum 5-Way and 6-Way Cuts in Graphs, Lecture Notes in Computer Science 1627, 1999, pp. 164 - 173

[OttWid96] T. Ottmann / P. Widmayer, Algorithmen und Datenstrukturen, 3. Auflage, Spektrum Verlag, Heidelberg 1996

[PreDie96] R. Preis, R. Diekmann, The PARTY Partitioning - Library User Guide - Version 1.1, ftp://ftp.uni-paderborn.de/doc/techreports/Informatik/tr-rsfb-96-024.psZ, 1996

[Rait99] M. Raitner, Effiziente Algorithmen zum Test der Planarität von Graphen, Diplomarbeit an der Universität Passau, 1999

[RoxSen97] T. Roxborough and A. Sen, Graph Clustering Using Multiway Ratio Cut (Software Demonstration), Lecture Notes in Computer Science 1353, 1997, pp. 291 - 296

[Sanchis89] L. A. Sanchis, Multiple-Way Network Partitioning, IEEE TRANSACTIONS ON COMPUTERS VOL. 38 NO 1, 1989, pp. 62 - 81

[Sanchis99] L. A. Sanchis, Source-Code zu [Sanchis89] in C, auf Nachfrage von ihr persönlich erhalten, September 1999

[SGI98] Silicon Graphics Computer Systems Inc., Standard Template Library Programmer's Guide, http://www.sgi.com, 1998

[Stroob99] D. Stroobandt, Pin Balancing in Ratio Cut Partitioning, Proceedings of the Swiss Conference of CAD/CAM, http://www.elis.rug.ac.be/~dstr/dstr.html, 1999

[WeiChe91] Y. C. Wei and C. K. Cheng, Ratio Cut Partitioning for Hierarchical Designs, IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN VOL. 10 NO.7, 1991, pp. 911 - 921

Eidesstattliche Erklärung

Hiermit versichere ich, daß diese Diplomarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt wurde und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind. Diese Diplomarbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Salzweg, den 16. März 2000

Christian Bachmaier