

Diplomarbeit zum Thema

Cache-effiziente Speicherlayouts für Graphen

zur Erlangung des akademischen Grades
Diplom-Informatiker

vorgelegt dem
Hochschulprüfungsamt der
Universität Trier

Holger Hink
Matrikelnummer: 631954
19. Juli 2007

Erstgutachter: Prof. Dr. Stefan Näher
Zweitgutachter: Prof. Dr. Peter Sturm

Inhaltsverzeichnis

Erklärung zur Diplomarbeit	1
1 Einleitung und Motivation	2
1.1 Cache-Effizienz auf Anwendungsentwicklungsebene	3
1.1.1 Beispiel: Feldzugriffe	4
1.2 Prefetching-Strategien	5
1.2.1 Prefetch On Miss	5
1.2.2 Tagged Prefetch	6
1.3 Zum Thema: Graphen	6
1.3.1 Ausgangssituation	6
1.3.2 Zielsetzung	7
1.4 Überblick	7
2 Das Kantensummen-Problem	9
2.1 Ein Beispiel	9
2.2 Formale Darstellung	10
2.3 Vereinfachung	12
2.4 Verwandte Probleme	14
3 Der Bandbreiten-Graph	15
3.1 Die Adjazenzmatrix	15
3.2 Vereinfachung: Ein zweites Kostenmaß	18
3.3 Der abstrakte Datentyp	20
3.3.1 Komponenten des Bandbreiten-Graphen	21
3.3.2 Operationen auf dem Bandbreiten-Graphen	21
3.4 Der konkrete Datentyp	21
4 NP-Vollständigkeit	23
4.1 Das Minimum Linear Arrangement Problem	23
4.2 NP-Vollständigkeit von MIBS	24
4.2.1 Idee	25
4.2.2 Formaler Beweis	30
4.2.3 Ein komplexeres Beispiel	35
5 Algorithmen	39
5.1 Interne Kantenordnung	39

Inhaltsverzeichnis

5.2	Greedy-Algorithmus	41
5.3	Iteration nach dem arithmetischen Mittel	43
5.4	Dynamische Programmierung	45
6	Experimente	52
6.1	Partiell vollständige Max-Flow-Probleme	52
6.2	Genrmf Generator	55
6.3	Bewertung des Greedy-Algorithmus	57
6.4	Iteration nach dem arithmetischen Mittel	58
6.5	Fazit	58
7	Zusammenfassung und Ausblick	59
	Literaturverzeichnis	60

Erklärung zur Diplomarbeit

Hiermit erkläre ich, dass ich die Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe. Die Diplomarbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher nicht veröffentlicht.

Trier, den 19. Juli 2007

HOLGER HINK

1 Einleitung und Motivation

Aufgrund der immer größer werdenden Diskrepanz zwischen Prozessorgeschwindigkeiten auf der einen, und Speicherzugriffszeiten auf der anderen Seite, gewinnt eine effiziente Cache-Ausnutzung immer mehr an Bedeutung. So verdoppelt sich die Prozessorgeschwindigkeit nach dem Moore'schen Gesetz etwa alle 18 Monate, während der Geschwindigkeitszuwachs bei Hauptspeichern mit nur 7% pro Jahr weit zurück liegt [3]. Abbildung 1.1 aus [5] zeigt die Entwicklung dieser Lücke von 1980 bis 2000.

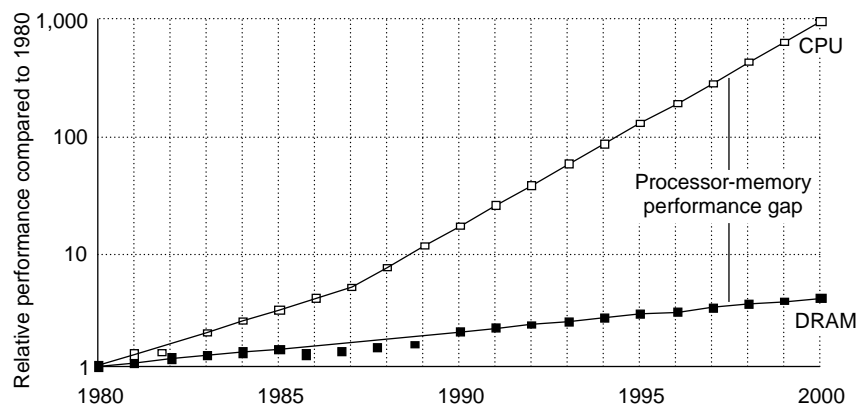


Abbildung 1.1: Prozessor-Hauptspeicher Geschwindigkeitslücke

Hierdurch wird deutlich, wie wichtig Caches sowohl im Software- als auch im Hardware-Design sind. Ziel ist es nun, die Anzahl der Cache-Fehlschläge zu minimieren, so dass nur möglichst wenig Speicherzugriffe aus dem langsamen Hauptspeicher bedient werden müssen. Prinzipiell gibt es drei verschiedene Ebenen, auf denen hier angesetzt werden kann [4].

Als unterste Ebene ist hierbei die Hardware zu nennen. So führen größere Caches sowie erhöhte Assoziativität im Allgemeinen auch zu einer erhöhten Trefferrate, jedoch auch zu höheren Kosten und konstruktionsbedingten langsameren Zugriffszeiten.

Der nächste Ansatzpunkt ist auf der Compiler-Ebene zu finden. So könnten „wohldurchdachte“ Platzierungen der Daten im Hauptspeicher, sowie eine „geschickte“ Umordnung der Ausführungsreihenfolge der Befehle zu einer erhöhten Ausnutzung der Referenzlokalität führen - sowohl in zeitlicher, als auch in räumlicher Hinsicht. Da dies bei einem Compiler automatisch geschehen muss und nicht auf spezielle Probleme bzw. Algorithmen begrenzt werden kann, erreicht man hier schnell die Grenze des Realisierbaren. Jedoch gibt es auch hier Ansätze, wie beispielsweise das „Loop Tiling“. Die Kernidee

ist hier, eine Schleife, welche auf einem großen Datenbereich arbeitet, derart in kleinere Schleifen aufzuteilen, dass möglichst viele Speicherzugriffe aus dem Cache bedient werden können. Für weitergehende Informationen sei der Leser auf [2] verwiesen.

Somit sind wir auf der dritten Ebene, welche sich an den Anwendungsentwickler richtet. Im Gegensatz zu einem Compiler hat dieser - zumindest im Idealfall - eine viel tiefer gehende Kenntnis über sein Programm, und damit auch weitergehende Möglichkeiten der „Umgestaltung“. Da dies auch die Stelle ist an der diese Arbeit ansetzt, wollen wir uns dieser nun etwas ausführlicher zuwenden.

1.1 Cache-Effizienz auf Anwendungsentwicklungsebene

Wie schon angedeutet, gibt es zwei verschiedene Ansatzpunkte zur Erhöhung der Cache-Trefferrate. Der erste ist gegeben durch die zeitliche Referenzlokalität und der hiermit verbundenen *LRU-Verdrängungsstrategie* der Caches. Vereinfacht dargestellt ist das Ziel hierbei, zwischen zwei Speicherzugriffen auf die gleiche Zelle A nur so viele paarweise verschiedene abweichende Speicherzugriffe zuzulassen, wie der Cache noch zu speichern vermag. Bei einer erneuten Anfrage der Zelle A kann diese aus dem Cache bedient werden. Das dies nicht global für alle möglichen Speicherzugriffe geschehen kann, ist offensichtlich. Allerdings lassen sich durch lokale Änderungen oftmals erhebliche Leistungssteigerungen erreichen. Hierzu sei beispielhaft auf die Arbeit „Lists Revisited: Cache Conscious STL Lists“ [7] verwiesen. Hier werden STL-Listen unter Einbeziehungen der Cache-Effizienz neu implementiert, was gegenüber vergleichbaren LEDA oder GCC Listen zu einer 5-10 fachen Geschwindigkeitssteigerung beim Traversieren und einer 3-5 fachen Steigerung beim Sortieren führt.

Der zweite Ansatz basiert auf einer weiteren Cache-Eigenschaft, dem sogenannten *Prefetching*. Zur Erklärung gehen wir kurz auf die Cache- bzw. Hauptspeicherorganisation ein. Für weitergehende Informationen sei beispielhaft auf das Werk „Computer Architecture. A Quantitative Approach“ [12] verwiesen. Sowohl der Cache als auch der Hauptspeicher sind in Blöcke fester Größe aufgeteilt. Im Zusammenhang von Caches redet man auch von (Cache-)Zeilen. Typische Größen sind 64 oder 128 Byte. Wird nun ein Datum von der CPU angefordert, welches nicht im Cache vorgehalten ist, so wird nicht nur dieses einzelne Datum, sondern der ganze Block in den Cache geladen. Dieses Vorgehen begründet sich auf zwei Feststellungen. Zum einen ist die räumliche Referenzlokalität eine inhärente Programmeigenschaft (man denke nur an sequentielle Feldzugriffe oder an den Programmcode selbst), so dass nachfolgende Speicherzugriffe mit einer gewissen Wahrscheinlichkeit aus dem Cache bedient werden können. Zum anderen kriegt man dieses erweiterte Einlesen aufgrund der Rechnerarchitektur quasi geschenkt. An einem Beispiel soll nun verdeutlicht werden, was dieses Prefetching im Extremfall bewirken kann.

1.1.1 Beispiel: Feldzugriffe

Wir wollen nun ein einfaches Programm schreiben, an welchem sich die potentiellen Zugewinne an der Cache-Trefferquote und der hiermit verbundenen gesamten Leistungssteigerung durch Ausnutzung des Prefetchings erkennen lassen. Hierzu initialisieren wir ein Integer-Feld `data` der Größe `sz` mit den ersten `sz - 1` natürlichen Zahlen. Hierüber bilden wir nun die Summe. In einem ersten Versuch durch einen sequentiellen Lauf über das Feld, anschließend durch einen randomisierten. Dies realisieren wir durch folgende C++-Codezeile:

```
for (int i = 0; i < sz; i++) sum += data[access[i]];
```

`access` ist hierbei ebenfalls ein Integer-Feld der Größe `sz` und enthält entweder fortlaufend oder in einer zufälligen Permutation die Zahlen 0 bis $n - 1$. Abbildung 1.2 verdeutlicht dieses Vorgehen.

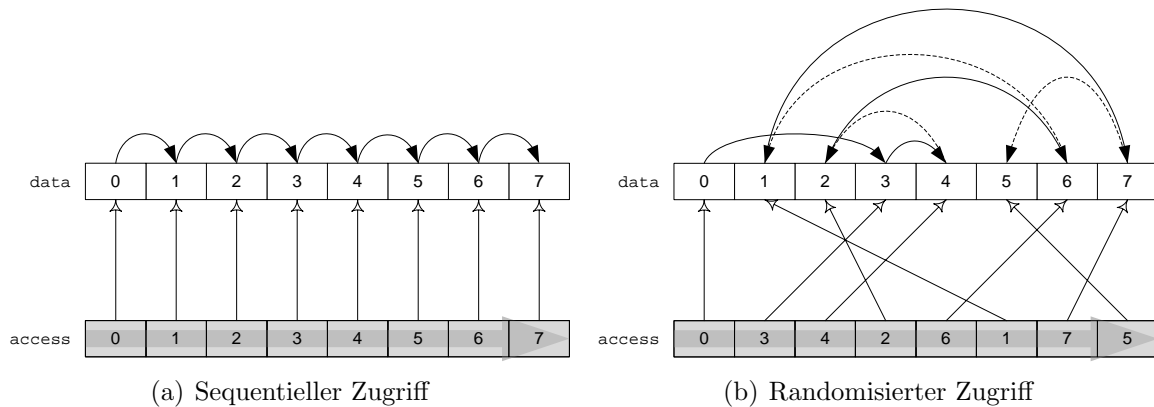


Abbildung 1.2: Auslesen eines Feldes

Bevor wir das Experiment starten, wollen wir kurz unsere Erwartungen darlegen. Die Größe einer Cache-Zeile auf der Zielplattform beträgt 64 Byte, kann also 8 Integer-Werte (a 4 Byte) umfassen. Bei einem Cache-Fehlschlag wird stets die komplette Zeile gefüllt, womit bei der sequentiellen Variante bereits die nächsten 7 benötigten Zahlen im Cache zur Verfügung stehen. Bei der randomisierten Variante ist dies (höchstwahrscheinlich) nicht der Fall, es ist also bei jedem Zugriff mit einem Cache-Fehlschlag zu rechnen. Die Fehlschlag-Rate erhöht sich also um den Faktor 8. Nun besteht das Programm aber eben nicht nur aus diesen Feldzugriffen, welche auch aus dem Cache bedient nicht völlig umsonst sind, weshalb die Verlangsamung etwas niedriger ausfallen sollte.

Tatsächlich ergibt sich jedoch bei der randomisierten Variante auf der getesteten Zielplattform eine Verlangsamung um einen Faktor größer als 10. Somit reicht unser oben beschriebener Prefetch-Ansatz als Erklärung nicht aus, und es muss weitere Caching-Strategien geben, die zusätzliche Verbesserungen erzielen. Im nächsten Abschnitt wollen wir ein Verfahren vorstellen, mit dem sich diese Lücke schließen lässt.

Trotz allem hat dieses Beispiel gezeigt, wie stark der Einfluß des Prefetchings von Caches sein kann. Zugegebenermaßen ist es sehr konstruiert, und hat aus zweierlei Gründen nicht allzu viel praktische Relevanz. Zum einen wird dieses Programm extrem (und unrealistisch) stark durch den Feldzugriff dominiert, so dass hier die Cache-Fehlschlagrate deutlich durchschlägt. Zum anderen hat man es in der Praxis meist mit komplexeren Datenstrukturen als einfachen (Integer-)Feldern zu tun. So auch in dieser Arbeit, wie der Titel bereits andeutet.

1.2 Prefetching-Strategien

Wie im letzten Abschnitt festgestellt, muss es Prefetching-Strategien geben, die über das Vorauslesen einzelner Cachezeilen hinausgehen. Wir wollen hier kurz eine Strategie vorstellen, die leicht (in Hardware) zu implementieren ist, und als *One Block Lookahead*-Strategie (OBL) bezeichnet wird. Sie gehört zur Familie der *Sequential Prefetching*-Verfahren.

Die Kernidee besteht darin, den Block $b + 1$ schon vor auszulesen, falls ein bestimmter Zugriff auf Block b statt findet. Bevor wir die Zugriffsart näher charakterisieren, wollen wir uns kurz überlegen, worin der Unterschied dazu besteht, einfach die Blockgröße zu verdoppeln. Zum einen können wir das Vorauslesen von einer bestimmten Zugriffsart abhängig machen. Zum anderen wirkt sich aber auch die LRU-Verdrängungsstrategie, wie sie üblicherweise bei Caches Anwendung findet, anders aus. Stellen wir uns hierzu einen Block vor, welcher ein häufig referenziertes Wort enthält. Der Rest dieses Blockes wird jedoch überhaupt nicht benötigt. Die Verdopplung der Blockgröße führt nun dazu, dass der gesamte Block ständig im Cache vorgehalten wird, obwohl nur dieses eine Wort häufig benötigt wird. Bei zwei kleinen Blöcken jedoch kann hier unterschieden werden. Enthält der eine das häufig referenzierte Wort komplett, so kann der andere verdrängt werden.

Nun wollen wir kurz zwei Varianten vorstellen, welche aus zwei verschiedenen Zugriffsarten resultieren.

1.2.1 Prefetch On Miss

Bei dieser Art des OBL wird der Block $b + 1$ genau dann vorausgelesen, wenn ein Zugriff auf Block b zu einem Fehlschlag führt. Falls der Block $b + 1$ schon im Cache vorgehalten ist, so wird hierfür kein Hauptspeicherzugriff initiiert. Offensichtlich nachteilig an diesem Verfahren ist, dass einem Vorauslesen zwangsläufig ein Fehlschlag vorangegangen sein muss. Hieraus resultiert die Idee des zweiten Verfahrens.

1.2.2 Tagged Prefetch

Beim sogenannten *Tagged Prefetch* ist mit jedem Speicherblock ein Markierungsbit assoziiert. Zu Beginn ist keines dieser Bits gesetzt. Bei einem Fehlschlag wird nun genau wie beim *Prefetch On Miss* der Folgeblock in den Cache geladen. Gleichzeitig wird aber auch das Markierungsbit (des Folgeblocks) gesetzt. Wird nun ein Wort aus einem solchen markierten Block angefordert, so wird auch hier der nachfolgende Block in den Cache geladen und das Bit wieder gelöscht. Dieses Bit dient also der Feststellung, ob ein vorausgelesener Block bereits referenziert wurde. Handelt es sich um einen Erstzugriff, so wird auch in diesem Falle (neben dem eines Cache-Fehlschlag) ein weiterer Block vorausgelesen.

Untersuchungen haben gezeigt, dass *Tagged Prefetch* die Cache-Fehlschlagrate um 50 bis 90% reduzieren kann, während *Prefetch On Miss* nur etwa halb so gut ist. Für weitere Verfahren und Ergebnisse sei auf das Paper „Data Prefetch Mechanisms“ [17] verwiesen.

1.3 Zum Thema: Graphen

Nach dieser allgemeinen Einführung wollen wir uns nun dem Thema dieser Arbeit nähern, den Graphen. Zunächst soll dargelegt werden, von welchen Voraussetzungen bzw. Grundlagen wir hier ausgehen. Hierbei soll insbesondere die speicherinterne Repräsentation von Graphen erläutert werden. Dies geschieht anhand des LEDA-Graphdatentyps. Anschließend beenden wir dieses Kapitel mit der Frage, was wir denn überhaupt erreichen wollen.

1.3.1 Ausgangssituation

Als erstes wollen wir nun kurz darlegen, wie ein LEDA-Graph denn überhaupt im Speicher abgelegt wird. Im wesentlichen werden hier die Knoten und die Kanten in zwei Listen gespeichert. Für jeden Knoten werden nun neben den Knoten-Informationen noch zwei Listen mit Verweisen auf eingehende bzw. ausgehende Kanten gespeichert. Die Kanten selbst werden in der Reihenfolge der Erzeugung im Speicher abgelegt. In den meisten Fällen werden zuerst die Knoten erzeugt, anschließend für jeden Knoten die ausgehenden Kanten eingefügt. Dies ist insbesondere der Fall, wenn ein Graph aus einem dimacs-File¹ generiert wird. Dies hat zur Folge, dass ausgehende Kanten kompakt im Speicher lie-

¹dimacs ist ein ascii-basiertes Datei-Format zur Speicherung von Netzwerken für Fluß-Probleme, siehe <http://dimacs.rutgers.edu/Challenges/>

gen, d.h. dass es zwischen aufeinanderfolgenden ausgehenden Kanten keine Lücken gibt. Sollen nun die ausgehenden Kanten eines Knotens durchlaufen werden, so finden im Speicher keinerlei Sprünge statt. In Bezug auf das Caching-Verhalten ist dieses Verhalten offensichtlich optimal.

Anders sieht es hingegen bei den eingehenden Kanten aus. Hier ist eine solche Kompaktheit im Allgemeinen nicht gegeben.

1.3.2 Zielsetzung

Ziel dieser Arbeit ist es nun, eben diese Sprünge der eingehenden Kanten mit dem Ziel eines besseren Caching-Verhaltens zu verändern. Wir wollen also eine Anordnung der Kanten finden, die bzgl. des Caching-Verhaltens möglichst effizient ist. Hierzu bieten sich grundlegend zwei verschiedene Ansatzmöglichkeiten an. Wir könnten die Kompaktheit der ausgehenden Kanten aufgeben, und somit jede mögliche Anordnung zulassen. Nun müßten wir aber nicht nur einen Effizienzgewinn bei den eingehenden Kanten erreichen, sondern zusätzlich noch den Verlust bei den ausgehenden Kanten ausgleichen.

In dieser Arbeit verfolgen wir daher den zweiten Ansatz, bei dem wir die Kompaktheitseigenschaft ausgehender Kanten aufrecht erhalten. Durch diese Einschränkung ist es uns im wesentlichen nur möglich ganze Kantenblöcke, also jeweils alle ausgehenden Kanten eines Knotens simultan, zu „verschieben“. Was dies genau bedeutet, wird in Kapitel 2 erörtert.

1.4 Überblick

Nach dieser informellen Einführung werden wir das Problem in Kapitel 2 zunächst formal erfassen. Basierend auf dieser formalen Darstellung entwickeln wir ein Gütemaß, nach welchem es später zu optimieren gilt. Um einen leichteren Zugang zu dem Problem zu erhalten, werden wir es in zwei Teilprobleme zerlegen, welche wir dann separat zu lösen versuchen.

In Kapitel 3 stellen wir eine zweite Sichtweise auf unser Problem vor. Wir starten mit der Adjazenzmatrix-Darstellung eines Graphen, und entwickeln darauf basierend schließlich einen Datentypen, der als Grundlage für unsere Algorithmen und damit auch ihrer Implementierung dient.

Kapitel 4 stellt dann insofern einen wesentlichen Teil dieser Arbeit dar, als dass er uns eine Rechtfertigung dafür liefert, dass wir keinen optimalen Algorithmus entwickeln werden. Dies hätte nämlich zur Folge, dass die beiden Komplexitätsklassen P und NP zusammen fallen würden, was nach allgemeiner Auffassung als „höchstgradig unwahrscheinlich“ angesehen wird.

Wir müssen uns also in Kapitel 5 mit Näherungslösungen zufrieden geben, zumindest

wenn wir uns nicht auf kleinste Graphen (der Größenordnung bis etwa 20 Knoten) beschränken wollen. Es stellt sich zwangsläufig die Frage, wie wir diese Lösungen überhaupt beurteilen können. Da wir unsere Ergebnisse aus einsichtigen Gründen nicht mit dem tatsächlichen Optimum vergleichen können, erscheint dies zunächst gar nicht möglich. Um zumindest einen Eindruck zu gewinnen, wenden wir zwei Strategien an. Zum einen machen wir uns doch noch auf die Suche nach optimalen Lösungen, die wir zumindest für kleine Graphen mittels vollständiger Suche durchaus finden können. Zum anderen werden wir speziell generierte Netzwerke heranziehen, die uns sowohl in einer sehr guten, als auch in einer sehr schlechten Version erstellt werden können.

Anschließend führen wir in Kapitel 6 Experimente mit den zuvor entwickelten Algorithmen durch. Wir untersuchen für verschiedene Graphen bzw. Netzwerke, inwieweit sie sich in Hinblick auf unser Gütemaß verbessern lassen. Spezielle (optimierte) Max-Flow-Netzwerke dienen schließlich als Eingabe eines Max-Flow-Algorithmus. In der Hoffnung auf verbesserte Laufzeiten versuchen wir hiermit wieder den Zusammenhang zu der eigentlichen Cache-Problematik herzustellen.

Kapitel 7 schließt die Arbeit mit einer Zusammenfassung und der Benennung weiterer Aspekte dieser Problematik ab.

2 Das Kantensummen-Problem

Wie bereits in der Einführung angekündigt, wollen wir uns dem Problem jetzt von der formalen Seite nähern. Fassen wir noch einmal kurz das Wichtigste zusammen. Wir haben einen Graphen im Speicher liegen, die ausgehenden Kanten der einzelnen Knoten jeweils kompakt, die eingehenden willkürlich angeordnet. Unser Ziel ist es nun, die Kanten unter Beibehaltung der ausgehenden Kompaktheit bezüglich eines besseren Caching-Verhaltens umzuordnen. Wir werden also irgendwie versuchen müssen, die Sprünge im Speicher nach irgendwelchen Kriterien zu minimieren. Eine präzisere Zielsetzung erfolgt nach der formalen Darstellung. Abschließen wollen wir das Kapitel mit einer Übersicht über verwandte Graph-Probleme.

2.1 Ein Beispiel

Wir wollen uns zunächst das Problem an einem konkreten Graphen anschauen. Hierzu versehen wir die Kanten mit den jeweiligen Adressen im Speicher, wie in Abbildung 2.1(a) dargestellt.

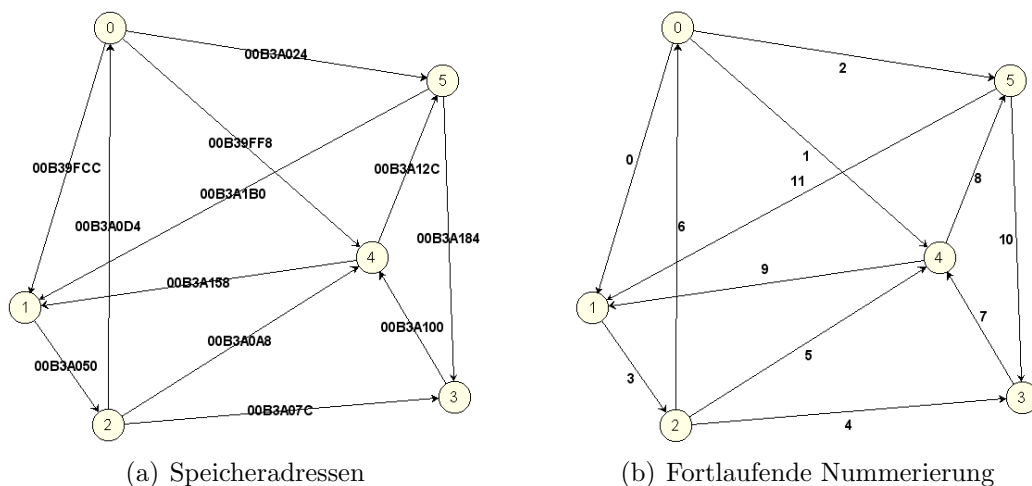


Abbildung 2.1: Graph mit Kantenbeschriftung

Was sehen wir nun an diesem Beispiel? Berücksichtigt man, dass eine Kante 44 Byte an Speicher belegt (bzw. 2C Byte in hexadezimaler Schreibweise), so überzeugt man sich leicht von der Kompaktheit der ausgehenden Kanten. Außerdem zeigt sich, dass mit den Kanten ein fortlaufender Speicherbereich allokiert wird (von B39FCC bis B3A1B0+2C). Wir können unsere m Kanten also genauso gut ordnungserhaltend auf die Zahlen von 0 bis $m - 1$ abbilden, ohne hierdurch Informationen zu verlieren. Dies ist der Ausgangspunkt unserer formalen Darstellung.

2.2 Formale Darstellung

Beginnen wir mit einer Definition der grundlegenden Begriffe und Bezeichnungen.

Definition 2.1. Ein **gerichteter Graph** G ist ein Paar (V, E) zweier disjunkter, endlicher Mengen, wobei $E \subset V \times V$ gilt. V heißt hierbei die Menge der **Knoten**, E die Menge der **Kanten** von G .

Für eine Kante $e = (u, v) \in E$ nennen wir u den **Startknoten** und v den **Endknoten** von e . u bezeichnen wir mit $\text{source}(e)$, v mit $\text{target}(e)$. e ist hierbei eine **ausgehende Kante** von u und eine **eingehende Kante** von v . Die Menge aller eingehender Kanten eines Knotens $v \in V$ werde mit $\text{inedges}(v)$ bezeichnet, die Menge aller ausgehender Kanten mit $\text{outedges}(v)$. Die jeweilige Anzahl der Elemente dieser Menge heißt der **Eingangsgrad** bzw. **Ausgangsgrad** von v und wird mit $\text{indeg}(v)$ bzw. $\text{outdeg}(v)$ bezeichnet.

Hierauf aufbauend wollen wir in der folgenden Definition ein erstes Maß für die Güte einer Kanten-Anordnung bereitstellen.

Definition 2.2. Gegeben sei ein gerichteter Graph $G = (V, E)$, ein Knoten $v \in V$, sowie eine bijektive Abbildung $\varphi : E \rightarrow \{0, \dots, |E| - 1\}$. Die **lokale Kantenbandbreite (der eingehenden Kanten) von v bzgl. φ** ist definiert als

$$S(v, \varphi) := \max\{\varphi(e) \mid e \in \text{inedges}(v)\} - \min\{\varphi(e) \mid e \in \text{inedges}(v)\}$$

Was für eine Bedeutung hat nun dieser Kantenbandbreiten-Begriff für unser Problem? Nehmen wir uns hierzu einen Knoten v aus unserem Graphen, dessen eingehende Kanten wir traversieren wollen. Weiterhin sei eine Speicheranordnung der Kanten durch eine Abbildung φ gegeben.

Die eingehenden Kanten von v erstrecken sich nun im Speicher auf einen Bereich der Größe $S(v, \varphi) * \text{size}(\text{edge})$. Hierbei gibt $\text{size}(\text{edge})$ den benötigten Speicherplatz für eine einzelne Kante an, in obigem Beispiel also 44 Byte.

Nach unseren in der Einführung dargelegten Überlegungen zum Prefetch-Verhalten von Caches ist es für eine effiziente Cache-Ausnutzung sicherlich sinnvoll, diesen Bereich

möglichst klein zu machen. Aus diesem Grund bildet obige Definition die Basis unserer späteren Optimierungsziele. Wir wollen an dieser Stelle jedoch nicht unerwähnt lassen, dass man sich viele andere, auch bessere Gütemaße überlegen kann, sofern man detaillierter auf die konkreten Beschaffenheiten der Caches eingeht. So kann man sich beispielsweise vorstellen, dass ein großer Sprung besser ist als viele kleine, auch wenn sie in der Summe kleiner sein mögen. Denn schlimmstenfalls invalidiert jeder der kleinen Sprünge den vorhergesagten Cache-Inhalt, während dies bei dem großen Sprung unter Umständen nur einmal der Fall ist.

Trotzdem wollen wir es hier im wesentlichen bei diesem Maß belassen. Denn zum einen wollen wir unsere Verfahren möglichst allgemein halten, zum anderen würde die Einbeziehung weiterer Aspekte den Rahmen dieser Arbeit sprengen.

Statt der so definierten Bandbreite könnten wir uns auch auf die Summe der Lücken beschränken, was wie folgt aussähe. Seien e_1, \dots, e_k die bzgl. φ aufsteigend sortierten eingehenden Kanten von v . Insbesondere gilt also

$$e_1 = \min\{\varphi(e) | e \in \text{inedges}(v)\} \text{ und } e_k = \max\{\varphi(e) | e \in \text{inedges}(v)\}. \quad (*)$$

Die Summe der Lücken ergibt sich nun zu

$$S_2(v, \varphi) := \sum_{i=2}^k (e_i - (e_{i-1} + 1))$$

Dies lässt sich aber wie folgt umformen

$$\begin{aligned} S_2(v, \varphi) &= \sum_{i=2}^k (e_i - (e_{i-1} + 1)) - \sum_{i=2}^k 1 \\ &= e_k - e_1 + k - 1 && (\text{Teleskopsumme}) \\ &= S(v, \varphi) + \text{indeg}(v) - 1 && (*) \end{aligned}$$

Es gibt also keinen gravierenden Unterschied zwischen diesen beiden Maßen. Insbesondere spielt es keine Rolle, welches wir minimieren wollen.

Nun haben wir uns bis jetzt auf einen einzelnen Knoten beschränkt. Die Erweiterung auf den gesamten Graphen ist Gegenstand der

Definition 2.3. Gegeben sei ein gerichteter Graph $G = (V, E)$, sowie eine bijektive Abbildung $\varphi : E \longrightarrow \{0, \dots, |E| - 1\}$. Die **Bandbreiten-Summe (der eingehenden Kanten) von G bzgl. φ** ist definiert als

$$S(G, \varphi) := \sum_{v \in V} S(v, \varphi) = \sum_{v \in V} \max\{\varphi(e) | e \in \text{inedges}(v)\} - \min\{\varphi(e) | e \in \text{inedges}(v)\}$$

Bemerkung. Im folgenden sprechen wir auch oft nur noch von den Kosten (einer Kantenummerierung) und meinen damit stets die Bandbreiten-Summe gemäß Definition 2.3.

Für die Formulierung unseres Ziels brauchen wir schließlich noch den Begriff der Kompaktheit, wie er in der folgenden Definition bereitgestellt wird.

Definition 2.4. Sei $G = (V, E)$ ein gerichteter Graph. Eine bijektive Abbildung $\varphi : E \rightarrow \{0, \dots, |E| - 1\}$ heißt **kompakt bzgl. ausgehender Kanten** (kurz **kompakt**) genau dann, wenn gilt:

$$\forall v \in V \exists j \in \mathbb{N} \quad \varphi(\text{outedges}(v)) = \{j, \dots, j + \text{outdeg}(v) - 1\}$$

Ziel. Finde eine bijektive Abbildung φ unserer Kantenmenge E in die ersten $|E| - 1$ natürlichen Zahlen, welche den beiden folgenden Bedingungen genügt:

1. φ ist kompakt
2. Unter allen bijektiven und kompakten Abbildungen $\varphi' : E \rightarrow \{0, \dots, |E| - 1\}$ ist die Bandbreiten-Summe bzgl. φ minimal

Kurz zusammengefaßt stellt sich unser Problem wie folgt dar:

MIBS

Gegeben: Gerichteter Graph $G = (V, E)$

Gesucht: Bijektive und kompakte Abbildung $\varphi : E \rightarrow \{0, \dots, |E| - 1\}$ mit
 $S(G, \varphi) = \min\{S(G, \varphi) | \varphi : E \rightarrow \{0, \dots, |E| - 1\} \text{ bijektiv und kompakt}\}$

Bevor wir einen Datentypen für eine effiziente Behandlung des Problems bereitstellen, wollen wir uns im folgenden Abschnitt noch eine Vereinfachung des Problems überlegen.

2.3 Vereinfachung

Um dieses sehr komplexe Problem ein wenig handhabbarer zu machen, wollen wir es in zwei Teilprobleme zerlegen, welche wir unabhängig voneinander lösen. Schauen wir uns hierzu den zulässigen Lösungsraum, welcher aus allen kompakten Permutationen besteht, etwas genauer an. Ganz naiv könnten wir hergehen und wirklich jede Permutation testen. Zum einen, ob sie tatsächlich kompakt ist, zum anderen, wie es sich mit den Kosten verhält. Vernünftiger wäre es jedoch, generell nur solche Permutationen zu erzeugen, die auch tatsächlich als Lösung in Betracht kommen können. Hierzu fassen wir die Kanten in Blöcken zusammen. Jeweils alle ausgehenden Kanten eines Knotens bilden einen solchen Block. Sowohl die Kanten innerhalb der Blöcke, als auch die Blöcke selbst ordnen wir nun linear an und nummerieren die Kanten bei 0 beginnend gemäß dieser Anordnungen durch. Abbildung 2.2 zeigt ein Beispiel.

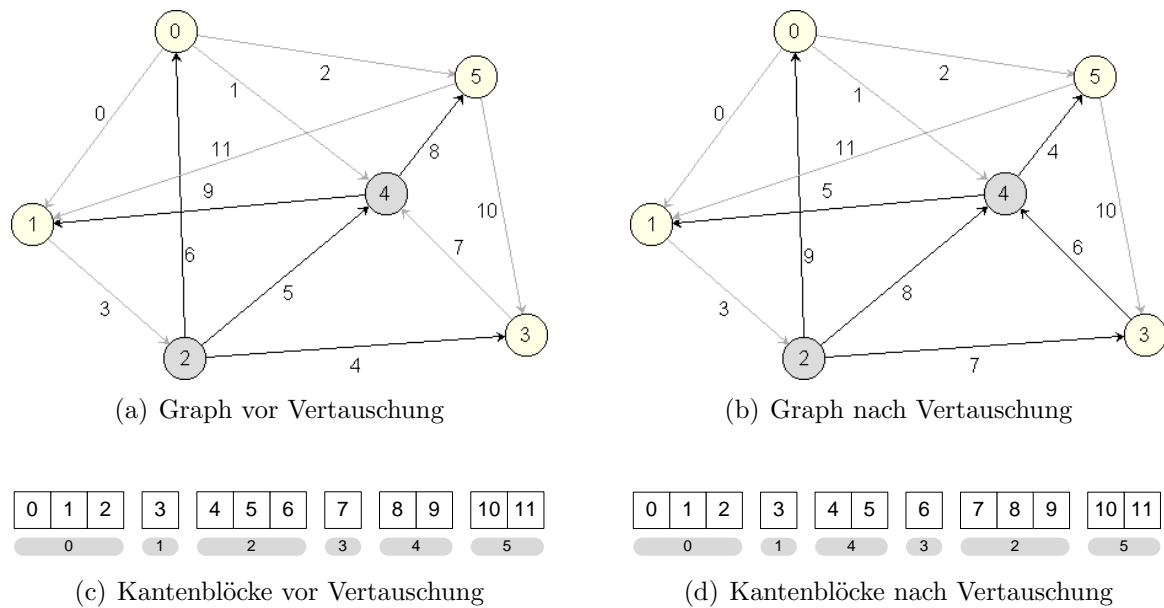


Abbildung 2.2: Externe Umordnung durch Vertauschung der Kantenblöcke 2 und 4

Wir unterscheiden zwei Gruppen möglicher Umordnungen:

Interne Umordnung Permutation der Kanten innerhalb eines Blocks

Externe Umordnung Permutation der Kantenblöcke

In naheliegender Weise ergibt sich durch diese Gruppierung eine Zerlegung des Problems. In einem ersten Schritt wollen wir die Kosten mittels externer Umordnungen minimieren bzw. senken. Erst im Anschluß kümmern wir uns in einer zweiten Phase um die interne Anordnung der Kanten. Als logische Konsequenz dieser Vorgehensweise ergibt sich, dass wir für die Kostenberechnung in der ersten Phase die interne Kantenanordnung völlig unberücksichtigt lassen. Was dies konkret bedeutet, werden wir im nächsten Abschnitt sehen.

Zuvor wollen wir jedoch noch auf eine Problematik unserer Vorgehensweise aufmerksam machen. Nun ist es keineswegs klar, dass eine optimale Lösung des Gesamtproblems aus einer optimalen Teillösung nach der externen Umordnungsphase hervorgeht. Es kann also durchaus sein, dass durch diese zweistufige Minimierung Optimallösungen gar nicht erst gefunden werden können. Doch diese Gefahr nehmen wir zugunsten der deutlichen Vereinfachung in Kauf. Wie wir später sehen werden, ist das Problem „höchstwahrscheinlich“ sowieso nicht effizient lösbar, weshalb wir gar nicht umhin kommen, (möglicherweise optimale) Lösungen außer Acht zu lassen.

2.4 Verwandte Probleme

Abschließend wollen wir eine Übersicht über verwandte Graph-Probleme geben. Es sei angemerkt, dass den folgenden Problemen stets ein ungerichteter Graph zugrunde liegt. Zunächst stellen wir die Klasse der sogenannten Graph-Layout-Probleme vor. Unter einem solchen Layout verstehen wir eine Nummerierung der Knoten mit paarweise verschiedenen ganzzahligen Werten. Ziel ist nun, ein solches Layout zu finden welches eine bestimmte Kostenfunktion optimiert. Ein Überblick über derartige Probleme ist in [6] gegeben. Herausgreifen wollen wir das sogenannte *Bandbreiten-Problem*. Hier ist das Ziel, ein solches Layout der Knoten zu finden, welches die maximale Differenz benachbarter Knoten minimiert. Bereits im Jahre 1976 konnte in [15] die NP-Vollständigkeit dieses Problems gezeigt werden.

Wählt man nun nicht die Minimierung der maximalen Differenzen als Optimierungsziel, sondern die Minimierung der entsprechenden Summen, so führt dies zum *Minimum Linear Arrangement-Problem*, welches in der Literatur auch unter den Namen *Bandwidth-Sum-* oder *Edgesum-Problem* [19] zu finden ist. Auch hier konnte die NP-Vollständigkeit nachgewiesen werden [10]. Dieses Problem werden wir in Kapitel 4 ausführlich vorstellen und mit dessen Hilfe die NP-Vollständigkeit unseres Kantensummen-Problems nachweisen.

Als letztes möchten wir das sogenannte *Kantenbandbreiten-Problem* vorstellen. Hier suchen wir, analog zum (Knoten-)Bandbreiten-Problem, eine Nummerierung der Kanten, welche die maximale Differenz zweier inzidenter Kanten minimiert. Indem wir zu einem gegebenen Graphen G seinen *Line-Graphen* konstruieren, erhalten wir eine Instanz des Knotenbandbreiten-Problems, dessen Lösung uns sofort eine Lösung unseres Ausgangsproblems liefert. Hierbei ist die Knotenmenge des Line-Graphen gerade durch die Kanten des Ursprungsgraphen definiert. Zwei solche Knoten sind benachbart genau dann, wenn die entsprechenden Kanten in G inzident sind. Damit ist das Kantenbandbreiten-Problem aus komplexitätstheoretischer Sicht zumindest nicht schwerer als das Bandbreiten-Problem. Es ist offen, ob dieses Problem überhaupt NP-hart ist [13].

3 Der Bandbreiten-Graph

Wie bereits angekündigt, wollen wir nun einen Datentypen bereitstellen, mit Hilfe dessen wir das beschriebene Problem bearbeiten können. Überlegen wir uns zunächst einmal, welche Operationen wir überhaupt benötigen. Zum einen müssen wir die Kosten gemäß obiger Definition berechnen können. Hierzu benötigen wir Zugriff auf die größte und kleinste eingehende Kante eines jeden Knotens. Dies sollte mit unserem Datentyp in Zeit $O(1)$ möglich sein. Zum anderen müssen wir oben beschriebene interne und externe Kantenumordnungen durchführen können.

3.1 Die Adjazenzmatrix

Auf der Suche nach einer geeigneten Repräsentation unseres Graphen schauen wir uns in einem ersten Schritt einmal seine Adjazenzmatrixdarstellung an. Hierzu zunächst folgende

Definition 3.1. Sei $G = (V, E)$ ein gerichteter Graph mit der Knotenmenge $V = \{v_0, \dots, v_{n-1}\}$. Die **Adjazenzmatrix** $A = (a_{ij})_{n \times n} \in \{0, 1\}^n$ ist definiert durch

$$a_{ij} := \begin{cases} 1 & \text{falls } (v_j, v_i) \in E \\ 0 & \text{falls } (v_j, v_i) \notin E \end{cases}$$

Wir haben in dieser Defintion bewusst Zeilen und Spalten gegenüber der allgemein üblichen Defintion vertauscht. Wir werden nachher sehen warum. Betrachten wir zunächst das Beispiel in Abbildung 3.1(b). Hierin steht ein X in Zeile i und Spalte j für eine Kante von Knoten i zu Knoten j , entspricht also der 1 in obiger Defintion. Kein X bedeutet dementsprechend 0. Die Lücken zwischen eingehenden Kanten sind grau dargestellt.

Wie können wir nun diese Matrix für unser Problem verwenden? Hierzu überlegen wir uns als erstes, wie wir mit Hilfe einer solchen Matrix eine (gültige) Nummerierung der Kanten herbeiführen können. Eine Spalte der Matrix repräsentiert ja nun genau die Menge der ausgehenden Kanten des entsprechenden Knotens, welche es kompakt zu halten gibt. Wir müssen also für jede Spalte fortlaufende Nummern für alle Einträge vergeben. Hierbei wollen wir links (in der ersten Spalte) beginnen, und schrittweise

3 Der Bandbreiten-Graph

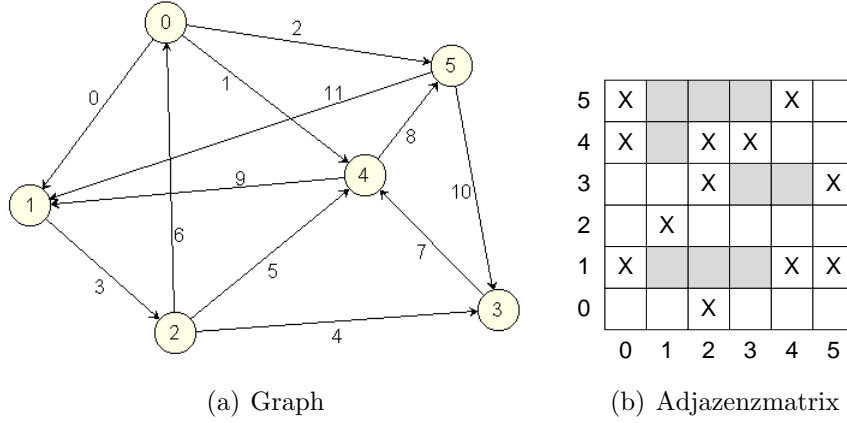


Abbildung 3.1: Adjazenzmatrixdarstellung unseres Beispielgraphen

weiter nach rechts vordringen, bis zur letzten Spalte. Eine externe Umordnung - wie oben beschrieben - findet nun einfach durch Spaltenvertauschungen statt. Innerhalb der Spalten können wir die Nummern frei verteilen, was gerade den internen Umordnungen entspricht.

Da die kleinste Nummer einer Spalte implizit durch die Position der Spalte gegeben ist, brauchen wir hier nur mit einem entsprechenden Offset zu arbeiten. Wir erweitern unsere Adjazenzmatrix entsprechend dieser Überlegungen in

Definition 3.2. Sei $G = (V, E)$ ein gerichteter Graph mit der Knotenmenge $V = \{v_0, \dots, v_{n-1}\}$. Eine **erweiterte Adjazenzmatrix** $A_e = (a_{ij})_{n \times n} \in \{0, 1\}^n$ ist definiert durch

$$a_{ij} := \begin{cases} \text{offset}_j((v_j, v_i)) & \text{falls } (v_j, v_i) \in E \\ -1 & \text{falls } (v_j, v_i) \notin E \end{cases}$$

Hierbei sind

$$\text{offset}_i : \text{outedges}(v_i) \longrightarrow \{0, \dots, \text{outdeg}(v_i) - 1\}, i = 0 \dots n - 1$$

bijektive Abbildungen.

Wie oben skizziert, induziert eine solche Matrix in natürlicher Weise eine Nummerierung der Kanten des zugrunde liegenden Graphen. Spaltenvertauschungen führen zu externen Umordnungen, die Wahl der offset_i -Funktionen bestimmt wiederum die internen Ordnungen. Dies halten wir fest in

Definition 3.3. Sei $G = (V, E)$ ein gerichteter Graph mit $V = \{v_0, \dots, v_{n-1}\}$, A_e die erweiterte Adjazenzmatrix von G bzgl. $(\text{offset}_j : \text{outedges}(v_j) \longrightarrow \{0, \dots, \text{outdeg}(v_j) - 1\})_{j=0}^{n-1}$. Weiterhin sei eine bijektive Abbildung $\pi : V \longrightarrow \{0, \dots, n - 1\}$ gegeben. Die Abbildung

$$\text{num} : E \longrightarrow \{0, \dots, |E| - 1\}$$

3 Der Bandbreiten-Graph

definiert durch

$$(v_j, v_i) \mapsto \sum_{k=0}^{\pi(v_j)-1} \text{outdeg}(\pi(v_k)) + \text{offset}(v_i) \quad \left(\sum_{k=0}^{-1} \dots := 0 \right)$$

heißt die von A_e **induzierte Nummerierung** der Kanten von G .

Somit sind wir auch in der Lage, die Kosten zu berechnen. Da wir jetzt jeder Kante in eindeutiger Weise eine gültige Nummer zuordnen können, wäre dies mittels Definition 2.3 ohne weiteres möglich. Doch kommen wir hierzu auch ohne absolute Kantennummern aus. Dies wollen wir an folgendem Beispiel in Abbildung 3.2 verdeutlichen.

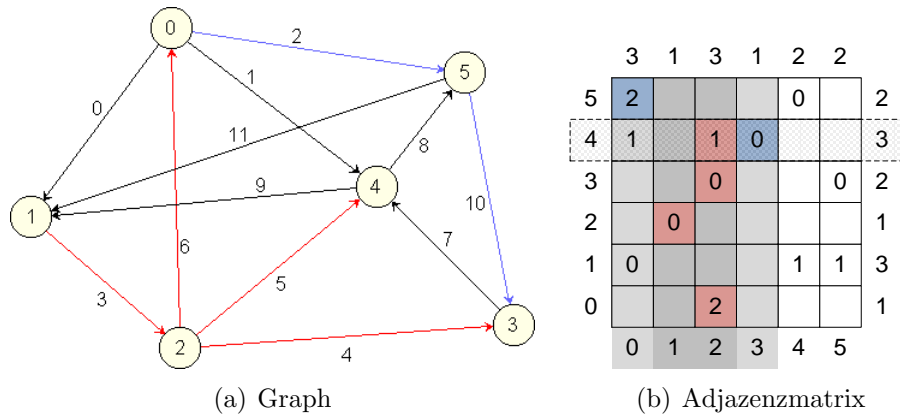


Abbildung 3.2: Erweiterte Adjazenzmatrixdarstellung unseres Beispielgraphen

Hier haben wir den Graphen aus Abbildung 3.1(b), zusammen mit einer erweiterten Adjazenzmatrix. Die X-Markierungen wurden durch die entsprechenden Offsets ersetzt, welche sich aus dem Graphen ergeben. Außerdem wurde die Matrix um die Ausgangsgrade (erste Zeile) bzw. Eingangsgrade (letzte Spalte) ergänzt. Nun wollen wir die lokale Bandbreite des Knotens 4 bestimmen, dessen Zeile in der Matrix hervorgehoben ist. Die zu seiner Bestimmung relevanten Spalten sind grau hinterlegt: Die Spalten der eingehenden Startkante und Endkante hellgrau, die Spalten der dazwischen liegenden Knoten dunkelgrau. Gemäß der Definition der induzierten Nummerierung haben alle Kanten im dunkelgrauen Bereich eine Nummer zwischen derer von Start- bzw. Endkante, verursachen also Kosten von eins. Die Gesamtkosten einer solchen Spalte sind gegeben durch den Ausgangsgrad aus der obersten Zeile. Die entsprechenden Kanten sind sowohl in der Matrix als auch im Graphen rot dargestellt. Nun müssen wir noch im hellgrauen Bereich überprüfen, welche Offsets größer des Offsets der Startkante bzw. kleiner gleich dessen der Endkante sind (vgl. Definition 2.2). Diese sind jeweils blau dargestellt. Insgesamt ergeben sich für den Knoten 4 also Kosten in Höhe von 6.

3.2 Vereinfachung: Ein zweites Kostenmaß

Wir wollen nun auf die bereits angesprochene Vereinfachung aus Abschnitt 2.3 zurückkommen. Zunächst werden wir diese an einer alternativen Darstellung unserer Adjazenzmatrix einführen, bevor wir sie anschließend formal erfassen. Abbildung 3.3 zeigt eine solche „Bandbreiten-Darstellung“ unseres Beispielgraphen.

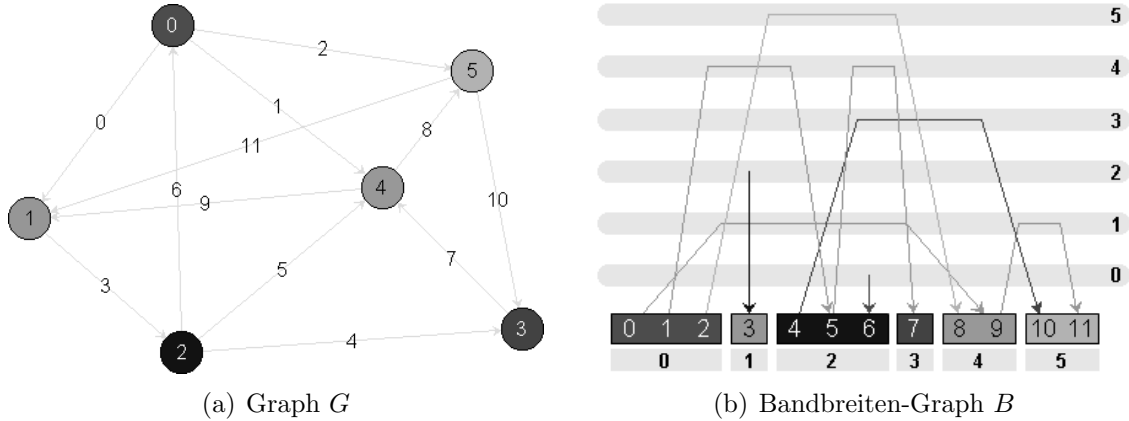


Abbildung 3.3: Bandbreiten-Graph-Darstellung

Was hat sich gegenüber der Adjazenzmatrix-Darstellung geändert? Für jeden Knoten in G haben wir in B einen Knoten der Länge des entsprechenden Ausgangsgrades. Waren die eingehenden Kanten eines Knotens v in der Matrixdarstellung durch ein X gekennzeichnet, so werden in dieser Darstellung benachbarte eingehende Kanten von v durch eine Kante verbunden. Dies auf der Höhe der entsprechenden Knotennummer. Schließlich werden die Offsets durch die entsprechende Start- bzw. Endposition innerhalb des Knotens repräsentiert. Die lokale Bandbreite eines Knotens ist nun durch die Summe der Längen der entsprechenden Kanten in B gegeben.

Die bereits beschriebene Vereinfachung besagt nun, dass wir uns zunächst nur um die externe Ordnung kümmern, also die internen Ordnungen innerhalb der Knoten erst einmal völlig außer Acht lassen. Wie stellen wir dies am geschicktesten an? Zunächst bemerken wir kurz, dass die exakte Position einer Kante innerhalb eines Knotens nur für Start- und Endknoten eines Pfades relevant sind. Dies diskutieren wir ausführlicher im Kapitel 5. Bleibt also nur die Frage, was wir mit Start- und Endkanten machen. Da wir noch nicht wissen, wie die exakte Ordnung innerhalb der Knoten später einmal aussehen wird, erreichen wir die beste Annäherung durch die Wahl der Mittelposition. Abbildung 3.4 veranschaulicht dies. Hierbei ist jeder Knoten mit seiner Länge gekennzeichnet.

Diese Überlegungen führen zu einem neuen Kostenbegriff, welchen wir in Definition 3.5 bereitstellen wollen. Zuvor wollen wir jedoch noch den Begriff der Start- bzw. Endknoten präzisieren in

3 Der Bandbreiten-Graph

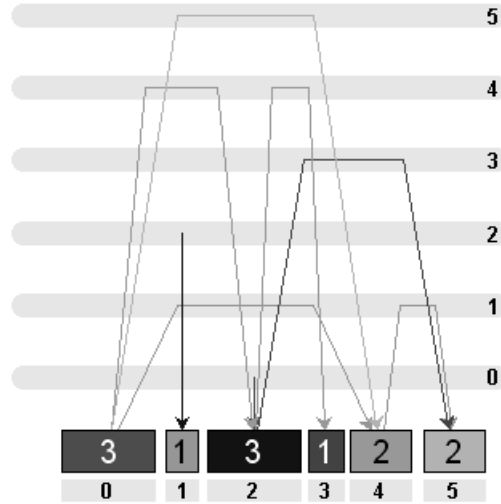


Abbildung 3.4: Vereinfachung - Kanten starten und enden in der Knotenmitte

Definition 3.4. Seien $G = (V, E)$ ein gerichteter Graph, π eine bijektive Abbildung auf V und $v \in V$ gegeben. Wir definieren

$$\text{NB}(v) := \{\text{source}(e) \mid e \in \text{inedges}(v)\}$$

als die Menge der **Nachbarn von v** (bzgl. eingehender Kanten). Hierbei nennen wir $u \in \text{NB}(v)$ mit $\pi(u) = \min\{\pi(w) \mid w \in \text{NB}(v)\}$ den **kleinsten Nachbarn von v** bzgl. π und schreiben für u $\text{min_neighbor}_\pi(v)$. Die entsprechende Kante $(\text{min_neighbor}_\pi(v), v)$ nennen wir **minimale (eingehende) Kante von v** und schreiben hierfür $\text{min_edge}_\pi(v)$. In analoger Weise ist der größte Nachbar und die maximale Kante von v definiert.

Die minimale Kante eines Knotens v ist also diejenige eingehende Kante von v , welche bzgl. π den kleinsten Sourceknoten besitzt. Die maximale Kante ist dementsprechend diejenige mit dem größten Sourceknoten.

Definition 3.5. Sei $G = (V, E)$ ein gerichteter Graph und π eine bijektive Abbildung auf V . Die Abbildung $\text{len} : V \rightarrow \mathbb{N}$ sei definiert durch $v \mapsto \text{outdeg}(v)$, $v \in V$. Die **Näherungskosten von G bzgl. π** sind definiert als

$$\text{approx_cost}_\pi(G) := \sum_{v \in V} \left(\sum_{\substack{u \in V \\ m_v \leq \pi(u) \leq M_v}} \text{len}(u) - \frac{1}{2}(\text{len}(m_v) + \text{len}(M_v)) \right)$$

wobei $m_v := \text{min_neighbor}_\pi(v)$ und $M_v := \text{max_neighbor}_\pi(v)$, $v \in V$. Hierbei nennen wir π auch eine **Knotenanzuordnung** oder kurz **Anordnung** von V .

In einer letzten Definition führen wir noch einen Positionsbegriff ein, welcher uns im Anschluss eine leichtere Berechnung der Näherungskosten ermöglicht.

3 Der Bandbreiten-Graph

Definition 3.6. Sei $G = (V, E)$ ein gerichteter Graph. Eine **Platzierung** der Knotenmenge V ist eine Abbildung $\text{pos} : V \rightarrow \mathbb{N}$. Für einen Knoten $v \in V$ heißt $\text{pos}(v)$ die Position von v . Eine Platzierung heißt **gültig**, falls

$$\forall v \in V \text{ pos}(v) = \sum_{\substack{u \in V \\ \text{pos}(u) \leq \text{pos}(v) \\ u \neq v}} \text{len}(u)$$

Für eine Anordnung der Knotenmenge $\pi : V \rightarrow \{0, \dots, |V| - 1\}$ heißt die durch

$$\text{pos}_\pi : V \rightarrow \mathbb{N}, v \mapsto \sum_{\substack{u \in V \\ \pi(u) < \pi(v)}} \text{len}(u)$$

definierte Abbildung **die von π induzierte Platzierung von V** .

Hiermit gilt für die Näherungskosten der folgende

Satz 3.1. Sei $G = (V, E)$ ein gerichteter Graph. Setze $m_v := \min_neighbor_\pi(v)$, $M_v := \max_neighbor_\pi(v)$, $v \in V$. Dann gilt.

$$\text{approx_cost}(G) = \sum_{v \in V} \left(\text{pos}_\pi(M_v) - \text{pos}_\pi(m_v) + \frac{1}{2}(\text{len}(M_v) - \text{len}(m_v)) \right).$$

Beweis. Sei $v \in V$ beliebig. Es gilt

$$\begin{aligned} & \text{pos}_\pi(M_v) - \text{pos}_\pi(m_v) + \frac{1}{2}(\text{len}(M_v) - \text{len}(m_v)) \\ & \stackrel{3.6}{=} \sum_{\substack{u \in V \\ \pi(u) < \pi(M_v)}} \text{len}(u) - \sum_{\substack{u \in V \\ \pi(u) < \pi(m_v)}} \text{len}(u) + \frac{1}{2}(\text{len}(M_v) - \text{len}(m_v)) \\ & = \sum_{\substack{u \in V \\ \pi(m_v) \leq \pi(u) \leq \pi(M_v)}} \text{len}(u) - \text{len}(M_v) + \frac{1}{2}(\text{len}(M_v) - \text{len}(m_v)) \\ & = \sum_{\substack{u \in V \\ \pi(m_v) \leq \pi(u) \leq \pi(M_v)}} \text{len}(u) - \frac{1}{2}(\text{len}(M_v) - \text{len}(m_v)) \end{aligned}$$

□

3.3 Der abstrakte Datentyp

In diesem Abschnitt wollen wir zunächst noch einmal kurz die wesentlichen Bestandteile unseres Bandbreiten-Graphen `bw_graph` zusammentragen. Anschließend werden wir die wichtigsten Operationen auf diesem Datentypen definieren.

3.3.1 Komponenten des Bandbreiten-Graphen

Grundlegend für unseren Bandbreiten-Graphen ist ein gerichteter Graph $G = (V, E)$ zusammen mit einer Anordnung der Knoten π . Die Knoten aus V versehen wir gemäß unseren vorangegangenen Überlegungen noch mit zwei zusätzlichen Informationen. Zum einen hat jeder Knoten eine Länge len , welche seinem Ausgangsgrad entspricht, zum anderen eine durch π induzierte Position pos .

Die Kanten werden mit einem offset versehen, welcher entweder die tatsächliche Position innerhalb eines Knotens enthält, oder aber in der externen Umordnungsphase dessen Mittelpunkt.

3.3.2 Operationen auf dem Bandbreiten-Graphen

Nun wollen wir die grundlegenden Operationen unseres Datentypen definieren, wie wir sie auch in Kapitel 5 zur Beschreibung unserer Algorithmen benötigen.

Definition 3.7. Sei $G = (V, E)$ ein gerichteter Graph, $V = \{v_0, \dots, v_{n-1}\}$, $i \in \{0, \dots, n-2\}$. Unter der **Inkrementierung des Knotens** v_i verstehen wir die Überführung einer Anordnung π in eine Anordnung π' , für die gilt:

$$\pi'(j) = \begin{cases} \pi(j) + 1 & \text{falls } j = i \\ \pi(j) - 1 & \text{falls } \pi(j) = i + 1 \\ \pi(j) & \text{sonst} \end{cases}$$

In analoger Weise ergibt sich

Definition 3.8. Sei $G = (V, E)$ ein gerichteter Graph, $V = \{v_0, \dots, v_{n-1}\}$, $i \in \{1, \dots, n-1\}$. Unter der **Dekrementierung des Knotens** v_i verstehen wir die Überführung einer Anordnung π in eine Anordnung π' , für die gilt:

$$\pi'(j) = \begin{cases} \pi(j) - 1 & \text{falls } j = i \\ \pi(j) + 1 & \text{falls } \pi(j) = i - 1 \\ \pi(j) & \text{sonst} \end{cases}$$

Weitere Operationen werden in Kapitel 5 an entsprechender Stelle eingeführt.

3.4 Der konkrete Datentyp

In diesem letzten Abschnitt wollen wir noch kurz einen Blick auf die grundlegende Datenstruktur für unseren Bandbreiten-Graphen werfen. Zunächst einmal bemerken wir,

3 Der Bandbreiten-Graph

dass es äußerst teuer wäre, tatsächlich eine Matrix als ein 2-dimensionales Feld zu implementieren. Stattdessen speichern wir die Knoten in einem Feld entsprechender Größe. Für jeden Knoten stellen wir ein weiteres Feld der Größe des jeweiligen Ausgangsgrades bereit, in dem die Target-Knoten der ausgehenden Kanten gespeichert werden. Schauen wir uns dies in Abbildung 3.5 für unseren Beispielgraphen G an.

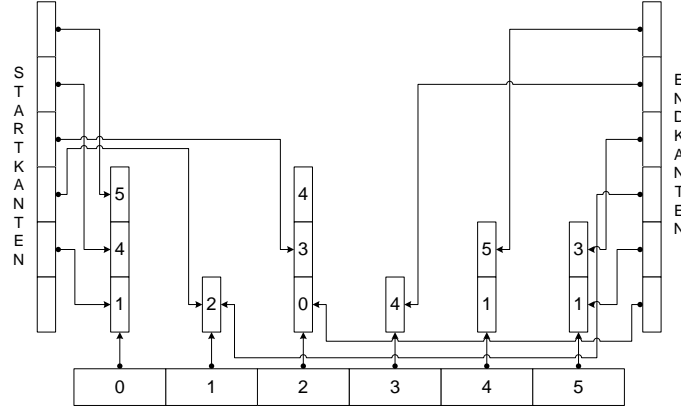


Abbildung 3.5: Datentyp bwgraph

Verweise für Start- und Endkanten sind in zwei entsprechenden Feldern gespeichert. Dies ermöglicht uns die Berechnung der Kosten in Zeit $O(n)$. Hierbei ist n die Anzahl der Knoten von G .

4 NP-Vollständigkeit

Bevor wir uns auf die Suche nach Lösungs- bzw. Approximationsverfahren machen, erscheint es sinnvoll, sich zunächst mit der Komplexität des Problems zu befassen. Hierzu ziehen wir ein verwandtes Problem heran, welches uns eine komplexitätstheoretische Einordnung von MIBS ermöglicht. Bei diesem Problem handelt es sich um das sogenannte Minimum Linear Arrangement Problem. Zunächst wollen wir dieses vorstellen, wie es üblicherweise in der Literatur vorzufinden ist [14]. Anschließend stellen wir einen Zusammenhang zu unserem Kantensummen-Problem her.

4.1 Das Minimum Linear Arrangement Problem

Starten wir mit zwei grundlegenden Definitionen, welche uns im Anschluß eine formale Darstellung des Problems ermöglichen.

Definition 4.1 (Lineares Arrangement). Sei $G = (V, E)$ ein ungerichteter Graph mit $|V| = n$. Ein **lineares Arrangement** von G ist eine bijektive Abbildung $\pi : V \rightarrow \{0, \dots, n-1\}$.

Definition 4.2 (Kosten eines Linearen Arrangements). Sei $G = (V, E)$ ein ungerichteter Graph. Die **Kosten eines Linearen Arrangements** π von G sind definiert durch

$$LA_\pi(G) := \sum_{\langle i, j \rangle \in E} |\pi(i) - \pi(j)|$$

Das Ziel ist nun, unter allen möglichen linearen Arrangements ein solches zu finden, welches die Kosten minimiert. Formal stellt sich das Minimum Linear Arrangement Problem (MINLA) wie folgt dar:

MINLA

Gegeben: Ungerichteter Graph $G = (V, E)$

Gesucht: $\pi : V \rightarrow \{0, \dots, |V| - 1\}$ bijektiv mit
 $LA_\pi(G) = LA(G) := \min\{LA_\pi(G) \mid \pi : V \rightarrow \{0, \dots, |V| - 1\} \text{ bijektiv}\}$

Die Entscheidungsvariante läßt sich wie folgt formulieren:

MINLA_D

Gegeben: Ungerichteter Graph $G = (V, E)$, $C \in \mathbb{N}$

Frage: $\exists \pi : V \longrightarrow \{0, \dots, |V| - 1\}$ bijektiv mit $LA_\pi(G) < C$?

Unsere späteren Überlegungen basieren auf folgendem wichtigen Satz, für dessen Richtigkeit wir hier nur auf die Literatur verweisen können [9].

Satz 4.1. *Die Entscheidungsvariante von MINLA ist NP-vollständig.*

Dieses Problem wollen wir nun so zu MIBS in Beziehung setzen, dass wir dieses Ergebnis darauf übertragen können.

4.2 NP-Vollständigkeit von MIBS

In diesem Abschnitt wollen wir zeigen, dass es sich bei MIBS um ein NP-vollständiges Problem handelt. Da wir dies durch eine Reduktion von MINLA bewerkstelligen werden, sollen zuvor noch einmal die Entscheidungsvarianten beider Probleme dargelegt werden:

MINLA

Gegeben: Ungerichteter Graph $G = (V, E)$, $n := |V|$, $C \in \mathbb{N}$

Frage: $\exists \pi : V \longrightarrow \{0, \dots, n - 1\}$ bijektiv mit $\sum_{\langle u, v \rangle \in E} |\pi(v) - \pi(u)| \leq C$?

MIBS

Gegeben: Gerichteter Graph $G = (V, E)$, $B \in \mathbb{N}$

Frage: $\exists \varphi : E \longrightarrow \{0, \dots, |E| - 1\}$ bijektiv und kompakt¹ mit $\sum_{v \in V} (\max\{\varphi(e) | e \in \text{inedges}(v)\} - \min\{\varphi(e) | e \in \text{inedges}(v)\}) \leq B$?

Satz 4.2. *MIBS ist NP-vollständig.*

Bevor wir einen formalen Beweis führen, wollen wir zunächst die grundlegende Idee der Reduktion darlegen.

¹kompakt bzgl. ausgehender Kanten, siehe Definition 2.4

4.2.1 Idee

Wie bereits erwähnt, wollen wir das Minimum Linear Arrangement Problem auf unser Kantensummen-Problem reduzieren. Hierzu müssen wir zu einer gegebenen Instanz $I = (G, C)$ von MINLA eine Instanz $I' = (G', C')$ von MIBS konstruieren, so dass folgendes gilt: Es gibt eine Anordnung der Knoten von G mit Kosten kleiner oder gleich C genau dann, wenn es eine kompakte Nummerierung der Kanten von G' mit Kosten kleiner oder gleich C' gibt. Hierbei müssen wir C' berechnen können, ohne dass wir auf eine konkrete Knotenanordnung bzw. Kantenummerierung zurückgreifen können. Die Reduktion wollen wir am Beispiel des linearen Arrangements in Abbildung 4.1 vorstellen. Hierbei lassen wir zunächst die zweite Implikation - also dass wir von den Kosten von G' auch wieder auf die Kosten von G zurückschließen können - außer Betracht.

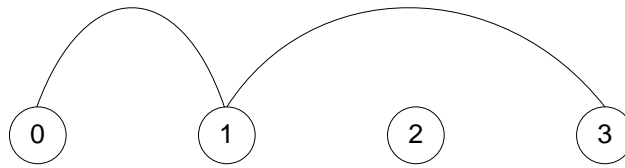


Abbildung 4.1: Ein einfaches lineares Arrangement G

Die Linear-Arrangement-Kosten ergeben sich nun im wesentlichen aus den Lücken bzgl. einer Kante benachbarter Knoten (die genauen Kosten nach Definition 4.2 ergeben sich durch Addition der Kantenzahl).

Die Lücken wollen wir nun auf eine entsprechende MIBS-Instanz übertragen, deren Kosten gerade durch minimale und maximale eingehende Kanten (bzgl. einer Kantenummerierung) entstehen. Es bietet sich also an, für jede Kante von G einen Knoten in G' mit jeweils genau zwei eingehenden Kanten zu konstruieren. Diesen Kanten müssen wir nun derart Nummern geben können, dass sich die resultierenden Kosten (Betrag der Differenzen) aus der entsprechenden Kante von G herleiten lassen. Hierzu übernehmen wir die Knoten mit ihrer Anordnung aus G . Ausgehende Kanten gehen zu den jeweiligen „Kanten-Knoten“ von G' , wie in Abbildung 4.2 dargestellt.

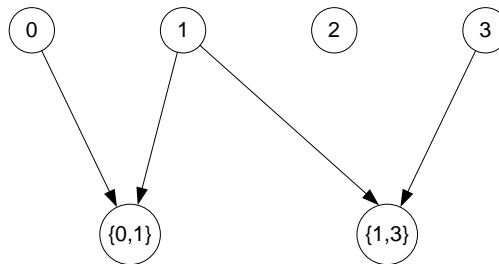


Abbildung 4.2: Konstruierter MIBS-Graph G'

Wir wollen nun eine *ordnungserhaltende Kantenummerierung* finden. Dies soll bedeuten, dass die Ordnung der Knoten von G durch die Kantenummerierung widerspiegelt

wird. Eine Präzisierung findet sich in

Definition 4.3. Sei $G = (V, E)$ ein ungerichteter Graph, $\pi : V \rightarrow \{0, \dots, |V| - 1\}$ bijektiv. Weiterhin sei $G' = (V', E')$ ein gerichteter Graph mit $V \subset V'$. Eine bijektive und kompakte Abbildung $\varphi : E' \rightarrow \{0, \dots, |E'| - 1\}$ heißt **ordnungserhaltend (bzgl. π)**, falls für alle $u, v \in V \subset V'$ gilt

$$\forall e \in \text{outedges}(u), \forall e' \in \text{outedges}(v) : \pi(u) < \pi(v) \implies \varphi(e) < \varphi(e')$$

In einem ersten Ansatz erhalten nun die Kanten die Nummern der jeweiligen Startknoten.

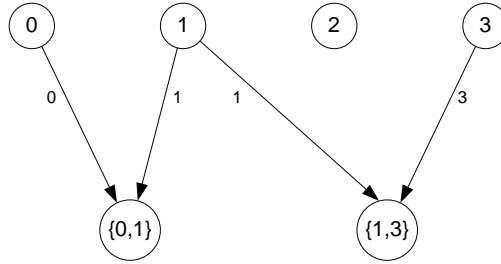


Abbildung 4.3: Konstruierter MIBS-Graph G' mit Kantennummerierung

Tatsächlich wären auch hier die Kosten sogar gleich C . Jedoch verletzen wir die Anforderungen an eine Kantennummerierung, welche nach Definition bijektiv sein muss. Doch weder die Injektivität, noch die Surjektivität ist hier gegeben. Wie können wir dies korrigieren, so dass wir den Graphen trotzdem noch für unsere Reduktion gebrauchen können? Aufgrund der ordnungserhaltenden Eigenschaft entsteht eine der beiden in Abbildung 4.4 dargestellten Situationen.

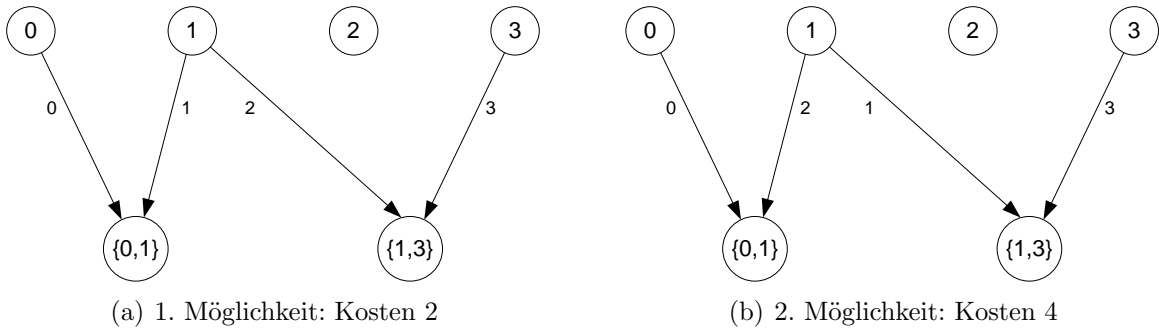


Abbildung 4.4: Korrigierter MIBS-Graph (Bijektivität)

Jetzt haben wir zwar die Bijektivität der Kantennummerierung hergestellt, doch haben die Kosten nichts mehr mit denen des zugrunde liegenden MINLA-Problems zu tun. Wie eingangs beschrieben, wollten wir aber auch gar nicht die eigentlichen Kosten

übertragen, sondern nur die Lücken. Nun verursachen die Lücken nach unserer bisherigen Konstruktion noch überhaupt keine Kosten. Dies lässt sich ändern, indem wir die oberen Knoten (jene aus dem MINLA-Graphen G) mit „Dummy-Kanten“ erweitern. Eine Dummy-Kante soll hierbei eine neue ausgehende Kante mit einem neuen Zielknoten sein. Die Anzahl dieser Kanten soll nun so gewählt werden, dass folgende zwei Bedingungen erfüllt werden:

1. Der Ausgangsgrad der Knoten von G' ist konstant
2. Die Gesamtkosten sollen von den Lücken dominiert werden

Der zweite Punkt soll bedeuten, dass wir die anderen Kosten (die eben nicht durch die Lücken entstehen) in unserer Kostenabschätzung völlig vernachlässigen können. Zur Herstellung der ersten Eigenschaften brauchen wir uns nur den Maximalgrad von G anzuschauen (in unserem Beispiel also 2), und die Knoten um entsprechend viele Dummy-Kanten zu ergänzen. In Abbildung 4.5 ist unser neuer Graph dargestellt. Hierbei haben wir zusätzlich eine Klassifizierung der Kanten vorgenommen, welche wir weiter unten besprechen werden.

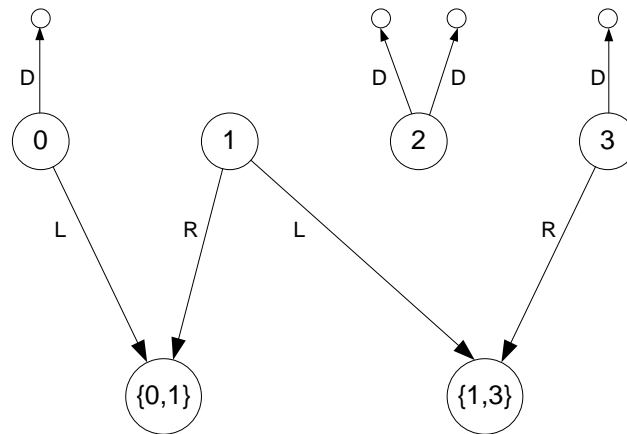


Abbildung 4.5: 1. Erweiterung von G' mit Kanten-Klassifizierung

Wir bemerken zunächst, dass jede Lücke mit Kosten in Höhe des Grades von G zu den Gesamtkosten von G' beiträgt. Zur Herstellung der zweiten Eigenschaft betrachten wir eine kostenoptimale ordnungserhaltende Nummerierung der Kanten. Um eine solche zu erhalten, dient uns die Klassifizierung der Kanten. Neben den bereits besprochenen Dummy-Kanten (D) gibt es noch sogenannte Linkskanten (L) und Rechtskanten (R). Zur Erklärung betrachten wir einen Knoten, der eine Kante von G repräsentiert. Ein solcher Knoten hat nach unserer Konstruktion stets zwei eingehende Kanten. Die zugehörigen Quellknoten sind über das lineare Arrangement von G geordnet. Die Kanten mit den jeweils kleineren Quellknoten sind nun gerade unsere Linkskanten, die mit den größeren Quellknoten unsere Rechtskanten.

Nun ordnen wir unter Beachtung der Ordnungserhaltung den Linkskanten möglichst

große Nummern, den Rechtskanten möglichst kleine Nummern zu. Mit den verbleibenden Zahlen werden die Dummy-Kanten nummeriert. Wie man sich leicht überzeugt, ist hiermit gerade eine optimale ordnungserhaltende Kantenummerierung gegeben. Abbildung 4.6 zeigt dies an unserem Beispiel.

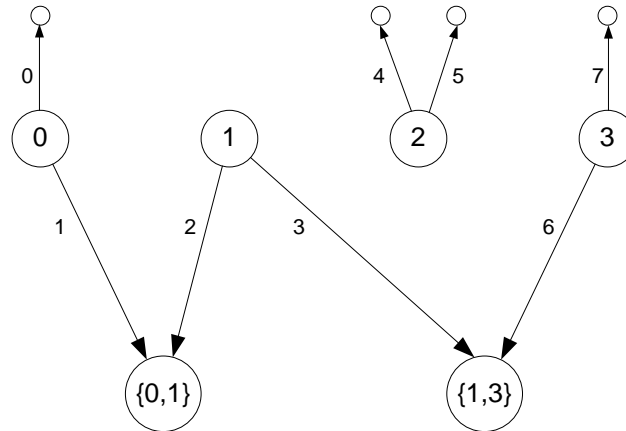


Abbildung 4.6: 1. Erweiterung von G' mit Kanten-Nummerierung

Nun können wir die Gesamtkosten aufteilen in Kosten, die unmittelbar durch die Lücken entstehen (welche wir als *externe Kosten* bezeichnen), und die verbleibenden Kosten (die wir als *interne Kosten* bezeichnen). Letztere sind nun zwar abhängig von einer konkreten Anordnung der Knoten, können jedoch nach oben abgeschätzt werden. Bevor wir eine obere Schranke bestimmen, gehen wir zunächst davon aus, dass diese durch s gegeben sei. Die zweite Bedingung an die Anzahl der Dummy-Kanten wird nun dadurch erfüllt, dass wir den Ausgangsgrad der Knoten aus $V \subset V'$ nicht nur auf den Grad von G erhöhen, sondern auf $s + 1$. Die erste Bedingung bleibt hiermit erhalten.

Nun wollen wir eine solche obere Schranke s bestimmen. Betrachten wir hierzu einen Knoten aus $V \subset V'$, den wir entsprechend unseren Überlegungen um Dummy-Kanten ergänzt haben. In Abbildung 4.7 ist ein solcher Knoten in der Länge seines Ausgangsgrads $s + 1$ dargestellt.

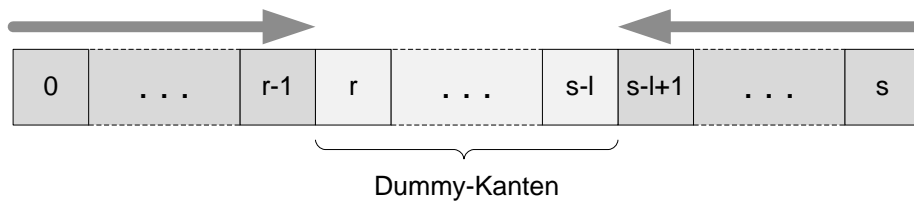


Abbildung 4.7: Interne Ordnung der ausgehenden Kanten eines Knotens $v \in V \subset V'$

Wir haben hier noch einmal unser Verfahren zur Nummernvergabe veranschaulicht. Die r Rechtskanten erhalten die ersten r Nummern des jeweiligen Kantenblocks, die l Linkskanten die letzten l Nummern. Da nun die Linkskanten gerade unsere minimalen eingehenden Kanten in G' sind, und die Rechtskanten die maximalen, lassen sich die internen

Kosten c_{int} , die durch einen Knoten induziert werden, wie folgt berechnen:

$$c_{\text{int}} = \sum_{i=0}^{l-1} i + \sum_{i=0}^r i$$

Dass wir in der zweiten Summe das i nicht auch nur bis $r-1$ laufen lassen kommt daher, dass wir hier nicht nur die Lücken eingehender Kanten betrachten, sondern die Differenz. Für jedes eingehende Kantenpaar ist diese Differenz um eins größer als die entsprechende Lücke. Dies berücksichtigen wir bei den Rechtskanten. An unserem Beispiel wollen wir uns in Abbildung 4.8 die Aufteilung der Kosten anschauen.

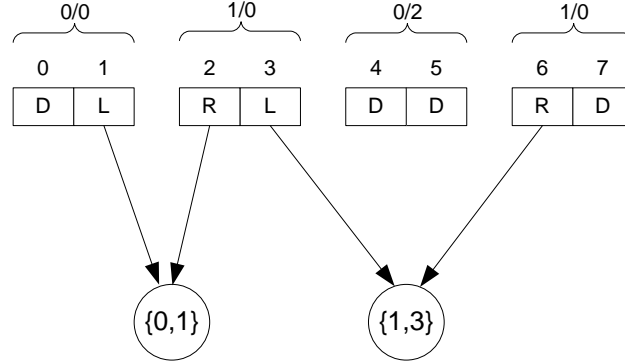


Abbildung 4.8: Intern/Extern induzierte Kosten der Knoten aus $V \subset V'$

Wie wir sehen, haben wir hier interne Kosten von 2, und ebenfalls externe Kosten von 2. Berechnen wir die Kosten nach Definition mittels der Differenzen eingehender Kanten, so ergeben sich ebenfalls Gesamtkosten von $(2-1) + (6-3) = 4$. Desweiteren wird in dieser Abbildung deutlich, dass die internen Kosten unabhängig von der Anzahl der Dummy-Kanten sind. Jedoch sind diese, wie bereits erwähnt, abhängig von der Anordnung der Knoten von G . Eine Umordnung dieser Knoten hat im Allgemeinen Einfluß auf die Anzahl der jeweiligen Links- bzw. Rechtskanten. Zwar ist die Gesamtheit beider Kantenarten stets gleich, jedoch führt dies im Allgemeinen nicht zu gleichen internen Kosten, wie aus obiger Berechnung von c_{int} ersichtlich ist. Eine obere Schranke für die internen Kosten erhalten wir nun, wenn wir von dem (unmöglichen) Fall ausgehen, dass jede Kante eine Rechtskante ist. Jede andere (mögliche) Aufteilung in Links- und Rechtskanten verursacht geringere Kosten. Berücksichtigen wir schließlich noch, dass die Gesamtanzahl dieser Kanten durch den Grad des entsprechenden Knotens in G gegeben ist, so liefert uns

$$c_{\text{int}} = \sum_{i=0}^{l-1} i + \sum_{i=0}^r i \leq \sum_{i=0}^{l+r} i$$

eine benötigte Abschätzung für s . Dieses können wir nun berechnen mittels

$$s = \sum_{v \in V} \sum_{i=0}^{\deg_G(v)} i$$

Für unser Beispiel berechnet sich diese Schranke also zu $s = (0 + 1) + (0 + 1 + 2) + 0 + (0 + 1) = 5$. Abbildung 4.9 zeigt den endgültigen Graphen G' .

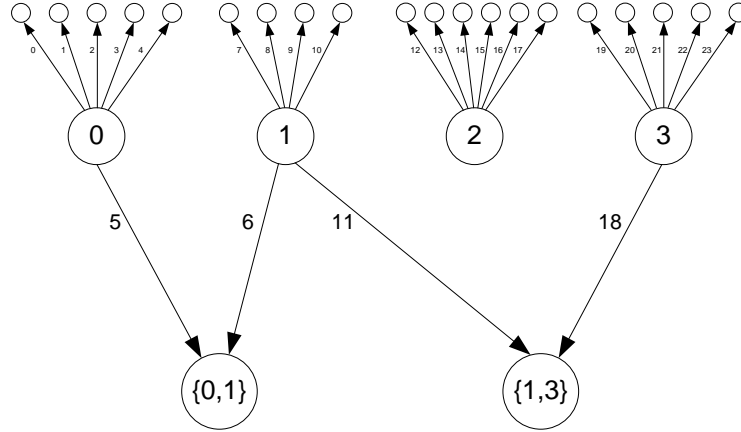


Abbildung 4.9: Endgültige Version unseres Graphen G'

Wir haben zu $G = (V, E)$ mit $|E| = m$ also einen Graphen $G' = (V', E')$ konstruiert, der folgendes leistet: Falls es eine Anordnung der Knoten von G mit Kosten kleiner oder gleich C gibt, so können wir eine Nummerierung der Kanten von G' finden, so dass die entstehenden Kosten kleiner oder gleich $(s + 1) \cdot (C - m) + s$ sind. $C - m$ ist eine obere Schranke für die Summe der Lücken in unserem Linearen Arrangement.

Zur Umkehrung bemerken wir, dass eine bijektive und kompakte Nummerierung der Kanten eine Ordnung der Knoten von G induziert (vgl. Definition 4.3). Von dieser Nummerierung können wir nun aber o.B.d.A. annehmen, dass diese bzgl. der Ordnung der Knoten von G optimal ist (falls sie es nicht ist, erhalten wir eine solche durch eine entsprechende Umnummerierung der Kanten mit geringeren Kosten). Von dieser optimalen Nummerierung ausgehend können wir nun aber wieder zurückschließen auf die Kosten des Linearen Arrangements.

4.2.2 Formaler Beweis

Aufbauend auf den vorausgegangenen Überlegungen wollen wir nun den formalen Beweis führen.

Beweis. Die Zugehörigkeit $\text{MIBS} \in NP$ ergibt sich durch den üblichen Guess & Check-Ansatz („rate“ Permutation und berechne in Polynomialzeit die entstehenden Kosten). Zum Nachweis der NP-Härte wollen wir das Minimum Linear Arrangement Problem reduzieren, also zeigen, dass es eine Polynomialzeitreduktion $\text{MINLA} \leq_P \text{MIBS}$ gibt.

Sei also eine Instanz $(G = (V, E), C)$ mit $V = \{0, \dots, n - 1\}$, $E \subset \{\{u, w\} | u, w \in V\}$

4 NP-Vollständigkeit

mit $|E| =: m$ und $C \in \mathbb{N}$ von MINLA gegeben. Wir konstruieren nun einen gerichteten Graphen $G' = (V', E')$ wie folgt:

$$V' := V \cup E \cup D, \quad E' := E_I \cup E_D$$

Zur Bestimmung der Knotenmenge D setzen wir

$$s := \sum_{v \in V} \sum_{i=0}^{\deg_G(v)} i$$

und definieren hiermit

$$D := \bigcup_{v \in V} \bigcup_{i=0}^{s-\deg(v)} \{d_{v,i}\}$$

so dass $D \cap (E \cup V) = \emptyset$ gilt. D ist also die Menge der Dummy-Knoten gemäß unserer obigen Überlegungen.

Die Kantenmengen E_I , E_D setzen sich wie folgt zusammen:

$$E_I := \{(u, \{u, v\}), (v, \{u, v\}) \mid \{u, v\} \in E\} \subset V \times E$$

$$E_D := \bigcup_{v \in V} E_v \text{ mit } E_v := \bigcup_{i=0}^{s-\deg(v)} \{(v, d_{v,i})\}$$

E_I liefert uns also für jeden Knoten $v \in V$ ein Paar eingehender Kanten, welche die Kosten in unserer MIBS-Instanz verursachen. E_D erweitert für jeden Knoten $v \in V$ die Menge seiner ausgehenden Kanten auf den konstanten Wert $s + 1$ (Dummy-Kanten), was für unsere spätere Kostenabschätzung benötigt wird. Für die Anzahl der Dummy-Knoten und -Kanten gilt hiermit:

$$|D| = |E_D| = \sum_{v \in V} (s - \deg(v) + 1) = n \cdot (s + 1) - \sum_{v \in V} \deg_G(v) = n \cdot (s + 1) - 2m$$

Für die Gesamtanzahl der Knoten und Kanten folgt hieraus:

$$\begin{aligned} |V'| &= n + m + n \cdot (s + 1) - 2m = n \cdot (s + 2) - m \\ |E'| &= 2m + n \cdot (s + 1) - 2m = n \cdot (s + 1) \end{aligned}$$

Wir setzen

$$B := (C - m) \cdot (s + 1) + s$$

und behaupten

$$\begin{aligned}
 \exists \pi : V \longrightarrow \{0, \dots, n-1\} \text{ bijektiv mit } LA_\pi(G) \leq C \\
 \iff \\
 \exists \varphi : E' \longrightarrow \{0, \dots, n \cdot (s+1) - 1\} \text{ bijektiv und kompakt mit} \\
 S(G', \varphi) \leq B
 \end{aligned} \tag{4.1}$$

Hierbei bezeichnet $S(G', \varphi)$ die Kosten von G' gemäß Definition 2.3.

(\implies). Sei eine bijektive Abbildung $\pi : V \longrightarrow \{0, \dots, n-1\}$ mit $LA_\pi(G) \leq C$ gegeben. Für $v \in V$ setze

$$\begin{aligned}
 V_L(v) &:= \{u \in V \mid \pi(u) < \pi(v), \{u, v\} \in E\} \\
 V_R(v) &:= \{w \in V \mid \pi(w) > \pi(v), \{v, w\} \in E\}
 \end{aligned}$$

$V_L(v)$ ist also die Menge der linken Nachbarn von v bzgl. π , $V_R(v)$ die der rechten Nachbarn. Für einen Knoten $v \in V$ können wir nun mittels dieser Mengen seine ausgehenden Kanten aus E_1 zerlegen in

$$E_L(v) := \bigcup_{w \in V_R(v)} \{(v, (v, w))\} \quad (|E_L(v)| =: l_v) \tag{4.2}$$

$$E_R(v) := \bigcup_{u \in V_L(v)} \{(v, (u, v))\} \quad (|E_R(v)| =: r_v) \tag{4.3}$$

Es folgt

$$|V_R(v)| = l_v \text{ und } |V_L(v)| = r_v \tag{4.4}$$

l_v bzw. r_v gibt also an, wie oft ein Knoten v linker bzw. rechter Nachbarknoten ist. Obige Mengen E_L , E_R definieren nun gerade die Links- und Rechtskanten gemäß unserer Vorüberlegungen. Hiermit sind wir in der Lage, eine bijektive und kompakte Abbildung

$$\varphi : E' \longrightarrow \{0, \dots, n \cdot (s+1) - 1\}$$

zu definieren, welche den folgenden Bedingungen genügt:

$$\varphi(E_R(v)) = \bigcup_{i=0}^{r_v-1} \{\pi(v) \cdot n \cdot (s+1) + i\} \tag{4.5}$$

$$\varphi(E_L(v)) = \bigcup_{i=0}^{l_v-1} \{(\pi(v) + 1) \cdot n \cdot (s+1) - 1 - i\} \tag{4.6}$$

$$\varphi(E_v) = \bigcup_{i=0}^{s-\deg(v)} \{\pi(v) \cdot n \cdot (s+1) + r_v + i\} \tag{4.7}$$

Wir müssen zeigen, dass unter diesen Bedingungen tatsächlich eine bijektive Abbildung definiert werden kann. Hierzu ist es hinreichend, die folgende Gleichheitskette zu beweisen:

$$\begin{aligned}
 & |E_R(v) \cup E_L(v) \cup E_v| \\
 = & |\varphi(E_R(v))| + |\varphi(E_L(v))| + |\varphi(E_v)| \\
 = & |\varphi(E_R(v)) + \varphi(E_L(v)) + \varphi(E_v)|
 \end{aligned}$$

Aus der Konstruktion unserer Kantenmengen folgt $|E_R(v) \cup E_L(v) \cup E_v| = s + 1$. Damit ergibt sich die erste Gleichheit unmittelbar aus der Definition von φ , denn es gilt $r_v + l_v + s - \deg(v) + 1 = s + 1$. Zum Nachweis der zweiten Gleichheit bemerken wir, dass die drei Bildmengen paarweise disjunkt sind.

Wie eine Abbildung φ unter diesen Voraussetzungen nun konkret aussieht, ist hier nicht von Bedeutung. Zur Berechnung der Kosten von G' bzgl. φ stellen wir folgende Überlegungen an. Zunächst erinnern wir uns, dass sich die Knotenmenge V' von G' gerade durch $V \cup E \cup D$ zusammensetzt. Für Knoten aus D und V entstehen überhaupt keine Kosten, denn diese besitzen nur eine einzige eingehende Kante (D) oder nur ausgehende Kanten (V). Die Kosten C_E der Kantenmenge E lassen sich wie folgt berechnen:

$$\begin{aligned}
 C_E & \stackrel{(4.2),(4.3)}{=} \sum_{v \in V} \sum_{e \in E_R(v)} \varphi(e) - \sum_{v \in V} \sum_{e' \in E_L(v)} \varphi(e') \\
 & = \sum_{v \in V} \left(\sum_{e \in E_R(v)} \varphi(e) - \sum_{e' \in E_L(v)} \varphi(e') \right) \\
 & \stackrel{(4.5),(4.6)}{=} \sum_{v \in V} \left(\sum_{i=0}^{r_v-1} ((\pi(v) \cdot n \cdot (s+1)) + i) \right. \\
 & \quad \left. - \sum_{j=0}^{l_v-1} ((\pi(v) + 1) \cdot n \cdot (s+1) - 1 - j) \right) \\
 & = n \cdot (s+1) \cdot \sum_{v \in V} \left(\sum_{i=0}^{r_v-1} \pi(v) - \sum_{j=0}^{l_v-1} (\pi(v) + 1) \right) \\
 & \quad + \sum_{v \in V} \left(\sum_{i=0}^{r_v-1} i + \sum_{j=0}^{l_v-1} (j+1) \right) \\
 & = n \cdot (s+1) \cdot \sum_{v \in V} (r_v \cdot \pi(v) - l_v \cdot (\pi(v) + 1)) \\
 & \quad + \sum_{v \in V} \left(\sum_{i=0}^{r_v-1} i + \sum_{j=0}^{l_v-1} (j+1) \right)
 \end{aligned}$$

Wir setzen nun

$$\begin{aligned} c_1 &:= \sum_{v \in V} \left(\sum_{i=0}^{r_v-1} i + \sum_{j=1}^{l_v-1} (j+1) \right) \\ c_2 &:= n \cdot (s+1) \cdot \sum_{v \in V} (r_v \cdot \pi(v) - l_v \cdot (\pi(v) + 1)) \end{aligned}$$

und zeigen

$$1. \ c_1 \leq s = \sum_{v \in V} \sum_{i=0}^{\deg_G(v)} i \text{ und}$$

$$2. \ c_2 = (C - m) \cdot n \cdot (s + 1)$$

Hieraus folgt dann unmittelbar die Implikation „ \implies “ unserer Behauptung (4.1). Für den ersten Punkt benutzen wir die Abschätzung

$$\sum_{i=0}^{r_v-1} i \leq \sum_{i=l_v}^{l_v+r_v-1} i \leq \sum_{i=l_v+1}^{l_v+r_v} i$$

und erhalten

$$c_1 \leq \sum_{v \in V} \left(\sum_{i=l_v+1}^{l_v+r_v} i + \sum_{j=0}^{l_v-1} (j+1) \right) = \sum_{v \in V} \sum_{i=0}^{l_v+r_v} i = \sum_{v \in V} \sum_{i=0}^{\deg(v)} i = s$$

Für Punkt 2 ist folgende Gleichheit zu zeigen

$$n \cdot (s+1) \cdot \sum_{v \in V} (r_v \cdot \pi(v) - l_v \cdot (\pi(v) + 1)) = (C - m) \cdot n \cdot (s+1),$$

nach Vereinfachung unter Beachtung von $C = LA_\pi(G)$ also

$$LA_\pi(G) = \sum_{v \in V} (r_v \cdot \pi(v) - l_v \cdot (\pi(v) + 1)) + m \quad (4.8)$$

Rechter Hand ergibt sich nach Ausmultiplizieren der l_v unter Berücksichtigung der Identität $\sum_{v \in V} l_i = m$

$$\sum_{v \in V} (r_v \cdot \pi(v) - l_v \cdot \pi(v)) - \sum_{v \in V} l_i + m = \sum_{v \in V} r_v \cdot \pi(v) - \sum_{v \in V} l_v \cdot \pi(v) \quad (4.9)$$

Linker Hand greifen wir zur Berechnung der Kosten auf die Mengen $V_L(v)$ linker Nachbarknoten von v zurück, womit sich die Kosten wie folgt berechnen lassen:

$$\begin{aligned} LA_\pi(G) &= \sum_{v \in V} \sum_{u \in V_L(v)} (\pi(v) - \pi(u)) \\ &\stackrel{(4.4)}{=} \sum_{v \in V} r_v \cdot \pi(v) - \sum_{v \in V} \sum_{u \in V_L(v)} \pi(u) \\ &\stackrel{(*)}{=} \sum_{v \in V} r_v \cdot \pi(v) - \sum_{v \in V} l_v \cdot \pi(v) \end{aligned} \quad (4.10)$$

Für (*) müssen wir uns noch von der Gleichheit

$$\sum_{v \in V} \sum_{u \in V_L(v)} \pi(u) = \sum_{v \in V} l_v \cdot \pi(v)$$

überzeugen. Um diese einzusehen, schauen wir uns zunächst die linke Seite der Gleichung an. Für jeden Knoten $v \in V$ gehen die Positionen $\pi(u)$ aller seiner linken Nachbarknoten u in die Summe ein. Dies bedeutet aber für einen Knoten u , dass dieser so oft berücksichtigt wird, wie er linker Nachbar eines Knotens $v \in V$ ist. Diese Anzahl ist aber gerade durch die Anzahl l_i seiner rechten Nachbarn gegeben, woraus die zu zeigende Gleichheit resultiert. Mit 4.9 und 4.10 haben wir also die Gleichheit in 4.8 bewiesen.

(„ \Leftarrow “). Sei nun eine Nummerierung der Kanten durch $\varphi : E' \rightarrow \{0, \dots, n \cdot (s + 1) - 1\}$ mit Kosten $S(G, \varphi) \leq (C - m) \cdot (s + 1) + s$ gegeben. Wir müssen zeigen, dass es eine Knotenanordnung $\pi : V \rightarrow \{0, \dots, n - 1\}$ gibt, welche $LA_\pi(G) \leq C$ erfüllt. Hierzu gehen wir zunächst davon aus, dass die Abbildung φ unsere obigen Bedingungen 4.5 bis 4.7 an eine Kantenummerierung erfüllt. Aufgrund der Konstruktion von φ und unseren Überlegungen im ersten Teil des Beweises können wir sofort auf die Existenz einer Knotenanordnung $\pi : V \rightarrow \{0, \dots, n - 1\}$ schließen, welche $LA_\pi(G) \leq C$ erfüllt. Hierzu setzen wir für alle $v \in V$

$$\pi(v) := \frac{\min\{\varphi(v, u) \mid (v, u) \in E'\}}{s + 1}.$$

Falls φ obige Anforderungen nicht erfüllt, so erhalten wir durch entsprechende Umnummerierungen der Kanten eine Abbildung $\varphi' : E' \rightarrow \{0, \dots, n \cdot (s + 1) - 1\}$, welche diese erfüllt. Da eine solche Anordnung geringere Kosten hat, erhalten wir mittels dieser („erst recht“) eine gewünschte Knotenanordnung π . \square

Abschließend wollen wir uns den Beweis noch an einem etwas komplexeren Beispiel verdeutlichen.

4.2.3 Ein komplexeres Beispiel

Gegeben sei folgende Instanz eines Minimum-Linear-Arrangement-Problems:

$$\begin{aligned} G &= (V, E) \text{ mit} \\ V &= \{0, 1, 2, 3, 4, 5\} \\ E &= \{\{0, 3\}, \{1, 2\}, \{1, 5\}, \{2, 3\}, \{2, 4\}, \{3, 5\}\} \\ C &= 13 \end{aligned}$$

In Abbildung 4.10 ist der Graph dargestellt. Wir wollen zeigen, wie wir ausgehend von einer Knotenanordnung eine Kantenummerierung mit den gewünschten Kosten finden.

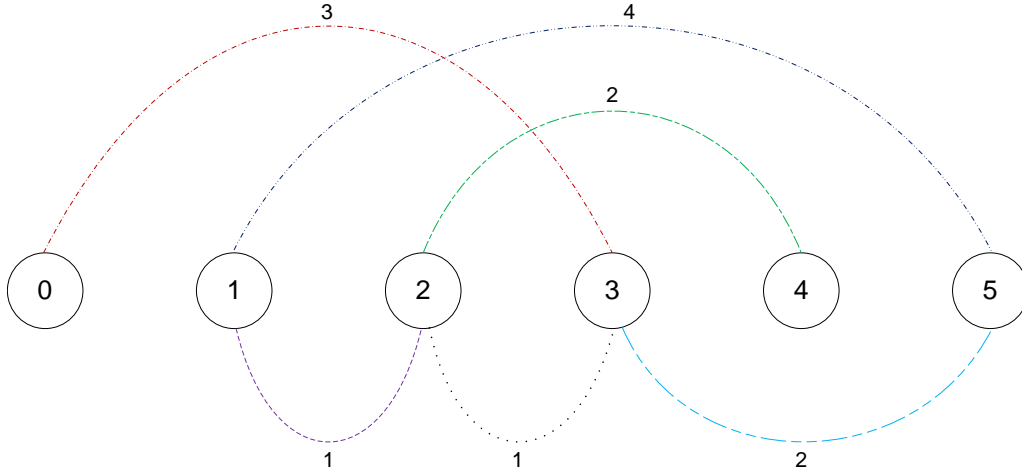


Abbildung 4.10: Instanz eines MinLA Problems mit den Kosten 13

Als lineare Anordnung der Knoten genügt hier die Identitätsabbildung ($\pi := \text{id}_V$), da sie Kosten von 13 ($\leq C$) hat.

In folgender Tabelle sind die Grade der Knoten von G dargestellt:

Knoten	0	1	2	3	4	5
Grad	1	2	3	3	1	2

Hiermit können wir nun die obere Abschätzung s für die internen Kosten berechnen:

$$s = \underbrace{(0+1)}_0 + \underbrace{(0+1+2)}_1 + \underbrace{(0+1+2+3)}_2 + \underbrace{(0+1+2+3)}_3 + \underbrace{(0+1)}_4 + \underbrace{(0+1+2)}_5 = 20$$

Nun konstruieren wir gemäß obigen Beweises eine Instanz $G' = (V', E')$ eines MIBS-Problems mit Kosten kleiner oder gleich $(C - m) \cdot (s + 1) + s = (13 - 7) \cdot 21 + 20 = 167$. Die Knotenmenge V' setzt sich aus den Knoten und Kanten von G und den Dummy-Knoten zusammen. Hierbei ist die Anzahl der Dummy-Knoten für einen Knoten v gegeben durch $s - \deg(v) + 1$. Es gilt also

$$V' = \{0, 1, 2, 3, 4, 5\} \cup \{\{0, 3\}, \{1, 2\}, \{1, 5\}, \{2, 3\}, \{2, 4\}, \{3, 5\}\} \cup D,$$

wobei die Menge der Dummy-Knoten gegeben ist durch

$$\begin{aligned}
 D = & \{d_{0,0}, \dots, d_{0,19}\} \\
 & \cup \{d_{1,0}, \dots, d_{1,18}\} \\
 & \cup \{d_{2,0}, \dots, d_{2,17}\} \\
 & \cup \{d_{3,0}, \dots, d_{3,17}\} \\
 & \cup \{d_{4,0}, \dots, d_{4,19}\} \\
 & \cup \{d_{5,0}, \dots, d_{5,18}\}
 \end{aligned}$$

Die Kantenmenge E' setzt sich aus den beiden Mengen E_I und E_D zusammen, wobei sich E_I gemäß obiger Konstruktionsvorschrift wie folgt zusammensetzt:

$$\begin{aligned}
 E_I := & \{(0, \{0, 3\}), (3, \{0, 3\})\} \\
 & \cup \{(1, \{1, 2\}), (2, \{1, 2\})\} \\
 & \cup \{(1, \{1, 5\}), (5, \{1, 5\})\} \\
 & \cup \{(2, \{2, 3\}), (5, \{2, 3\})\} \\
 & \cup \{(2, \{2, 4\}), (4, \{2, 4\})\} \\
 & \cup \{(3, \{3, 5\}), (5, \{3, 5\})\}
 \end{aligned}$$

Diese erweitern wir nun mittels der Kantenmenge E_D um die entsprechenden Dummy-Kanten, so dass die Knoten aus V einen konstanten Ausgangsgrad von $s+1 = 21$ haben. Eine Darstellung des so konstruierten Graphen findet sich in Abbildung 4.11.

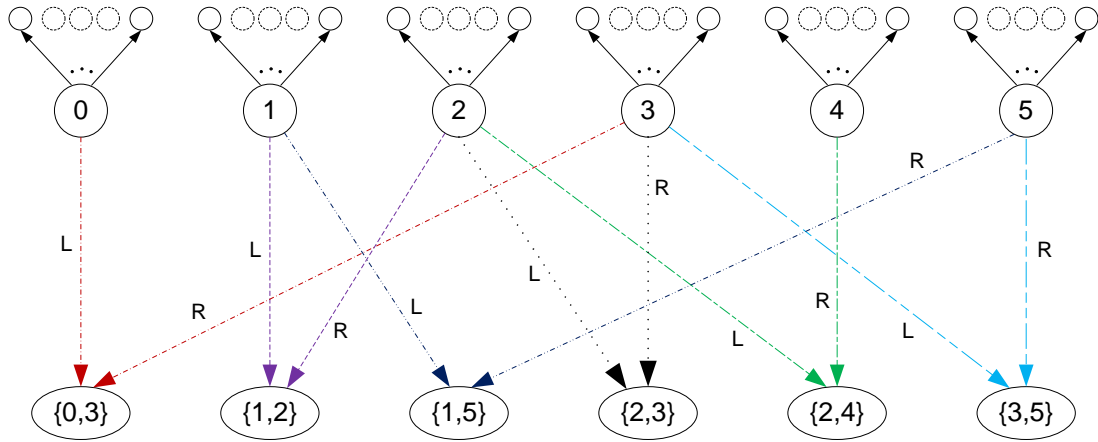


Abbildung 4.11: Zu G konstruiertes MIBS-Problem

Hier haben wir bereits eine Klassifizierung der Kanten in Linkskanten (L) und Rechtskanten (R) vorgenommen, die uns direkt eine optimale Nummerierung liefert, wie sie in Abbildung 4.12 zu finden ist.

Die Kosten dieser Probleminstanz berechnen sich nun also zu

$$S(G', \varphi) = 43 + 1 + 65 + 2 + 23 + 23 = 157$$

4 NP-Vollständigkeit

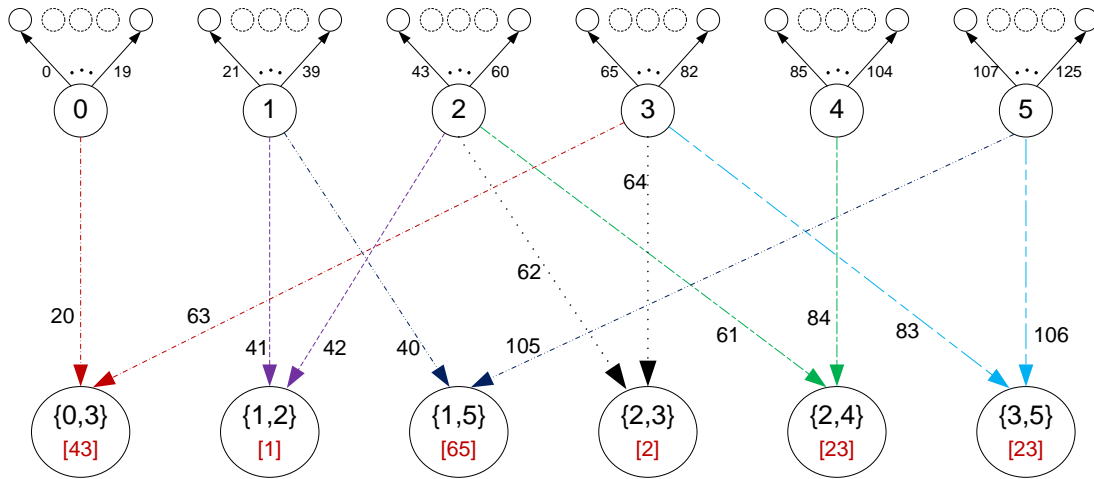


Abbildung 4.12: G' mit Kantenummerierung φ und lokalen Bandbreiten der Knoten

Wir haben also tatsächlich eine Nummerierung der Kanten mit Kosten kleiner gleich 167 gefunden.

5 Algorithmen

In diesem Kapitel wollen wir Algorithmen vorstellen, mit denen wir die Kosten eines gegebenen Graphen reduzieren können. Wie bereits gezeigt, haben wir es hier allerdings mit einem NP-harten Problem zu tun. Wir können also keineswegs erhoffen oder gar erwarten, in akzeptabler Zeit optimale Lösungen zu finden. Was wir demzufolge brauchen sind heuristische Verfahren, mit denen wir eine optimale Lösung im Sinne der obigen Kostendefinition möglichst gut annähern können. Doch was soll dieses „möglichst gut“ überhaupt bedeuten? Um dies zumindest im Ansatz klären zu können stellen wir zum Schluß noch ein exaktes Verfahren vor, welches uns dann die tatsächlich optimale Lösung liefert. Dies ist jedoch aufgrund der Schwere des Problems leider nur für relativ kleine Graphen möglich.

Wie bereits dargelegt, haben wir das Problem in zwei separate Teilprobleme zerlegt, der externen Umordnung (der Kantenblöcke) und der internen Umordnung (der Kanten). Diese werden wir auch - mit Ausnahme bei der vollständigen Suche - getrennt lösen. Auch wenn später die externe Umordnung der internen vorausgeht, wollen wir hier mit der Beschreibung letzterer beginnen, da wir hierauf nachfolgend noch Bezug nehmen werden.

5.1 Interne Kantenordnung

In diesem Abschnitt gehen wir von einer festen Anordnung der Knoten aus, wie sie beispielsweise mit später noch vorgestellten Algorithmen berechnet worden ist. Unter dieser Voraussetzung wollen wir nun versuchen, die interne Anordnung der Kanten zu optimieren. Hierzu klassifizieren wir die Kanten eines Graphen G gemäß folgender Gruppen (vgl. Definition 3.4):

Start-Kanten Menge aller minimalen eingehenden Kanten von G

End-Kanten Menge aller maximalen eingehenden Kanten von G

Zwischen-Kanten Menge aller Kanten, die weder Start- noch End-Kanten sind

Schauen wir uns hierzu einen Bandbreiten-Graphen an:

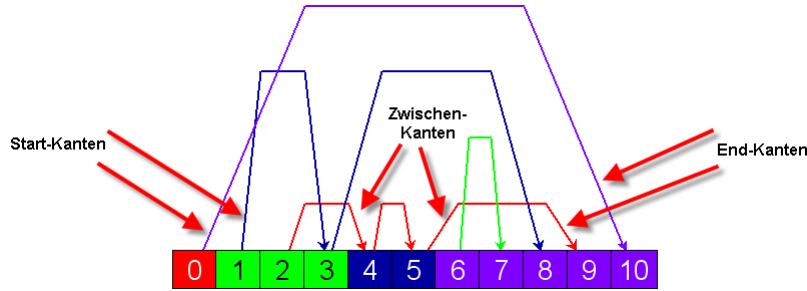


Abbildung 5.1: Klassifizierung der Kanten eines Bandbreiten-Graphen

Nun ist es ja unser Ziel, die Länge der Pfade (also die Summe über die Längen aller Kanten des Pfades) zu minimieren. Hieraus ergibt sich unmittelbar, dass wir

1. Start-Kanten möglichst weit rechts, und
2. End-Kanten möglichst weit links

innerhalb eines Knotens platzieren. Die Positionierung der Zwischen-Kanten ist völlig irrelevant, da stets die gesamte Knotenlänge zu den Gesamtkosten beiträgt. Aufgrund der Regeln 1 und 2 ergibt sich, dass wir

3. Zwischen-Kanten zwischen Start- und End-Kanten

platzieren sollten. Weiterhin sind allerdings noch Self-Loops zu berücksichtigen. Diese entstehen, falls der Ausgangsgraph parallele Kanten besitzt. Hierbei betrachten wir nun die Anzahl der Self-Loops, die mit einer Kante zusammenhängen. Gemäß dieser Anzahl sortieren wir bei den End-Kanten aufsteigend, bei den Start-Kanten absteigend. Dies ergibt sich einfach aus der Tatsache, dass wir jeweils möglichst wenig Self-Loops überspringen wollen. Bei den Zwischen-Kanten spielen auch Self-Loops keine Rolle. Insgesamt ergibt sich nun folgendes Bild für die Platzierungen der Kanten:



Abbildung 5.2: Optimale interne Kantenanordnung

Hieraus resultiert Algorithmus 5.1. Der besseren Darstellbarkeit halber verzichten wir hier auf eine Behandlung der Self-Loops und verweisen nur auf unsere Überlegungen.

Listing 5.1: Interne Ordnung der Kanten

```

1  /*
2  Eingabe: Gerichteter Graph  $G = (V, E)$ ,  $V = \{0, \dots, n-1\}$ , Anordnung der Knoten  $\pi \in S_n$ 
3  Ausgabe: Kantenfeld offset (indiziert mit Kanten  $e \in E$ )
4           offset[e] enthält den optimalen offset der Kante  $e$  bzgl.  $\pi$ 
5  */
6
7  foreach_node  $v \in V$ 
8  do
9      next_left := 0;
10     next_right := len(v) - 1;
11
12     foreach_edge  $(u, v) \in G$ 
13     do
14          $e := (u, v)$ ;
15         if "e ist Start-Kante" then
16             do
17                 offset[e] := next_left;
18                 next_left := next_left + 1;
19             od
20         if "e ist End-Kante" then
21             do
22                 offset[e] := next_right;
23                 next_right := next_right - 1;
24             od
25         else do
26             inside := inside  $\cup \{e\}$ ;
27         od
28     od
29     foreach  $e \in \text{inside}$ 
30     do
31         offset[e] := next_left;
32         next_left := next_left + 1;
33     od;

```

Intuitiv sollte klar sein, dass die hierdurch erzeugte Kantenanordnung optimal ist. Auf einen formalen Beweis wollen wir deshalb an dieser Stelle verzichten.

Wir kommen nun zu den Algorithmen, die sich mit der Umordnung der Knoten beschäftigen.

5.2 Greedy-Algorithmus

Zu Beginn wollen wir einen einfachen Greedy-Algorithmus vorstellen. Hierbei konstruieren wir unseren Graphen sukzessive durch Hinzunahme einzelner Knoten. Für einen hinzuzufügenden Knoten wählen wir jeweils eine gültige Position, welche minimale Gesamtkosten verursacht.

Sei nun also $G = (V, E)$ unser Ausgangsgraph, aus dem wir sukzessive unseren Bandbreiten-Graphen S konstruieren. Es gelte $V = \{1, \dots, n\}$, $\text{outedges}(i) = \{e_{i,1}, \dots, e_{i,k_i}\}$, $i = 0, \dots, n-1$ (also $k_i = \text{outdeg}(i)$).

Der folgende Algorithmus liefert eine Ordnung der Knoten gemäß oben skizzierter Greedy-Strategie.

Listing 5.2: Greedy Algorithmus

```

1  S = ∅;
2  for i = 0 to n-1
3  do
4      P[i] := i;
5      V(S) := V(S) ∪ {vi} // erweitere Knotenmenge von S, sei V(i) := vi
6      for j = 1 to ki do
7          E(S) := E(S) ∪ {eij} // erweitere Kantenmenge von S
8
9      // Finde günstigste Position für Knoten vi
10     min_cost = cost(S);
11     min_pos = P[i];
12     for j = 1 to i
13     do
14         Dekrementiere Knoten vi;
15         Aktualisiere P gemäß Definition 3.8;
16         Aktualisiere Kantenverkettung von S;
17         if (cost(S) < min_cost) then
18             do
19                 min_pos = P[i];
20                 min_cost = cost(S);
21             od
22     od
23
24     // Bewege Knoten vi an günstigste Position min_pos
25     for k = 1 to min_pos (k ≥ 1)
26     do
27         Inkrementiere Knoten vi;
28         Aktualisiere P gemäß Definition 3.8;
29         Aktualisiere Kantenverkettung;
30     od
31 od

```

Wir wollen nun die Laufzeit des vorgestellten Algorithmus analysieren. Wie man leicht sieht, wird diese Laufzeit dominiert von der **for**-Schleife in den Zeilen 12-22. Das Dekrementieren des Knotens v_i wird bestimmt durch die Neuverkettung der Kanten. Diese benötigt Zeit $O(\text{outdeg}(v_i) + \text{outdeg}(\text{pred}(v_i)))$. Hiermit ergibt sich als Gesamtlaufzeit für die Schleife

$$O\left(\sum_{j=0}^{i-1} \text{outdeg}(v_i) + \text{outdeg}(v_j)\right) = O\left(i \cdot \text{outdeg}(v_i) + \sum_{j=0}^{i-1} \text{outdeg}(v_j)\right)$$

Insgesamt ergibt sich damit als Laufzeit also

$$\begin{aligned}
& O\left(\sum_{i=0}^{n-1}\left(i \cdot \text{outdeg}(v_i) + \sum_{j=0}^{i-1} \text{outdeg}(v_j)\right)\right) \\
&= O\left(\sum_{i=0}^{n-1}(i \cdot \text{outdeg}(v_i)) + \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} \text{outdeg}(v_j)\right) \\
&= O\left(\sum_{i=0}^{n-1}(n \cdot \text{outdeg}(v_i)) + \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \text{outdeg}(v_j)\right) \\
&= O\left(n \cdot \sum_{i=0}^{n-1}(\text{outdeg}(v_i)) + \sum_{i=0}^{n-1} m\right) \\
&= O(n \cdot m + n \cdot m) \\
&= O(n \cdot m)
\end{aligned}$$

5.3 Iteration nach dem arithmetischen Mittel

Nachteilig an vorigem Algorithmus ist seine schlechte Laufzeit. Für jeden Knoten werden alle möglichen Positionen durchprobiert. Dies wollen wir in diesem Abschnitt ändern. Doch wenn wir nicht mehr alle Positionen durchprobieren dürfen, wohin verschieben wir den Knoten dann? Wir formulieren als grobes Ziel, dass wir die Kanten auf allen Ebenen möglichst dicht zusammenführen wollen. Hierzu müssen wir die entsprechenden Knoten irgendwie bzgl. ihrer Kanten bewegen. Als Kriterium wählen wir hierfür das arithmetische Mittel der Längen aller nach rechts und links führenden Kanten. Erstere werden hierbei positiv, letztere negativ bewertet.

Schauen wir uns zunächst an, wie wir das arithmetische Mittel berechnen.

Listing 5.3: Arithmetic Mean

```

1 arithmetic_mean(node v_i)
2 {
3     for j = 0 to k_i
4     do
5         target = target(e_ij);
6         dist = pos(target) - pos(v_i);
7         dist_sum = dist_sum + dist;
8     od
9     return (dist_sum / k_i);
10 }
```

Nun müssen wir noch klären, wie wir mit Hilfe dieses Mittelwertes eine gültige Platzierung P in eine *gültige* Platzierung P' überführen. Seien hierzu $G = (V, E)$ mit

5 Algorithmen

$V = \{v_0, \dots, v_{n-1}\}$ und eine gültige Platzierung $\text{pos} : V \rightarrow \mathbb{Z}$ von V gegeben. Betrachten wir den Fall, dass wir den Knoten $v_i \in V$ um „etwa“ $\delta > 0$ nach rechts verschieben wollen. Eine Verschiebung um $\delta < 0$ ist völlig symmetrisch und wird hier nicht weiter behandelt.

Zunächst definieren wir die Menge

$$U := \{v_j \in V \mid \text{pos}(v_i) < \text{pos}(v_j) < \text{pos}(v_i) + \delta\}$$

der zwischen der Start- und der Zielposition von v_i liegenden Knoten. Weiterhin seien

$$v_{\max} \in U \text{ mit } \forall u \in U, u \neq v_{\max} \quad u < v_{\max}$$

das Maximum bzgl. der Position und

$$s := \sum_{u \in U} \text{len}(u)$$

die Summe der Knotenlängen dieser Menge. Hiermit bestimmen wir nun

$$m := \begin{cases} |U| & \text{falls } \delta - s > \frac{\text{len}(\text{succ}(v_{\max}))}{2} \\ |U| + 1 & \text{falls } \delta - s \leq \frac{\text{len}(\text{succ}(v_{\max}))}{2} \end{cases},$$

die Anzahl der auszuführenden Inkrement-Operationen für den Knoten v . Abbildung 5.3 soll die Situation verdeutlichen.

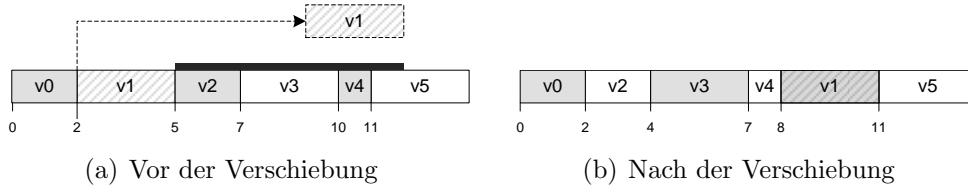


Abbildung 5.3: `move_node(v1, 7)`

Betrachten wir zunächst die Ausgangssituation in 5.3(a). Der Knoten **v1** soll um 7 Einheiten nach rechts verschoben werden. Diese Distanz ist ab dem Nachfolger **v2** schwarz markiert. Die vollständig abgedeckten Knoten sind gerade diejenigen der Menge U , wobei gilt $v_{\max} = \mathbf{v4}$. Der Knoten **v5** ist nur zu einem Drittel markiert. Schauen wir uns hierzu die Definition von m an. Da dieser Knoten nicht mindestens zur Hälfte abgedeckt ist, findet hier keine weitere Inkrement-Operation für **v1** mehr statt, die neue Position ist also *vor* **v5**, wie in 5.3(b) dargestellt.

Mit diesen Grundlagen können wir nun folgenden Algorithmus formulieren. Wir beschränken uns wieder auf eine Verschiebung nach rechts.

Listing 5.4: Move Node

```

1 move_node(v, delta)
2 {
3     m := 0;
```

```

4
5  if (delta > 0) then
6  do
7      u := v;
8      total_move := 0;
9      while (total_move + len(succ(u)) < delta)
10     do
11         m := m + 1;
12         u := succ(u);
13         total_move += len(u);
14     od
15     if (delta - total_move > length(succ(u)) / 2) then
16         m := m + 1;
17     }
18
19     // Fall (delta < 0) symmetrisch
20
21     move_node_to_rank(v, rank(v) + m);
22 }

```

Jetzt haben wir alles für unseren iterativen Algorithmus nach dem arithmetischen Mittel beisammen.

Listing 5.5: Arithmetic Mean Iteration

```

1  for (int r = 0; r < rounds; r++)
2  {
3      foreach_node(v, G)
4      do
5          med = arithmetic_mean(v);
6          G.move_node(v, med);
7      od
8  }

```

Es bleibt zu klären, wie wir die Anzahl der Runden bestimmen sollen. Wir könnten versuchen, experimentell einen geeigneten Wert zu finden. Ein anderer Ansatz besteht darin, einen Knoten nur dann zu verschieben, wenn damit eine echte Verbesserung erzielt wird. Dies hat zur Folge, dass das Verfahren zwangsläufig konvergiert. Hierbei läßt sich die Konvergenz einfach dadurch feststellen, dass in einem kompletten Durchlauf keine Knotenverschiebung mehr stattfindet.

5.4 Dynamische Programmierung

Wie in der Einführung angekündigt, wollen wir nun noch ein exaktes Verfahren vorstellen, mit dem wir die tatsächlich beste Lösung berechnen können. Eine leichte Abwandlung dieses Verfahrens liefert uns dann gleichzeitig die schlechteste Lösung, was uns eine Einordnung der zuvor vorgestellten Algorithmen ermöglicht. Doch wie finden wir eine optimale Lösung dieses NP-harten Problems? In einem naiven Ansatz könnten wir

5 Algorithmen

hergehen, und nacheinander für alle möglichen Permutationen der Knoten die Kosten berechnen. Die Laufzeit würde jedoch fakultativ mit der Anzahl der Knoten n wachsen, also nur durch ein $O(n!)$ nach oben beschränkt sein. Wir wollen untersuchen, ob sich die Laufzeit nicht wenigstens auf ein exponentielles Wachstum beschränken läßt. Ein Blick auf Tabelle 5.1 verdeutlicht den Gewinn durch ein solches Verfahren.

n	$n!$	2^n
10	3622880	1024
12	479001600	4096
14	87178291200	16384
16	20922789888000	65536
18	6402373705728000	262144
20	2432902008176640000	1048576
22	1124000727777607680000	4194304

Tabelle 5.1: Fakultatives vs. Exponentielles Wachstum

Beginnen wir mit einer Definition, auf welcher die Kernidee unseres Algorithmus basiert.

Definition 5.1. Sei $G = (V, E)$ ein gerichteter Graph, $\pi : V \longrightarrow \{0, \dots, n-1\}$ eine Anordnung der Knoten und $\text{len} : V \longrightarrow \mathbb{N}$ definiert durch $v \mapsto \text{outdeg}(v)$, $v \in V$. pos_π sei die von π induzierte Platzierung von V (vgl. Definition 3.6). Für alle $v \in V$ setze $m_v := \min_neighbor(v)$, $M_v := \max_neighbor(v)$. Weiterhin sei eine Zerlegung der Knotenmenge in $L = \{v_0, \dots, v_{l-1}\}$ und $R = \{v_l, \dots, v_{n-1}\}$ gegeben. Die **lokalen Näherungskosten von L bzgl. π** sind definiert durch

$$\begin{aligned} \text{approx_cost}_\pi(G|L) := \sum_{v \in V} & \left(\text{pos}_\pi(\min\{v_{l-1}, M_v\}) - \text{pos}_\pi(\min\{v_{l-1}, m_v\}) \right. \\ & \left. + \frac{1}{2} (\text{len}(\min\{v_{l-1}, M_v\}) - \text{len}(\min\{v_{l-1}, m_v\})) \right). \end{aligned}$$

Analog sind die **lokalen Näherungskosten von R bzgl. π** definiert durch

$$\begin{aligned} \text{approx_cost}_\pi(G|R) := \sum_{v \in V} & \left(\text{pos}_\pi(\max\{v_{l-1}, M_v\}) - \text{pos}_\pi(\max\{v_{l-1}, m_v\}) \right. \\ & \left. + \frac{1}{2} (\text{len}(\max\{v_{l-1}, M_v\}) - \text{len}(\max\{v_{l-1}, m_v\})) \right). \end{aligned}$$

Hierbei gilt $\min\{u, v\} := \min_\pi\{u, v\} := \begin{cases} u & \pi(u) < \pi(v) \\ v & \text{sonst} \end{cases}$.

Analog ist \max definiert.

Hiermit gilt der folgende, einfache

Satz 5.1. *Seien ein gerichteter Graph $G = (V, E)$, eine Anordnung der Knoten $\pi : V \longrightarrow \{0, \dots, n-1\}$ und eine Zerlegung der Knotenmenge in L und R gegeben. Dann gilt:*

$$\text{approx_cost}_\pi(G) = \text{approx_cost}_\pi(G|L) + \text{approx_cost}_\pi(G|R)$$

Beweis.

$$\begin{aligned} & \text{approx_cost}_\pi(G|L) + \text{approx_cost}_\pi(G|R) \\ &= \sum_{v \in V} \left(\text{pos}_\pi(v_{l-1}) + \text{pos}_\pi(M_v) - \text{pos}_\pi(v_{l-1}) - \text{pos}_\pi(m_v) \right. \\ & \quad \left. + \frac{1}{2} (\text{len}(v_{l-1}) + \text{len}(M_v) - \text{len}(v_{l-1}) - \text{len}(m_v)) \right) \\ &= \sum_{v \in V} \left(\text{pos}_\pi(M_v) - \text{pos}_\pi(m_v) + \frac{1}{2} (\text{len}(M_v) - \text{len}(m_v)) \right) \\ &= \text{approx_cost}_\pi(G) \end{aligned}$$

□

Als Folgerung hieraus ergibt sich unmittelbar:

Folgerung. Die lokalen Näherungskosten von L sind völlig unabhängig von der Anordnung der Knoten in R .

Wie können wir uns dieses nun zu Nutze machen? Betrachten wir hierzu eine Teilmenge der Knoten $S \subset V$, welche auf den Plätzen $\{0, \dots, |S| - 1\}$ angeordnet werden soll. Ist eine solche Anordnung π_S bzgl. der lokalen Näherungskosten von S optimal, so ist dies auch in globaler Hinsicht der Fall, was folgendes bedeutet: Haben wir eine optimale Gesamtlösung, bei der eben diese Teilmenge S auf den Plätzen $\{0, \dots, |S| - 1\}$ angeordnet wird, so können wir die Knoten von S auch gemäß π_S anordnen, ohne dass sich an den Gesamtkosten etwas ändert.

Dies ist auch das Kernstück unseres Algorithmus. Wir berechnen sukzessive für die ersten i Plätze ($i = 1, \dots, |V|$) die optimale Anordnung einer i -elementigen Teilmenge von V . Hierbei greifen wir jeweils auf die beste Anordnung der entsprechenden $i - 1$ elementigen Menge zurück.

Listing 5.6: Dynamisches Programmieren

```

1  /*
2  Eingabe: Gerichteter Graph  $G = (V, E)$ ,  $V = \{0, \dots, n-1\}$ 
3  Ausgabe: Vektor ordering – enthält optimale Reihenfolge der Knoten.
4
5  Datenstruktur:
6  Felder cost und right_vtx, jeweils indiziert mit Knotenmengen  $S \subset V$ 
7  cost[ $S$ ] enthält die (aktuell) optimalen Kosten von  $S$ 
8  right_vtx[ $S$ ] enthält den rechtesten Knoten von  $S$  (an Stelle  $|S|-1$ )
9  2-dimensionales Feld outside, indiziert mit Knotenmengen  $S \subset V$  und Knoten  $v \in V$ 
10 outside[ $S, v$ ] enthält Anzahl der eingehenden Kanten von  $v$ , welche nicht in  $S$  beginnen
11 */
12
13 // Initialisierung
14 foreach  $S \subset V$ 
15 do
16     cost[ $S$ ] =  $\infty$ ;
17     foreach_node  $v \in V$  do
18         outside[ $S, v$ ] = indeg( $v$ );
19     od
20 cost[ $\emptyset$ ] = 0;
21
22 // Hauptschleife
23 for  $j = 0$  to  $n-1$ 
24 do
25     foreach  $S \subset V$ ,  $|S| = j$ 
26     do
27         foreach_node  $u \in V \setminus S$ 
28         do
29             Berechne Kosten new_cost von  $S \cup \{u\}$ ;
30             if (new_cost < cost[ $S \cup u$ ]) then
31                 do
32                     cost[ $S \cup \{u\}$ ] := new_cost;
33                     right_vtx[ $S \cup \{u\}$ ] :=  $u$ ;
34                     foreach_node  $v \in V$ 
35                     do
36                         if  $((v, u) \in E)$  then do
37                             outside[ $S \cup \{u\}, v$ ] := outside[ $S, v$ ] - 1;
38                         else do
39                             outside[ $S \cup \{u\}, v$ ] := outside[ $S, v$ ];
40                         od
41                     fi
42                 od
43             od
44         od
45     od
46 // Bestimme optimale Reihenfolge der Knoten
47  $S := V$ ;
48 for  $i = 0$  to  $n-1$ 
49 do
50      $v = \text{right\_vtx}(S)$ ;
51     ordering[ $i$ ] =  $v$ ;
52      $S := S - \{v\}$ ;
53 od

```

Es bleibt zu klären, wie wir in Zeile 29 die neuen Kosten **new_cost** von $S \cup \{u\}$ berechnen. Wir wollen kurz anmerken, dass wir uns mit Kosten hier auf die Lücken eingehender Kanten beziehen (siehe auch Bemerkung zu Definition 2.2). Zur Berechnung dieser Ko-

5 Algorithmen

sten durchlaufen wir alle Knoten $w \in V$ und unterscheiden jeweils sechs Fälle.

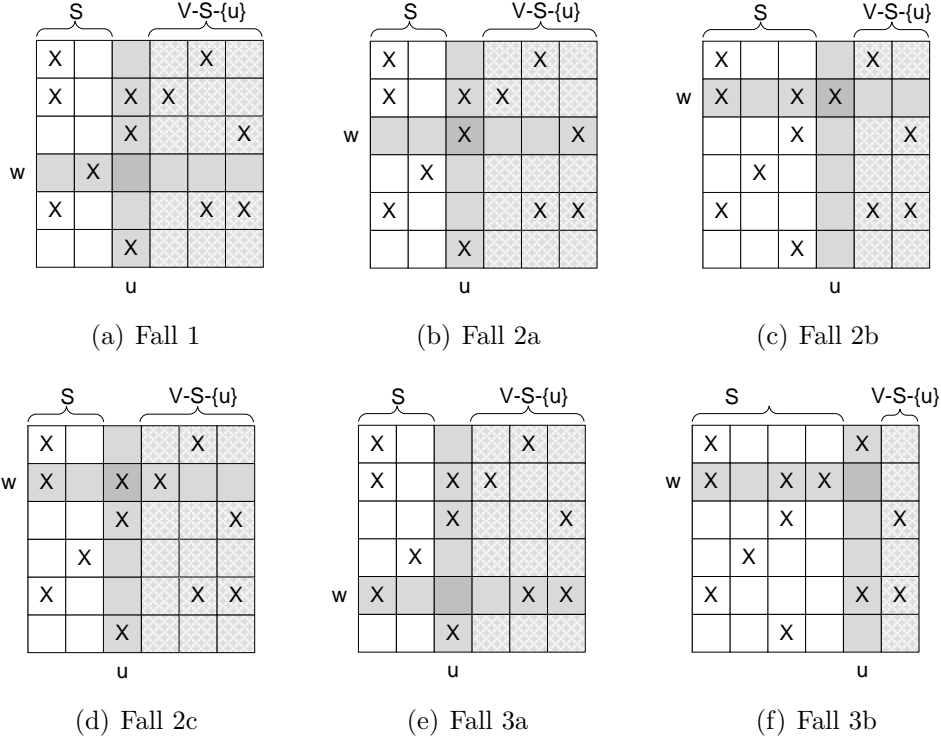


Abbildung 5.4: Fallunterscheidung für (u, w)

1. $\text{indeg}(w) \leq 1$
Für maximal eine eingehende Kante entstehen keine Kosten.
2. $\text{indeg}(w) > 1$ und $(u, w) \in E$
 - a) $\nexists v \in S$ mit $(v, w) \in E$
 (u, w) ist also minimale eingehende Kante von w (Start-Kante). Die Kosten ergeben sich aus den Überlegungen in Abschnitt 5.1.
 - b) $\nexists v \in V - (S \cup \{i\})$ mit $(v, w) \in E$
 (u, w) ist also maximale eingehende Kante von w (End-Kante). Die Kosten ergeben sich aus den Überlegungen in Abschnitt 5.1.
 - c) $\exists v_1 \in S, v_2 \in V - (S \cup \{i\})$ mit $(v_1, w), (v_2, w) \in E$
 (u, w) ist also Zwischen-Kante von w . Die Kosten ergeben sich aus der Länge von u . Da wir hier die Lücken betrachten, erhöhen sich selbige um $\text{len}(u) - 1$.
3. $\text{indeg}(w) > 1$ und $(u, w) \notin E$

- a) $\exists v_1 \in S, v_2 \in V - (S \cup \{w\})$ mit $(v_1, w), (v_2, w) \in E$
 w hat also Nachbarknoten bzgl. eingehender Kanten, welche größer bzw. kleiner als u sind. Damit erhöhen sich die Kosten um $\text{len}(u)$.
- b) $\forall v \in V$ mit $(v, w) \in E$ gilt $v \in S$ oder
 $\forall v \in V$ mit $(v, w) \in E$ gilt $v \in V - S$ w hat also nur Nachbarknoten bzgl. eingehender Kanten, welche alle größer oder alle kleiner als u sind. Es entstehen somit keine Kosten.

Wir wollen uns diese Fallunterscheidung am Beispiel der Adjazenzmatrixdarstellung unseres Beispielgraphen G aus Abschnitt 3.1 in Abbildung 5.4 verdeutlichen. Es sei daran erinnert, dass ein X in Spalte u und Zeile w für eine Kante (u, w) von G steht.

Es stellt sich schließlich die Frage, wie wir diese Fälle in unserem Algorithmus unterscheiden können. Für einen Knoten $w \in V$ ist dies möglich mit der Anzahl $\text{outside}[S, w]$, der außerhalb von S beginnenden eingehenden Kanten von w .

Zum Beispiel ist eine Kante $(u, w) \in E$ genau dann die erste eingehende Kante von w , wenn alle eingehenden Kanten von w außerhalb der schon betrachteten Menge S beginnen, wenn also gilt $\text{outside}[S, w] = \text{indeg}(w)$. Die weiteren Fälle gehen aus Listing 5.7 hervor.

Listing 5.7: Berechnung der Kosten **new_cost** von $S \cup \{u\}$

```

1  foreach_node w ∈ V
2  do
3      // Fall 1
4      if (indeg(w) ≤ 1) then do continue;
5
6      // Fall 2
7      if ((u, w) ∈ E) then
8      do
9
10         // Fall 2a
11         if (outside[S, w] == indeg(w) then do
12             new_cost := new_cost + start_edge_cost;
13             start_edge_cost := start_edge_cost + 1;
14         od
15
16         // Fall 2b
17         else if (outside[S, w] == 0) then do
18             new_cost := new_cost + end_edge_cost;
19             end_edge_cost := end_edge_cost + 1;
20         od
21
22         // Fall 2c
23         else do
24             new_cost := new_cost + len(u) - 1;
25         od
26     od
27
28     // Fall 3
29     else // (u, w) ∉ E
30     do

```

5 Algorithmen

```
31
32     // Fall 3a
33     if (0 < outside[S,w] < indeg(w)) then do
34         new_cost := new_cost + len(u);
35     od
36
37     // Fall 3b
38     else do continue;
39 od
40 od
```

6 Experimente

In diesem Kapitel wollen wir die vorgestellten Algorithmen auswerten. Zum einen interessiert uns hier natürlich die (verbesserte) Bandbreitensumme. Konnten wir tatsächlich eine signifikante Verbesserung erreichen? Wie unterscheidet sich der Gewinn bei unterschiedlichen Klassen von Graphen bzw. Netzwerken?

Unser primäres Ziel war es jedoch nicht, diese Bandbreitensumme zu reduzieren, sondern ein besseres Cache-Verhalten zu erreichen. Der Frage, ob und inwieweit wir auch dieses Ziel erreichen konnten, werden wir durch weitere Experimente nachgehen.

6.1 Partiiell vollständige Max-Flow-Probleme

Wir stellen als erstes einen Generator vor, der uns spezielle Netzwerke mit zwei verschiedenen Kantenummerierungen erzeugt. Eine ist bezüglich der Bandbreitensumme besonders gut, die andere ist besonders schlecht. Schauen wir uns zunächst an, wie der zugrundeliegende Graph $G = (V, E)$ aussieht. Hierzu sei eine $(n \cdot m + 2)$ -elementige Knotenmenge gegeben durch

$$V = \{s, t\} \cup \bigcup_{i=0}^{m-1} \bigcup_{j=0}^{n-1} \{v_{ij}\}$$

Hierauf ist die Kantenmenge $E := E_s \cup E_r \cup E_c \cup E_t$ definiert durch

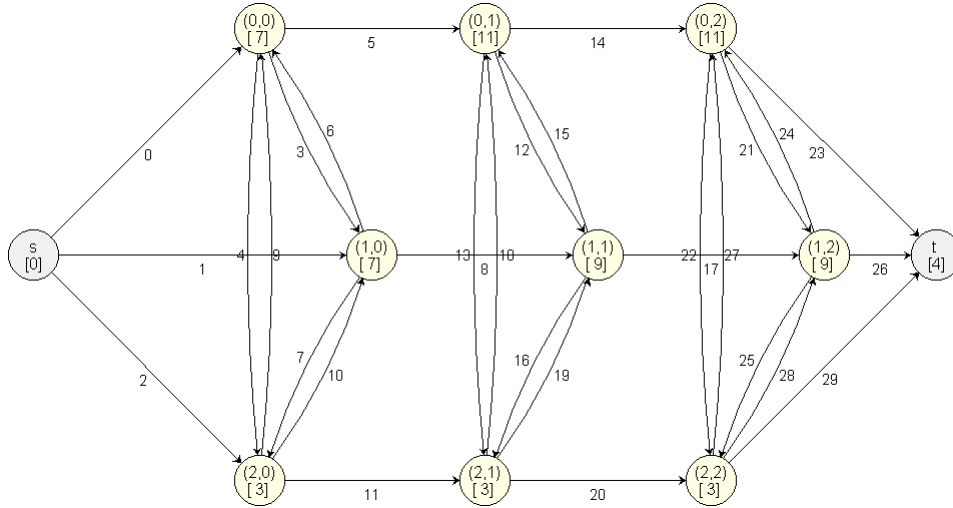
$$\begin{aligned} E_s &:= \{(s, v_{i,0}) | 0 \leq i \leq m-1\} \\ E_r &:= \{(v_{i,j}, v_{i,j+1}) | 0 \leq i \leq m-1, 0 \leq j \leq n-2\} \\ E_c &:= \{(v_{l,j}, v_{k,j}) | 0 \leq l, k \leq m-1, l \neq k, 0 \leq j \leq n-1\} \\ E_t &:= \{(v_{i,n-1}, t) | 0 \leq i \leq m-1\} \end{aligned}$$

Wir können uns die Knoten des Graphen (ohne s und t) also in einer $n \times m$ -Matrix angeordnet vorstellen. Innerhalb der Zeilen sind die Knoten durch ausgehende Kanten mit ihren jeweiligen rechten Nachbarn verbunden (E_r). Die Spalten bilden vollständige Teilgraphen (E_c). Für ein Max-Flow-Netzwerk benötigen wir noch die beiden ausgezeichneten Knoten s und t , welche die Quelle bzw. Senke darstellen. s erhält zu jedem Knoten

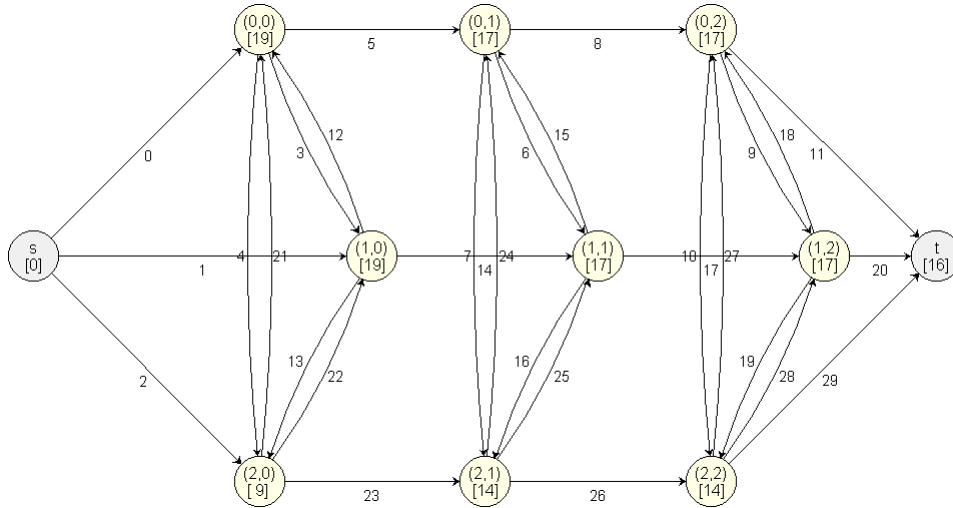
6 Experimente

der ersten Spalte noch eine ausgehende Kante (E_s), entsprechend liefert jeder Knoten der letzten Spalte eine zusätzliche eingehende Kante für t (E_t). Schließlich versehen wir die Kanten noch mit zufälligen Kapazitätswerten.

Um nun eine „gute“ Nummerierung zu erhalten, durchlaufen wir die Knoten des Graphen spaltenweise und vergeben jeweils fortlaufende Nummern für die ausgehenden Kanten. Eine zeilenweise Traversierung führt zu einer entsprechend „schlechten“ Nummerierung. Wir wollen uns dies an einem Beispiel in Abbildung 6.1 verdeutlichen. Als Kosten haben wir hier die Lücken der eingehenden Kanten gewählt. Entsprechend sind in den Knoten jeweils die Summen der Lücken eingehender Kanten eingetragen.



(a) Spaltenweise Kantennummerierung: Kosten 67



(b) Zeilenweise Kantennummerierung: Kosten 159

Abbildung 6.1: „Gute“ und „schlechte“ Kantennummerierung eines 3×3 -Netzwerkes

Tabelle 6.1 zeigt eine Übersicht der hier verwendeten Netzwerke:

Name	Nodes	Edges	Indeg	Good	Bad	Random
mf10x10	102	1010	9.90	12716	89990	83942
mf20x20	402	8020	19.95	220931	3039980	2915987
mf30x30	902	27030	29.97	1149646	23489970	22921314
mf40x40	1602	64040	39.98	3683861	99839960	97564392
mf50x50	2502	125050	49.98	9068576	306249950	300948283
mf100x100	10002	1000100	99.99	147524651	9899999900	9803196098
mf150x150	22502	3375150	149.99	750993226	75431249850	74946113833

Tabelle 6.1: Max-Flow-Netzwerke

Good und *Bad* bezeichnen hier die Kosten des entsprechenden Netzwerks mit spaltenweiser bzw. zeilenweiser Nummerierung. Die Spalte *Random* enthält die Kosten einer zufälligen Kantenummerierung. Wir bemerken, dass sich diese sehr nahe an den Kosten der schlechten Anordnung bewegen. Und je größer das Netzwerk wird, desto geringer wird die relative Abweichung.

Als erstes wollen wir nun unseren Greedy-Algorithmus anwenden. Diesen führen wir sowohl auf der schlechten Kantenummerierung aus, als auch auf einer zufälligen Permutation. Die Ergebnisse sind in den Tabellen 6.2 und 6.3 dargestellt. Hierbei gibt *Orig Cost* die ursprünglichen Kosten der schlechten bzw. randomisierten Nummerierung und *Imp Cost* die verbesserten Kosten nach unserem Greedy-Algorithmus an. *Imp* repräsentiert die relative Verbesserung in Prozent. In der Spalte *Time* sind die Laufzeiten angegeben, die unser Algorithmus benötigt hat. Sämtliche Tests wurden auf einem Rechner mit einer Intel Pentium 4 CPU mit 2.99 GHz Taktfrequenz und 2 GB Hauptspeicher ausgeführt.

Name	Nodes	Edges	Orig Cost	Imp Cost	Imp	Time
mf10x10.bad	102	1010	89990	12636	85.96	0.02s
mf20x20.bad	402	8020	3039980	220571	92.74	0.06s
mf30x30.bad	902	27030	23489970	1148806	95.11	0.39s
mf40x40.bad	1602	64040	99839960	3682341	96.31	1.75s
mf50x50.bad	2502	125050	306249950	9066176	97.04	5.45s
mf100x100.bad	10002	1000100	9899999900	147514851	98.51	165.09s
mf150x150.bad	22502	3375150	75431249850	750971026	99.00	1265.00s

Tabelle 6.2: Greedy-Konstruktion auf schlechter Kantenummerierung

Es fällt auf, dass der Algorithmus bei schlechter Ausgangsnummerierung bessere Ergebnisse liefert als bei zufälliger (besserer) Ausgangsnummerierung.

Im Folgenden wollen wir nun anhand eines Max-Flow-Algorithmus (MFS) untersuchen, ob sich die deutlich verbesserte Bandbreite auch tatsächlich in einer verbesserten Laufzeit widerspiegelt. Hierbei beschränken wir uns auf die beiden großen Netzwerke, da die Laufzeiten der kleineren zu gering und damit überhaupt nicht aussagekräftig sind. Die Ergebnisse der Messungen sind in Tabelle 6.4 dargestellt.

Name	Nodes	Edges	Orig Cost	Imp Cost	Imp	Time
mf_10_10.rnd	102	1010	82007	19991	75.62	0.02s
mf_20_20.rnd	402	8020	2868382	540720	81.15	0.09s
mf_30_30.rnd	902	27030	22899776	3578986	84.37	0.51s
mf_40_40.rnd	1602	64040	97798414	15276187	84.38	2.09s
mf_50_50.rnd	2502	125050	299311462	43071885	85.61	6.17s
mf_100_100.rnd	10002	1000100	9810878848	1287667770	86.88	178.80s
mf_150_150.rnd	22502	3375150	74977430618	8252768640	88.99	1339.30s

Tabelle 6.3: Greedy-Konstruktion auf randomisierter Kantenummerierung

Name	Bad	Good	Greedy (Bad)	Greedy (Rnd)
mf_100_100	0.17	0.09	0.09	0.09
mf_150_150	0.64	0.39	0.39	0.45

Tabelle 6.4: Laufzeiten MFS in s

Die Spalten *Bad* und *Good* zeigen die Laufzeiten des Max-Flow-Algorithmus auf den entsprechend generierten Netzwerken. *Greedy (Bad)* und *Greedy (Rnd)* stellen die Laufzeiten auf den durch den Greedy-Algorithmus verbesserten Ausgangsnetzwerken dar. Wir sehen also, dass die gute Nummerierung des Generators gegenüber der schlechten zu einer um den Faktor 1,6 bis 1,9 verbesserten Laufzeit führt. Gleiche Geschwindigkeitssteigerungen konnten bei unseren optimierten Netzwerken erzielt werden, sofern diese auf der schlechten Nummerierung gestartet sind. Basierend auf der zufälligen Kantenummerierung konnte immerhin noch eine um den Faktor 1,8 bis 1,9 verbesserte Laufzeit erzielt werden. Wir sehen also, dass mit der verbesserten Bandbreite auch tatsächlich eine verbesserte Laufzeit einhergeht, welche auf eine effizientere Nutzung der Caches zurückzuführen ist.

Nun wurde der hier vorgestellte Generator speziell für unsere Problematik konzipiert. Im nächsten Abschnitt wollen wir Netzwerke heranziehen, die ihren Ursprung außerhalb dieses Kontextes haben und untersuchen, ob sich auch hier signifikante Verbesserungen erzielen lassen.

6.2 Genrmf Generator

Für unsere nächsten Tests nutzen wir einen Generator, wie er von Goldfarb und Grigoriadis entwickelt wurde. Eine Beschreibung dieses Generators findet sich in „A Computational Comparison of the Dinic and Network Simplex Methods for Maximum Flow“ [11]. Für weiterführende Informationen sei auch auf http://www.avglab.com/andrew/CATS/maxflow_synthetic.htm verwiesen. Als Ausgangssituation dient hier eine zufällige Nummerierung der Kanten. Tabelle 6.2 stellt eine Übersicht der verwendeten Netz-

werke dar.

Name	Nodes	Edges	Indeg	Rand
genrmf.128.1.rnd	16384	65024	3.97	635439394
genrmf.128.2.rnd	32768	146432	4.47	3032442298
genrmf.128.3.rnd	49152	227840	4.64	7199231471
genrmf.128.4.rnd	65536	309248	4.72	13119318950
genrmf.128.5.rnd	81920	390656	4.77	20828803678
genrmf.128.6.rnd	98304	472064	4.80	30332206807
genrmf.128.7.rnd	114688	553472	4.83	41523909029
genrmf.128.8.rnd	131072	634880	4.84	54624078440
genrmf.128.9.rnd	147456	716288	4.86	69354980601
genrmf.128.10.rnd	163840	797696	4.87	85854378635

Tabelle 6.5: Genrmf-Netzwerke

Unser Greedy-Algorithmus liefert die in Tabelle 6.6 zusammengefassten Ergebnisse.

Name	Nodes	Edges	Orig Cost	Imp Cost	Imp	Time
genrmf.128.1	16384	65024	635439394	221973067	65.07	57.69s
genrmf.128.2	32768	146432	3032442298	1478828006	51.23	297.86s
genrmf.128.3	49152	227840	7199231471	3796628806	47.26	721.86s
genrmf.128.4	65536	309248	13119318950	7104283290	45.85	1327.39s
genrmf.128.5	81920	390656	20828803678	11521420529	44.69	2119.17s
genrmf.128.6	98304	472064	30332206807	16964913971	44.07	3097.36s
genrmf.128.7	114688	553472	41523909029	23481478020	43.45	4257.81s
genrmf.128.8	131072	634880	54624078440	31056146951	43.15	5604.56s
genrmf.128.9	147456	716288	69354980601	39521084016	43.02	7132.08s
genrmf.128.10	163840	797696	85854378635	49344918441	42.52	8849.84s

Tabelle 6.6: Greedy-Konstruktion

Wir sehen, dass die Verbesserungen hier deutlich schlechter ausfallen als bei den Netzwerken unseres ersten Generators. Weiterhin fällt auf, dass die erzielte Verbesserung mit dem durchschnittlichen Eingangsgrad der Netzwerke korreliert.

Wie auch schon bei unserem ersten Generator wollen wir nun die Laufzeiten des Max-Flow-Algorithmus untersuchen. Die Ergebnisse sind in Tabelle 6.7 dargestellt.

Wie aufgrund der geringeren Verbesserung der Bandbreitensumme zu erwarten war, fällt auch der Geschwindigkeitszuwachs deutlich niedriger aus als bei den Netzwerken des ersten Generators. Jedoch konnten wir auch hier durchschnittliche Leistungssteigerungen von etwa 25% erzielen.

Name	Random	Greedy	Speedup
genrmf.128.3	2.32	1.71	1.36
genrmf.128.4	2.64	2.36	1.12
genrmf.128.5	3.25	2.77	1.17
genrmf.128.6	4.75	3.94	1.21
genrmf.128.7	6.19	4.83	1.28
genrmf.128.8	6.93	5.23	1.33
genrmf.128.9	7.92	6.38	1.24
genrmf.128.10	8.08	6.18	1.31

Tabelle 6.7: Laufzeiten MFS in s

6.3 Bewertung des Greedy-Algorithmus

In diesem Abschnitt wollen wir einen Eindruck von der Güte unseres Greedy-Algorithmus gewinnen. Hierzu stellen wir eine Reihe zufälliger Graphen bereit, welche wir mit dem Verfahren der dynamischen Programmierung tatsächlich optimieren werden. Eine Abwandlung dieses Verfahrens liefert uns gleichzeitig die schlechtest mögliche Bandbreiten-summe. Aufgrund der exponentiellen Laufzeit müssen wir uns hierbei jedoch auf sehr kleine Graphen beschränken. Bereits bei einer Knotenanzahl von 20 erreichen wir die Grenze des praktisch berechenbaren. Die Ergebnisse unserer Experimente finden sich in Tabelle 6.8.

Name	Nodes	Edges	Min	Orig	Max	Greedy	ImpO	ImpG
rnd_20_10	20	10	0	6	14	0	100.00	100.00
rnd_20_20	20	20	2	48	77	2	97.40	95.83
rnd_20_30	20	30	16	140	184	30	91.30	78.57
rnd_20_40	20	40	40	218	293	72	86.35	66.97
rnd_20_50	20	50	80	369	483	111	83.44	69.92
rnd_20_60	20	60	197	515	654	244	69.88	52.62
rnd_20_70	20	70	318	679	874	371	63.62	45.36
rnd_20_80	20	80	471	858	1087	554	56.67	35.43
rnd_20_90	20	90	648	1121	1270	789	48.98	29.62
rnd_20_100	20	100	806	1310	1507	992	46.52	24.27
rnd_20_110	20	110	908	1407	1624	1071	44.09	23.88
rnd_20_120	20	120	1034	1596	1801	1194	42.59	25.19
rnd_20_130	20	130	1274	1835	1986	1476	35.85	19.56
rnd_20_140	20	140	1411	2063	2268	1659	37.79	19.58
rnd_20_150	20	150	1652	2299	2461	1743	32.87	24.18

Tabelle 6.8: Vergleich der Ergebnisse des Greedy-Algorithmus mit Optimum

Die Spalten *Min* und *Max* geben hier die Kosten der jeweils besten bzw. schlechtesten

Kantennummierung an. *Orig* und *Greedy* bezeichnen die Kosten des (zufälligen) Ausgangsgraphen bzw. des durch den Greedy-Algorithmus verbesserten Ergebnisgraphen. Die letzten beiden Spalten geben die entsprechenden prozentualen Verbesserungen an. Abgesehen von den Graphen mit durchschnittlichen Eingangsgraden bis drei erreichen wir mit unserem Greedy-Algorithmus eine Verbesserung, die etwa 15 bis 20 Prozent schlechter als das Optimum ist.

Desweiteren zeigt die Tabelle sehr deutlich, wie sehr die Optimierungsmöglichkeiten, zumindest bei zufälligen Graphen, mit zunehmenden Eingangsgrad der Knoten abnehmen.

6.4 Iteration nach dem arithmetischen Mittel

In diesem Abschnitt wollen wir kurz die entsprechenden Ergebnisse unseres zweiten heuristischen Verfahrens zusammenfassen. Wir haben dieses bisher völlig außer Acht gelassen, da es durchweg schlechtere Resultate erzielt als der Greedy-Algorithmus. Beispielfhaft ist dies anhand dreier Max-Flow-Netzwerke unseres ersten Generators in Tabelle 6.9 dargestellt.

Name	Bad	ArithMean	Imp	Rounds	Time
mf_50_50.bad	306372500	9307454	96.96	17	8.47s
mf_100_100.bad	9900990000	150091309	98.48	32	287.49s
mf_150_150.bad	75434602500	762512075	98.99	46	2282.73s

Tabelle 6.9: Arithmetic Mean Iteration

Rounds gibt hierbei die Runden an, die das Verfahren bis zur Konvergenz benötigte. Dies bedeutet, dass in der entsprechenden Rundenzahl keine Knotenverschiebung mehr stattgefunden hat, da hiermit keine echte Verbesserung mehr erzielt werden konnte. Ein Vergleich mit Tabelle 6.2 zeigt, dass dieses Verfahren sowohl bezüglich der verbesserten Kosten, als auch in Bezug auf die Laufzeit schlechter ist als die Greedy-Konstruktion.

6.5 Fazit

Wir konnten mit unseren Experimenten nachweisen, dass mit einem Cache-berücksichtigenden Speicherlayout von Graphen tatsächlich eine verbesserte Laufzeit von Algorithmen erzielt werden kann. Am Beispiel eines Max-Flow-Algorithmus haben wir gesehen, dass sich die Laufzeit nur aufgrund einer besseren Anordnung des Graphen im Speicher annähernd halbieren ließ.

7 Zusammenfassung und Ausblick

Wir wollen noch einmal die wichtigsten Punkte unserer Arbeit zusammenfassen. Gestartet sind wir von einem gerichteten Graphen, der bzgl. seiner ausgehenden Kanten kompakt im Speicher abgelegt ist. Unser Ziel war es nun, die eingehenden Kanten in Hinblick auf ein besseres Cache-Verhalten umzuordnen. Auf Basis einer formalen Darstellung unserer Ausgangssituation konnten wir dann präzise unser Ziel formulieren und beweisen, dass das entsprechende Problem NP-vollständig ist. Mit diesem Wissen haben wir uns dann auf die Suche nach Näherungsverfahren gemacht. Durch Experimente konnte gezeigt werden, dass mit einer Verbesserung unseres Optimierungskriteriums tatsächlich auch eine verbesserte Laufzeit entsprechender Algorithmen auf den Graphen einherging. Naturgemäß konnten wir das Thema im Rahmen dieser Diplomarbeit nicht erschöpfend behandeln. So wollen wir die Arbeit mit der Benennung weiterer Aspekte und Denkanstöße abschließen, auf die wir hier nicht näher eingehen konnten.

So haben wir zwar in der Arbeit die NP-Vollständigkeit des Problems bewiesen, wie aber sieht es aus, wenn wir uns auf spezielle Klassen von Graphen beschränken? Gibt es vielleicht effiziente Algorithmen, wenn wir die Graphen speziellen Einschränkungen unterwerfen?

Desweiteren könnte man noch mal über unsere Kostenfunktion nachdenken, die es zu minimieren galt. Mit der Bandbreiten-Summe eingehender Kanten haben wir zwar ein Maß gefunden, welches sich auch in der Cache-Effizienz widerspiegelt, jedoch könnte man sich hier sicher noch „mehr“ vorstellen, also Kriterien, die in engerem Zusammenhang mit der Arbeitsweise der Caches stehen. Darüber hinaus könnte man sich fragen, ob man überhaupt an der Kompaktheit ausgehender Kanten festhalten sollte. Können nicht vielleicht bessere Ergebnisse erzielt werden, wenn man diese aufgibt?

Der letzte Punkt betrifft die Algorithmen. Hier bieten sich sicher noch viele Ansätze, denen man nachgehen könnte. Seien dies bekannte Heuristiken wie das „Hillclimbing“ oder „Simulated Annealing“, die man hier anzuwenden versucht, oder auch völlig neue (problemspezifische) Ideen. Ein anderer Ansatzpunkt ergibt sich, wenn wir uns noch einmal die Ergebnisse unserer Experimente anschauen. Es hat sich gezeigt, dass es für unseren Greedy-Algorithmus einen großen Einfluß hat, in welcher Reihenfolge wir die Knoten unserem Graphen hinzufügen. So waren die Ergebnisse deutlich besser, als wir auf unserem „schlecht generierten“ Netzwerk gestartet sind, als dies bei einer zufälligen Permutation der Kantenummerierung der Fall war. Interessanterweise führte hier eine bessere Ausgangslage zu einem schlechteren Endergebnis. Man könnte sich also Gedanken um eine Vorsortierung der Knoten machen, auf welcher dann erst in einer zweiten Phase der eigentliche Algorithmus ansetzt.

7 Zusammenfassung und Ausblick

Sowohl mit der allgemeinen Problematik eines Cache-berücksichtigenden Speicherlayouts für Graphen, als auch mit dem hieraus entstandenen NP-vollständigen Problem MIBS, haben wir hier zwei Bereiche vorgestellt, welche nach unseren Recherchen noch relativ unbeleuchtet sind und noch eine Menge Möglichkeiten zur Weiterentwicklung bieten.

Literaturverzeichnis

- [1] M. Bender, R. Cole, E. Demaine, and M. Farach-Colton. Scanning and traversing: Maintaining data for traversals in memory hierarchy. In *Proc. Annual European Symposium on Algorithms*, pages 152–164, 2002.
- [2] K. Beyls and E. D’Hollander. Reuse distance as a metric for cache behavior, 2001.
- [3] Kristof Beyls. The processor-memory gap: Cache remapping and related techniques.
- [4] Kristof Beyls, Erik H. D’Hollander, and Yijun Yu. Visualization enables the programmer to reduce cache misses. In *IASTED PDCS*, pages 774–781, 2002.
- [5] Neal Cardwell Richard Fromm Kimberly Keeton Christoforos Kozyrakis Randi Thomas David Patterson, Thomas Anderson and Katherine Yelick. A case for intelligent ram, 1997.
- [6] Josep Díaz, Jordi Petit, and Maria Serna. A survey of graph layout problems. *ACM Comput. Surv.*, 34(3):313–356, 2002.
- [7] Leonor Frias, Jordi Petit, and Salvador Roura. Lists revisited: Cache conscious stl lists. In *WEA*, pages 121–133, 2006.
- [8] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS ’99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285, Washington, DC, USA, 1999. IEEE Computer Society.
- [9] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
- [10] M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some simplified np-complete problems. In *STOC*, pages 47–63, 1974.
- [11] Donald Goldfarb¹ and Michael D. Grigoriadis. A computational comparison of the dinic and network simplex methods for maximum flow. *Annals of Operations Research*, 13(1), 1998.

- [12] John L. Hennessy and David A. Patterson. *Computer Architecture. A Quantitative Approach*. 2006.
- [13] Tao Jiang, Dhruv Mubayi, Aditya Shastri, and Douglas B. West. Edge-bandwidth of graphs. *SIAM Journal on Discrete Mathematics*, 12(3):307–316, 1999.
- [14] Yehuda Koren and David Harel. A multi-scale algorithm for the linear arrangement problem. In *WG*, pages 296–309, 2002.
- [15] C. Papadimitriou. The np-completeness of the bandwidth minimization problem. *Computing*, 16:263–270, 1976.
- [16] Sandeep Sen, Siddhartha Chatterjee, and Neeraj Dumir. Towards a theory of cache-efficient algorithms. *J. ACM*, 49(6):828–858, 2002.
- [17] Steven P. VanderWiel and David J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.
- [18] Sung-Eui Yoon, Peter Lindstrom, Valerio Pascucci, and Dinesh Manocha. Cache-oblivious mesh layouts. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 886–893, New York, NY, USA, 2005. ACM Press.
- [19] Kenneth Williams Yung-Ling Lai. A survey of solved problems and applications on bandwidth, edgesum, and profile of graphs. *Journal of Graph Theory*, 31(2):75–94, 1999.