View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

Algorithmen für Matchingprobleme in speziellen Graphklassen

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Martin Löhnertz

aus

Würselen

Bonn (September) 2009

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

1. Referent: Prof. Dr. N. Blum
2. Referent: Prof. Dr. M. Karpinski
Tag der Promotion: 3.2.2010
Erscheinungsjahr: 2010

Vorwort

Seit mir im Rahmen des Bundeswettbewerbs Informatik Matchingprobleme erstmals begegneten, dort der gewichtete Fall als Subroutine des Algorithmus von Christophides [4], bin ich davon fasziniert, dass zu einem strukturell so einfach erscheinenden Problem noch nicht alle Fragen abschließend geklärt werden konnten. Diese Begeisterung begleitete mich durch mein gesamtes Studium, beeinflusste meine Nebenfachwahl "Diskrete Mathematik" und die Wahl meines Diplomarbeitsthemas [26].

Somit war es nur natürlich, dass diese Problematik bei der Auswahl eines Forschungsthemas sofort erste Wahl war. Ich bin von zwei Seiten an das Gebiet herangetreten: Einerseits habe ich versucht, eigene schnellere Algorithmen für das allgemeine Problem zu finden, aus dieser Bestrebung ist Teil II entstanden. Andererseits habe ich Forschungsergebnisse meines Betreuers Prof. Dr. Blum zum allgemeinen Matching mit anderen Verfahren verbunden, um auf diese Weise dessen Arbeit auf diesem Gebiet zu ergänzen. Dies führte zu den Resultaten im Teil I.

Mein besonderer Dank gilt meinem Betreuer, Herrn Prof. Dr. Blum, unserem Abteilungsleiter Prof. Dr. Karpinski, sowie meinen Kollegen Dr. Mathias Hauptmann, Matthias Kretschmer, Johannes Langguth, Hans-Hermann Leinen und Dr. Peter Wegner, die in fruchtbaren Diskussionen meine Arbeit bereicherten.
Diese Arbeit wäre nicht möglich gewesen ohne die Unterstützung meiner Eltern Werner und Lilli Anna Löhnertz, die mit großer Geduld meine Promotionszeit ertragen haben und denen ich für ihre beständige Ermutigung danke.

Bonn, 23.9.2009

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Einführung 1

1.2 Notation 2

1.3 Einordnung 6

1.4 Grundlegende Algorithmen 8

1.4.1 Der Algorithmus von Hopcroft und Karp 9

1.4.2 Der Algorithmus von Dinic 12

I Algorithmen für das nichtbipartite Matchingproblem **15**

2 Einleitung **17**

2.1 Der Algorithmus von Feder und Motwani 18

2.1.1 Überblick 18

2.1.2 Durchführung der Graphenkompression 22

2.1.3 Das Komprimieren von Graphen 27

2.2 Eigenschaften Nichtbipartiter Graphen 29

2.3 Der nichtbipartite Fall als bipartites Problem 34

2.3.1 Umwandlung in ein Erreichbarkeitsproblem 35

2.3.2 Komprimieren des umgewandelten Graphen 36

3 Explizite Kompression **38**

3.1 Verringerung der Kantenzahl 38

3.2 Konstruktion von CMA-Graphen 50

3.2.1 Superkonzentratoren 50

3.2.2 Konstruktion von Konzentratoren 52

3.2.3 Umwandlung von Superkonzentratoren 54

4 Implizite Kompression 58

4.1 Einleitung 58
4.2 Der Matching-Algorithmus von Blum 60
4.2.1 Gesamtaufbau 60
4.2.2 MBFS 62
4.2.3 Der Übergang von MBFS nach MDFS 69
4.2.4 MDFS 72
4.3 Algorithmische Suche in Cliquen und Sternen 85
4.3.1 Einleitung 85
4.3.2 Anwendung im bipartiten Fall 86
4.4 Die Beschleunigung von MBFS 91
4.4.1 Beschleunigung der Vorwärtssuche 91
4.4.2 Beschleunigung der Rückwärtssuche 94
4.4.3 Laufzeitanalyse 99
4.5 Die Beschleunigung von MDFS 101
4.5.1 Aufbereitung der Ergebnisse von MBFS für MDFS 101
4.5.2 Beschleunigung von Vorwärts- und Rückwärtssuche von MDFS . 102
4.5.3 Laufzeitanalyse 108
4.6 Gesamtergebnis 110

5 Anmerkungen und Erweiterungen 115

5.1 Realisierung der Union-Find Algorithmen 115
5.2 Beschleunigung von MBFS durch explizite Kompression 115
5.3 Verkleinerung der CMA-Graphen 118
5.4 Suchstrukturen für Graphen 119

II Bipartite Graphen mit zahlreichen Matchings 123

6 Einleitung 125

7 Beschreibung des Algorithmus 127

7.1 Übersicht 127

7.2 Finden eines nahezu regulären Subgraphen 130
7.3 Verwenden eines annähernd regulären Teilgraphen 135
7.4 Ausbalancieren der beiden Teile 137
7.5 Der Algorithmus von Cole, Ost und Schirra 137

8 Anmerkungen und Erweiterungen 144

8.1 Anmerkungen 144
8.1.1 Der bestmögliche Fall 144
8.1.2 Parameterwahl 144
8.1.3 Vereinfachung der Implementierung 145
8.2 Verallgemeinerungen 148
8.3 Anwendung von Graphenkompression 153
8.3.1 Beschleunigung des dritten Teils 153
8.3.2 Beschleunigung des ersten Teils 155

III Zusammenfassung und Verzeichnisse 161

9 Zusammenfassung 163

Verzeichnis der Definitionen, Sätze und Lemmata 166

Abbildungsverzeichnis 169

Verzeichnis der Algorithmen 172

Literaturverzeichnis 173

Kapitel 1

Einleitung

1.1 Einführung

Das Matchingproblem ist ein klassisches Problem der Graphentheorie. Ein gutes Beispiel zur Veranschaulichung von Matchingproblemen ist eine Tanzveranstaltung: In einem Saal sind einige Tänzer versammelt. Jeder gibt auf einem Fragebogen an, mit welcher der anderen Personen er gerne tanzen würde. Nun soll eine möglichst große Zahl von Paaren gebildet werden (vgl. Abb. 1.1). In dem hier dargestellten Beispiel gehen wir

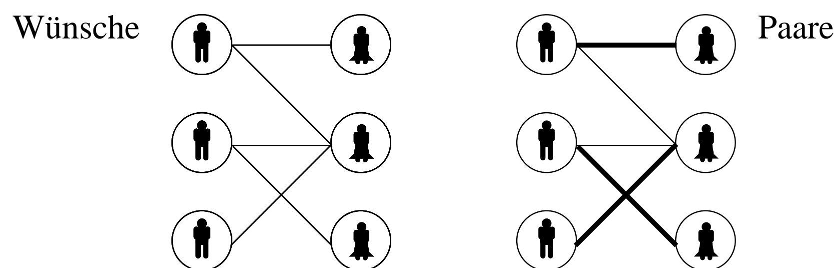

Abbildung 1.1: Paarbildung bei Tanzveranstaltung

davon aus, dass jeweils ein Mann mit einer Frau tanzt. Die Menge der Tänzer kann also in zwei disjunkte Mengen ("Männer" und "Frauen") zerlegt werden, so dass zu jedem Paar jeweils ein Mann und eine Frau gehören. Diese besondere Situation werden wir als das sogenannte "bipartite" "Matchingproblem" bezeichnen. Die allgemein als Matchingproblem bezeichnete Fragestellung sieht keine solche Trennung vor - sie entspricht beispielsweise der Bildung von Arbeitsgruppen aus je zwei Personen.
Das Ziel dieser Arbeit ist es, Algorithmen zum schnellen Finden solcher paarweisen Zuordnungen ("Matchings") zu entwickeln.
Fragestellungen aus dem Umfeld des Matchings treten auch in zahlreichen anderen Anwendungen - z.B. in der automatischen Stundenplanerstellung - als Teilprobleme auf, so dass eine Beschleunigung der Verfahren zum Finden von Matchings unmittelbare Auswirkungen auf viele andere Bereiche hat. Häufig treten in diesen Kontexten Spezialfälle des Matchingproblems auf, die effizienter gelöst werden können als das

allgemeine Problem. In dieser Arbeit werden wir uns insbesondere mit solchen Problemen auseinandersetzen, in denen viele verschiedene Lösungen möglich sind, und zeigen, dass - der Anschauung entsprechend - in diesen Fällen eine Lösung effizienter gefunden werden kann.

1.2 Notation

Bei Bezügen auf Sätze, Lemmata etc. die mehr als zwei Seiten überspringen, geben wir die zugehörige Seitennummer in Form eines kleinen Indexes an. Zusätzlich finden sich Verzeichnisse am Ende dieser Arbeit.
Definitionen, die auf ein Kapitel beschränkt sind, finden sich dort. Wir verwenden ansonsten die in der Graphentheorie etablierten Notationen (vgl. z.B. [3] oder [22]):
Ein *Graph* $G = (V, E)$ besteht aus einer Menge von *Knoten* V und einer Menge von *Kanten* E. Graphen können *gerichtet* oder *ungerichtet* sein. Im *ungerichteten* Fall ist jede Kante ein ungeordnetes Paar von Knoten, also $E \subseteq \binom{V}{2}$. Bei einem *gerichteten Graphen* ist jede Kante ein geordnetes Paar von Knoten, also $E \subseteq V \times V$. Man stellt eine Kante entsprechend als das Tupel (u, v) dar, wobei man u als den "Schwanz" und v als den "Kopf" der Kante bezeichnet. Will man Graphen zeichnerisch darstellen, so verwendet man für die Knoten zumeist Punkte und für die Kanten diese Punkte verbindende Strecken bzw. im gerichteten Fall Pfeile vom "Schwanz" zum "Kopf". Diese anschauliche Darstellung wird auch in die Sprache übernommen: Man sagt, dass eine Kante zwei Knoten "verbindet", bzw. im gerichteten Fall von einem Knoten zum anderen Knoten "führt". Zu einem gegebenen Graphen G bezeichnet $E(G)$ seine Kanten- und

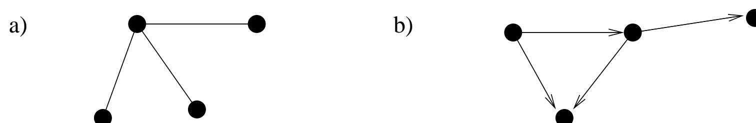

Abbildung 1.2: Zeichnerische Darstellung eines ungerichteten (a) und eines gerichteten Graphen (b)

$V(G)$ seine Knotenmenge. Wenn nicht anders angegeben, kürzen wir die Kardinalität der Knotenmenge $|V(G)|$ mit n und die der Kantenmenge $|E(G)|$ mit m ab.
Eine Kante heißt *inzident* zu einem Knoten, wenn sie diesen mit einem anderen Knoten verbindet (bzw. in Mengennotation: wenn sie diesen enthält). Zwei Knoten werden *adjazent* genannt, wenn es eine Kante gibt, die zu beiden inzident ist. Gibt es zu je zwei Knoten höchstens eine Kante, die zu beiden inzident ist, heißt der Graph *einfach*. Mengen von paarweise nicht adjazenten Knoten werden als *unabhängige* (Knoten-) Mengen bezeichnet.
Ein *Teilgraph* eines Graphen G ist ein Graph $H = (V', E')$ mit $V' \subset V$ und $E' \subset E$. Wir schreiben vereinfachend $H \subset G$. Ein Teilgraph heißt *spannend*, wenn er alle Knoten des Graphen enthält, also $V = V'$ gilt. Ein Teilgraph heißt *induziert*, wenn bei fest gewählten

Knoten des Teilgraphen alle Kanten zwischen diesen Knoten aus dem ursprünglichen Graphen übernommen werden, also $E' = \{(u, v) \in E | u, v \in V'\}$.
Der *Schnitt* zweier Graphen $G \cap F$ ergibt sich als Schnitt der jeweiligen Knoten- und Kantenmengen. Er ist der größte gemeinsame Teilgraph beider Graphen.
Die *Nachbarschaft* $\Gamma(v)$ eines Knotens v ist die Menge der zu ihm adjazenten Knoten.
Der *Grad* $\delta(v)$ eines Knotens v ist die Zahl der zu diesem Knoten inzidenten Kanten, im Fall einfacher Graphen gilt entsprechend $\delta(v) = |\Gamma(v)|$. Ein Graph heißt *regulär*, wenn alle Knoten den gleichen Grad haben. Mit $\Delta(G)$ bezeichnen wir den *Maximalgrad* des Graphens G also $\Delta(G) := \max_{v \in V(G)} \delta(v)$. Ist ein Teilgraph H eines Graphen G gegeben, so bezeichnet $\delta_H(v)$ den Grad des Knotens v in diesem Teilgraphen.

Ein Graph heißt *bipartit*, wenn sich seine Knotenmenge so in zwei nichtleere disjunkte Teile A und B (z.B Tänzer und Tänzerinnen) zerlegen läßt, dass alle Kanten Knoten aus A mit Knoten aus B verbinden, es also keine Kanten zwischen zwei Knoten aus A oder zwei Knoten aus B gibt. Wir bezeichnen die Menge aller ungerichteten Kanten, die eine Knotenmenge A mit einer anderen Knotenmenge B verbinden, als $A \otimes B := \{\{u, v\} \mid V \times V, u \in A, v \in B\}$. Entsprechend gilt für ungerichtete bipartite Graphen $E \subseteq A \otimes B$ und für gerichtete bipartite Graphen $E \subseteq A \times B \cup B \times A$.

Ein *Matching* in einem ungerichteten Graphen ist eine Menge von Kanten M, in der keine zwei Kanten einen Knoten gemeinsam haben: $\{v, w\} \cap \{x, y\} = \emptyset \; \forall \{v, w\}, \{x, y\} \in M$.
Man sagt auch dass die Kanten eines Matchings eine *unabhängige Kantenmenge* bilden.
Die *Größe* $\nu(M)$ eines Matchings ist seine Kardinalität.
Ist ein Matching gegeben, so nennt man Knoten, die nicht zu einer Kante des Matchings gehören, *frei*, und solche, die dazugehören, *überdeckt*. Analog bezeichnet man Kanten, die nicht zum Matching gehören, als *freie Kanten* und die dazugehörigen Kanten als *Matchingkanten*. Bei einem gegebenen Matching M in einem Graphen G ist die Zahl der überdeckten Knoten $2|M|$ und die Zahl der freien Knoten $V(G) - 2|M|$.
Ein Matching M heißt

- *perfekt*, wenn es jeden Knoten überdeckt, also $\bigcup_{\{x,y\} \in M} \{x, y\} = V$.
- *nicht erweiterbar*, wenn es nicht möglich ist, dem Matching weitere Kanten hinzuzufügen, es also kein Matching M' mit $M \subset M', M \neq M'$ gibt.
- *maximal*, wenn es kein größeres Matching M', $|M'| > |M|$ gibt.

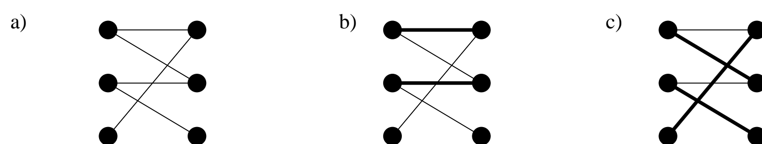

Abbildung 1.3: a) Ausgangsgraph, b) nicht erweiterbares Matching, c) maximales Matching

Man bezeichnet ein Matchingproblem mit einem zugrundeliegenden bipartiten Graphen auch als *bipartites Matchingproblem* - ansonsten als nichtbipartites oder allgemeines Matchingproblem.

Ein *Pfad* ist eine Folge von Knoten $(v_1, \ldots, v_r)$, bei der je zwei aufeinanderfolgende Knoten adjazent sind $(v_i, v_{i+1}) \in E, i = 1, \ldots, r-1$; er heißt *einfach*, wenn die Knoten paarweise verschieden sind. Die Kanten des Pfades $E(P)$ sind die Kanten $\{e_i = (v_i, v_{i+1})\}_{i=1,\ldots,r-1}$, die die Knoten jeweils miteinander verbinden. $P|_{v_i,v_j}$ bezeichnet den Teilpfad von P, der bei Knoten v_i beginnt und bei Knoten v_j endet. Ein *Kreis* ist ein einfacher Pfad, bei dem Anfangs- und Endknoten identisch sind. Die Länge eines Pfades $P = (v_1, \ldots v_r)$ bezeichnen wir mit $l(P) := |E(P)| = r - 1$.
Ein *alternierender Pfad* zu einem gegebenen Matching M ist ein einfacher Pfad, der abwechselnd Kanten aus M und aus $E \backslash M$ enthält. Er heißt M*-augmentierend*, wenn er dabei mit jeweils einer Kante aus $E \backslash M$, d.h. mit einer freien Kante beginnt und endet. Wenn keine Unklarheiten bezüglich M bestehen, sprechen wir einfach von augmentierenden Pfaden.
Diese Pfade sind für die Konstruktion von Matching-Algorithmen von besonderer Bedeutung. Nimmt man von den Kanten eines solchen Pfades P die Kanten, die zu einem Matching M gehören, aus dem Pfad heraus und fügt die anderen Kanten des Pfades dafür in das Matching ein, so erhält man ein größeres Matching M' (vgl. Abb. 1.4). Der Rest des Graphen bleibt unverändert.

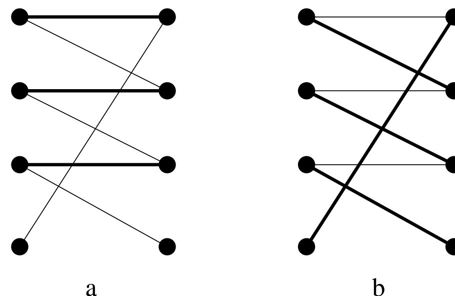

Abbildung 1.4: Augmentierender Pfad / Augmentierung

Nach dieser sogenannten *Augmentierung* besteht das neue Matching M' also aus den Kanten, die zum Pfad P und nicht zum Matching M gehören und aus den Kanten, die zu M aber nicht zu P gehören. Diese Menge bezeichnet man auch als die *symmetrische Differenz* von M und P. Allgemeiner: Sind A und B zwei Mengen, so bezeichnen wir mit $A \oplus B$ ihre symmetrische Differenz $(A \backslash B) \cup (B \backslash A) = (A \cup B) \backslash (A \cap B)$.

Graphen werden oft zur Modellierung von Transportproblemen eingesetzt. Im einfachsten Modell geht man davon aus, dass Objekte ausgehend von einem Startknoten s (*Quelle*) über Pfade zu einem Zielknoten t (*Senke*) transportiert werden sollen. Hierbei kann über jede Kante nur eine begrenzte Menge an Objekten transportiert werden. Entsprechend definiert man ein *Netzwerk* $N = (V, E, c)$ als einen gerichteten Graphen mit Knotenmenge V und Kantenmenge E sowie mit einer Funktion $c : E \rightarrow \mathbb{R}_+$, der sogenannten *Kapazität*. Sind s und t zwei verschiedene Knoten aus V, so ist ein s-t*-Fluss* eine Abbildung $f : E \rightarrow \mathbb{R}_+$, die die folgende Bedingungen erfüllt:

- Aus jedem Knoten, außer s und t, fließt soviel hinaus, wie hineinfließt.
- Keine Kante wird über ihre Kapazität hinaus belastet.

Formal ausgedrückt:

$$\sum_{(a,v)\in E} f(a,v) = \sum_{(v,b)\in E} f(v,b) \quad \forall v \in V\backslash\{s,t\} \tag{1.1}$$

$$f(e) \leq c(e) \;\; \forall e \in E \tag{1.2}$$

Die *Gesamtstärke* des Flusses f ist die Größe des aus s heraus und in t hineinfließenden Flusses

$$\sum_{(s,b)\in E} f(s,b) = \sum_{(a,t)\in E} f(a,t).$$

Vollständig ausgelastete Kanten e mit $f(e) = c(e)$ werden als *gesättigt* bezeichnet. Kanten mit $f(e) > 0$ sind *flusstragende* Kanten. Ist $c(e) = 1 \; \forall e \in E$, so spricht man auch von einem Einheitskapazitätsnetzwerk.

Unter dem *Identifizieren* einer Knotenmenge $V' \subset V$ mit sich selbst verstehen wir die Operation, die diese Knoten durch einen einzelnen Knoten ersetzt und diesen mit all den Knoten verbindet, mit denen die Knoten aus V' verbunden waren. Wir schreiben dies als

$$\begin{aligned} G/V' \;\; := \;\; & ((V \cup \{v_{V'}\}\backslash V'), (\{(v,w) \in E \mid v,w \notin V'\} \\ & \cup\{(v_{V'},w) \mid \exists v' \in V' : (v',w) \in E\} \\ & \cup\{(w,v_{V'}) \mid \exists v' \in V' : (w,v') \in E\})) \end{aligned}$$

Ein gerichteter Graph heißt *schleifenfrei*, wenn es keine Kanten der Form (v,v) (sog. *Schleifen*) gibt.

Bei der Analyse von Laufzeiten werden wir wie allgemein üblich konstante Faktoren vernachlässigen und entsprechend die approximativen Notationen einsetzen, die wir hier nach Blum [3] und Corman et al. [38] wiedergeben:

- $f(n) \in \mathcal{O}(g(n)) \Leftrightarrow \exists c \in \mathbb{R}_+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 : f(n) < cg(n)$, d.h. f wächst asymptotisch langsamer als g.
- $f(n) \in \Omega(g(n)) \Leftrightarrow \exists c \in \mathbb{R}_+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 : f(n) > cg(n)$, d.h. f wächst asymtotisch schneller als g.
- $f(n) \in \mathbf{o}(g(n)) \Leftrightarrow \forall c \in \mathbb{R}_+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 : f(n) < cg(n)$, d.h. f wächst asymtotisch deutlich langsamer als g.
- $f(n) \in \omega(g(n)) \Leftrightarrow \forall c \in \mathbb{R}_+ \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0 : f(n) > cg(n)$, d.h. f wächst asymtotisch deutlich schneller als g.

1.3 Einordnung

Wir geben hier nur einen sehr kurzen Überblick und beschränken uns zumeist auf die algorithmische Sichtweise. Für die zahlreichen algebraischen und strukturellen Eigenschaften von Matchings sei auf Lovasz und Plummer [28], Karpinski[23] oder Schrijver [37] verwiesen.
Die meisten Matching-Algorithmen basieren auf augmentierenden Pfaden. Wenn man einen augmentierenden Pfad findet, kann das Matching vergrößert werden, indem die Kanten auf dem Pfad, die zum Matching gehören, aus diesem entfernt werden, und die Kanten, die nicht Bestandteil des Matchings sind, hinzugefügt werden (vgl. Abb. 1.5).

Abbildung 1.5: Augmentierender Pfad

Das heißt, solange es noch augmentierende Pfade gibt, kann das Matching weiter vergrößert werden. Der Erfolg dieser Methode beruht darauf, dass auch die Umkehrung gilt:

Satz 1.1 (Berge 1952) *Ein Matching M ist genau dann maximal, wenn es keinen augmentierenden Pfad gibt.*

> **Beweis:** Offensichtlich kann es bezüglich eines maximalen Matchings keine augmentierenden Pfade mehr geben.
> Die Rückrichtung wird durch einen Widerspruchsbeweis gezeigt: Es sei angenommen, das Matching ist nicht maximal. Es ist also zu zeigen, dass es noch einen augmentierenden Pfad gibt.
> Man betrachtet nun die symmetrische Differenz zwischen dem gegebenen Matching M und einem maximalen Matching M'. Jeder Knoten kann maximal zu je einer Kante aus M und einer Kante aus M' inzident sein. Also hat jeder Knoten höchstens Grad zwei. Der Graph, der aus den Kanten dieser Differenz gebildet wird, besteht somit nur aus Pfaden und Kreisen. Kreise müssen zudem jeweils gleichviele Kanten aus M und M' enthalten, da keine zwei Kanten aus M (ebenso M') zum selben Knoten inzident sein können.
> Da $|M'| > |M|$ muss es mindestens einen Pfad geben, der mehr Kanten aus M' als aus M enthält. Dies ist nur möglich, wenn die erste und die letzte Kante dieses Pfades aus M' sind. Dieser Pfad ist also ein augmentierender Pfad. □

Algorithmische Verfahren zum Lösen des bipartiten Problems wurden schon vor dem zweiten Weltkrieg von König und Egervary angedeutet [28] und finden sich bei Kuhn und Hall (vgl. [37]): Um ein maximales Matching zu finden, beginnt man mit einem beliebigen (zufälligen) Matching und vergrößert dieses nach und nach über augmentierende Pfade. Wir werden in Abschnitt 1.4[8] sehen, dass es möglich ist, einen solchen Pfad in $\mathcal{O}(m)$ zu finden. Da ein Matching maximal die Größe n haben kann, hat dieser einfachste Algorithmus eine Laufzeit von $\mathcal{O}(mn)$.
Die erste und wesentlichste Beschleunigung für allgemeine bipartite Graphen gelang Dinic [8] 1970 bzw. Hopcroft und Karp [21] 1972:

Satz 1.2 (Hopcroft, Karp 1972) *In einem bipartiten Graphen kann ein Matching in Zeit $\mathcal{O}(\sqrt{n}m)$ gefunden werden.*

Wir werden in Kapitel 1.4 genauer auf dieses Resultat eingehen.

Die bisher genannten Algorithmen funktionieren nicht im nichtbipartiten Fall. In nichtbipartiten Graphen bereiten die sogenannten "Blüten" Probleme. Dies sind spezielle Kreise ungerader Länge, die ein solcher Algorithmus unter bestimmten Umständen für einen augmentierenden Pfad halten könnte, obwohl keiner vorhanden ist. Das folgende Beispiel (Abb. 1.6) kann nur einen ersten oberflächlichen Eindruck bezüglich dieser Problematik vermitteln.

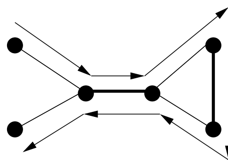

Abbildung 1.6: Blüte

Das nichtbipartite Problem wurde 1965 von Edmonds [9] gelöst, der beobachtete, dass man die Blüten so zu einem Knoten kontrahieren kann, dass der resultierende Graph genau dann einen augmentierenden Pfad hat, wenn der Ausgangsgraph einen besaß. Genaueres hierzu geben wir in Teil I_{17} an.
Erst viele Jahre später gelang die Verallgemeinerung von Satz 1.2 auf nichtbipartite Graphen:

Satz 1.3 (Micali, Vazirani 1980) *In einem nichtbipartiten Graphen kann ein Matching in Zeit $\mathcal{O}(\sqrt{n}m)$ gefunden werden.*

Die Sätze 1.2 und 1.3 bilden aus praktischer Sicht nach wie vor den Status Quo für Matching-Algorithmen.
Für bipartite Graphen mit hoher Kantenzahl gelang Feder und Motwani [11] durch das Verwenden sogenannter Graph-Kompressionsverfahren eine weitere Verbesserung, auf die wir ebenfalls in Kapitel I_{17} ausführlich eingehen werden.

Satz 1.4 (Feder, Motwani 1980-1994) *In einem bipartiten Graphen kann ein Matching in $\mathcal{O}(\sqrt{n}m\frac{\log\frac{2n^2}{m}}{\log n})$ Schritten gefunden werden.*

Wiederum war die Übertragung auf den nichtbipartiten Fall die nächste naheliegende Fragestellung. Erste Ansätze im Jahr 1995 von Goldberg und Karzanov [18] wurden von diesen 1998 teilweise revidiert [24], um dann 2002 erneut und in verbesserter Form formuliert zu werden. Zwischenzeitlich hatten aber auch Jungnickel und Freymuth-Päger (2001) und Löhnertz (2001,2004) das gleiche Ergebnis erzielen können:

Satz 3.15, 4.35 (Jungnickel, Freymuth-Paeger 2001; Goldberg, Karzanov 1995, 2002; Löhnertz 2001,2004)
In einem Graphen G mit n Knoten und m Kanten kann ein Matching in Zeit $\mathcal{O}(\sqrt{n}m\frac{\log\frac{2n^2}{m}}{\log n})$ gefunden werden.

Es gab zu jeder Zeit Versuche, spezielle Graphklassen zu identifizieren, bei denen das Finden von Matchings leichter möglich ist. Zu nennen sind hier reguläre Graphen [6] planare Graphen [29], chordale Graphen [7] und insbesondere Graphen, die genau ein maximales Matching enthalten [16]. Wir fügen in Kapitel II$_{125}$ dieser Liste einen weiteren Fall hinzu, nämlich bipartite Graphen, die zahlreiche disjunkte Matchings enthalten:

Satz 6.1 (Löhnertz 2002) *In einem bipartiten Graphen, der $\sigma^3\sqrt{n}$ disjunkte perfekte Matchings enthält, kann ein perfektes Matching in Zeit $\mathcal{O}\left(\frac{m\sqrt{n}}{\sigma}\right)$ gefunden werden.*

1.4 Grundlegende Algorithmen

Unsere Algorithmen in beiden Teilen der Arbeit bauen auf den Algorithmen von Hopcroft und Karp[21] bzw. Dinic [8] auf. Diese stellen wir daher in diesem Kapitel gesondert vor. Bei einem durch den Satz von Berge (Satz 1.1$_6$) inspirierten Vorgehen werden immer wieder neue augmentierende Pfade gesucht, über die dann wiederum augmentiert wird. Hier ergibt sich die Frage, ob es augmentierende Pfade gibt, die "besser geeignet" als andere sind. Da das Augmentieren über einen Pfad die Struktur der alternierenden Pfade im Graphen stark verändert, erscheint es sinnvoll, möglichst viele disjunkte Pfade zu finden, über die dann in einem Schritt augmentiert wird, damit man mit einem bestimmten unveränderten Zustand des Graphen möglichst lange arbeiten kann. Eine hohe Zahl disjunkter Pfade impliziert aber, dass die einzelnen Pfade nicht zu lang sein dürfen.
Es hat sich herausgestellt, dass es tatsächlich vorteilhaft ist, die jeweils kürzesten augmentierenden Pfade zu wählen. Der Beweis dieser Tatsache folgt aber einem anderen Gedankengang, als die gerade angedeutete Herleitung:

Zunächst stellt sich die Frage, wieviele disjunkte augmentierende Pfade es überhaupt geben kann. Dies klärt die folgende Erweiterung des Satzes 1.1$_6$:

Lemma 1.5 *Sei $G = (V, E)$ ein Graph und seien M und M' Matchings in G mit $|M'| > |M|$. Dann enthält der Graph $G' := (V, M \oplus M')$ genau $|M'| - |M|$ knotendisjunkte Pfade, die in Bezug auf M augmentierend sind.*

Beweis: Da alle Knoten in $(V, M \oplus M')$ höchstens zu einer Kante aus jeder Menge inzident sind, ist jeder Knoten zu höchstens zwei Kanten inzident, hat also höchstens Grad 2. Der Graph G' besteht also aus knotendisjunkten Pfaden und Kreisen gerader Länge. Da keine zwei Kanten aus M oder M' zum selben Knoten inzident sein können, müssen die Kanten auf diesen Pfaden und Kreisen abwechselnd (alternierend) aus M und M' sein. Es gehört also

nur jede zweite Kante zu M. Nur Pfade, die mit einer Kante aus M' beginnen und enden, können mehr Kanten aus M' als aus M enthalten, und zwar genau eine mehr. Daher muss es $|M'| - |M|$ dieser Pfade geben. □

Korollar 1.6 *Sei G ein Graph und seien M und M' Matchings in G mit $|M'| > |M|$. Dann kann in G eine Menge von knotendisjunkten M-augmentierenden Pfaden konstruiert werden, die mindestens $|M'| - |M|$ Pfade enthält.*

Beweis: Die $|M'| - |M|$ Pfade aus Lemma 1.5 bilden eine solche Menge mit genau $|M'| - |M|$ Elementen. □

1.4.1 Der Algorithmus von Hopcroft und Karp

Der Algorithmus von Hopcroft und Karp versucht, jeweils über kürzeste augmentierende Pfade zu augmentieren. Die entscheidende Eigenschaft dieses Vorgehens ist die, dass die Länge der kürzesten augmentierenden Pfade dabei (schwach) monoton steigt:

Satz 1.7 (Hopcroft, Karp) *Augmentiert man über einen kürzesten augmentierenden Pfad P, so gibt es danach keinen kürzeren augmentierenden Pfad.*

Beweis: Sei P' ein augmentierender Pfad bezüglich $M \oplus E(P)$, also dem Matching, das entsteht, wenn man M über P augmentiert. Es sei $N := M \oplus E(P) \oplus E(P')$ das Matching, das entsteht, wenn man auch noch über P' augmentiert. Dann ist $|N| = |M| + 2$ und es muß in $(V, M \oplus N)$ gemäß Korollar 1.5 zwei Pfade P_1 und P_2 geben, die in Bezug auf M augmentierend sind. Da für $\oplus$ das Assoziativ- und das Kommutativgesetz gelten und $A \oplus A = \emptyset$ gilt, kann man $N = M \oplus E(P) \oplus E(P')$ umformen zu $M \oplus N = \underbrace{M \oplus M}_{=\emptyset} \oplus E(P) \oplus E(P')$. Da $E(P_1)$ und $E(P_2)$ disjunkt in $M \oplus N$ liegen und $M \oplus N = E(P) \oplus E(P')$ gilt, muß $|E(P) \oplus E(P')| \geq |E(P_1)| + |E(P_2)|$ gelten. Da aber P_1 und P_2 mindestens die Länge $|P|$ haben, folgt daraus $|E(P) \oplus E(P')| \geq 2|E(P)|$. Wegen

$$|E(P) \oplus E(P')| = |E(P)| + |E(P')| - |E(P) \cap E(P')|$$

$$\Rightarrow 2|E(P)| \leq |E(P)| + |E(P')| - |E(P) \cap E(P')|$$

ergibt sich somit $|E(P')| \geq |E(P)| + |E(P) \cap E(P')|$, d.h. $|E(P')| \geq |E(P)|$. □

Um einen Fortschritt zu garantieren, reicht diese schwache Monotonie noch nicht aus. Es muss also sichergestellt werden, dass irgendwann alle augmentierenden Pfade einer bestimmten Länge bearbeitet wurden.

Satz 1.8 (Hopcroft, Karp 1973) *Augmentiert man über eine nicht erweiterbare Menge alternierender knotendisjunkter Pfade minimaler Länge $\mathcal{P}$, so sind alle augmentierenden Pfade im resultierenden Graphen länger als die Pfade aus $\mathcal{P}$.*

Beweis: Sei l die Länge der Pfade in $\mathcal{P}$ und P' ein augmentierender Pfad mit Länge $|P'| \leq l$ bezüglich $M \oplus \bigcup_{P \in \mathcal{P}} E(P)$, also bezüglich dem Matching, das nach Augmentieren über alle Pfade aus $\mathcal{P}$ entsteht. Sei außerdem $N := M \oplus \bigcup_{P \in \mathcal{P}} E(P) \oplus E(P')$. Dann ist $|N| = |M| + |\mathcal{P}| + 1$ und es muss in $(V, M \oplus N)$ somit $t := |\mathcal{P}| + 1$ knotendisjunkte augmentierende Pfade $P_1, \dots P_t$ geben. Da analog zum Beweis von Lemma 1.7 $M \oplus N = \bigcup_{P \in \mathcal{P}} E(P) \oplus E(P')$ gilt, muß $|\bigcup_{P \in \mathcal{P}} E(P) \oplus E(P')| \geq \sum_{i=1}^{t} |P_i|$ gelten. Daraus folgt $|\bigcup_{P \in \mathcal{P}} E(P) \oplus E(P')| \geq tl$. Da die Pfade in $\mathcal{P}$ aber disjunkt waren und somit

$$|\bigcup_{P \in \mathcal{P}} E(P) \oplus P'| = |\bigcup_{P \in \mathcal{P}} E(P)| + |E(P')| - |E(P') \cap \bigcup_{P \in \mathcal{P}} E(P)|$$

$$= (t-1)l + |E(P')| - |E(P') \cap \bigcup_{P \in \mathcal{P}} E(P)|$$

$$\Rightarrow tl \leq (t-1)l + |E(P')| - |E(P') \cap \bigcup_{P \in \mathcal{P}} E(P)|$$

gilt, ergibt sich $|E(P')| \geq l + |E(P') \cap \bigcup_{P \in \mathcal{P}} E(P)|$. Somit kann $|E(P')|$ wiederum nicht kleiner als l sein. Wäre die Länge von P' aber l, dann müsste $|E(P') \cap \bigcup_{P \in \mathcal{P}} E(P)| = 0$ sein. Dann wären P' und die Pfade in $\mathcal{P}$ aber kantendisjunkt. In diesem Fall müssen sie auch knotendisjunkt sein, da keine zwei Kanten aus M zum selben Knoten inzident sein können. Also hätte man $|P'|$ zu $\mathcal{P}$ hinzufügen können. Dies ist aber nicht möglich, da P eine nicht erweiterbare Menge augmentierender Pfade war. □

Dies alleine ist noch nicht hinreichend, um einen schnelleren Algorithmus zu konstruieren, da es augmentierende Pfade der Längen $1, \dots, n-1$ geben könnte. Um die Zahl dieser Suchphasen zu beschränken, können wir nun Lemma 1.5_8 einsetzen, um die Zahl der jeweils verbleibenden augmentierenden Pfade abzuschätzen:
Wenn wir c Schritte gemäß Satz 1.8 ausführen, also jeweils nicht erweiterbare Mengen augmentierender Pfade minimaler Länge suchen und über diese augmentieren, erhalten wir ein neues Matching M. Jeder M-augmentierende Pfad muß eine Länge von mindestens c haben. Es kann aber in der symmetrischen Differenz $(V, M \oplus M')$ höchstens $\frac{n}{c}$ knotendisjunkte Pfade mit jeweils mindestens c Knoten geben, da insgesamt nur $n \geq c\frac{n}{c}$ Knoten vorhanden sind. Nach Lemma 1.5_8 kann also ein maximales Matching M' höchstens $\frac{n}{c}$ Kanten mehr haben als das Matching M. Um das gefundene Matching zu einem maximalen Matching zu ergänzen, genügt es also, $\frac{n}{c}$ mal einen beliebigen augmentierenden Pfad zu suchen.
Wie wir zeigen werden, können sowohl eine nicht erweiterbare Menge von augmentierenden Pfaden als auch ein einzelner augmentierender Pfad in Zeit $\mathcal{O}(m)$ gefunden werden,

so dass die Gesamtlaufzeit $\mathcal{O}\left(cm + \frac{n}{c}m\right)$ beträgt. Dieser Ausdruck wird im Sinne der $\mathcal{O}$-Notation minimal für $c := \sqrt{n}$, womit sich eine Laufzeit von $\mathcal{O}(\sqrt{n}m)$ ergibt.

Um den Beweis von Satz 1.2$_7$ abzuschließen muss noch gezeigt werden, dass eine nicht erweiterbare Menge von augmentierenden Pfaden entsprechend Satz 1.8 in Linearzeit gefunden werden kann:

Lemma 1.9 *Eine nicht erweiterbare Menge kürzester augmentierender Pfade kann in bipartiten Graphen mit $\mathcal{O}(m)$ Operationen gefunden werden.*

Beweis: Es müssen drei Probleme gelöst werden:

1. Bei der Pfadsuche müssen abwechselnd Kanten, die zum Matching gehören, und Kanten, die nicht dazu gehören, verwandt werden.

2. Es muss bestimmt werden, welche Kanten und Knoten zu einem kürzesten augmentierenden Pfad gehören können.

3. Es muss eine nicht erweiterbare Menge von Pfaden gefunden werden, die nur Kanten verwendet, die im zweiten Schritt gefunden wurden.

zu 1.: Die Suche nach alternierenden Pfaden: Um das erste Problem zu lösen, genügt im bipartiten Fall die Beobachtung, dass ein augmentierender Pfad stets in verschiedenen Partitionen beginnt und endet; wenn man in Partition B startet, gelangt man stets mit einer freien Kante von B nach A und mit einer Matchingkante von A nach B. Indem man den Graphen richtet und jeweils freie Kanten von B nach A orientiert und Matchingkanten umgekehrt (vgl. Abb. 1.7), kann man das Problem des Findens eines augmentierenden Pfades in ein herkömmliches Erreichbarkeitsproblem in einem gerichteten Graphen umwandeln. Hier stehen Tiefen- und Breitensuche zur Lösung dieses Problems zur Verfügung (vgl. [3]).

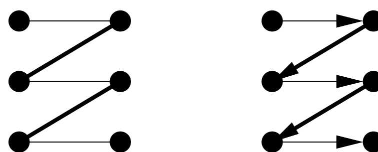

Abbildung 1.7: Umwandlung in ein Erreichbarkeitsproblem

zu 2.: Die Bestimmung der Kanten, die auf einem kürzesten Pfad liegen können: Um die Länge eines kürzesten (augmentiernden) Pfades zu ermitteln, kann man nun Breitensuche (BFS) einsetzen. Mit dieser erzielt man noch einen weiteren Nebeneffekt: Ordnet man jedem Knoten die Länge eines kürzesten Weges von einem freien Knoten zu ihm zu, so muss jede Kante auf einem kürzesten Pfad stets von einem Knoten mit Abstand i zu einem solchen mit Abstand $i+1$ führen. Somit kann man die Breitensuche auch einsetzen, um alle Knoten und Kanten zu identifizieren, die auf einem kürzesten augmentierenden

Pfad liegen, und die anderen Kanten entfernen.
Der so entstehende Graph ist (im gerichteten Sinn) kreisfrei, da die Abstände auf jedem Pfad monoton wachsen. Jeder Pfad von einem freien Knoten in B zu einem freien Knoten in A in diesem Graphen ist automatisch ein kürzester augmentierender Pfad.

zu 3.: Die Bestimmung einer nicht erweiterbaren Menge von Pfaden: Um eine nicht erweiterbare Menge von disjunkten Pfaden zu finden, entfernt man aus dem so konstruierten Graphen so lange Pfade, bis keiner mehr vorhanden ist. Das heißt, man sucht einen Pfad von einem freien Knoten in B zu einem freien Knoten in A, speichert diesen, und löscht dann alle seine Knoten und Kanten aus dem Graphen. Dies ist mit Tiefensuche in Zeit $\mathcal{O}(m)$ möglich, da nach dem Löschen eines Pfades zum Finden eines weiteren Pfades nur bisher nicht untersuchte Kanten betrachtet werden müssen:

Lemma 1.10 *Wird bei einer Tiefensuche der erste gefundene Pfad P gelöscht, so muss zum Auffinden weiterer Pfade keine bis zu diesem Punkt betrachtete Kante später nochmals betrachtet werden.*

> **Beweis:** Wir nehmen mit dem Ziel eines Widerspruchs an, dass über eine bereits betrachtete Kante e später ein augmentierender Pfad P' laufen könnte. e kann nicht zu P gehören, da sie ansonsten gelöscht wurde. Gehört sie aber nicht zu P, so ist die Suche über e erfolglos zurückgekehrt. Dies ist aber ebenfalls nicht möglich, da in diesem Fall von e aus über das verbleibende Reststück von P' bereits vor dem Finden von P ein anderer Pfad gefunden worden wäre. $\square$

Wir können also nach dem Finden eines Pfades einfach erneut eine Tiefensuche von einem anderen freien Knoten aus starten. $\square_{\text{L1.9}}$

Korollar 1.11 *Ein einzelner augmentierender Pfad kann ebenfalls in $\mathcal{O}(m)$ gefunden werden.*

1.4.2 Der Algorithmus von Dinic

Der Algorithmus von Dinic sucht nach maximalen Flüssen in Netzwerken. Wir werden aber sehen,

1. dass man jedes bipartite Matchingproblem in ein Flussproblem transformieren kann.

2. dass der Algorithmus von Dinic ähnliche Strategien anwendet wie der Algorithmus von Hopcroft und Karp

zu 1.: Transformation in ein Flussproblem: Sei $G = (A \dot{\cup} B, E)$ der betrachtete Graph. Wir übernehmen die Knoten und fügen eine Quelle s und eine Senke t hinzu. Für jede Kante $\{b, a\}, a \in A, b \in B$ in G wird eine gerichtete Kante (b, a) eingefügt. Die Quelle wird mit allen Knoten aus B und alle Knoten aus A werden mit der Senke verbunden. Alle Kanten erhalten die Kapazität 1 (vgl. Abb. 1.8).
Sei M ein maximales Matching in diesem Graphen. Setzt man den Fluss auf jeder Kante $(b, a) \in M$ auf 1, ebenso $f(s, b)$ und $f(a, t)$ so erhält man einen Fluss der Größe $|M|$. Damit muss ein maximaler Fluss mindestens die Größe $|M|$ haben. Andererseits kann jeder Knoten nur von einer "Flusseinheit" passiert werden, so dass die flusstragenden Kanten eines ganzzahligen Flusses zwischen A und B stets aus unabhängigen Kanten bestehen und somit ein Matching bilden müssen.

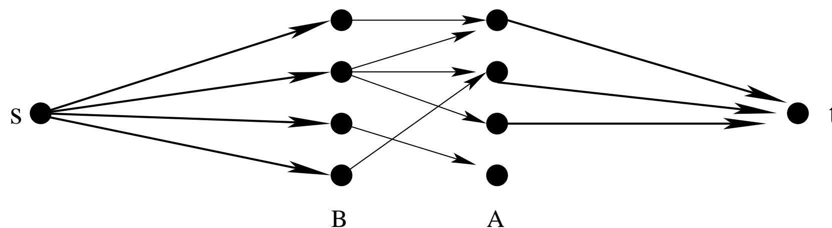

Abbildung 1.8: Umwandlung in ein Flussproblem

zu 2.: Lösen des Flussproblems:

Satz 1.12 *In einem Netzwerk, in dem*

1. *alle Kanten Kapazität 1 haben und*
2. *jeder Knoten außer s und t entweder Eingangsgrad oder Ausgangsgrad 1 hat,*

kann ein maximaler ganzzahliger Fluss in Zeit $\mathcal{O}(\sqrt{n}m)$ gefunden werden.

Beweisskizze: Ausgehend von einem gegebenen Fluss versuchen wir, diesen nach und nach zu vergrößern. Weiterer Fluss kann nur über solche Kanten geschickt werden, deren Kapazität noch nicht ausgelastet ist. Bei Kanten, die bereits einen positiven Fluss tragen, kann dieser reduziert werden. Dies kann man sich auch als einen Rückfluss und somit als einen neuen Fluss über eine entgegengesetzt gerichtete Kante vorstellen. Um herauszufinden, ob weiterer Fluss von der Quelle zur Senke geschickt werden kann, muss geprüft werden, ob über die eben beschriebenen Kanten noch ein Weg von der Quelle zur Senke möglich ist. Hierzu definiert man das sogenannte *residuale* Netzwerk:

Definition 1.13 *Zu einem gegebenen Netzwerk $N = (V, E, c)$ und einem Fluss $f : E \to \mathbb{R}_+$ auf diesem Netzwerk definieren wir das* residuale Netzwerk *als $N' = (V, E', c')$ mit $(v, w) \in E'$ falls $f((v, w)) < c((v, w))$ (noch Kapazität frei) oder $f((w, v)) > 0$ (noch Rückfluss möglich). In diesem Fall sei $c'((v, w)) := c((v, w)) - f((v, w)) + f((w, v))$ die neue Kapazität.*

Es gilt eine analoge Version des Satzes von Berge 1.1:

Beobachtung 1.14 *Ein Fluss ist genau dann maximal, wenn es im residualen Netzwerk keinen Pfad von s nach t gibt.*

Satz 1.7 samt Beweis lassen sich von augmentierenden Pfaden auf Pfade im residualen Netzwerk übertragen. Diese Pfade werden wir im Flusskontext ebenfalls als augmentierend bezeichnen.
Die Laufzeitanalyse entspricht weitgehend der beim Algorithmus von Hopcroft und Karp. Es kann gezeigt werden, dass die Länge eines kürzesten Pfades im residualen Netzwerk immer gleich bleibt oder erhöht wird, wenn man den Fluss über einen kürzesten Pfad vergrößert. Jeder Knoten in $A \cup B$ kann nur von einer Flusseinheit passiert werden, da jeder Knoten entweder Eingangs- oder Ausgangsgrad 1 hat. Da die Flüsse also knotendisjunkte Pfade bilden müssen, läßt sich Satz 1.5_8 ebenfalls analog anwenden. D.h. man augmentiert wiederum über eine nicht erweiterbare Menge augmentierender Pfade. Diese Mengen werden im Fluss-Kontext "*blockierende Flüsse*" genannt.
Um eine Menge kürzester Pfade im residualen Netzwerk zu finden, ordnen wir zunächst jedem Knoten v seinen Abstand zur Quelle $l(v)$ zu. Dies kann mittels BFS in Zeit $\mathcal{O}(m)$ geschehen. Danach entfernen wir alle Kanten, die nicht von einem Knoten mit łi zu einem Knoten mit ł$i + 1$ führen, da dies wiederum eine notwendige Eigenschaft für Kanten auf einem kürzesten Pfad ist.
In dem so gewonnenen *geschichteten Netzwerk* finden wir mittels DFS eine nicht erweiterbare Menge disjunkter s-t-Pfade. Dies ist in $\mathcal{O}(m)$ möglich, da jede Kante nur eine Flusseinheit tragen kann, also nur von einem solchen Pfad verwandt wird. Dann finden wir einen neuen größeren Fluss, indem wir jeweils eine Flusseinheit über jeden gefundenen Pfad schicken. War der Fluss vorher ganzzahlig, so ist er es auch nachher, da jeder Flusswert nur um $+1$ oder -1 geändert wird.
Da nun jeder augmentierende Pfad die Länge $\sqrt{n}$ haben muss und jeder der n Knoten nur von jeweils einem Pfad passiert werden kann, kann es nur noch $\sqrt{n}$ Pfade geben. Diese weiteren $\mathcal{O}(\sqrt{n})$ augmentierenden Pfade lassen sich mit DFS in Zeit $\mathcal{O}(m)$ pro Pfad finden. $\Diamond_{S1.12}$

Sowohl der Algorithmus von Hopcroft und Karp als auch der Algorithmus von Dinic finden also in $\mathcal{O}(\sqrt{n}m)$ Zeit ein maximales Matching in einem bipartiten Graphen.

Teil I

Algorithmen für das nichtbipartite Matchingproblem

Kapitel 2

Einleitung

Im Jahr 1991 haben Feder und Motwani [11][12] die erste und bisher einzige Verbesserung des Algorithmus von Hopcroft und Karp auf bipartiten Graphen erreicht: Sie ersetzen vor dem eigentlichen Matching-Algorithmus vollständige Teilgraphen durch Teilgraphen mit wesentlich weniger Kanten, die sich im Rahmen des Matching-Algorithmus wie die ursprünglichen Teilgraphen verhalten. Da die Laufzeit der Matching-Algorithmen wesentlich von der Zahl der Kanten abhängt, werden diese somit beschleunigt. (Mit diesem Vorgehen konnten auch andere Algorithmen entsprechend verbessert werden.)
Man erhält durch diese Ersetzungen eine kleinere Repräsentation des Graphen, aus der man den ursprünglichen Graphen verlustfrei rekonstruieren könnte. Dieses Vorgehen wird daher auch als "Graphenkompression" bezeichnet.

Das Verfahren von Feder und Motwani reduziert die Zahl der Kanten auf $m\frac{\log \frac{2n^2}{m}}{\log n}$. Auf diese Weise konnten Feder und Motwani einen Algorithmus konstruieren, der in einem Graphen $G = (V, E)$ mit $|V| = n$ Knoten und $|E| = m$ Kanten ein maximales Matching in Zeit $\mathcal{O}\left(\sqrt{n}m\frac{\log \frac{2n^2}{m}}{\log n}\right)$ bestimmt.
Das Ausmaß der Beschleunigung ist von der Dichte des Graphen abhängig, d.h. von dem Verhältnis zwischen Kantenzahl und Knotenzahl. In sehr dichten Graphen mit $m \geq kn^2, 0 < k < 1$ erreicht das Verfahren mit $\mathcal{O}(\frac{\sqrt{n}m}{\log n})$ eine Beschleunigung um einen logarithmischen Faktor. Für weniger dichte Graphen mit $m \leq n^{2-\delta}, 0 \leq \delta \leq 2$ ist die Laufzeit $\mathcal{O}\left(\sqrt{n}m\frac{\log \frac{2n^2}{n^{2-\delta}}}{\log n}\right) = \mathcal{O}\left(\sqrt{n}m\frac{\delta \log n}{\log n}\right)$ mit der des Algorithmus von Hopcroft und Karp [21] identisch.

Unser Ziel in diesem Kapitel ist es, dieses auf bipartite Graphen erfolgreich angewendete Graphenkompressionsverfahren auf den nichtbipartiten Fall zu übertragen. Wir beschleunigen damit ein $\mathcal{O}(\sqrt{n}m)$-Matchingverfahren, konkret den Algorithmus von Blum [2]. Auf diese Weise können wir ein Matching auch im nichtbipartiten Fall in Zeit $\mathcal{O}\left(\sqrt{n}m\frac{\log \frac{2n^2}{m}}{\log n}\right)$ bestimmen.

Wir werden dabei wie folgt vorgehen:

1. Wir beschreiben in Abschnitt 2.1.2$_{22}$ den auf Graphenkompression basierenden Algorithmus von Feder und Motwani [11] zum Finden von Matchings im bipartiten Fall.
2. Wir gehen in Abschnitt 2.2$_{29}$ auf die besonderen Probleme des nichtbipartiten Falles ein.
3. Wir erläutern in Abschnitt 2.3$_{34}$ die Transformation des nichtbipartiten Problems in ein bipartites Problem mit Nebenbedingungen.
4. Wir beschreiben zwei Algorithmen, die die komprimierte bipartite Repräsentation des Graphen nutzen, um ein Matching schnell zu bestimmen. Der erste Algorithmus erzeugt daraus wieder umgekehrt eine komprimierte nichtbipartite Form (Kapitel 3$_{38}$). Der zweite Algorithmus arbeitet durchgehend mit der bipartiten Repräsentation (Kapitel 4$_{58}$).

2.1 Der Algorithmus von Feder und Motwani

2.1.1 Überblick

Die Grundlage des Algorithmus von Feder und Motwani [11] ist die Beobachtung, dass sich vollständige bipartite Teilgraphen von Graphen in Bezug auf Matching-Algorithmen besonders einfach verhalten. Wir bezeichnen diese vollständigen Teilgraphen als "bipartite Cliquen".

Definition 2.1 *Sei $G = (V, E)$ ein Graph. Ein Teilgraph $C = (A\dot{\cup}B, E_C)$ ist eine* bipartite Clique *oder auch* Biclique*, wenn in C alle Knoten aus A mit allen Knoten aus B verbunden sind, d.h. $A \otimes B = E_C \subset E$. Ist die genaue Größe der Clique von Interesse, so setzen wir $\alpha := |A|$ und $\beta := |B|$ und sprechen von einer (α, β)*-Clique.

Bemerkung 2.2 *Es ist damit nicht verlangt, dass A oder B in G unabhängige Mengen sein müssen. Ebenso verlangt dies nicht, dass C ein induzierter Teilgraph sein muss.*

Ein bipartite Clique $C = (A\dot{\cup}B, E_C)$ hat die besondere Eigenschaft, dass es zu je zwei gleichgroßen Teilmengen von A und B $A' \subset A$ und $B' \subset B$ $|A'| = |B'|$ stets ein Matching gibt, das A' und B' überdeckt: Solange es in A' und B' noch unüberdeckte Knoten gibt, gibt es stets eine Kante, die zwei davon verbindet. Im Prinzip genügt es also zu wissen, welche Knoten von A und B durch Kanten innerhalb der Clique verbunden werden müssen. Es kann dann immer ein Matching konstruiert werden, das diese überdeckt.

Man könnte also die Kantenmengen der Cliquen im Prinzip vernachlässigen:

Lemma 2.3 *Sei $G = (V, E)$ ein Graph und $C = (A\dot{\cup}B, E_C)$ eine bipartite Clique in G. Sei M' ein Matching in $G' = (V, E\backslash E_C)$, für das*

$$|M'| + \min(|\{a \in A \mid a \text{ frei}\}|, |\{\text{b} \in \text{B} \mid \text{b frei}\}|)$$

maximal ist. Dann gibt es ein maximales Matching M in G mit $M' \subset M$.

Beweis: Man kann M' als Matching in G um $\min(|\{a \in A|a \text{ frei}\}|, |\{\mathrm{b} \in \mathrm{B}|\mathrm{b} \text{ frei}\}|)$ Kanten zu einem Matching M vergrößern: Sei o.B.d.A angenommen, dass $|\{a \in A|a \text{ frei}\}| \leq |\{\mathrm{b} \in \mathrm{B}|\mathrm{b} \text{ frei}\}|$. Dann kann man die $|\{a \in A|a \text{ frei}\}|$ unüberdeckten Knoten durch Kanten aus E_C mit Knoten aus B verbinden, da dort mindestens genausoviele Knoten unüberdeckt sind.

Um einzusehen, dass M bereits maximal ist, sei M_+ ein maximales Matching. Entfernt man die zu E_C gehörigen Kanten aus M_+, so erhält man ein Matching M'_+ in G' mit $|M'_+| + \min(|\{a \in A|a \text{ frei}\}|, |\{\mathrm{b} \in \mathrm{B}|\mathrm{b} \text{ frei}\}|) = |\mathrm{M}_+|$. Aufgrund der Wahl von $|M'|$ gilt also

$$|M| = |M'| + \min(|\{a \in A|a \text{ frei}\}|, |\{\mathrm{b} \in \mathrm{B}|\mathrm{b} \text{ frei}\})| \geq |\mathrm{M}_+|.$$

□

Da die Anwendung des Kompressionsverfahrens bei Flussalgorithmen anschaulicher darstellbar ist, gehen wir vorläufig zu der in der Einleitung (Kapitel 1.4.2$_{13}$) beschriebenen Repräsentation des Matchingproblems als Flussproblem über: Wir wandeln einen bipartiten Graphen in ein gerichtetes Netzwerk um, indem wir eine Quelle s hinzufügen, die wir mit allen Knoten aus B verbinden, und eine Senke t, die wir mit allen Knoten aus A verbinden. Wir richten die Kanten von s über B nach A und von dort nach t. Alle Kanten erhalten Kapazität 1. Wie wir in Kapitel 1.4.2$_{13}$ gesehen haben, bilden bei einem maximalen ganzzahligen Fluss die Kanten mit positivem Fluss zwischen A und B ein maximales Matching.

Betrachtet man Flüsse in bipartiten Graphen, so haben bipartite Cliquen $C = (A \dot\cup B, E_C)$ die Eigenschaft, dass es zu jeder Wahl von gleichgroßen Teilmengen $B' \subset B$ und $A' \subset A$ möglich ist, $|A'| = |B'|$ Flusseinheiten von B' nach A' zu schicken.
Die Idee von Feder und Motwani ist es nun, eine solche Clique durch einen anderen Teilgraphen zu ersetzen, der ebenfalls diese Eigenschaft hat, aber aus weniger Kanten besteht. Hierbei ist zu beachten, dass wir nur die Kantenmenge der Clique durch eine andere Struktur ersetzen. Die Knoten und deren inzidente, aber nicht zur Clique gehörenden Kanten bleiben davon unberührt.

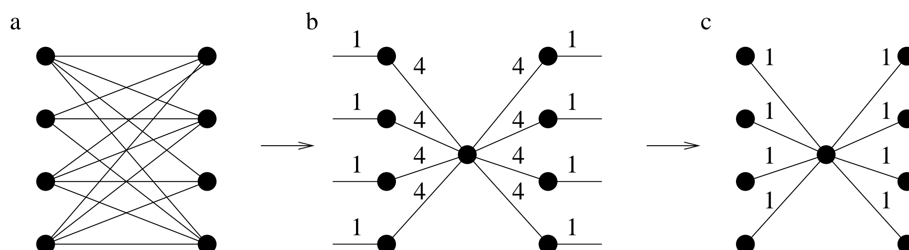

Abbildung 2.1: Ersetzen einer Clique durch einen Stern im Flussfall

Abb. 2.1 veranschaulicht, dass man die Kanten zwischen den beiden Seiten einer Clique (A', B') - mit von B' nach A' gerichteten Kanten - durch einen “Stern” ersetzen kann, ohne dass sich die Größe eines maximalen Flusses ändert: Zunächst ist klar, dass man den

gesamten Fluss durch einen zentralen Knoten leiten könnte, wenn man die Kapazitäten der Kanten zu diesem Knoten hoch genug wählt (Schritt von a nach b). Im vorliegenden Fall haben alle Knoten Eingangs- bzw. Ausgangsgrad 1, so dass die Kapazitäten zum zentralen Knoten wieder auf 1 reduziert werden können, da ohnehin nicht mehr Fluss über diese Kanten fließen kann (b nach c). Diese Reduktion wiederum erlaubt es, einen für den Fall c gefundenen Fluss wieder auf Fall a zu transformieren: Man verbindet paarweise Knoten, die Fluss zum zentralen Knoten schicken, und solche, die von diesem Knoten Fluss empfangen.
Es ist also möglich,

- die Kanten der bipartiten Clique durch diesen “Stern” zu ersetzen,
- in dem so entstehenden Graphen einen maximalen Fluss zu bestimmen,
- und aus diesem Fluss einen maximalen Fluss im ursprünglichen Graphen zu rekonstruieren.

Die Zahl der Kanten im Graphen wird durch diese Kompression verringert, da eine Clique mit $|A'||B'|$ Kanten durch einen Graphen mit $|A'| + |B'|$ Kanten ersetzt werden konnte.
Wir belassen es zunächst bei dieser anschaulichen Beschreibung des Ersetzungsvorganges, wir werden ihn formal exakt mit Definition 3.8$_{43}$ beschreiben. Den jeweiligen zentralen Knoten der Sterne werden wir entsprechend als *Zentralknoten* bezeichnen.

Bemerkung 2.4 *Eine solche Ersetzung durch Sterne ist im Kontext alternierender Pfade nicht möglich, da stets nur ein Pfad den zentralen Knoten des Sterns passieren kann.*

Zum Finden von Flüssen in diesen Graphen wollen wir nun den Algorithmus von Dinic [8] einsetzen.
Bei der Laufzeitanalyse dieses Algorithmus in Kapitel 1.4.2$_{13}$ sind wir davon ausgegangen, dass es höchstens $\sqrt{n}$ augmentierende Pfade der Länge $\sqrt{n}$ geben kann. Darauf basierend konnten wir zeigen, dass - sobald es nur noch Pfade mindestens dieser Länge gibt - das Matching in Zeit $\mathcal{O}(\sqrt{n}m)$ zu einem maximalen Matching vergrößert werden kann.
Die Zahl der verbleibenden augmentierenden Pfade konnte abgeschätzt werden, da jeder Knoten Ein- oder Ausgangsgrad 1 hatte, so dass jeder Knoten nur von jeweils einem augmentierenden Pfad durchquert werden konnte. Dies ist nun nicht mehr der Fall, da einige Knoten ,nämlich die Zentralknoten der Sterne, nicht mehr Ein- oder Ausgangsgrad 1 haben.
Wie Feder und Motwani [11] gezeigt haben, funktioniert das Verfahren von Dinic aber auch in diesem Fall in $\mathcal{O}(\sqrt{n}m)$. Es sind nie zwei Zentralknoten zueinander adjazent. Damit besitzt zumindestens jeder zweite Knoten auf jedem Pfad noch die Eigenschaft, Ein- oder Ausgangsgrad 1 zu haben. Jeder augmentierende Pfad der Länge $\sqrt{n}$ benötigt also mindestens $\frac{\sqrt{n}}{2}$ der Knoten, die nur von einer Flusseinheit passiert werden können.

Da es nach wie vor n Stück davon gibt, kann es höchstens $2\sqrt{n}$ augmentierende Pfade dieser Länge geben.

Um diese Ersetzung einer Clique zu einer Beschleunigung des gesamten Algorithmus auszubauen, ist es notwendig, eine ausreichend große Zahl von möglichst großen Cliquen zu finden und zu ersetzen.

Definition 2.5 *Eine* bipartite Cliquenzerlegung *eines Graphen G ist eine Menge von bipartiten Cliquen $\mathcal{C} = C_1, \ldots, C_r \subset G$, so dass jedes $e \in E(G)$ für genau ein $i \in \{1, \ldots, r\}$ zu $E(C_i)$ gehört.*

Als gleichzeitiges Maß für Größe und Zahl der Cliquen führen wir den Begriff der Gesamtgröße ein:

Definition 2.6 *Die* Gesamtgröße $||\mathcal{C}||$*einer Cliquenzerlegung $\mathcal{C}$ ist die Summe der Ordnungen der Teilgraphen, also $||\mathcal{C}|| := \sum_{C \in \mathcal{C}} |V(C)|$.*

Bemerkung 2.7 *Ein Knoten kann in mehreren Cliquen enthalten sein und somit mehrmals zur Gesamtgröße beitragen.*

Lemma 2.8 *Ersetzt man ausgehend von einer Cliquenzerlegung $\mathcal{C}$ mit Gesamtgröße m^* die Kantenmengen der Cliquen durch sternförmige Ersatzgraphen, so erhält man einen Graphen mit m^* Kanten.*

Beweis: Jeder Knoten erhält für jede Clique, zu der er gehört, eine Kante zum (bzw. vom) Zentralknoten des entsprechenden Sterns. Man betrachte eine $\{0, 1\}$-Matrix D mit $|V|$ Spalten und $|\mathcal{C}|$ Zeilen, in der ein Eintrag (i, j) genau dann den Wert 1 hat, wenn der $i$te Knoten zur j-ten Clique gehört. Dann entspricht die gerade gegebene Beschreibung der Kantenanzahl einer spaltenweisen Summierung und die Definition der Gesamtgröße einer zeilenweisen Summierung.

$$m^* = \sum_{i=1}^{|V|} \sum_{j=1}^{|\mathcal{C}|} D(i, j) = \sum_{j=1}^{|\mathcal{C}|} \sum_{i=1}^{|V|} D(i, j) = ||\mathcal{C}||$$

□

Korollar 2.9 *Ist ein bipartiter Graph $G = (V, E)$ und eine Cliquenzerlegung $\mathcal{C}$ mit Gesamtgröße m^* gegeben, so kann ein maximales Matching in diesem Graphen in Zeit $\mathcal{O}(\sqrt{n}m^*)$ gefunden werden.*

Beweis: Man wandelt das Matchingproblem in ein Flussproblem um und ersetzt wie oben beschrieben die Cliquen durch Sterne. In dem so entstandenen Graphen mit $\mathcal{O}(m^*)$ Kanten kann mittels des Algorithmus von Dinic ein maximaler Fluss in Zeit $\mathcal{O}(\sqrt{n}m^*)$ gefunden werden. Dies läßt sich in Zeit $\mathcal{O}(m)$ in ein maximales Matching in G umwandeln. □

2.1.2 Durchführung der Graphenkompression

An dieser Stelle präsentieren wir das Graphen-Kompressionsverfahren von Feder und Motwani [11], das Grundlage ihres Algorithmus für das bipartite Matchingproblem ist.

Zunächst zeigen wir, dass eine hohe Kantenzahl hinreichend dafür ist, dass der Graph ausreichend große Cliquen enthält. Danach geben wir wieder, wie man eine solche findet und entfernt. Wir zeigen dann, wie man dieses Finden und Entfernen iterieren kann, um letztendlich den Graphen in solche Cliquen und einen relativ kleinen Restgraphen zu zerlegen.
Um der formalen Definition einer Cliquenzerlegung (vgl. Definition 2.6) zu entsprechen, können die Kanten dieses Restgraphen dann noch als 1-Kanten Cliquen interpretiert werden. Wir nehmen an, dass $|A| = |B| = n$.

2.1.2.1 Die Existenz einer großen Clique

Der Algorithmus von Feder und Motwani findet Cliquen ungleichmäßiger Größe:

Definition 2.10 ([11]) *Sei* $0 \leq \delta \leq 1$ *und* $k(n, m, \delta) := \left\lfloor \frac{\delta \log n}{\log \frac{2n^2}{m}} \right\rfloor$. *Dann ist eine* δ*-Clique eine* (α, β)*-Clique mit* $\alpha = \lceil n^{1-\delta} \rceil$ *und* $\beta = k(n, m, \delta)$.

Wir zeigen zunächst, dass jeder bipartite Graph für jedes δ eine δ-Clique enthält. Dies erlaubt es uns später, den Parameter δ beliebig zu wählen. Der Beweis des Satzes liefert zudem die Grundidee für den Algorithmus zum Finden einer δ-Clique:

Satz 2.11 (Feder, Motwani[11]) *Jeder bipartite Graph hat eine* δ*-Clique.*

Beweis: Der Beweis wird zunächst als reiner Existenzbeweis mit Abzählargumenten geführt. Wir werden später zeigen, wie sich daraus ein Konstruktionsverfahren für eine δ-Clique herleiten läßt.
Eine (α, β)-Clique ist im Prinzip eine Teilmenge von B der Größe β, die aus Knoten besteht, die (mindestens) α gemeinsame Nachbarn in A haben. Wenn wir zeigen können, dass die durchschnittliche Zahl gemeinsamer Nachbarn aller Teilmengen der Größe β größer ist als α, so können wir sicher sein, dass mindestens eine dieser Mengen eine entsprechend große gemeinsame Nachbarschaft haben muss.
Aus algorithmischen Gründen, die wir erst in Kapitel 2.1.2.2[24] erläutern können, bestimmen wir allerdings nicht den Durchschnitt der Nachbarschaftsgröße über alle Teilmengen, sondern sogar über alle geordneten Teilmengen. Dies ändert nichts an der zugrundeliegenden Argumentation.

Hierzu sei $x^{\underline{k}}$ definiert als $x(x-1)(x-2)\cdots(x-k+1)$, also die Zahl der k-elementigen geordneten Teilmengen einer x-elementigen Menge. Des Weiteren bezeichne $B^{\underline{k}} := K_1, \ldots, K_{n^{\underline{k}}}$ die Menge der k elementigen geordneten Teilmengen von B und $(a_i)_{i=1\cdots n}$ die Knoten aus A.

Den Grad des i-ten Knoten $d(a_i)$ kürzen wir mit d_i ab. Dann kann die Existenz einer Teilmenge von B mit vielen gemeinsamen Nachbarn in A durch doppelte Summation über eine $\{0,1\}$- Matrix gezeigt werden: Sei für $1 \le i \le n$ und $1 \le j \le n^{\underline{k}}$

$$M_{i,j} := \begin{cases} 1 & \text{falls } K_j \subseteq \Gamma(a_i) \\ 0 & \text{sonst.} \end{cases}$$

D.h. es steht genau dann eine 1 in Eintrag (i,j), wenn Knoten a_i ein gemeinsamer Nachbar aller Knoten aus der j-ten Teilmenge ist. So ist die Zahl der Einsen in einer Zeile i die Zahl der geordneten Teilmengen, die völlig in der Nachbarschaft von a_i liegen. Die Zahl der Einsen in einer Spalte j ist die Zahl von Knoten, in deren Nachbarschaft die die j-te Teilmenge liegt.
Sei die Gesamtzahl von Einsen in dieser Matrix mit $C_k := \sum_{i=1}^{n} \sum_{j=1}^{n^{\underline{k}}} M_{i,j}$ bezeichnet.
Für einen Knoten $a_i \in A$ ergibt sich $\sum_{j=1}^{n^{\underline{k}}} M_{ij} = d(a_i)^{\underline{k}} =: d_i^{\underline{k}}$, da man aus den $d_i := d(a_i)$ Nachbarn von a_i genau diese Zahl k-elementiger geordneter Teilmengen wählen kann. Summiert man über alle Knoten, so erhält man wiederum die Gesamtsumme: $\sum_{i=1}^{n} d_i^{\underline{k}} = C_k$.
Die Summe über die Zeilensummen ist gleich der Summe über die Spaltensummen. Somit erhält man die durchschnittliche Zahl von Knoten, in deren gemeinsamer Nachbarschaft eine geordnete Teilmenge liegt, indem man die Gesamtsumme durch die Zahl der geordneten k-elementigen Teilmengen teilt:

$$\frac{C_k}{n^{\underline{k}}} = \frac{\sum_{i=1}^{n} d_i^{\underline{k}}}{n^{\underline{k}}}$$

Um den Ausdruck $\sum_{i=1}^{n} d_i^{\underline{k}}$ in Abhängigkeit von der Gesamtkantenzahl abzuschätzen, muss nun betrachtet werden, wie er sich - je nachdem wie die Kanten auf die Knoten verteilt sind - verhält.

Lemma 2.12 (Feder, Motwani[11]) *Bei konstanter Kantenzahl im Graphen $m = \sum_{i=1}^{n} d_i$ nimmt der Ausdruck $\sum_{i=1}^{n} d_i^{\underline{k}}$ sein Minimum an, wenn die d_i alle gleich groß sind.*

Somit ergibt sich für den Durchschnitt

$$\frac{C_k}{n^{\underline{k}}} = \frac{\sum_{i=1}^{n} d_i^{\underline{k}}}{n^{\underline{k}}} \ge \frac{n\left(\left\lfloor \frac{m}{n} \right\rfloor\right)^{\underline{k}}}{n^{\underline{k}}}$$
$$\ge \frac{n(\frac{m}{n} - k)^k}{n^k}$$

Wegen $k(n,m,\delta) \le \frac{\delta \log n}{\log \frac{2n^2}{m}} \le \frac{m}{2n}$ und $\frac{-\delta \log n}{\log \frac{m}{2n^2}} = \log_{\frac{m}{2n^2}} n^{-\delta}$ erhält man als untere Grenze für den Durchschnitt

$$\frac{C_k}{n^{\underline{k}}} \ge n \left(\frac{\left(\frac{m}{n} - \frac{m}{2n}\right)}{n} \right)^k = n \left(\frac{m}{2n^2} \right)^k \ge n^1 \left(\frac{m}{2n^2} \right)^{\frac{-\delta \log n}{\log \frac{m}{2n^2}}} = n^{1-\delta}$$

Nach dem Schubfachprinzip muss aber mindestens eine Zeile der Matrix mindestens so viele Einsen enthalten wie dieser Durchschnitt. D.h. es gibt eine geordnete Teilmenge der Größe k, die in der Nachbarschaft von mindestens $\lceil n^{1-\delta} \rceil$ Knoten liegt. $\square_{\text{S2.11}}$

2.1.2.2 Das Finden einer Clique

Im vorangegangenen Kapitel haben wir festgestellt, dass jeder bipartite Graph eine δ-Clique enthält, da es mindestens eine k-elementige Teilmenge von B mit ausreichend vielen gemeinsamen Nachbarn geben muss.
Für einen Algorithmus, der die Cliquen explizit ersetzten soll, genügt diese reine Existenzaussage natürlich nicht, sondern eine solche Menge muss auch gefunden werden. Da ein vollständiges Durchsuchen aller geordneten k-elementigen Teilmengen aus Zeitgründen nicht möglich ist, würde sich die folgende Variante einer binären Suche anbieten: Man wählt beliebig einen Knoten $b \in B$ und bestimmt mit den im vorangegangenen Kapitel gezeigten Rechnungen, wie groß die durchschnittliche gemeinsame Nachbarschaft aller k-elementigen geordneten Teilmengen von B ist, die b enthalten. Sei dieser Wert $\bar{C}_b$. Ist $\bar{C}_b$ größer als der Durchschnitt aller Mengen, so fügt man b in B' ein und sucht ein weiteres Element. Ansonsten weiß man, dass der Durchschnitt $\bar{C}_{\not b}$ über alle Mengen, die b nicht enthalten ausreichend groß sein muss, da der Gesamtdurchschnitt der Mittelwert von $\bar{C}_b$ und $\bar{C}_{\not b}$ ist.

Leider könnten für diese Bestimmung von B' dann $\mathcal{O}(n)$ dieser aufwändigen Auswahlprozesse benötigt werden, wenn man gerade zufällig die Knoten zuletzt ausprobiert, die tatsächlich die gesuchte Menge bilden. Feder und Motwani[11] haben daher das Auswahlverfahren so abgeändert, dass dieses im Prinzip binärer Suche ähnelnde Verfahren nicht nur zur Suche der gesamten Menge, sondern auch zur Suche jedes einzelnen Elementes für diese Menge verwandt wird.
Um das erste Element für B' zu finden, wird die Menge B in zwei gleichgroße Mengen zerlegt: B_0 und B_1. Damit erhält man auch eine Zerlegung der Menge aller geordneten k-elementigen Teilmengen von B, $B^{\underline{k}}$, in zwei gleichgroße Mengen: In die Mengen, die als erstes Element ein Element aus B_0 haben und die Mengen, die als erstes Element eins aus B_1 haben. Für eine dieser beiden Mengen von Teilmengen muss gelten, dass die durchschnittliche Größe der gemeinsamen Nachbarschaften ihrer Teilmengen größer gleich dem Gesamtdurchschnitt ist. Ist dies o.B.d.A. für B_1 der Fall, so zerlegt man diese wieder in zwei Mengen B_{10} und B_{11} und wiederholt das Ganze so lange, bis die verbleibende Menge nur noch einen einzelnen Knoten b_1 enthält. Dieser wird zum ersten Knoten von B'. Danach beschränkt man sich auf die Mengen, die mit b_1 anfangen und sucht auf gleiche Weise b_2 usw. Die den Elementen hierbei mehr oder weniger aufgezwungene Reihenfolge ist der eigentliche Grund, warum hier mit geordneten Teilmengen gearbeitet wird.

Um dieses Verfahren formal beschreiben zu können, muss zunächst ein relativ großer Aufwand an Definitionen betrieben werden. Es sei daher zur Vereinfachung angenommen, dass $n = 2^r$ für ein $r \in \mathbb{N}$.

Definition 2.13 *Sei $w \in \{0,1\}^{\leq r}$ eine binäre Zeichenkette mit Länge kleiner gleich r. Dann bezeichne B_w die Menge der $b_i \in B$, für die gilt, dass w ein Prefix des Binärcodes von i ist. Insbesondere ist z.B. die dem leeren Wort ϵ zugeordnete Menge $B_\epsilon = B$.*
$d_{i,w}$ sei die Zahl der Nachbarn, die der Knoten $a_i \in A$ in B_w hat.

Definition 2.14 *Die Verknüpfung zweier Strings stellen wir mit dem Operator "." dar, also $a.b = ab$.*

Definition 2.15 *Für $U \subset A$ und $\mathcal{C} \subset B^{\underline{k}}$ bezeichne $C(U, \mathcal{C})$ die Zahl der Einsen in der durch U und $\mathcal{C}$ definierten Untermatrix von M.*

Definition 2.16 *Wurde bereits bestimmt, dass die ersten t Elemente der gesuchten Menge B' die Knoten $(b_1, \ldots, b_t)$ sein sollen, so bezeichne B_t die Menge $B \backslash \{b_1, \ldots, b_t\}$. A_t sei die Menge der $a \in A$, so dass $b_1, \ldots, b_t \in \Gamma(a)$.*

Definition 2.17 *Mit $S_{t,w}$ sei die Menge derjenigen geordneten k-elementigen Teilmengen von B bezeichnet, deren $t-1$ ersten Elemente $b_1, \ldots b_{t-1}$ sind und deren t-tes Element in B_w enthalten ist.*

Der Algorithmus bestimmt nun nach und nach die Elemente der Menge B'. Durch jede Wahl eines weiteren Elements wird die Zahl der Mengen, die noch in Frage kommen, verringert. Wir versuchen nun, die Wahl jeweils so zu treffen, dass die durchschnittliche Größe einer gemeinsamen Nachbarschaft bei diesen Mengen mindestens so hoch bleibt, wie dies vorher beim Gesamtdurchschnitt der Fall war.
Zu Beginn ist $\bar{C} := \frac{C_k}{n^{\underline{k}}} = \frac{C(A, S_{0,\epsilon})}{|S_{0,\epsilon}|}$ dieser Mittelwert.
Jede Menge $S_{t,w}$ läßt sich nun in die zwei gleichgroßen Mengen $S_{t,w.0}$ und $S_{t,w.1}$ zerlegen. Nach dem Schubfachprinzip muss für eine der beiden der Mittelwert der Anzahl der Einsen in der zugehörigen Teilmatrix mindestens genauso groß sein wie für die Gesamtmenge. Wählt man also jeweils diejenige Erweiterung von w, die zu der Menge mit dem größeren Wert korrespondiert, so erhält man schließlich für $|w| = r$ ein einzelnes Element aus B, das man dann als $(t+1)$tes Element wählen kann. Nach k Iterationen dieses Vorgehens ist somit eine Menge mit der gewünschten Eigenschaft gefunden.
Voraussetzung für dieses Vorgehen ist es, zu jedem t und w die Größe $C(A, S_{t,w})$ bestimmen zu können. Zu diesem Zweck werden sogenannte "Nachbarschaftsbäume" verwendet.

Definition 2.18 *Ein "Nachbarschaftsbaum" zu einem Knoten a_i ist ein Binärbaum der Tiefe r, dessen Knoten in der Weise mit den Strings $w \in \{0,1\}^{\leq r}$ bezeichnet werden, dass die beiden Söhne eines Knotens jeweils den String ihres Vaters verlängert um 0 bzw 1 erhalten. Jedem Knoten zugeordnet sei der Wert $d_{i,w}$, der angibt, wieviele der Knoten aus B_w in der Nachbarschaft von a_i also $\Gamma(a_i)$, liegen.*

Lemma 2.19 $C(A_t, S_{t,w}) = \sum_{a_i \in A_t} d_{i,w}(d_i - 1)^{\underline{k-t}}$

> **Beweis:** Die Einsen in dieser Submatrix können zeilenweise addiert werden. Es genügt also zu zeigen, dass die Zahl der Einsen in der Zeile von a_i gleich $d_{i,w}(d_i - 1)^{\underline{k-t}}$ ist. Die ersten $t-1$ Elemente jeder Menge sind gleich und per Definition in der Nachbarschaft von a_i enthalten. Das $t$te Element kommt aus $\Gamma(a_i) \cap B_w$, wofür es $d_{i,w}$ Möglichkeiten gibt. Die verbleibenden $k-t$ Elemente können beliebig aus der Menge der $d_i - 1$ verbleibenden Nachbarn gewählt werden. Hierbei ist zu beachten, dass d_i ebenso wie alle $d_{i,w}$ jedesmal angepasst wird, wenn ein Element für B' gefunden wurde. □

Bemerkung 2.20 *Der Wert $\sum_{a_i \in A_t} d_{i,w}(d_i - 1)^{\underline{k-t}}$ kann in Zeit $\mathcal{O}(nk)$ berechnet werden, wenn $d_{i,w}$ gegeben ist.*

Sobald ein weiteres Element von B' bestimmt wurde, müssen die $d_{i,w}$-Werte neu bestimmt werden. Dieser Vorgang wird auch als die "Aktualisierung der Nachbarschaftsbäume" bezeichnet. Die einzigen Änderungen, die sich durch die späteren Modifikationen des Algorithmus für den nichtbipartiten Fall ergeben werden, betreffen dieses Aktualisieren der Nachbarschaftsbäume. Wir stellen daher hier diesen Teil genauer dar:
Sobald ein Knoten für B' gefunden wurde, müssen vor der nächsten Suche alle n Nachbarschaftsbäume an diese Veränderung angepaßt werden. Für jeden Knoten b' müssen bei jedem Nachbarschaftsbaum die Werte aller Knoten auf dem Pfad, der mit $B_w = \{b'\}$ endet, um jeweils 1 verringert werden. Dies benötigt pro Baum eine Laufzeit von $\mathcal{O}(\log n)$, kann also bei insgesamt n Bäumen und k Aufrufen in $\mathcal{O}(kn \log n)$ durchgeführt werden.

Algorithmus 1 Cliquenabspaltung nach [11]

1: **procedure** SPLIT(k)
2: Erzeuge Zeiger $z_i, i = 1, \ldots, n$ zur Wurzeln des Nachbarschaftsbaums von a_i
3: $A_t := A$
4: $B_t := B$
5: **for** $t := 1$ to k **do**
6: $w := \epsilon$
7: $c_0 := C(A_t, S_{t,w\bullet 0})$
8: $c_1 := C(A_t, S_{t,w\bullet 1})$
9: **if** $c_0 \geq c_1$ **then**
10: $w := w \bullet 0$
11: **else**
12: $w := w \bullet 1$
13: **end if**
14: Aktualisiere die Zeiger der Nachbarschaftsbäume z_i auf den Sohn w
15: **if** $|w| < r$ **then**
16: GOTO 7
17: **end if**
18: $b_t := b'$ mit $B_w = \{b'\}$
19: $B_{t+1} := B_t \backslash b_t$
20: $A_{t+1} := \{u \in A_t | b_t \in \Gamma(u)\}$
21: Aktualisiere die Nachbarschaftsbäume und setze die Zeiger wieder auf die Wurzeln.
22: **end for**
23: Return $A' = A_k, B' = \{b_1, \ldots b_k\}$
24: **end procedure**

Satz 2.21 (Feder, Motwani 1991[11]) *Der Clique-Stripping Algorithmus bestimmt eine δ-Clique in Zeit $\mathcal{O}(nk^2 \log n)$ und aktualisiert gleichzeitig die Nachbarschaftsbäume. Die erste Berechnung der Bäume ist in der obigen Schranke nicht berücksichtigt.*

Beweis: Für jedes der k zu bestimmenden Elemente muss $\log n$ mal der Wert von $\sum_{a_i \in A_l} d_{i,w}(d_i - 1)^{\underline{k-t}}$ bestimmt werden, was Zeit $\mathcal{O}(nk)$ kostet. Das Ak-

tualisieren der Nachbarschaftsbäume benötigt daher insgesamt $\mathcal{O}(kn \log n)$ Schritte. □

2.1.3 Das Komprimieren von Graphen

Um einen Graphen zu komprimieren, wendet man den Clique-Stripping Algorithmus so oft an, bis die Größe der durch Satz 2.11_{22} garantierten Cliquen so weit gesunken ist, daß ein weiteres Ersetzen keine Verbesserung mehr erzielt.
Zur Vorbereitung ist es zunächst notwendig, die Nachbarschaftsbäume zu erzeugen.

Lemma 2.22 *Die n Nachbarschaftsbäume können in Zeit $m \log n$ erzeugt werden.*

Beweis: Jede Kante (u, b) der m Kanten trägt jeweils 1 zu jedem Knoten auf dem Pfad von der Wurzel des zu u gehörenden Baumes T_u zu dem Knoten mit Markierung $\{b\}$ bei (s. z.B. das Paar $(u, b13)$ in Abb. 2.2, dargestellt ist der Nachbarschaftsbaum von u, T_u). Jeder dieser Pfade hat Länge $\log n$. Wir können also mit Bäumen anfangen, bei denen alle Werte Null sind und erhöhen für jede Kante alle Werte auf dem zugehörigen Pfad um 1. Somit können die Bäume durch sukzessives Erhöhen der Knotenwerte in der gewünschten Zeit erzeugt werden. □

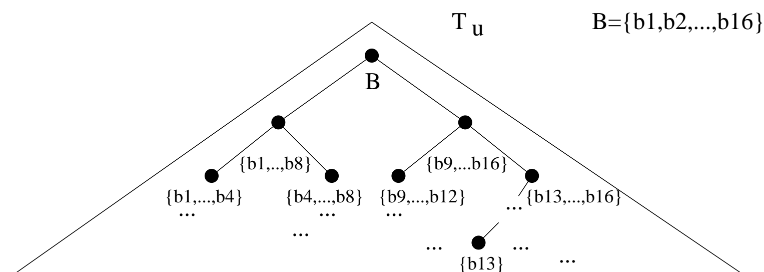

Abbildung 2.2: Nachbarschaftsbaum von Knoten u

Nun kann mit Hilfe der Nachbarschaftsbäume der Clique-Stripping Algorithmus (Algorithmus 2) angewandt werden. Zeile 10 ist notwendig, da die Nachbarschaftsbäume auch für einige Knoten geändert wurden, die später nicht in die jeweilige Clique aufgenommen wurden. Dieses Zurücksetzen kann aber natürlich mit dem selben Zeitaufwand geschehen, wie die vorangegangene Aktualisierung der Nachbarschaftsbäume.

Damit erhält man das bereits zu Beginn erwähnte Resultat, zu dessen vollständigem Beweis wir auf [11] verweisen:

Satz 2.23 (Feder, Motwani 1991[11]) *Sei G ein bipartiter Graph mit $2n$ Knoten und $m \geq n^{1-\delta}$ Kanten. Dann partitioniert der Algorithmus 2 die Kanten in kantendisjunkte Cliquen mit einer Gesamtgröße von $\mathcal{O}\left(\frac{m}{k(n,m,\delta)}\right)$ und benötigt hierzu Zeit $\mathcal{O}(mn^{\delta} \log^2 n)$.*

Algorithmus 2 Cliquenpartitionierung nach [11]

1: $i := 0;$
2: $n := |A|$
3: $m' := |E|$
4: Berechne die Nachbarschaftsbäume
5: **while** $m' \geq n^{2-\delta}$ **do**
6: $\quad i := i + 1$
7: $\quad k' := \left\lfloor \frac{\delta \log n}{\log \frac{2n^2}{m'}} \right\rfloor$
8: $\quad$ Finde Clique $C_i = (A_i, B_i)$ mit SPLIT(k')
9: $\quad E := E \backslash A_i \times B_i$
10: $\quad$ Setze die Nachbarschaftsbäume für alle Knoten außer denen in A_i zurück.
11: $\quad m' := |E|$
12: **end while**
13: Alle verbleibenden Kanten bleiben unangetastet (oder zählen als (1,1)Cliquen)

Beweisskizze:
Der Algorithmus hält die angegebene Laufzeitschranke ein:

1. Wie in Lemma 2.22 gezeigt, können die Nachbarschaftsbäume in Schritt 4 in Zeit $\mathcal{O}(m \log n)$ erzeugt werden.
2. Jede Ausführung der Splitting-Routine benötigt $\mathcal{O}(nk'^2 \log n)$ Zeit. Dabei werden mindestens $k'n^{1-\delta}$ Kanten entfernt. Verteilt man die Operationen auf die Kanten, so ergeben sich $\mathcal{O}\left(\frac{nk'^2 \log n}{n^{1-\delta}k}\right) = \mathcal{O}(n^\delta k' \log n) \subset \mathcal{O}(n^\delta \log^2 n)$ Operationen pro Kante, da k' stets durch $\mathcal{O}(\log n)$ beschränkt bleibt.
3. Es können insgesamt höchstens m Kanten entfernt werden, so dass die Gesamtzeit für das Ausführen der Schleife durch $\mathcal{O}(mn^\delta \log^2 n)$ beschränkt ist.

Der Beweis der Kompressionsgüte erfolgt auf dem Weg, dass man beobachtet, dass jede entfernte Kante amortisiert durch höchstens $\frac{k'+n^{1-\delta}}{k'n^{1-\delta}}$ Kanten in der komprimierten Darstellung ersetzt wird. Die Zahl der am Ende verbleibenden, nicht von der Kompression erfassten Kanten ist in $\mathcal{O}(n^{2-\delta})$. $\diamondsuit$

Bemerkung 2.24 *Die bei der Kompression übrig gebliebenen Kanten werden dabei als Cliquen mit nur einer Kante interpretiert. Dies steht nicht im Widerspruch zur Definition der Cliquen.*

Korollar 2.25 *Mit Hilfe des Algorithmus "Partition" kann man in Zeit $\mathcal{O}(mn^\delta \log^2 n)$ zu einem Graphen G einen Graphen G' mit $V(G') \supset V(G)$ konstruieren, der höchstens $\mathcal{O}\left(\frac{m}{k(n,m,\delta)}\right)$ Kanten hat und in dem zwei Knoten v und w aus G genau dann durch einen Pfad der Länge höchstens 2 verbunden sind, wenn sie in G adjazent waren.*

Beweis: Man erhält G', indem man, wie in Kapitel 2.1.1[18] beschrieben, die Cliquen durch "Sterne" ersetzt. Entsprechend werden die bei der Kompression übrig gebliebenen 1-Kanten-Cliquen durch Pfade der Länge 2 ersetzt. □

Wir können also ein Matching in einem Graphen schnell bestimmen, indem wir

1. den Graphen komprimieren,
2. den komprimierten Graphen in ein Netzwerk umwandeln,
3. in diesem mit dem Algorithmus von Dinic einen maximalen Fluss bestimmen,
4. diesen Fluss in den nicht komprimierten Graphen zurück übertragen
5. und dessen flusstragende Kanten wieder als Matching interpretieren.

2.2 Eigenschaften Nichtbipartiter Graphen

Bevor wir beschreiben, wie sich das Graphenkompressionsverfahren auf Matching-Algorithmen für nichtbipartite Graphen übetragen läßt, stellen wir dar, welche zusätzlichen Schwierigkeiten im nichtbipartiten Fall auftreten. Dazu betrachten wir zunächst, welche Unterschiede zwischen bipartiten und nichtbipartiten Graphen bestehen.

In bipartiten Graphen haben alle Kreise eine gerade Zahl von Kanten, da stets abwechselnd Knoten aus den beiden Partitionen benutzt werden müssen. Es gilt sogar der Satz:

Satz 2.26 *Ein Graph ist genau dann bipartit, wenn er keine Kreise ungerader Länge enthält.*

Beweis: Es bleibt, die Rückrichtung zu zeigen. Sei also $G = (V, E)$ ein Graph ohne Kreise ungerader Länge. Wir beweisen zunächst, dass alle Pfade zwischen zwei beliebigen Knoten entweder ausschließlich gerade oder ausschließlich ungerade Länge haben: Sei mit dem Ziel eines Widerspruchs angenommen, dass u' und v' ein Gegenbeispiel bilden mit zwei $u - v$ Pfaden P_1 und P_2. Dabei sei o.B.d.A angenommen, dass P_1 gerade Länge hat und P_2 ungerade Länge hat. Weiter nehmen wir an, dass u und v so gewählt wurden, dass $|P_1| + |P_2|$ minimal ist unter allen Gegenbeispielen.
Haben P_1 und P_2 nur u' und v' gemeinsam, so bilden sie zusammen einen Kreis ungerader Länge im Widerspruch zur ursprünglichen Annahme. Haben P_1 und P_2 einen weiteren Knoten z gemeinsam, so bildet eins der Paare (u', z) oder (v', z) ein Gegenbeispiel mit kürzeren Pfadlängen: Dies gilt, da von $P_2|_{u,z}$ und $P_2|_{z,v}$ einer gerade und einer ungerade Länge haben muss, während $P_1|_{u,z}$ und $P_1|_{z,v}$ entweder beide gerade oder beide ungerade Länge haben müssen.

Wählt man also einen beliebigen Knoten $x \in V$, so zerfällt die Knotenmenge eindeutig in eine Menge A von Knoten, die zu x einen geraden Abstand haben und in eine Menge B von Knoten, die zu x einen ungeraden Abstand haben. Um einzusehen, dass diese beiden Mengen die gewünschten zwei Partitionen sind, ist zu zeigen, dass es keine Kanten innerhalb von A bzw B gibt. Seien o.B.d.A $a_1, a_2 \in A$ zwei Knoten mit einer gemeinsamen Kante (a_1, a_2). Dann gibt es von x aus zu a_1 nach Definition von A einen Pfad gerader Länge. Es gibt aber auch einen Pfad P_g gerader Länge zu a_2. Benutzt dieser (a_1, a_2) nicht, so ist $P_g \circ (a_2, a_1)$ ein Pfad ungerader Länge zu a_1. Ansonsten ist (a_1, a_2) die letzte Kante von P_g zu a_2 und P_g ohne diese Kante ist ein Pfad ungerader Länge zu a_1. Dann gibt es aber zwischen x und a_1 sowohl einen Pfad gerader als auch einen Pfad ungerader Länge. Dies kann nicht sein. □

Die Ursache für das prinzipielle Scheitern algorithmischer Ansätze aus dem bipartiten Bereich bei nichtbipartiten Graphen liegt entsprechend in der Behandlung von Kreisen ungerader Länge. Sind auf einem solchen Kreis die Matchingkanten wie in Abbildung 2.3 verteilt, so kann einem nach augmentierenden Pfaden suchenden Algorithmus eine solcher Pfad vorgetäuscht würden, obwohl keiner vorhanden ist. Die zweite Darstellung

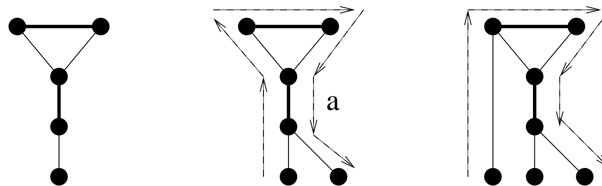

Abbildung 2.3: Kreis ungerader Länge und augmentierende Pfade

zeigt, wie dies geschehen kann. Die Pfeile deuten den Pfad an, den alternierende Tiefensuche für einen augmentierenden Pfad halten könnte. Über diesen Pfad kann nicht augmentiert werden, da er die Kante a zweimal enthält.

Das dritte Bild belegt, dass sich das Problem nicht dadurch lösen läßt, dass man die problematischen Knoten einfach als schon abgearbeitet ansieht. In diesem Fall könnten andere augmentierende Pfade, die eine solche Struktur von außen betreten, übersehen werden. Derartige Strukturen werden wir als *Blumen* bezeichnen. Im Laufe der Zeit wurden verschiedene Definitionen für diese Teilgraphen aufgestellt. Um nicht in Widerspruch zu diesen Definitionen zu geraten, werden wir zunächst "einfache" Blumen und "einfache" *Blüten* definieren. Wenn wir später von Blumen und Blüten sprechen, so meinen wir - falls nicht explizit erwähnt- diese einfache Variante.

Definition 2.27 *Eine* einfache Blume *ist ein Kreis ungerader Länge $2l + 1$, von dessen Kanten l Matchingkanten sind. Der eine Knoten b, der nicht von diesen Matchingkanten überdeckt wird, ist über einen alternierenden Pfad mit einem freien Knoten s verbunden. Der Kreis wird auch* einfache Blüte *genannt und der Pfad* Stengel *(vgl. das linke Bild von Abb. 2.4). b wird* Basis *der einfachen Blüte genannt.*

Bemerkung 2.28 *Nicht jeder Kreis der Länge $2l+1$ mit l Matchingkanten ist eine einfache Blüte. Sie muss Teil einer einfachen Blume sein, d.h. ein Stengel muss vorhanden sein. Der Stengel kann die Länge 0 haben, d.h. die Basis kann frei sein.*

Die meisten Methoden zur Behandlung dieser Blüten basieren auf der Beobachtung, dass, wann immer ein augmentierender Pfad eine einfache Blüte durchquert, es einen augmentierenden Pfad gibt, der durch den Stengel verläuft. Somit verhält sich der Kreis einer Blüte nach außen hin in Bezug auf die Existenz augmentierender Pfade durch die Blüte wie ein einzelner Knoten. Entsprechend kann er durch einen solchen ersetzt werden:

Definition 2.29 *Das* Schrumpfen einer einfachen Blüte *bezeichnet das Identifizieren der Knoten, die zu dem Kreis der Blüte gehören, miteinander. Es bezeichne G/B den Graphen G nach Schrumpfen der Blüte B und v_B denjenigen Knoten, der das Ergebnis der Identifizierung ist (vgl. Abb. 2.4).*
Sind $v_1,\dots,v_{2l+1}$ die Knoten der Blüte, so erhalten wir entsprechend:

$$V(G/B) = V\backslash\{v_1,\dots,v_{2l+1}\}\cup\{v_B\}$$

und

$$\begin{aligned} E(G/B) &= E\backslash\{\{u,z\}\mid\{u,z\}\cap\{v_1,\dots,v_{2l+1}\}\neq\emptyset\}\\ &\cup\ \{\{u,v_B\}\mid\{u,v_i\}\in E, u\notin\{v_1,\dots,v_{2l+1}\}, i=1,\dots,2l+1\}\end{aligned}$$

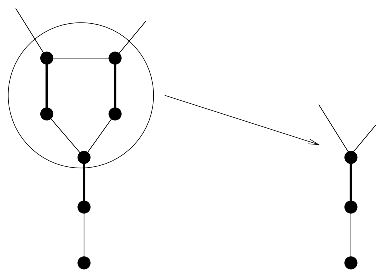

Abbildung 2.4: Schrumpfen einer Blüte

Es bleibt also zu zeigen,

1. dass dieses Schrumpfen tatsächlich keine Auswirkungen auf die Existenz eines augmentierenden Pfades hat.

2. dass es möglich ist, aus einem augmentierenden Pfad im Graphen G/B einen augmentierenden Pfad in G zu erzeugen.

Ersteres liefert das folgende Lemma von Edmonds [9], das wir hier nach Hochstättler, Kromberg und Moll [41] wiedergeben.
Hierzu erinnern wir zunächst an grundlegende Eigenschaften der symmetrischen Differenz von Matchings aus Kapitel 1.3$_6$:

Lemma 2.30 *Seien M und M' zwei Matchings in einem Graphen $M = (V, E)$ mit $|M'| > |M|$. Sei $G' = (V, M \oplus M')$ mit $M \oplus M' = (M \cup M')\backslash(M \cap M')$ der Graph, der die symmetrische Differenz von M und M' als Kantenmenge hat. Dann gilt:*

1. *In G' hat jeder Knoten höchstens Grad 2, also $\Delta(G) \leq 2$.*
2. *G' besteht aus Pfaden, Kreisen und isolierten Knoten.*
3. *Es gibt in G' einen Pfad, der mit einer Kante aus M' beginnt und endet - also bezüglich M augmentierend ist.*

Lemma 2.31 (Blossom Expansion Lemma,[9],[41]) *Sei $G = (V, E)$ ein Graph, M ein Matching in G, $s, t \in V$ freie Knoten und B eine einfache Blüte mit Basis v_0. Sei W_{s,v_0} ein alternierender Pfad von s nach v_0 und $W_{s,t}$ ein augmentierender Pfad von s nach t. Sei M_B die Menge der Matchingkanten, die nicht in der Blüte liegen. Dann gibt es in G/B einen M_B augmentierenden Pfad von s nach t.*

Beweis: Der Graph G' sei der Teilgraph von G, der aus der Blüte, dem Stengel und dem Weg $W_{s,t}$ besteht. Wenn es in diesem Teilgraphen nach dem Schrumpfen einen solchen Weg gibt, gibt es ihn auch im G/B.
Fall 1: Der Pfad betritt (oder symmetrisch: verläßt) die Blüte erstmals durch die Basis: In diesem Fall kann man ihn ab der Basis mit dem Stück fortsetzen, das nach dem letztmaligen Verlassen der Blüte zum freien Knoten führt. Dieses Stück beginnt mit einer freien Kante, die nach dem Schrumpfen dann vom Basisknoten wegführt.
Fall 2: Es sei daher angenommen, dass er die Blüte in zwei anderen Knoten erstmals betritt bzw. endgültig verläßt (vgl. Abb. 2.5 a, die Pfeile geben den Pfad an).
Man betrachte nun zwei Matchings in diesem Graphen:
1. $\tilde{M} := M \oplus W_{s,v_0}$. Bei diesem Matching sind v_0 und t frei. Keine Matchingkante verbindet die Blüte mit anderen Knoten (Bild b).
2. $\bar{M} := M \oplus W_{s,t}$ ist ein perfektes Matching in G' (vgl. Abb. 2.6 Bild c).
Es muss in $\bar{M} \oplus \tilde{M}$ einen Pfad von v_0 nach t geben (d). Sei v_1 derjenige Knoten, an dem dieser Pfad die Blüte endgültig verläßt; da v_0 zur Blüte gehört, muss es einen solchen Knoten geben. $W_{v_1,t}$ sei das entstehende Teilstück dieses Pfades.
Da $\tilde{M}$ keinen Knoten, der zur Blüte gehört, mit einem anderen Knoten verbindet, muss $W_{v_1,t}$ mit einer Kante aus $\bar{M}$ beginnen und enden (d).
Nun betrachten wir den Graphen G'/B, der durch Schrumpfen der Blüte entsteht.
Da $\tilde{M}$ keinen Knoten der Blüte mit Knoten aus G' verbindet und M nur einen Knoten, nämlich v_0, mit anderen Knoten verbindet, können wir beide Matchings durch Löschen der Kanten, die ganz in der Blüte liegen, nach G'/B übertragen. Wir bezeichnen das Ergebnis als M_B bzw. $\tilde{M}_B$. Es gilt $|\tilde{M}_B| = |M_B|$, da gleichviele Matchingkanten hierbei entfernt wurden. In

$\tilde{M}_B$ sind v_B und t frei. Da v_1 mit v_b identifiziert wurde und $W_{v_1,t}$ ein v_1, t-augmenierender Pfad war, ist $N := \tilde{M}_B \oplus W_{v_1,t}$ ein perfektes Matching in G/B (vgl. Abb. 2.7[34] e). Daher muss $M_B \oplus N$ einen $s - t$ Pfad enthalten (f), der entsprechend in G/B ein augmentierender Pfad ist. □

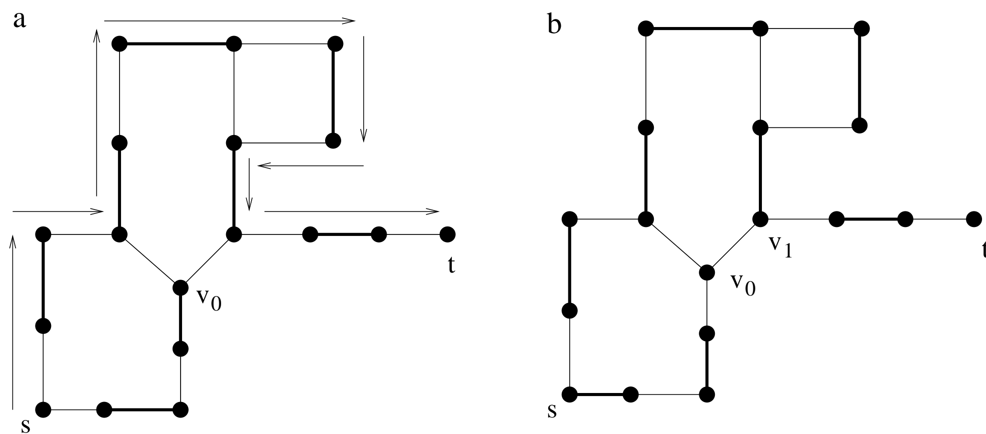

Abbildung 2.5: Beispiel zum Beweis von Lemma 2.31 Teil 1

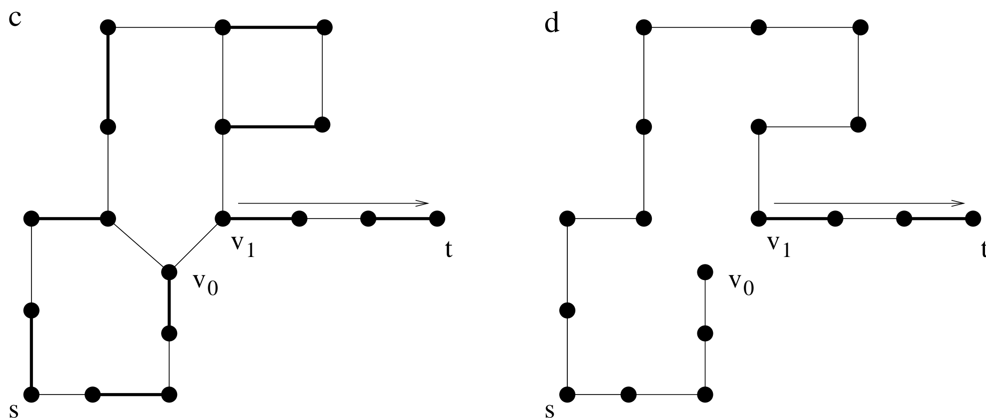

Abbildung 2.6: Beispiel zum Beweis von Lemma 2.31 Teil 2

Hat man nun einen augmentierenden Pfad P_B, der v_B verwendet, in M_B gefunden, so kann man ihn wie folgt nach G übertragen (vgl. Abb. 2.8):
Der Pfad muss v_B über den ursprünglichen Stengel betreten, da dies die einzige Matching-Kante war, die in die Blüte in G und somit in G/B in v_b führte. Er verlässt v_B über eine nicht zu M_B gehörige Kante (v_b, u). Diese entspricht einer Kante (v_j, u) in G. Da in einer Blüte Matchingkanten und freie Kanten abwechselnd vorkommen, gibt es genau einen alternierenden Pfad von v_0 zu v_j in der Blüte, der mit einer Matchingkante endet. Fügt man diesen Pfad anstelle von v_b in P_B ein, so erhält man einen augmentierenden Pfad P in G.

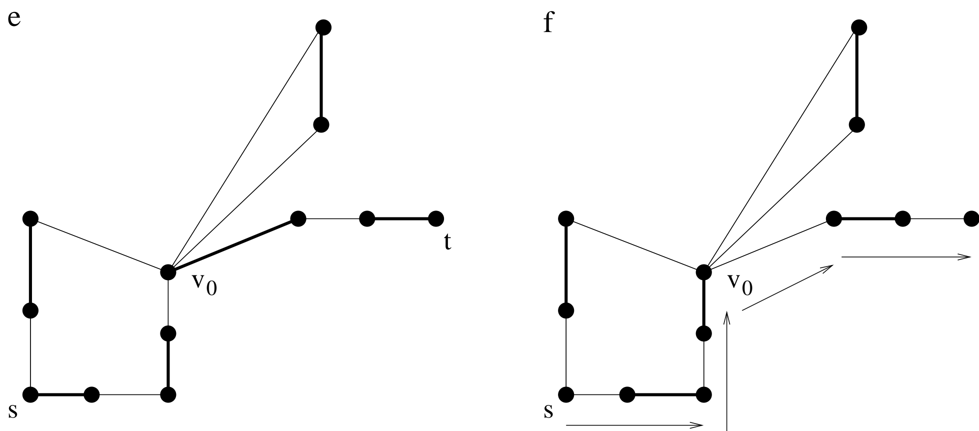

Abbildung 2.7: Beispiel zum Beweis von Lemma 2.31 Teil 3

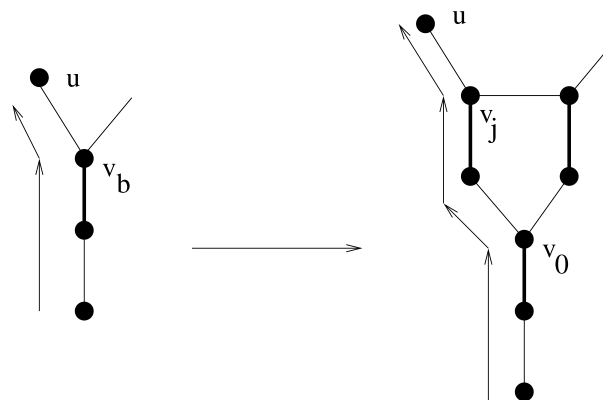

Abbildung 2.8: Übertragen eines augmentierenden Pfades von G/B nach G

2.3 Der nichtbipartite Fall als bipartites Problem mit Nebenbedingungen

Um das in Kapitel 2.1.1[18] vorgestellte Kompressionsverfahren auch auf den nichtbipartiten Fall übertragen zu können, wollen wir das nichtbipartite Matchingproblem als bipartites Matchingproblem mit geeigneten Nebenbedingungen formulieren.
Wir müssen dann klären, wie sich diese Nebenbedingungen auf den Kompressions- bzw. auf den Matching-Algorithmus auswirken.

Zunächst werden wir zeigen, wie man die Suche nach einer solchen Menge in ein spezielles Erreichbarkeitsproblem umwandelt. Das Ergebnis dieser Umwandlung wird ein bipartiter Graph sein. Danach wenden wir das Kompressionsverfahren von Feder und Motwani [11] auf diesen Graphen an und diskutieren, wie sich das Ergebnis verwenden lässt.

2.3.1 Umwandlung in ein Erreichbarkeitsproblem

Im bipartiten Fall kann das Problem der Suche nach augementierenden Pfaden in ein Erreichbarkeitsproblem umgewandelt werden, indem alle freien Kanten von B nach A und alle zum Matching gehörenden Kanten von A nach B gerichtet werden. Dann gibt es genau dann einen augmentierenden Pfad, wenn es einen gerichteten Pfad von einem freien Knoten in B zu einem freien Knoten in A gibt (vgl. Abschnitt 1.4.1[11]). Im nichtbipartiten Fall existiert eine solche Zerlegung nicht.

Um im nichtbipartiten Fall eine solche Zerlegung, die den Wechsel zwischen freien Kanten und Matchingkanten erzwingt, künstlich zu erzeugen, ersetzt man jeden Knoten $v \in V$ durch zwei Knoten $[v, A]$ und $[v, B]$. Jede Kante $(u, v) \in E$ wird ersetzt durch zwei Kanten:

$$([u, A], [v, B])\text{und}([v, A], [u, B]) \quad \text{falls} \quad (u, v) \in M$$
$$([u, B], [v, A])\text{und}([v, B], [u, A]) \quad \text{falls} \quad (u, v) \notin M$$

Wir definieren der Übersichtlichkeit halber $\bar{A} := B$ und $\bar{B} := A$, und nennen $[v, \bar{X}]$ den *Symmetriepartner* von $[v, X]$. Bezeichnen wir den Knoten nur mit dem einem Buchstaben u, so bezeichnet $\bar{u}$ dessen Symmetriepartner.
Der resultierende Graph G_B ist bipartit und ein gerichteter Pfad korrespondiert zu einem alternierenden Pfad in G, da man jeweils mit einer Matchingkante von A nach B gelangt und mit einer freien Kante von B nach A (vgl. Abb. 2.9, a, b, d, c, e, f ist ein augmentierender Pfad).

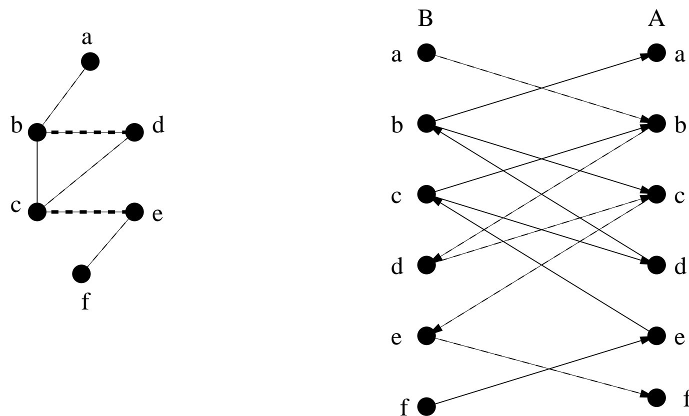

Abbildung 2.9: Umwandlung in ein bipartites Problem

Die besondere Problematik der Kreise ungerader Länge bei nichtbipartiten Graphen sorgt aber leider dafür, dass Erreichbarkeit nur ein notwendiges Kriterium für einen alternierenden Pfad ist. In Abbildung 2.10 ist $[f, B], [e, A], [c, B], [b, A], [d, B], [c, A], [e, B], [f, A]$ ein Pfad von einen freien Knoten in B zu einem freien Knoten in A. Dies ist aber kein augmentierender Pfad. Es handelt sich erneut um eine einfache Blüte (vgl. Kapitel 2.2[29]). Das entscheidende Problem ist, dass die Knoten e, f und c sowohl auf der A- als auch auf der B-Seite besucht werden. Wir müssen also dafür sorgen, dass ein Pfad niemals

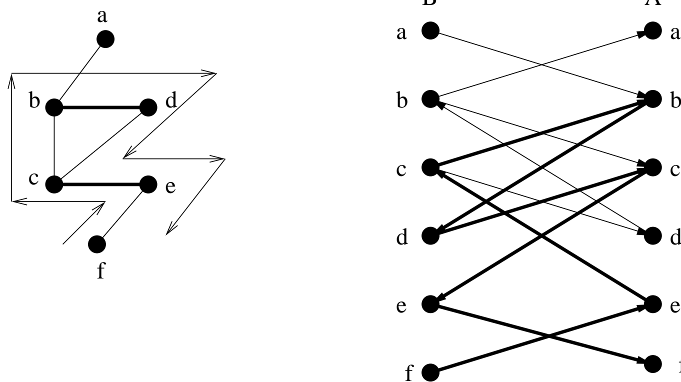

Abbildung 2.10: Besuch von $[c, A]$ und $[c, B]$

zwei Knoten $[x, A]$ und $[x, B]$ gleichzeitig enthalten kann, da dies einem Kreis ungerader Länge in G entspricht. Dies führt zur folgenden Definition:

Definition 2.32 *Ein streng einfacher Pfad in G_B ist ein gerichteter Pfad der weder einen Knoten mehrfach enthält, noch einen Knoten und dessen Symmetriepartner enthält.*

Somit ergibt sich

Lemma 2.33 ([1]) *Seien v und w freie Knoten in G. Es gibt genau dann einen augmentierenden Pfad zwischen ihnen, wenn es in G_B einen gerichteten streng einfachen Pfad zwischen $[v, B]$ und $[w, A]$ (und damit auch zwischen $[w, B]$ und $[v, A]$) gibt.*

Definition 2.34 *Zwei Knotenmengen U und U' heißen* streng disjunkt*, wenn U weder einen Knoten von U' noch den Symmetriepartner eines Knotens aus U' enthält.*

2.3.2 Komprimieren des umgewandelten Graphen

Im vorangegangenen Kapitel haben wir gesehen, wie man zu einem Graphen G und einem Matching M einen bipartiten gerichteten Graphen G_B so erzeugt, dass es in G genau dann einen augmentierenden Pfad gibt, wenn es in G_B einen gerichteten Pfad gibt, der gewisse Nebenbedingungen erfüllt.
In diesem Kapitel wollen wir nun das Kompressionsverfahren von Feder und Motwani [11], das wir in seinen Grundzügen in Kapitel 2.1.118 vorgestellt haben, auf diesen Graphen anwenden. Hierbei ergeben sich mehrere Probleme:

1. Das Verfahren arbeitet mit ungerichteten Graphen.

2. Im Rahmen des Algorithmus von Dinic [8] treten bis zu $2\sqrt{n}$ verschiedene Matchings auf und entsprechend gibt es auch ebensoviele verschiedene Graphen G_B. Da die Kompression aber bereits fast genausoviel Zeit ($\mathcal{O}(mn^\delta \log^2 n)$, $\delta < \frac{1}{2}$) in Anspruch nimmt, wie das Finden eines Matchings, kann sie nicht so oft durchgeführt werden.

3. Die Kompression kann die Symmetrie zerstören.

Wir werden in den folgenden Kapiteln (Kapitel 3[38] und Kapitel 4[58]) zwei verschiedene Methoden vorstellen, um diese Probleme anzugehen.

Zunächst sammeln wir noch einige Eigenschaften von G_B, die wir später benutzen werden, und tragen noch einige Notationen nach, um uns der etablierten Terminologie anzupassen.

1. Wenn es die Kante $([v, X], [w, \bar{X}])$ gibt, dann gibt es auch $([w, X], [v, \bar{X}])$. Damit ist G_B ein sogenannter *schiefsymmetrischer Graph*:

   Definition 2.35 *Ein Graph $G = (V, E)$ heißt* schiefsymmetrisch*, wenn es einen Graphenautomorphismus von G, d.h. eine Abbildung $g : V \to V$ mit $(u, v) \in E \Leftrightarrow (g(v), g(u)) \in E$, gibt, der fixpunktfrei ($g(u) \neq u \quad \forall u \in V$) und involutorisch ($g(g(u)) = u \quad \forall u \in V$) ist.*

   Im Fall von G_B ist g die Abbildung $[u, X] \mapsto [u, \bar{X}]$.

2. Da G keine Schleifen, also Kanten der Form (v, v) hat, enthält G_B keine Kanten des Typs $([v, X], [v, \bar{X}])$.

   Definition 2.36 *Ein bezüglich eines Automorphismus g schiefsymmetrischer Graph heißt* paarfrei (bezüglich g)*, wenn $(v, g(v)) \notin E \; \forall v \in V$ gilt.*

Proposition 2.37 *In G_B enthält keine Biclique gleichzeitig eine Kante und ihren Symmetriepartner.*

> **Beweis:** Angenommen, $([a, X], [b, Y])$ und $([b, \bar{Y}], [a, \bar{X}])$ bilden solch ein Paar, dann müßte $[a, X]$ mit $[a, \bar{X}]$ verbunden sein, da beide zur selben Clique gehören. Dies widerspricht aber der Paarfreiheit. □

Kapitel 3

Explizite Kompression

3.1 Verringerung der Kantenzahl durch Ersetzen von Teilgraphen

Die Laufzeit der meisten Matching-Algorithmen hängt wesentlich von der Zahl der Kanten in dem betrachteten Graphen ab. Eine mögliche Strategie, diese Algorithmen zu beschleunigen, ist es also, die Zahl der Kanten vorab zu reduzieren. Hierbei muss man sicherstellen, dass nach der Entfernung zumindestens ein maximales Matching übrigbleibt oder wiederhergestellt werden kann.
In Kapitel 2.1.1$_{18}$ haben wir gesehen, wie Feder und Motwani diese Strategie für bipartite Graphen realisiert haben: Sie haben bipartite Cliquen durch "Sterne" ersetzt, die sich im Matching-Kontext genauso verhalten, wie die jeweilige Clique. Hierzu mussten sie das Matchingproblem zunächst in ein Flussproblem umwandeln, da ihr Vorgehen insbesondere von der Tatsache Gebrauch machte, dass mehrere augmentierende Pfade im "Flussmodell" durch einen Knoten führen können.
Im nichtbipartiten Fall ist eine einfache Übertragung ins Flussmodell nicht mehr möglich. Ein Weg, dieses Problem zu lösen, ist die Einführung eines erweiterten Flussmodells: Die sogenannten schiefsymmetrischen Flüsse. Darauf basieren die Arbeiten von Goldberg und Karzanov [17] und Freymuth-Paeger und Jungnickel [14].
Da die Algorithmen, die wir beschleunigen wollen, insbesondere also der Algorithmus von Blum [1][2], auf augmentierenden Pfaden basieren, ist uns dieser Übergang zu Flüssen nicht möglich. Wir verwenden daher in diesem Kapitel an Stelle von "Sternen" andere "Ersatzgraphen", die, obwohl sie wenige Knoten haben, es mehreren augmentierenden Pfaden im "Matchingsinn" erlauben, gleichzeitig durch sie hindurchzuführen.
Da wir im bipartiten Fall Teilgraphen mit möglichst vielen Kanten benutzt haben, erscheint es zunächst naheliegend, im nichtbipartiten Fall das Gleiche zu tun, also mit den vollständigen Teilgraphen, den Cliquen, zu arbeiten. Leider ist aber kein Verfahren bekannt, das in annehmbarer Zeit eine ausreichend gute Cliquenüberdeckung eines Graphen liefern würde.
Es ist aber auch in nichtbipartiten Graphen möglich, bipartite Teilgraphen zu finden. Hierbei ist zu beachten, dass die zur Kompression eingesetzten Teilgraphen keine induzierten

Teilgraphen sein müssen. D.h. man muss in die Kantenmenge nicht alle Kanten zwischen den zugehörigen Knoten aufnehmen. Es wäre also z.B. möglich, eine Clique in zwei Teile A und B aufzuteilen und alle Kanten außer denen, die A mit B verbinden, wegzulassen. Auf diese Weise erhält man eine Biclique. Die dabei ignorierten Kanten gehören dann später entweder zu einer anderen großen Clique oder sie bilden eine 1-Kanten-Clique.

In Kapitel 2.1.1[18] haben wir gesehen, wie man in bipartiten Graphen eine geeignete Cliquenüberdeckung findet. In Kapitel 2.3.1[35] haben wir gezeigt, wie man durch Zerlegen jedes Knotens v eines Graphen G in zwei Knoten $[v, A]$ und $[v, B]$ eine bipartite Repräsentation G_B eines nichtbipartiten Graphen erzeugen kann.
Wir zeigen nun, wie das Kompressionsverfahren von Feder und Motwani [11] - angewandt auf diese bipartite Repräsentation G_B - eine geeignete Überdeckung des ursprünglichen Graphen mit Bicliquen liefern kann.

Um die in G_B gefundenen Cliquen nach G zurück übertragen zu können, ist es notwendig, dass eine Kante (u, v) aus G eindeutig einer Clique zugeordnet werden kann. Es darf also nicht der Fall sein, dass $([u, A], [v, B])$ und $([v, A], [u, B])$ zu verschiedenen Cliquen gehören.
Dies motiviert die folgende Definition:

Definition 3.1 *Eine Cliquenzerlegung $\mathcal{C}$ des Graphen G_B heißt symmetrisch, wenn für alle Kanten $([u, X], [v, Y])$ gilt, dass $([u, X], [v, Y])$ und $([v, \bar{Y}], [u, \bar{X}])$ zur gleichen Clique $C \subset \mathcal{C}$ gehören.*

Glücklicherweise verursacht die Einführung dieser zusätzlichen Bedingung keine Laufzeitprobleme:

Lemma 3.2 (Goldberg, Karzanov 2003[17]) *Eine symmetrische Kompression kann in der gleichen Zeit berechnet werden, wie eine nichtsymmetrische.*

Beweis: Zum Verständnis dieses Beweises ist eine genaue Kenntnis des Clique-Stripping Algorithmus notwendig vgl. Seite 26.
Sobald man eine Clique gefunden hat und diese entfernt wird, kann man sofort auch die Clique entfernen, die von den Symmetriepartnern der Kanten gebildet wird. Da $[u, X]$ und $[u, \bar{X}]$ nie in derselben Clique sein können, müssen diese beiden Cliquen disjunkt sein.
Das einzige Problem tritt bei der Behandlung der Nachbarschaftsbäume auf: Während die gefundene Clique von der Form $(n^{1-\delta}, k)$ war, hat ihr schiefsymmetrisches Pendant die Form $(k, n^{1-\delta})$. Da diese Clique mit B $n^{1-\delta}$ Knoten gemeinsam hat, würde es $nn^{1-\delta} \log n$ Operationen benötigen, alle n Bäume zu aktualisieren.
Wir haben aber bereits gesehen, dass die Bäume, die zu Knoten gehören, die nicht in die Clique aufgenommen werden, nachher wieder zurückgesetzt werden müssen. Die meisten dieser Operationen sind also unnötig. Da die symmetrische Clique nicht gefunden werden muss, sondern aus der bereits gefundenen konstruiert werden kann, müssen die Nachbarschaftsbäume bezüglich der symmetrischen Clique nicht beständig aktuell gehalten werden. Es genügt

also, die Bäume entsprechend anzupassen, wenn die symmetrische Clique entfernt wird.
Zum Aktualisieren der Nachbarschaftsbäume bezüglich des Entfernens der symmetrischen Clique müssen in k Bäumen jeweils insgesamt $n^{1-\delta}$ Pfade der Länge $\log n$ verändert werden. Dies benötigt insgesamt eine Zeit von $kn^{1-\delta} \log n$ und wird somit von der Zeit für das Aktualisieren der Bäume im Rahmen der Suche dominiert. □

Wir nehmen also im Folgenden stets an, dass eine symmetrische Kompression vorliegt. Wir können uns nun überlegen, dass wir - ausgehend von den Paaren symmetrischer Cliquen in G_B - auch in G Bicliquen erhalten.

Lemma 3.3 *Liegt symmetrische Kompression vor, so entsprechen zwei zueinander symmetrische Cliquen in G_B einem vollständigen bipartiten Teilgraphen im Ursprungsgraphen G.*

Beweis: 1. Aufgrund der Symmetrie repräsentieren die Clique und ihr Symmetriepartner dieselben Knoten und Kanten im ursprünglichen Graphen. D.h. wenn die eine Clique die Kante $([u, X], (v, \bar{X}])$ enthält, so enthält die andere $([u, \bar{X}], (v, X])$.
2. Keine der Bicliquen in G_B kann einen Knoten und seinen Symmetriepartner enthalten, da die beiden nach Konstruktion nicht miteinander verbunden sind. Dies überträgt sich auch auf die Kanten. Würde eine Biclique $([u, X], (v, \bar{X}])$ und $([u, \bar{X}], (v, X])$ enthalten, so enthielte sie damit natürlich auch $[u, X]$ und $[u, \bar{X}]$, was unmöglich ist.
Überträgt man zwei symmetrische Bicliquen C_{B1} und C_{B2} aus G_B nach G, indem man jedes Kantenpaar $([u, A], [z, B]), ([u, B], [z, A])$ in eine Kante (u, z) umwandelt, so entsteht stets ein bipartiter Teilgraph C:
Annahme des Gegenteils: In C liegt ein Kreis ungerader Länge vor. Wir betrachten nun die Urbilder der Kanten dieses Kreises in G_B, wobei wir die Urbilder aus beiden Cliquen C_{B1} und C_{B2} in G_B berücksichtigen.
Zu jedem Knoten v gibt es einen Knoten $[v, X]$ in der einen Clique und einen Knoten $[v, \bar{X}]$ in der anderen Clique. Betrachtet man nun zwei beliebige, im Kreis aufeinanderfolgende Knoten u und v, so muss die Kante $([u, A], [v, B])$ zu einer der beiden Cliquen o.B.d.A C_{B1} gehören. Damit müssen auch $[u, A]$ und $[v, B]$ zu C_{B1} gehören. Ihre Symmetriepartner müssen entsprechend zur anderen Clique C_{B2} gehören. D.h. ist $[u, A]$ in Clique C_{B1}, so ist $[v, A]$ in Clique C_{B2}. Da dies für jedes Paar aufeinanderfolgender Knoten gilt, müssen die jeweiligen Urbildknoten, die zu A-Partition gehören, entlang des Kreises immer abwechselnd zu C_{B1} und C_{B2} gehören. Dies entspräche einer Färbung des Graphen mit zwei Farben. Dies ist bei Kreisen ungerader Länge nicht möglich.
Bild 3.1 zeigt entsprechend, dass sich die Kanten aus G_B, die einem Kreis ungerader Länge entsprechen, nicht in zwei symmetrische Bicliquen zerlegen lassen (1), während dies für eine Biclique in G möglich ist (2). □

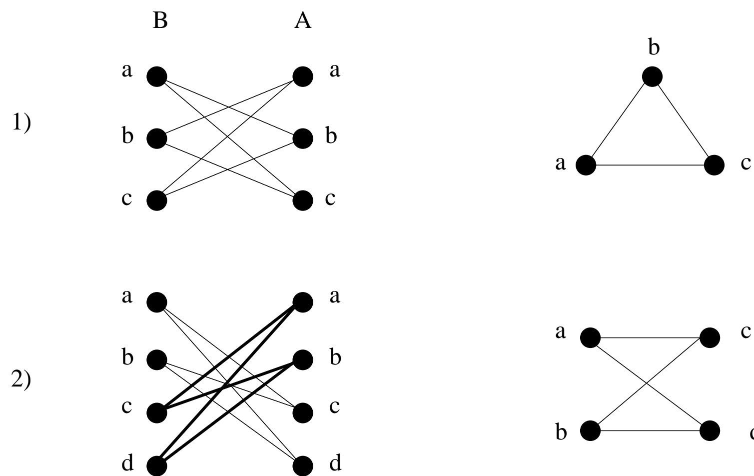

Abbildung 3.1: Cliquen in G und G_B

Korollar 3.4 *Sei $G = (V, E)$ ein nichtbipartiter Graph mit $n = |V|$ und $m = |E|$. Dann kann in Zeit $\mathcal{O}(mn^\delta \log^2 n)$ eine Überdeckung von G mit Bicliquen erstellt werden, die Gesamtgröße $\mathcal{O}\left(\frac{m \log \frac{2n^2}{m}}{\log n}\right)$ hat.*

Beweis: Der Übergang von G nach G_B verdoppelt die Kantenzahl. Bei der Rückübertragung werden die Kanten paarweise identifiziert, die Kantenzahl halbiert sich entsprechend wieder. Wenn die Zahl der Kanten in G_B um einen Faktor $\frac{\log \frac{2n^2}{m}}{\log n}$ reduziert werden kann, so verringert sich die Zahl der Kanten in G entsprechend um $2\frac{\log \frac{2n^2}{m}}{\log n}\frac{1}{2}$, also um den selben Wert. □

Wir wollen nun die Bicliquen wiederum durch einen geeigneten kleineren Teilgraphen ersetzen. Wir werden zunächst eine solche Biclique C exemplarisch betrachten und das beschriebene Verfahren später auf alle vom Kompressionsverfahren erkannten Bicliquen ausdehnen.

Wir betrachten die Einschränkung M_C eines Matchings M auf die Clique C. Die wesentliche Eigenschaft dieser Matchingkanten, die sich auf den Rest des Graphen auswirkt, ist die, dass die Kanten aus M_C stets gleichviele Knoten in der linken und der rechten Seite von C überdecken. Es wäre also möglich die Kanten dieser Biclique durch ein Konstrukt zu ersetzen, das genau diese Eigenschaft garantiert (vgl. Abb. 3.2).

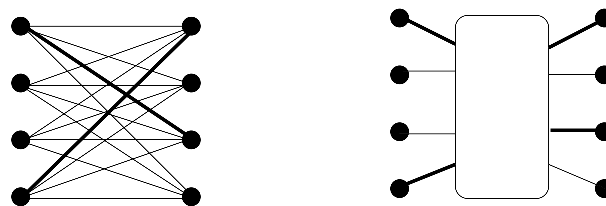

Abbildung 3.2: Ersetzen der Kanten einer Clique

Dies motiviert die folgende Definition:

Definition 3.5 (CMA-Graph) *Ein Graph $C_S = (V_L \dot{\cup} V_M \dot{\cup} V_R, E)$ mit*

$$E \subseteq (V_L \otimes V_M) \cup \binom{V_M}{2} \cup (V_M \otimes V_R)$$

heißt matching-äquivalent zu einer bipartiten Clique C (kurz CMA), wenn

1. *für jedes Matching, das alle Knoten aus V_M überdeckt, gilt, dass es gleichviele Knoten aus V_L und V_R überdeckt und*

2. *zu allen $V_L' \subset V_L$ und $V_R' \subset V_R$ mit $|V_L'| = |V_R'|$ ein Matching von C_S existiert, das genau $V_M \cup V_L' \cup V_R'$ überdeckt.*

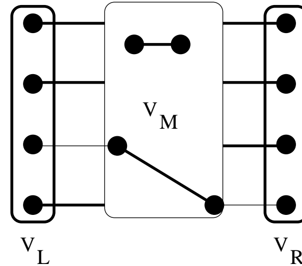

Abbildung 3.3: Abstraktes Beispiel eines CMA-Graphen

Bemerkung 3.6 1. *Für jedes Matching N, das V_M überdeckt, gilt*

$$|N \cap (V_L \otimes V_M)| = |N \cap (V_M \otimes V_R)|.$$

2. *Aus der Definition folgt, dass V_M ein perfektes Matching besitzt, da man $V_L' = V_R' = \emptyset$ wählen kann.*

Definition 3.7 *Ein CMA-Graph ist matching-äquivalent zu einer konkreten Biclique $C = (A \dot{\cup} B, A \otimes B)$, wenn $|V_L| = |A|$ und $|V_R| = |B|$ gilt.*

Wir wollen nun diese Graphen verwenden, um Bicliquen in G zu ersetzen. Es ist hierbei wichtig zu beachten, dass dabei nur Kantenmengen ersetzt werden. Entsprechend wurden die Knotenmengen V_L und V_R nur eingeführt, damit die CMA-Graphen korrekt (d.h. ohne Kanten mit nur einem Knoten) definiert werden konnten. D.h. die Mengen V_L und V_R verschmelzen mit Knoten aus G. Kanten dieser Knoten in G, die nichts mit der Biclique zu tun haben, bleiben völlig unberührt.
Wir geben nun eine formale Definition des Ersetzungsvorganges an, der in Abb. 3.4 nochmals graphisch dargestellt wird:

Definition 3.8 *Sei $G = (V, E)$ ein Graph, $C = (A\dot{\cup}B, A \otimes B)$ eine Biclique in G und $C_S = (V_L\dot{\cup}V_M\dot{\cup}V_R, E_{C_S})$ ein zu C matching-äquivalenter CMA-Graph. Seien zudem $\mu : A \to V_L$ und $\nu : B \to V_R$ beliebige bijektive Abbildungen. Dann entsteht durch Ersetzen von C durch C_S der Graph $G_{C_S/C}(\mu, \nu) = (V_{C_S/C}, E_{C_S/C})$ mit*

$$\begin{aligned} V_{C_S/C} &= V \cup V_M \\ E_{C_S/C} &= \begin{cases} E\backslash(A \otimes B) \\ \cup \{(u, v) \in E_{C_S} \mid u, v \in V_M\} \\ \cup \{(a, v) \in A \otimes V_M \mid (\mu(a), v) \in E_{C_S}\} \\ \cup \{(v, b) \in V_M \otimes B \mid (v, \nu(b)) \in E_{C_S}\} \end{cases} \end{aligned}$$

Die für verschiedene μ, ν entstehenden Graphen müssen im Gegensatz zu der Ersetzung durch Sterne (vgl. Kapitel 2.1.1[18]) nicht isomorph sein. Da die innere Struktur der CMA-Graphen für die hier benötigte - unter Permutation invariante - Wechselwirkung mit dem Rest des Graphen aber keine Rolle spielt, werden wir im weiteren auf die Angabe von μ und ν verzichten.

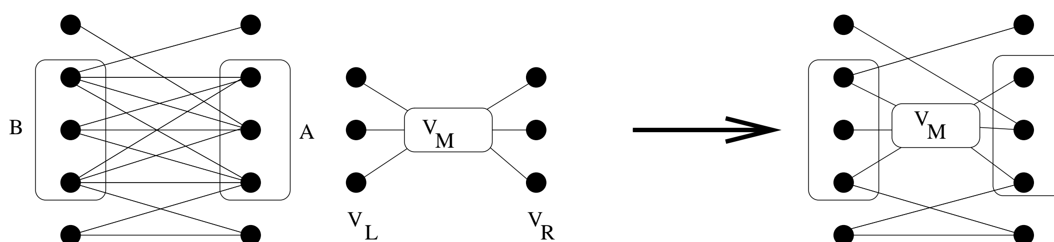

Abbildung 3.4: Einsetzen eines CMA-Graphen

An dieser Stelle würden wir nun gerne sagen, dass ein maximales Matching in $G_{C_S/C}$ genauso groß ist, wie ein maximales Matching in G und das wir aus einem maximalen Matching in $G_{C_S/C}$ leicht ein genauso großes Matching in G konstruieren können. Leider ist dies aus mehreren Gründen nicht möglich:

1. $G_{C_S/C}$ hat mehr Knoten und weniger Kanten als G. Daher können maximale Matchings in $G_{C_S/C}$ größer sein als maximale Matchings in G.

2. Wenn wir ein Matching in $G_{C_S/C}$ finden, muss es nicht notwendigerweise alle Knoten aus V_M überdecken.

Wir skizzieren das Vorgehen daher zunächst am Beispiel eines Graphen (Abb. 3.5) mit einem perfekten Matching, d.h. anhand eines Graphen, der ein Matching hat, das alle Knoten überdeckt. Den allgemeinen Fall können wir dann im Vergleich übersichtlicher darstellen.

Sei G ein Graph mit einem perfekten Matching M. Sei C eine Biclique in diesem Graphen und $G_{C_S/C}$ der Graph, der entsteht, wenn man die Clique nach obigem Schema

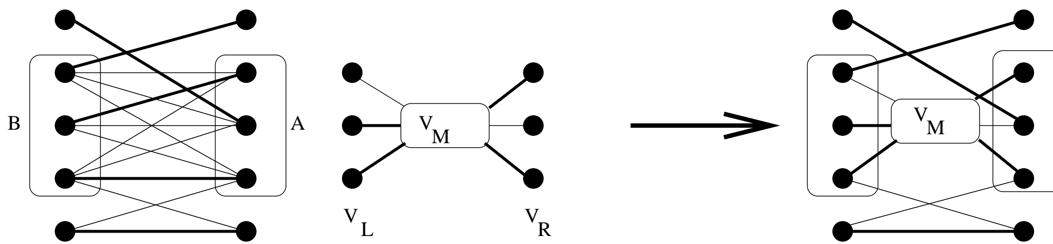

Abbildung 3.5: Korrespondierende Matchings in G und $G_{C_S/C}$

durch einen entsprechenden CMA-Graphen ersetzt (Abb. 3.5). Dann hat auch $G_{C_S/C}$ ein perfektes Matching:
Wir gehen von G und dem Matching M aus. Wir entfernen die zu C gehörigen Kanten E_C aus G.
Fall 1 (nicht dargestellt): Gehören keine Matchingkanten aus M zu E_C, so sind alle Knoten aus G nach wie vor überdeckt. Da der "hineingeklebte" Teilgraph V_M ein perfektes Matching besitzt, fügt man dieses einfach zu den Matchingkanten aus M hinzu und hat somit ein perfektes Matching in ganz $G_{C_S/C}$.
Fall 2 (vgl. Abbildung 3.5): Gehören Matchingkanten aus M zu E_C, so bleiben nach deren Entfernung Knoten aus G unüberdeckt. Da jede Matchingkante aus E_C einen Knoten von der einen (o.B.d.A "linken") Seite der Clique mit der anderen ("rechten") Seite der Clique verbindet, sind auf beiden Seiten gleichviele Knoten nicht überdeckt. Nach Definition von CMA-Graphen gibt es in diesem Fall ein Matching, das genau diese Knoten und V_M überdeckt. Damit entsteht ein perfektes Matching in $G_{C_S/C}$.

Damit haben wir nur die Existenz eines perfekten Matchings gezeigt. Wir wissen somit aber: Wenn wir einen Algorithmus zum Finden eines maximalen Matchings auf $G_{C_S/C}$ anwenden, so wird uns dieser ein konkretes perfektes Matching liefern. Wir wollen nun einsehen, dass man aus diesem Matching umgekehrt wieder eines in G erzeugen kann. Dazu zeigen wir wieder allgemein, dass dies mit jedem perfekten Matching in $G_{C_S/C}$ möglich ist:

Lemma 3.9 *Aus einem perfekten Matching in $G_{C_S/C}$ kann man ein perfektes Matching in G konstruieren.*

> **Beweis:** Ein perfektes Matching überdeckt auch alle Knoten aus V_M. Nach der Definition von CMA-Graphen sind also genausoviele Knoten in der linken, wie in der rechten Hälfte der Clique mit Knoten aus V_M gematcht. Entfernt man nun V_M, so sind also in beiden Hälften der Clique gleichviele Knoten nicht überdeckt. Wenn man nun, um G wieder herzustellen, die Kanten E_C wieder einfügt, so kann man diese Knoten paarweise miteinander verbinden. Somit erhält man ein perfektes Matching in G. □

Im Folgenden werden wir nun den Beweis für den allgemeinen Fall formal führen. Ab hier ist G also wieder ein allgemeiner Graph mit einer Biclique und $G_{C_S/C}$ der Graph, der bei Ersetzung dieser Biclique durch einen CMA-Graphen entsteht. Da $G_{C_S/C}$ mehr Knoten

hat als G und somit maximale Matchings in diesen Graphen zwangsläufig unterschiedliche Größen haben, können wir nicht mit den absoluten Größen $|M|$ dieser Matchings arbeiten. Wir verwenden stattdessen die Abweichung von einem perfekten Matching, d.h. die Zahl der nicht überdeckten Knoten.

Definition 3.10 (Defizit) *Das* Defizit *eines Matchings, ist die Zahl der vom Matching nicht überdeckten Knoten.* $D_M := |V| - 2|M|$

Der Beweis erfolgt in den folgenden Schritten, wobei die fett gedruckten Schritte in den nachfolgenden Lemmata gezeigt werden:

1. **Wir können aus einem maximalen Matching M in G mit Defizit D_M ein Matching in $G_{C_S/C}$ konstruieren, dessen Defizit nicht größer ist.**
2. Es gibt also in $G_{C_S/C}$ ein Matching, dessen Defizit nicht größer ist als D_M.
3. Jedes maximale Matching in $G_{C_S/C}$ hat also ein Defizit von höchstens D_M.
4. Wenden wir einen Algorithmus zum Finden maximaler Matchings auf $G_{C_S/C}$ an, so erhalten wir also ein konkretes Matching M' mit Defizit höchstens D_M.
5. **Aus jedem Matching in $G_{C_S/C}$ mit Defizit D' läßt sich in Linearzeit ein Matching in G konstruieren, dessen Defizit nicht größer ist als D'.**
6. Insbesondere läßt sich also aus dem vom Algorithmus gefundenen Matching M' in $G_{C_S/C}$, dessen Defizit kleiner gleich D_M ist, ein konkretes Matching M'' in G konstruieren, dessen Defizit ebenfalls nicht größer ist.
7. Da das Defizit des Matchings M'' in G kleiner gleich D_M ist und D_M das Defizit eines maximalen Matchings in G ist, muss M'' ein maximales Matching in G sein.
8. Wir können also ein konkretes maximales Matching in G auf diesem Wege bestimmen.

Die folgenden Lemmata vervollständigen diesen Beweis:

Lemma 3.11 *Sei $G = (V, E)$ ein Graph mit einer Biclique $C = (A\dot{\cup}B, A \otimes B)$. Sei $C_S = (V_L\dot{\cup}V_M\dot{\cup}V_R, E_{C_S})$ ein zu C matching-äquivalenter Graph und $G_{C_S/C}$ der Graph, der durch Einsetzen von C_S für C in G entsteht. Dann läßt ein maximales Matching M in G genausoviele Knoten unüberdeckt, wie ein maximales Matching $\tilde{M}$ in $G_{C_S/C}$, d.h. $D_{\tilde{M}} = D_M$.*

Beweis: Zum Beweis benötigen wir zunächst noch zwei weitere Lemmata:

Lemma 3.12 *Zu jedem maximalen Matching $\tilde{M}$ in $G_{C_S/C}$ gibt es ein Matching $\tilde{\tilde{M}}$ gleicher Größe, das V_M vollständig überdeckt.*

Beweis: 1. Es seien $\tilde{V}_L := \{u \in V_L \mid (u,v) \in \tilde{M}, v \in V_M\}$ und $\tilde{V}_R := \{w \in V_R \mid (z,w) \in \tilde{M}, z \in V_M\}$ die von $\tilde{M}$ überdeckten Knoten in V_L und V_R. Es sei zunächst angenommen, dass $|\tilde{V}_R| = |\tilde{V}_L|$. In diesem Fall gibt es gemäß der Definition von CMA-Graphen ein Matching $\tilde{\tilde{M}}$ in C_S, das alle Knoten aus $V_M \cup \tilde{V}_L \cup \tilde{V}_R$ überdeckt. $\tilde{\tilde{M}}$ überdeckt in V_L und V_R genausoviele Knoten wie $\tilde{M}$ und alle Knoten aus V_M. Somit gilt $\tilde{\tilde{M}} \geq \tilde{M}$. Da $\tilde{M}$ maximal war, folgt Gleichheit.

2. Nun sei o.B.d.A angenommen, dass $|\tilde{V}_R| < |\tilde{V}_L|$. Wählt man eine beliebige Teilmenge $\tilde{\tilde{V}}_L \subset \tilde{V}_L$ der Größe $|\tilde{V}_R|$, so gibt es ein Matching $\tilde{\tilde{M}}$, das V_M,$\tilde{V}_R$ und $\tilde{\tilde{V}}_L$ überdeckt. Die übrigen $|\tilde{V}_L| - |\tilde{V}_R|$ Knoten in V_L werden von $\tilde{\tilde{M}}$ nicht überdeckt. Wir zeigen nun, dass dies dadurch kompensiert wird, dass von $\tilde{M}$ in V_M ebensoviele, d.h. $|\tilde{V}_L| - |\tilde{V}_R|$ Knoten nicht überdeckt werden.
Da wir auf diesen Aspekt später nochmals gesondert zurückkommen werden, formulieren wir dies gleich als eigenständiges Lemma.
Das neue Matching $\tilde{\tilde{M}}$ ist also mindestens so groß wie $\tilde{M}$. $\square_{\text{L3.12}}$

Nun zeigen wir den eben übersprungenen Schritt:

Lemma 3.13 *Sei $C_S = (V_L, V_M, V_R, E)$ ein CMA-Graph und $\tilde{M}$ ein Matching in C_S, sei $\tilde{V}_L := \{u \in V_L \mid (u,v) \in M, v \in V_M\}$ die Menge der von $\tilde{M}$ überdecken Knoten aus V_L und $\tilde{V}_R$ analog definiert. Dann sind mindestens $||\tilde{V}_L| - |\tilde{V}_R||$ Knoten in V_M nicht überdeckt.*

> **Beweis:** Sei o.B.d.A $|\tilde{V}_R| < |\tilde{V}_L|$. Laut Definition von CMA-Graphen gibt es ein Matching $\tilde{\tilde{M}}$, das $V_M, \tilde{V}_R$ und eine beliebige Menge $\tilde{\tilde{V}}_L \subset \tilde{V}_L$ mit $|\tilde{\tilde{V}}_L| = |\tilde{V}_R|$ überdeckt. $\tilde{\tilde{M}}$ überdeckt somit in V_L insgesamt $||\tilde{V}_L| - |\tilde{V}_R||$ Knoten nicht. Wir nehmen nun mit dem Ziel eines Widerspruchs an, dass $\tilde{M}$ in V_M weniger als $||\tilde{V}_L| - |\tilde{V}_R||$ Knoten nicht überdeckt. Damit wäre $|\tilde{M}| > |\tilde{\tilde{M}}|$. Dann müsste es in der symmetrischen Differenz $\tilde{\tilde{M}} \oplus \tilde{M}$ einen Pfad P ungerader Länge geben, der mit jeweils einem in $\tilde{\tilde{M}}$ freien Knoten beginnt und endet. Diese Knoten können somit nur in $\tilde{V}_L \backslash \tilde{\tilde{V}}_L \subset V_L$ liegen, da $\tilde{\tilde{M}}$ ganz V_M und $\tilde{V}_R$ überdeckt. Dann würde aber das Matching $\tilde{\tilde{M}} \oplus P$ den Teilgraphen V_M ganz und in V_L zwei Knoten mehr überdecken als in V_R. Das ist aber nach Definition von C_S als CMA-Graphen nicht möglich, da in einem CMA-Graphen jedes Matching, das wie $\tilde{\tilde{M}}$ V_M ganz überdeckt, in V_L und V_R gleichviele Knoten überdeckt sein müssen. $\square$

Bemerkung 3.14 *Für die CMA-Graphen, die wir im nächsten Kapitel konstruieren werden, ist das gerade bewiesene Lemma trivial, da diese Graphen bipartit sind und V_L und V_R zu verschiedenen Partitionen gehören.*

Nun können wir Lemma 3.11 zeigen: $D_{\tilde{M}} \leq D_M$ ist erfüllt, da man zu jedem Matching M in G eines in $G_{C_S/C}$ konstruieren kann, das zusätzlich genau die Knoten aus V_M überdeckt: M überdeckt gleichviele Knoten der Clique, es gibt also nach Definition von C_S

ein Matching, das die gleichen Knoten und V_M überdeckt.
$D_{\tilde{M}} \geq D_M$ folgt daraus, dass man zu einem maximalen Matching M_C in $G_{C_S/C}$ gemäß Lemma 3.12 ein gleichgroßes M'_C finden kann, das V_M überdeckt. Nach Definition der CMA-Graphen ist in M'_C die Zahl der Matchingkanten aus $A \times V_M$ gleich der aus $V_M \times B$. Wenn wir dieses Matching M'_C von $G_{C_S/C}$ nach G zurück übertragen, so werden die Knoten aus V_M entfernt. Diese waren aber vollständig überdeckt, so dass sich die Zahl der nicht überdeckten Knoten hierdurch nicht ändert. Außerdem verlieren die Kanten aus A und B, deren Matchingpartner in V_M lag, ihren Partner. Die Zahl der Knoten mit dieser Eigenschaft ist - wie wir gerade gesehen haben - in A und B gleich.
Da (A, B) in G eine bipartite Clique ist, kann man die zugehörigen Knoten aus A und B in G paarweise verbinden. Somit sind in G in A und B genausoviele Knoten nicht überdeckt, wie in M'_C und somit in M_C. $\Box_{\text{L3.11}}$

Im nächsten Kapitel werden wir basierend auf den sogenannten Superkonzentratoren nach Pippenger [34] zu jeder Clique C einen Graphen $C_S = (A \dot{\cup} V_M \dot{\cup} B, E)$ konstruieren, der die folgende Eigenschaften hat:

- C_S ist ein zu C matching-äquivalenter CMA-Graph
- $|E|$ ist in $\mathcal{O}(|A| + |B|)$
- C^S kann in Zeit $\mathcal{O}(|A| + |B|)$ konstruiert werden.

Bevor wir dies tun, verwenden wir diese Eigenschaften, um den Hauptsatz dieses Abschnitts zu beweisen:

Satz 3.15 *Zu einem gegebenen nichtbipartiten Graphen $G = (V, E)$ mit $m := |E|$ und $n := |V|$ läßt sich in Zeit $\mathcal{O}(mn^\delta \log^2 n)$ ein Graph G_S mit $|E(G_S)| = \mathcal{O}\left(m \frac{\log \frac{2n^2}{m}}{\delta \log n}\right)$ Kanten konstruieren, so dass aus einem maximalen Matching M in G_S in Zeit $\mathcal{O}(m)$ ein maximales Matching M' in G berechnet werden kann.*

Beweis: Wir geben zunächst einen Überblick und zerlegen den Beweis dann in mehrere Lemata:

1. Wir gehen zur bipartiten Repräsentation G_B von G über.
2. Wir verwenden den Algorithmus von Feder und Motwani (s. S. 27), um eine symmetrische Cliquenzerlegung $\mathcal{C_B} = \{C_B^i\}_{i=1,\dots,2r}$ mit Gesamtgröße $2m \frac{\log \frac{2n^2}{m}}{\delta \log n}$ zu finden. Dies benötigt Zeit $\mathcal{O}(mn^\delta \log^2 n)$.
3. Wie in Lemma 3.3[40] gesehen, können wir jeweils symmetrische Paare dieser Bicliquen als eine Biclique in G interpretieren. Wir erhalten somit eine Cliquenzerlegung $\mathcal{C} = \{C^i\}_{i=1,\dots,r}$ von G.
4. Wir erstellen G_S, indem wir sämtliche gefundenen Cliquen C^i in G durch einen entsprechenden CMA-Graphen C_S^i ersetzen.

Zunächst wollen wir nun zeigen, dass dieser Graph G_S die geforderte Zahl von Kanten hat.

Bemerkung 3.16 *Wir verwenden im Folgenden wieder die Abkürzung* $m^* := m\frac{\log\frac{2n^2}{m}}{\delta\log n}$.

Lemma 3.17 $|E(G_S)| \in \mathcal{O}(m\frac{\log\frac{2n^2}{m}}{\delta\log n}) = \mathcal{O}(m^*)$

Beweis:

Wir erinnern an dieser Stelle nochmals an die Definition einer Cliquenzerlegung aus Lemma 2.5[21] und die Definition der Gesamtgröße 2.6[21]:
Eine bipartite Cliquenzerlegung eines Graphen $G = (V, E)$ *ist eine Menge von Bicliquen* $\mathcal{C} = C_1, \dots, C_r$*, so dass jede Kante* $e \in E$ *für genau ein* $i \in \{1, \dots, r\}$ *zu* $E(C_i)$ *gehört.*
Die Gesamtgröße ist die Summe der Knotenzahlen dieser Cliquen $||\mathcal{C}|| = \sum_{c\in\mathcal{C}} |V(C)|$.

In Korollar 3.4[41] haben wir gezeigt, dass die Gesamtgröße der Cliquenzerlegung, die der Algorithmus von Feder und Motwani liefert, durch m^* beschränkt ist.
Wir ersetzen jede Clique C^i mit mehr als einer Kante durch einen zu C^i matching-äquivalenten CMA-Graphen C^i_S und behalten die Ein-Kanten-Cliquen bei. Die neue Kantenzahl ist die Summe der Kanten in den CMA-Graphen zuzüglich der Zahl der Ein-Kanten-Cliquen.
Wir werden im nächsten Kapitel sehen, dass es CMA-Graphen $C_S = (V_L(C_S)\dot\cup V_M(C_S)\dot\cup V_R(C_S), E(C_S))$ gibt, die eine in der Zahl der Knoten aus $V_L(C_S)$ und $V_R(C_S)$ lineare Kantenzahl besitzen, also $|E(C_S)| < p(|V_L(C_S)| + |V_R(C_S)|)$ für ein festes $p > 1$ erfüllen. $|V_L(C_S)| + |V_R(C_S)|$ wiederum ist gleich der Zahl der Knoten der Clique C.
Die Ein-Kanten-Cliquen bleiben unverändert werden also durch eine Kante "ersetzt". Ihre Zahl wird sich also auf jeden Fall weniger als ver p-fachen. Somit ergibt sich:

$$\begin{aligned} |E(G_S)| &\leq \sum_{C\in\mathcal{C}} |E(C_S)| \\ &\leq \sum_{C\in\mathcal{C}} \max(1, p(|V_L(C_S)| + |V_R(C_S)|)) \\ &\leq \sum_{C\in\mathcal{C}} p|V(C)| \\ &\leq p\sum_{C\in\mathcal{C}} |V(C)| \\ &\leq pm^*. \end{aligned}$$

Die Zahl der Kanten von G_S ist also linear in der Gesamtgröße der Cliquenzerlegung. □

Nun ist zu zeigen, dass wir aus einem maximalen Matching in G_S eines in G konstruieren können. Dazu können wir Lemma 3.11[45] auf jede einzelne Biclique anwenden. Da diese kantendisjunkt sind, treten keinerlei Wechselwirkungen zwischen diesen Anwendungen auf.
Die folgenden Lemmata, in denen die eben gezeigten Ergebnisse von einer einzelnen Biclique auf eine Cliquenzerlegung übetragen werden, stellen daher weitgehend Wiederholungen dar und wir fassen sie entsprechend kurz.

Lemma 3.18 *Sei M ein maximales Matching in G und M_S ein maximales Matching in G_S. Dann läßt M mindestens genausoviele Knoten frei wie M_S, d.h.*

$$|V_G \setminus \cup_{e \in M} e| \geq |V_{G_S} \setminus \cup_{e \in M_S} e|.$$

Beweis: Zunächst nimmt man alle Kanten aus M, die nicht in einer der gefundenen Cliquen liegen in M_S auf. Danach verfährt man für jede Clique C_i wie im "$\leq$" Teil des Beweises von Lemma 3.11[45] und ersetzt sie durch den entsprechenden CMA-Graphen C_S^i. Seien $V_{i,L}$ und $V_{i,R}$ die Knoten auf der "linken" bzw. "rechten" Seite der Clique, die von Kanten aus $M \cap E(C_i)$ überdeckt werden. Da die Clique einen bipartiten Graphen bildet, gilt $|V_{i,L}| = |V_{i,R}|$. Damit gibt es nach Definition von C_S ein Matching von C_S^i, dass alle Knoten von C_S sowie genau $V_{i,1}$ und $V_{i,2}$ überdeckt. Wiederholt man dieses Verfahren für alle Cliquen, so sind in G_S genau diejenigen Knoten nicht überdeckt, deren Urbilder in G nicht überdeckt sind. □

Da das vom Algorithmus bestimmte Matching nicht genau das Matching aus Lemma 3.18 sein muss, müssen wir im nächsten Lemma wieder von einem beliebigen maximalen Matching ausgehen:

Lemma 3.19 *Sei ein maximales Matching M_S in G_S gegeben. Dann kann in Zeit $\mathcal{O}(m)$ ein Matching M' in G konstruiert werden, das höchstens soviele Knoten unüberdeckt läßt, wie M_S in G_S.*

Beweis: Wir zeigen, dass man jeden Ersatzgraphen so durch eine Clique und passende Matchingkanten zurückersetzen kann, dass höchstens so viele Knoten der Clique unüberdeckt bleiben, wie unüberdeckte Knoten im Ersatzgraphen vorkamen: Wir betrachten dazu wieder einen einzelnen CMA-Graphen $C_S^i = (V_L(C_S^i) \dot{\cup} V_M(C_S^i) \dot{\cup} V_R(C_S^i), E(C_S^i))$, der für eine Clique C^i eingesetzt wurde.
Seien dazu M_L und M_R die Knoten aus $V_L(C_S^i)$ bzw. $V_R(C_S^i)$, die vom Matching überdeckt wurden. Man setzt nun die ursprüngliche Clique $C^i = (A \dot{\cup} B, A \otimes B)$ wieder ein und wählt $\min(|M_L|, |M_R|)$ Kanten beliebig zwischen A und B für das Matching. Dies erhöht die Zahl der nicht überdeckten Knoten in G höchstens um $||M_L| - |M_R||$. Gemäß Lemma 3.13[46] müssen aber mindestens so viele unüberdeckte Knoten durch das Löschen von $V_M(C_S^i)$ bei der Ersetzung entfernt worden sein. □

Die Zahl der durch rekonstruierte Matching M' nicht überdeckten Knoten in G ist also kleiner gleich der von M_S nicht überdeckten Kanten in G_S. Da dies ein maximales Matching in G_S ist, ist diese Zahl wiederum kleiner gleich der von einem maximalen Matching M in G nicht überdeckten Knoten. Damit ist M' ein maximales Matching.

$$V(G) - 2|M'| \leq V(G_S) - 2|M_S| \leq V(G) - 2|M| \Leftrightarrow |M'| \geq |M|$$

$\square_{\text{S3.15}}$

3.2 Konstruktion von CMA-Graphen

3.2.1 Superkonzentratoren

Ziel dieses Abschnitts ist es, Graphen mit wenigen Kanten zu finden, durch die mehrere knotendisjunkte alternierende Pfade laufen können. Unter dem "Durchlaufen" verstehen wir, dass wir von einer bestimmten Teilmenge der Knoten, den "Eingängen", disjunkte Pfade zu einer anderen Teilmenge, den "Ausgängen", finden wollen.
Hierzu benötigen wir zunächst Graphen, die viele disjunkte Wege zulassen. Grundlage für unsere Konstruktion werden entsprechend sogenannte Superkonzentratoren sein.

Definition 3.20 *Ein (n, k) Superkonzentrator ist ein gerichteter azyklischer Graph, mit n Eingängen $A \subset V$, n Ausgängen $B \subset V$ und höchstens kn Kanten, so dass es für zwei beliebige Mengen von $t \leq n$ Eingängen $A' \subset A$ und t Ausgängen $B' \subset B$ stets t knotendisjunkte Pfade gibt, die Ein- und Ausgänge paarweise miteinander verbinden.*

Basierend auf Vorarbeiten von Pinsker [33] gelang es Valiant [39] einen Superkonzentrator linearer Größe ($k = 238$) zu erzeugen. Dies konnte durch Pippenger [34] auf $k = 39$ verbessert werden. All diese Konstruktionen waren probablistischer Natur. Eine deterministische Konstruktion mit $k = 504$ gelang erstmals Gabber und Galil [15], dies konnte von Alon und Capalbo [31] auf $k = 44$ verbessert werden.

Wir werden im folgenden die Konstruktion von Gabber und Galil [15] kurz vorstellen. Die von ihnen gefundenen Superkonzentratoren haben einen rekursiven Aufbau. Ein Superkonzentrator S besteht aus n direkten Verbindungen zwischen den n Ein- und Ausgängen, sowie einem um einen Faktor θ kleineren Superkonzentrator S', der mittels zweier sogenannter $(n, \theta n, k)$-Konzentratoren C und C' eingepaßt wird (vgl. Abbildung 3.6).

Definition 3.21 *Ein $(n, \theta n, k)$-Konzentrator ist ein gerichteter azyklischer Graph mit n Eingängen, $\theta n \leq n$ Ausgängen und höchstens nk Kanten, der es für jede θn elementige Teilmenge der Eingänge erlaubt, diese durch knotendisjunkte Wege mit den θn Ausgängen zu verbinden.*

Bemerkung 3.22 *All diese Konstruktionen können natürlich auch gegen die Kantenrichtung benutzt werden. Man könnte einen Konzentrator also auch definieren als einen gerichteten azyklischen Graph mit θn Eingängen, $n \geq \theta n$ Ausgängen und höchstens nk*

Kanten, der es für jede θn elementige Teilmenge der Ausänge erlaubt, diese durch knotendisjunkte Wege von den θn Eingängen aus zu erreichen. Wir setzen die Konzentratoren in beiden Richtungen ein.

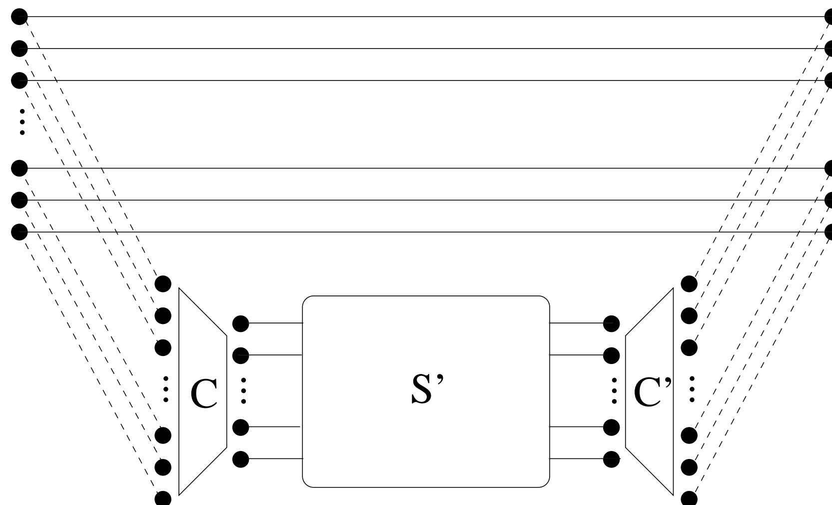

Abbildung 3.6: Grobstruktur eines Superkonzentrators

Lemma 3.23 *Sei S' ein $(\theta n, k)$ Superkonzentrator für $1 > \theta \geq \frac{1}{2}$ und seien C und C' $(n, \theta n, k)$ Konzentratoren (wobei C' vom umgedrehten Typ ist - vgl. Bemerkung 3.22). S sei der Graph mit n Eingangs- und n Ausgangsknoten, der entsteht, wenn man*

1. *die n Eingänge von S paarweise mit den n Ausgängen von S,*
2. *die n Eingänge von S paarweise mit den n Eingängen von C,*
3. *die θn Ausgänge von C paarweise mit den Eingängen von S',*
4. *die θn Ausgänge von S' paarweise mit den Eingängen von C'*
5. *und die n Ausgänge von C' paarweise mit den n Ausgängen von S*

verbindet (vgl. Abb. 3.6).
Dann ist S ein (n, k) Superkonzentrator.

Beweis: Bei einem Superkonzentrator muss jede Teilmenge der Eingänge der Größe t paarweise über disjunkte Pfade mit einer genauso großen Teilmenge der Ausgänge verbunden werden können. Wir nehmen anhand der Größe von t eine Fallunterscheidung vor:
Fall 1: $t \leq \frac{n}{2}$
Man betrachte eine beliebige Obermenge der Menge der Eingänge mit Größe θn. Nach Definition von C gibt es eine Möglichkeit, diese paarweise mit den θn Ausgängen von C und somit mit den Eingängen von S' zu verbinden. Dabei werden die t gewählten Eingänge von S durch C mit t Eingängen von S'

verbunden. Ebenso verbindet C' als umgedrehter Konzentrator die gewählten t Ausgänge von S rückwärts mit t Ausgängen von S'. Da S' ein Superkonzentrator ist, gibt es für jede beliebige Wahl von t Ein- und Ausgängen disjunkte Pfade, die diese paarweise verbinden, also auch für die von C und C' aus erreichten Ein- bzw. Ausgänge. Damit sind also alle t Ein- und Ausgänge von S paarweise durch disjunkte Pfade miteinander verbunden.
Fall 2: Ist $t > \frac{n}{2}$, so müssen wenigstens $t - \frac{n}{2}$ der Ein und Ausgänge direkt gegenüber liegen, so dass diese unmittelbar verbunden werden können. Die restlichen $\leq \frac{n}{2}$ Knoten werden wie im ersten Fall über S' verbunden. □

3.2.2 Konstruktion von Konzentratoren in Linearzeit

Wir haben im vorangegangenen Kapitel ein rekursives Konstruktionsschema für Superkonzentratoren vorgestellt. Jeder Superkonzentrator mit n Ein- und Ausgängen besteht aus einem kleineren Superkonzentrator, n direkten Kanten zwischen Ein- und Ausgängen sowie zwei Konzentratoren C und C' mit maximal n Ein- oder Ausgängen. Die Größe der rekursiv ineinander verschachtelten Superkonzentratoren nimmt dabei bei jedem Rekursionsschritt um den konstanten Faktor θ ab.
Um zu zeigen, dass die vorgestellten Superkonzentratoren in $\mathcal{O}(n)$ konstruiert werden können, postulieren wir, dass die Konzentratoren C und C' ebenfalls in Linearzeit in der Zahl ihrer Ein- und Ausgänge n erzeugt werden können: Haben diese Konzentratoren also jeweils die Größe cn mit einer Konstanten c, so ist die Gesamtgröße des Superkonzentrators beschränkt durch

$$\sum_{i=0}^{\infty} \underbrace{\theta^i n}_{\text{direkte Kanten}} \quad \underbrace{2c\theta^i n}_{\text{Konzentratoren}} = (n+2cn)\sum_{i=0}^{\infty}\theta^i = (n+2cn)\frac{1}{1-\theta} = n\left(\frac{2c+1}{1-\theta}\right)$$

Es bleibt also zu zeigen, dass die Konzentratoren in Linearzeit konstruiert werden können, was impliziert, dass ihre Kantenzahl ebenfalls in $\mathcal{O}(n)$ liegen muss. Wir beschreiben im Folgenden die Konzentratoren von Gabber und Galil[15], da die Konstruktion von Alon [31] diese voraussetzt. Dabei verzichten wir auf die maßtheoretischen Korrektheitsbeweise und beschränken uns auf die Komplexität der Konstruktion.

Zunächst gehen wir zu einer etwas schwächeren Form von Konzentratoren über: Eben haben wir verlangt, dass jede Teilmenge von Eingängen der Größe θn zu den genau θn Ausgängen verbunden werden kann. D.h. alle Ausgänge werden benutzt. Wir trennen nun die Zahl der benötigten Pfade von der Zahl der Ausgänge. Letztere bezeichnen wir nach wie vor mit θn, die Zahl der Pfade nennen wir αn. Wir verlangen, dass jede αn-elementige Menge von Eingängen mit irgendwelchen αn der insgesamt θn Ausgängen verbunden werden kann ($0 < \alpha < \theta < 1$).
Damit die eben beschriebene rekursive Konstruktion durchgeführt werden kann, genügt es, $\alpha \geq \frac{1}{2}$ zu erfüllen. θ muss lediglich kleiner als 1 sein. Dies ist hinreichend dafür,

1. dass die höchstens $\frac{n}{2}$ Verbindungen, die nicht direkt hergestellt werden können, weitergeleitet werden können und

2. dass die Größe der rekursiv verschachtelten Superkonzentratoren geometrisch abnimmt.

Es ist also möglich, den Superkonzentrator mit Hilfe der folgenden Konzentratoren zu konstruieren:

Definition 3.24 ([15]) *Ein (n, θ, k, α) beschränkter Konzentrator ist ein bipartiter Graph $G = (A \dot{\cup} B, E)$ mit n Eingängen A und θn Ausgängen B sowie höchstens kn Kanten, bei dem für alle Mengen X von Eingängen mit $|X| \leq \alpha n$ gilt, dass $|\Gamma(X)| \geq |X|$ ist.*

Der Ausdruck $|\Gamma(X)| \geq |X|$ stammt aus dem Satz von Hall 7.15$_{138}$. Er stellt - da diese Eigenschaft der Nachbarschaften hier implizit auch für alle Teilmengen von X gefordert wird - sicher, dass es zur Menge X ein Matching gibt, das diese überdeckt. Die Matchingkanten stellen dann die gewünschten Verbindungen zwischen Ein- und Ausgängen dar.

Gabber und Galil [15] konstruieren ihre beschränkten Konzentratoren mit Hilfe von sogenannten Expandern:

Definition 3.25 ([15]) *Ein (n, k', d)-Expander ist ein bipartiter Graph $G = (A \dot{\cup} B, E)$ mit $|A| = |B| = n$, der höchstens $|E| \leq nk'$ Kanten hat und bei dem für jede Teilmenge X von A gilt, dass $|\Gamma(X)| > (1 + d(1 - |X|/n))|X|$ erfüllt ist.*

Wir nehmen im Weiteren an, dass die Zahl n der Eingänge eine Quadratzahl ist, d.h. $n = q^2, q \in \mathbb{N}$. Dies vereinfacht die Darstellung. Für andere n ist es möglich, stattdessen zur nächstgrößeren Quadratzahl überzugehen.

Satz 3.26 ([15]) *Es gibt einen $(n, 5, (2 - \sqrt{3})/4)$-Expander mit $n = q^2$ für $q \in \mathbb{N}$.*

> **Beweisskizze:** Wir geben hier lediglich die Konstruktion an und verweisen für den Beweis, dass es sich tatsächlich um einen Expander mit den gewünschten Eigenschaften handelt, auf die hier beschriebene Arbeit von Gabber und Galil [15].
> Es ist möglich, jeden der jeweils n Knoten aus A und B mit jeweils genau einem Tupel $[i, j]$ mit $i, j \in \{0, \ldots, q-1\}$ zu bezeichnen. Die Knoten aus B, die zu einem Knoten $[i, j] \in A$ adjazent sind, sind dann gegeben durch die folgenden 5 Abbildungen:
>
> - $[i, j]$
> - $[i, i + j \bmod q]$
> - $[i, i + j + 1 \bmod q]$
> - $[i + j \bmod q, j]$
> - $[i + j + 1 \bmod q, j]$
>
> $\Diamond$

Beobachtung 3.27 *Das Bild jedes Elementes unter einer dieser Permutationen kann (Einheitskostenmaß vorausgesetzt, vergl. hierzu Kapitel 5.1*[115]*) in konstanter Zeit berechnet werden. Damit können alle* $5n$ *Kanten des Expanders in* $\mathcal{O}(n)$ *Schritten bestimmt werden.*

Lemma 3.28 ([15]) *Mit Hilfe der eben definierten Expander kann man einen* $(n, \frac{31}{32}, \frac{(5+1)31}{32}, \frac{1}{2})$ *beschränkten Konzentrator konstruieren.*

Beweis: Definitionsgemäß hat der Konzentrator n Eingänge und $\frac{31n}{32}$ Ausgänge. Die Eingänge werden in zwei Mengen zerlegt, die "große" Menge mit $\frac{31n}{32}$ und die "kleine" Menge mit $\frac{n}{32}$ Knoten. Die große Menge wird mittels eines $\left(\frac{31n}{32}, 5, \frac{2}{30}\right)$ Expanders mit den Ausgängen verbunden ($\frac{2}{30} \leq (2-\sqrt{3})/4$).
Jeder Knoten des "kleinen" Teils wird mit jeweils 31 Ausgängen verbunden, so dass jeder Ausgang mit einem Eingang aus dem "kleinen" Teil verbunden ist.
Sei nun X eine Teilmenge der Eingänge mit $|X| \leq \frac{1}{2}n$. Sei s die Zahl der Eingänge, die zum "großen Teil" gehören, und t die Zahl der Eingänge, die zum "kleinen Teil" gehören.
Ist $t > |X|/31$, so reichen die $31t$ Nachbarn der Knoten aus dem "kleinen Teil" bereits aus.
Anderenfalls ist $s \geq \frac{30}{31}|X|$. Wählt man eine beliebige Teilmenge Y der Größe $\left\lceil\frac{30}{31}|X|\right\rceil \leq \frac{1}{2}n$ der Eingänge aus dem "großen Teil", so ergibt sich aufgrund der Expander-Eigenschaften, dass diese mindestens

$$\left(1 + \frac{2}{30}\left(1 - \frac{1}{2}\right)\right)|Y| = \frac{31}{30}\left\lceil\frac{30}{31}|X|\right\rceil \geq |X|$$

Nachbarn haben muss. Da Y eine Teilmenge von X war, hat auch X mindestens ebensoviele Nachbarn. □

Beobachtung 3.29 *Auch die Konstruktionen aus Lemma 3.28 lassen sich in Linearzeit in der Kantenzahl realisieren. Somit können Superkonzentratoren mit* n' *Knoten in Zeit* $\mathcal{O}(n')$ *konstruiert werden.*

3.2.3 Umwandlung von Superkonzentratoren in CMA-Graphen

Im folgenden werden wir mit einer ungerichteten Variante der eben konstruierten Superkonzentratoren weiterarbeiten. Auf die ehemalige Orientierung werden wir aber hin und wieder Bezug nehmen. Wir wollen aus einem gegebenen Superkonzentrator S einen zu einer bipartiten Clique $C = (A\dot{\cup}B, E)$ matching-äquivalenten Graphen $C_S = (V_L\dot{\cup}V_M\dot{\cup}V_R, E)$ erzeugen.
Zur Vereinfachung betrachten wir zunächst eine Clique, bei der beide Seiten gleich groß sind, d.h. $|A| = |B|$ gilt. Wir wählen entsprechend einen Superkonzentrator S mit $|A|$

Ein- und Ausgängen und $\mathcal{O}(|A|)$ Kanten. Dass ein solcher existiert, haben wir im vorangegangenen Kapitel gezeigt.
Dann ersetzen wir jeden Knoten v in S durch zwei Knoten v_1 und v_2. Wir passen dabei die Kanten unter Verwendung der ehemaligen Kantenrichtung an und fügen für jede Kante $(u, w) \in E(S)$ die Kante (u_2, w_1) ein. Des weiteren verbinden wir für alle $v \in V(S)$ v_1 mit v_2.
Den resultierenden Graph bezeichnen wir mit C_S. C_S ist bipartit und die Zahl der Kanten wird im Vergleich zu S höchstens um $|V(S)|$ erhöht, so dass deren Zahl nach wie vor in $\mathcal{O}(|A|)$ liegt. Die bisher beschriebenen Knoten bilden V_M. Wir fügen zwei Knotenmengen V_L und V_R mit je $|A|$ Knoten hinzu (vgl. Abb. 3.7). Die Knoten aus V_L verbinden wir paarweise mit den Knoten aus der Menge $\{v_1|v$ war ein Eingangsknoten des Superkonzentrators $S\}$. Die Knoten aus V_R verbinden wir paarweise mit den Knoten der Menge $\{v_2|v$ war ein Ausgangsknoten von $S\}$.

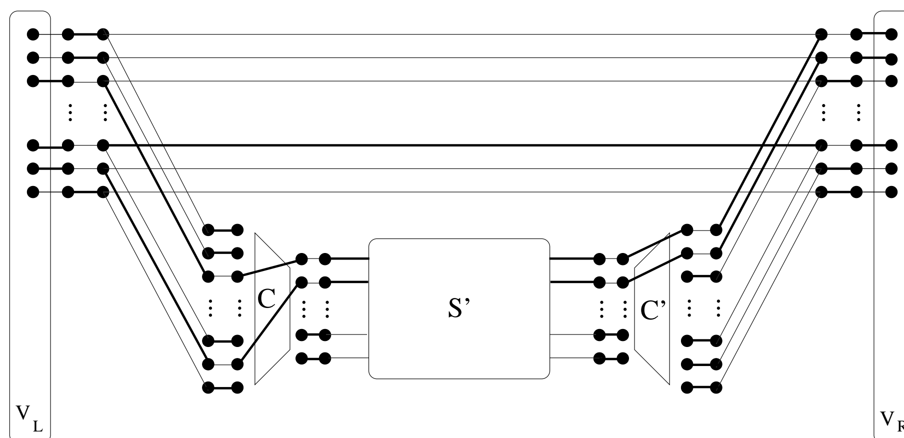

Abbildung 3.7: CMA Graph

Lemma 3.30 *Der so konstruierte Graph C_S ist matching-äquivalent zu einer bipartiten Clique mit $|A| = |B|$.*

Beweis: Dazu müssen die zwei definierenden Eigenschaften von CMA-Graphen nachgewiesen werden:
1. Jedes Matching, das V_M überdeckt, überdeckt gleich viele Kanten in V_R und V_L:
V_M ist nach Konstruktion bipartit mit den gleich großen Partitionen $P'_1 = \{v_1|v \in S\}$ und $P'_2 = \{v_2|v \in S\}$. Es gibt keine Kanten von V_L nach P'_2 und von V_R nach P'_1, so daß $(P_1 \dot{\cup} P_2, E(C_S))$ mit $P_1 := P'_1 \cup V_R$ und $P_2 := P'_2 \cup V_L$ ebenfalls ein bipartiter Graph ist.
Da auch V_L und V_R gleich groß sind, gilt $|P_1| = |P_2|$. In jedem bipartiten Graph überdeckt jedes Matching genausoviele Knoten in beiden Partitionen, da eine Kante jeweils Knoten aus verschiedenen Partitionen verbindet.
Sei also ein Matching M gegeben, das alle Knoten aus V_M überdeckt. Es überdeckt also jeweils $|M|$ Knoten in P_1 und P_2 und die zu V_M gehörenden

Teile P'_1 und P'_2 vollständig. Somit ist die Zahl der Knoten aus V_L, die das Matching überdeckt, gegeben durch $|M| - |P'_1|$ und die Zahl der überdeckten Knoten aus V_R als $|M| - |P'_2|$. Wegen $|P'_1| = |P'_2|$ überdeckt das Matching gleich viele Knoten in V_L und V_R.

2. Seien auf der anderen Seite zwei Teilmengen $V'_L \subset V_L$ und $V'_R \subset V_R$ mit $|V'_L| = |V'_R|$ gegeben. Dann ist zu zeigen, dass es ein Matching M von C_S gibt, das genau diese Knoten und V_M überdeckt.
Hierzu beobachten wir zunächst, dass auch C_S noch ein Superkonzentrator ist, da wir beim Aufspalten der Knoten die ehemalige Orientierung betrachtet haben und ein Pfad, der in S einen Knoten v passiert hat, nun v_1 und v_2 passieren kann. Es gibt also eine Menge $\mathcal{P}$ von $|V'_L|$ knotendisjunkten Pfaden $\{P_1, \ldots P_{|V'_L|}\}$ in C_S, die V'_L mit V'_R verbinden und zudem, wenn sie einen Knoten x_1 verwenden, stets auch x_2 passieren.

Wir konstruieren nun ein Matching M in der folgenden Weise: Sei $E(\mathcal{P})$ die Menge aller Kanten und $V(\mathcal{P})$, die Menge aller Knoten, die auf einem Pfad P aus $\mathcal{P}$ liegen. Für $v_1 \in V_M \backslash V(\mathcal{P})$ fügen wir jeweils die Kante (v_1, v_2) in das Matching ein. Diese Matchingkanten überdecken sicher alle Knoten in V_M, die nicht auf einem der Pfade liegen.
Des weiteren fügen wir alle Kanten aus $E(\mathcal{P})$ ein, die von der Form (u_2, x_1) sind, also jeweils die Kanten auf dem Pfad, die den Kanten des ursprünglichen Superkonzentrators entsprechen. Außerdem fügen wir die Anfangs- und Endstücke der Form $(v, u_1), v \in V'_L$ oder $(x_2, u), u \in V'_R$ hinzu.
Betrachtet man einen einzelnen Pfad P_i, so ist dieser von der Form $V'_L \ni v, v_1^1, v_2^1, v_1^2, v_2^2, \ldots, v_1^j, v_2^j, u \in V'_R$. Dabei sind $(v, v_1^1), (v_2^1, v_1^2), \ldots, (v_2^j, u)$ Matchingkanten. Die Knoten sind alle überdeckt, d.h. das Matching überdeckt ebenfalls V'_L und V'_R und alle Knoten in V_M, die auf einem Pfad liegen. Damit ist ganz V_M überdeckt. □

Bemerkung 3.31 *Alternativ kann man sich die Konstruktion auch so vorstellen, dass man zunächst alle Knotenpaare $\{v_1, v_2\}$ zu Matchingkanten erklärt und dann entlang der Pfade aus $\mathcal{P}$ alterniert.*

Proposition 3.32 *Ist $C = (A_C \dot\cup B_C, E)$ eine bipartite Clique mit $|A_C| > |B_C|$, so kann man einen zu dieser matching-äquivalenten Graphen C'_S mit $\mathcal{O}(|A_C|)$ Kanten und Knoten konstruieren.*

Beweis: Man konstruiert zunächst - wie eben beschrieben - einen CMA-Graphen C_S mit $|V_L|$=$|V_R|$=$|A_C|$. Dann löscht man $|A_C| - |B_C|$ Knoten beliebig aus V_R.
Der weitere Beweis verläuft analog zum obigen. Beide Beweisschritte bleiben - auch wenn diese Knoten gelöscht werden - korrekt. □

Verwendet man die von Alon [31] konstruierten Superkonzentratoren mit $k = 88$, so kann eine Clique mit $n^{1-\delta \frac{\log n}{\log \frac{2n^2}{m}}}$ Kanten, wie sie durch das Kompressionsverfahren von Feder

und Motwani [11] gefunden wird (vgl. Kapitel 2.1.1$_{18}$), durch einen CMA-Graphen mit $88n^{1-\delta}$ Kanten ersetzt werden.

Kapitel 4

Implizite Kompression

4.1 Einleitung

Im vorangegangenen Teil haben wir ein erstes Verfahren beschrieben, das Graphenkompression zur Beschleunigung von Matching-Algorithmen auf nichtbipartiten Graphen einsetzt. Wir haben zu einem Graphen G einen Graphen G_S konstruiert, der (asymptotisch bzgl. der Knotenzahl n) weniger Kanten als G hat und so beschaffen ist, dass man aus einem maximalen Matching in G_S ein maximales Matching in G konstruieren kann. G muss wegen der auftretenden Konstanten allerdings sehr viele Knoten haben, damit G_S auch absolut weniger Kanten hat als G.
In diesem Teil wollen wir ein Verfahren vorstellen, mit dem auch kleinere Graphen behandelt werden können. Wir erkaufen diesen Vorteil mit einer größeren Kompliziertheit von Algorithmen und Beweisen, da wir uns nun auf einen konkreten Matching-Algorithmus konzentrieren werden und seine spezifischen Eigenschaften berücksichtigen müssen. Wir werden zu diesem Zweck den Matching-Algorithmus von Blum [1][2] betrachten.

Wir gehen zunächst wiederum vom Kompressionsalgorithmus von Feder und Motwani [11] aus. Diese haben, ausgehend von einer Cliquenzerlegung des Graphen, bipartite Cliquen durch “Sterne” ersetzt. Entscheidend dabei war, dass durch diese Sterne der gleiche Netzwerkfluss möglich ist, wie durch die Cliquen. Problematisch ist wiederum, dass im nichtbipartiten Fall kein einfaches Flussmodell möglich ist und dass augmentierende Pfade diese Sterne nicht in gleicher Weise wie Flüsse durchqueren können (vgl. Kapitel 2.1.1[18]).
Im vorangegangenen Teil sind wir diesem Problem begegnet, indem wir statt Sternen wesentlich kompliziertere Ersatzgraphen verwendet haben. In diesem Teil betrachten wir einen Matching-Algorithmus unter der Fragestellung, welche Teile des Algorithmus den meisten Aufwand verursachen und zeigen, dass diese Teile im Prinzip mit Sternen als Ersatzgraphen beschleunigt werden können.

Alle auf der Suche nach augmentierenden Pfaden beruhenden Verfahren verwenden Varianten der sog. algorithmischen Suche auf Graphen [3], insbesondere Breitensuche und Tiefensuche. Diese Suchverfahren sind verantwortlich für den größeren der beiden Faktoren m in der Laufzeitschranke des Algorithmus von Hopcroft und Karp [21] und somit

auch des darauf basierenden Algorithmus von Blum [2]. Also werden wir versuchen, diese Phasen mit Hilfe von Graphenkompression zu beschleunigen.

Um Graphenkompression anwenden zu können, benötigen wir einen bipartiten Graphen. Wir haben in Kapitel 2.3.135 gesehen, dass man einen Graphen G in eine bipartite Repräsentation G_B überführen kann, indem man jeden Knoten u durch zwei Knoten $[u, A]$ und $[u, B]$ ersetzt und jede Kante (u, v) durch zwei Kanten $([u, B], [v, A])$ und $([v, B], [u, A])$ ersetzt. Die Richtung dieser Kanten war dabei von einem gegebenen Matching abhängig.
Bei der Kompression können wir auf diese Richtungen keine Rücksicht nehmen, da wir nur Zeit für einen Kompressionsvorgang haben, aber viele Matchings und somit viele verschiedene Orientierungen auftreten.
Wir nehmen also im folgenden an, dass der Graph G_B vollständig von B nach A gerichtet ist. Wir wenden auf diesen Graph das Kompressionsverfahren von Feder und Motwani[11] an, wobei als Ersatzgraphen Sterne eingesetzt werden. Den entstehenden Graphen nennen wir G'_B.

Um die Grundidee für die Beschleunigung der algorithmischen Suche anschaulicher darzustellen, vergleichen wir zunächst eine Tiefensuche auf G_B mit einer solchen auf G'_B. Später müssen wir die Suchvorgänge abstrahiert betrachten, da kompliziertere Suchmethoden - insbesondere die sogenannte “Double Depth First Search” - nicht mehr mit G'_B alleine realisiert werden können.
Als Beispiel für die Auswirkung der Kompression auf eine Tiefensuche betrachten wir Abbildung 4.1. Bild 1 zeigt den ursprünglichen Graphen G_B, Bild 2 seine komprimierte Darstellung G'_B. Die von den jeweils drei oberen Knoten gebildete Clique wurde durch einen Stern ersetzt. Bild 3 zeigt die Situation in G_B während einer Tiefensuche. Wir nehmen an, dass wir zunächst von Knoten b aus gesucht haben und nun eine neue Suche von Knoten v aus starten.
In Bild 3 gehen von v 4 unbetrachtete Kanten aus, die wir noch untersuchen müssen. Nur eine davon führt zu einem Knoten, den wir noch nicht erreicht haben. Bei Bild 4 - hier wieder in G'_B - sind es nur zwei unbetrachtete Kanten. Sobald wir den Zentralknoten des Sterns besucht haben, besuchen wir als nächstes sofort den neuen Knoten. Wir haben also zwei sinnlose Suchen vermieden. Dennoch ist klar, dass wir alle erreichbaren Knoten auf der rechten Seite erreichen.

Bevor wir beschreiben können, wie dieses einfache Schema abgewandelt werden muss, um auch im nichtbipartiten Fall zu funktionieren, müssen wir die dabei auftretenden Suchprobleme erst erläutern. Wir geben daher einen kurzen Überblick über den Aufbau des Algorithmus von Blum [1][2].

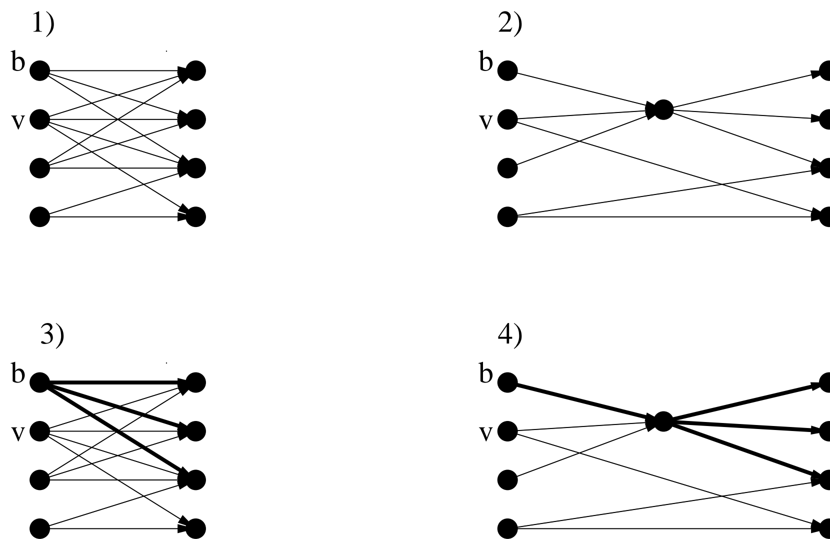

Abbildung 4.1: Beispiel für eine Tiefensuche ohne und mit Kompression

4.2 Der Matching-Algorithmus von Blum

4.2.1 Gesamtaufbau

In diesem Kapitel geben wir einen kurzen Überblick über den Matching-Algorithmus von Blum [1][2]. Die Grundidee ist, den Algorithmus von Hopcroft und Karp [21] bzw. Dinic [8] auf den nichtbipartiten Fall zu übertragen.
Entscheidend ist die Beobachtung, dass beim Beweis von

Satz 1.8$_{10}$ [Hopcroft, Karp 1973]
Augmentiert man über eine nicht erweiterbare Menge alternierender knotendisjunkter Pfade minimaler Länge $\mathcal{P}$, so sind alle augmentierenden Pfade im resultierenden Graphen länger als die Pfade aus $\mathcal{P}$.

nicht vorausgesetzt wurde, dass der Graph bipartit ist. Dieser Satz gilt also auch im nichtbipartiten Fall.
Ist es also auch im nichtbipartiten Fall möglich, eine nicht erweiterbare Menge kürzester augmentierender Pfade zu finden, so kann man durch $\sqrt{n}$-fache Wiederholung dieses Vorganges auch in diesem Fall erzwingen, dass alle verbleibenden augmentierenden Pfade mindestens die Länge $\sqrt{n}$ haben.
Da auch im nichtbipartiten Fall somit noch höchstens $\mathcal{O}(\sqrt{n})$ Knoten unüberdeckt bleiben, ist es möglich, das verbleibende Matching mit höchstens $\mathcal{O}(\sqrt{n})$ Tiefensuchen nach augmentierenden Pfaden und nachfolgenden Augmentierungen zu einem maximalen Matching zu erweitern.
Ein maximales Matching kann also mittels $\mathcal{O}(\sqrt{n})$ dieser Augementierungen über maximale Mengen kürzester augmentierender Pfade (Phase 1) gefolgt von höchstens $\mathcal{O}(\sqrt{n})$ Tiefensuchen nach augmentierenden Pfaden (Phase 2) gefunden werden.

Um auf diese Weise die angestrebte Laufzeit von $\mathcal{O}(\sqrt{n}m)$ zu erzielen, muss jeder $\sqrt{n}$

mal wiederholte Teil einer Phase in Zeit $\mathcal{O}(m)$ realisiert werden. Im bipartiten Fall wird dies in Phase 1 durch eine Kombination von Breiten- (BFS) und Tiefensuche (DFS) erreicht (vgl. Lemma 1.9$_{11}$), in Phase 2 wird nur Tiefensuche verwandt.
Im nichtbipartiten Fall realisiert Blum[1][2] dies mit zwei Variationen dieser Verfahren: Der modifizierten Breitensuche MBFS und der modifizierten Tiefensuche MDFS.

Mit Algorithmus 3 geben wir zunächst einen vereinfachenden Überblick über die einzelnen Phasen des Gesamtalgorithmus. In Phase 1 werden maximale Mengen kürzester augmentierender Pfade bestimmt. Hierzu ermittelt MBFS zunächst alle Knoten und Kanten, die zu irgendeinem kürzesten Pfad gehören. Dann sucht MDFS in dieser Menge eine nicht erweiterbare Menge augmentierender Pfade und über die Pfade aus dieser Menge wird augmentiert.
Sobald die Pfade länger werden als $\sqrt{n}$, wird so lange mit MDFS ein augmentierender Pfad gesucht und über diesen augmentiert, bis dies nicht mehr möglich und das Matching somit maximal ist (Phase 2).

Algorithmus 3 MBFS und MDFS[1] - Überblick

1: **while** kein augmentierender Pfad länger als $\sqrt{n}$ wurde gefunden **do** ▷ Phase 1
2: Suche mit MBFS alle Knoten und Kanten, die zu kürzesten augmentierenden Pfaden gehören
3: Suche in dieser Menge mit MDFS eine maximale Menge knotendisjunkter kürzester Pfade
4: augmentiere über die Pfade aus dieser Menge
5: **end while**
6: **while** Es werden noch augmentierende Pfade gefunden **do** ▷ Phase 2
7: Suche mit MDFS einen augmentierenden Pfad P
8: Augmentiere entlang von P
9: **end while**

Der nachfolgende Algorithmus 4 stellt bereits einige Details mehr vor. Er soll dem Leser helfen, die einzelnen Teile während der Lektüre besser einordnen zu können. Einen implementierbaren Pseudocode geben wir dann am Ende dieses Kapitels an.

In den nachfolgenden Kapiteln werden wir die Anwendung von MBFS und MDFS in Phase 1 im Detail beschreiben. Für Phase 2 kann reines MDFS - entsprechend der Beschreibung für Phase 1 - verwandt werden.

In Phase 1 wird maximal $\sqrt{n}$ mal eine Suche nach einer maximalen Menge kürzester augmentierender Pfade durchgeführt. Sobald eine solche Menge jeweils gefunden wurde, wird über diese augmentiert. Um eine Gesamtlaufzeit von $\mathcal{O}(\sqrt{n}m)$ sicherzustellen, genügt es zu zeigen, dass jede Suche in $\mathcal{O}(m)$ ausgeführt werden kann. Dass die Augmentierung in dieser Zeit möglich ist, ist offensichtlich.
Wir müssen also lediglich den Suchanteil eines Teils von Phase 1, d.h. die Ermittlung einer maximalen Menge kürzester augmentierender Pfade, genauer betrachten.

Es sei also im Folgenden angenommen, ein Matching M im Graphen ist gegeben und eine nicht erweiterbare Menge augmentierender Pfade kürzester Länge ist gesucht. Um diese zu finden, ermittelt der Algorithmus mit MBFS die Menge aller Kanten, die auf einem

Algorithmus 4 Realisierung des Algorithmus von Hopcroft und Karp im nichtbipartiten Fall unter Verwendung von MBFS und MDFS gemäß [1]

```
 1: while kein augmentierender Pfad länger als √n wurde gefunden do      ▷ Phase 1
 2:     l:=0;
 3:     while kein augmentierender Pfad wurde gefunden do
 4:         MBFS-Vorwärtsmarkierung(l)
 5:         MBFS-Rückwärtsmarkierung(l)
 6:         l:=l+1
 7:     end while
 8:     Aufbereitung der Ergebnisse von MBFS für MDFS
 9:     repeat   ▷ Bestimme nicht erweiterbare Menge augmentierender kürzester Pfade
10:         MDFS-SEARCH
11:         MDFS-RECONSTRUCT
12:         Entferne den von MDFS-RECONSTRUCT gefundenen Pfad und alle von
                MDFS-SEARCH betrachteten Kanten
13:     until MDFS liefert keinen Pfad mehr zurück
14:     Augmentiere über alle Pfade aus 𝒫
15: end while
16: while noch augmentierende Pfade gefunden werden do                  ▷ Phase 2
17:     Suche mit MDFS einen augmentierenden Pfad P
18:     Augmentiere entlang von P
19: end while
```

kürzesten Pfad liegen, und bestimmt dann mit MDFS in dieser Menge von Kanten eine nicht erweiterbare Menge von kürzesten Pfaden.

4.2.2 MBFS

Ab hier werden wir stets mit dem bipartiten Graphen G_B aus Kapitel 2.3.1[35] arbeiten, der entsteht, wenn man alle Knoten v in zwei Knoten $[v, A]$ und $[v, B]$ aufspaltet. Wir nehmen an, dass ein Matching M fest gegeben ist, so dass wir die Kanten wiederum so ausrichten, dass Matchingkanten von A nach B zeigen und und freie Kanten von B nach A.
Ein *streng einfacher* Pfad ist ein einfacher Pfad, der nicht gleichzeitig einen Knoten und seinen Symmetriepartner enthält. Wir haben gesehen, dass es in G genau dann einen augmentierenden Pfad gibt, wenn es in G_B einen streng einfachen Pfad zwischen zwei freien Knoten gibt (vgl. Kapitel 2.3.1[35]).
Der Einfachheit halber werden wir von der Menge der Knoten V und der der Kanten E sprechen, ohne explizit auf G_B zu verweisen.
Die “Modifizierte Breitensuche” MBFS ordnet jedem Knoten $[v, X]$ die Länge $\mathrm{l}([v, X])$ eines kürzesten alternierenden Pfades von einem freien Knoten zu ihm zu. Auf diese Weise werden wir den Graphen in Schichten zerlegen. Die Knoten mit gleichem Level sollen

dabei in den Mengen $\mathcal{L}(l) := \{v \in V \mid \mathrm{l}(v) = l\}$ gesammelt werden.[1]
Im Gegensatz zu herkömmlichem BFS muss dabei allerdings die Entfernung nicht zwangsläufig um 1 von dem entsprechenden Wert eines Vorgängerknotens verschieden sein, da Kreise ungerader Länge die Benutzung des dazugehörigen Weges unmöglich machen können:

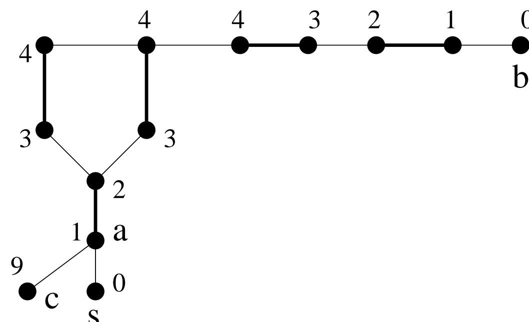

Abbildung 4.2: Überspringen eines Levels

In Abbildung 4.2 erhält Knoten a Level 1, sein Nachbar c aber Level 9, da der alternierende Pfad von s nach a nur "um die Blüte" herum zu c fortgesetzt werden könnte und daher ungültig ist. Das Level von c wird nun relativ zum freien Knoten b bestimmt.
Daher wird im folgenden dieser Markierungsschritt in zwei Phasen unterteilt, die "Vorwärtsmarkierung" und die "Rückwärtsmarkierung". Die Vorwärtsmarkierung übernimmt dabei die einfachere Aufgabe, die Markierungen solcher Knoten zu bestimmen, deren Wert durch einen unmittelbaren Nachbarknoten gegeben ist. Die Rückwärtsmarkierung setzt die Werte für identifizierte Blüten.

4.2.2.1 Vorwärtsmarkierung

Die Vorwärtsmarkierung zum Level l betrachtet alle Kanten, die einen Knoten mit Level l verlassen. Bei der Vorwärtsmarkierung können zwei Fälle unterschieden werden:
1. Ist der Knoten vom Typ $[v, A]$, so ist der einzige von diesem Knoten aus erreichbare Nachbarknoten sein Matchingpartner und dessen Level ist in diesem Fall eindeutig als das des momentanen Knotens $+1$ gegeben, da er umgekehrt auch nur von dem momentan betrachteten Knoten aus erreicht werden kann.
2. Ist der Knoten von der Form $[v, B]$ könnten bei der Vorwärtsmarkierung drei Unterfälle auftreten.
Es sei angenommen, dass in der $l$ten Runde eine Kante $([v, B], [w, A])$ mit $l = l([v, B])$ betrachtet wird.

1. $l([w, A]) > l$ und es gibt einen streng einfachen Pfad von einem freien Knoten nach $[w, A]$, der $[w, B]$ nicht enthält

[1] Eine Trennung in *even-* und *oddlevel*, wie im Algorithmus von Micali und Vazirani[30], ist dabei nicht notwendig, da jeder Knoten zweimal vorkommt.

2. $l([w,A]) > l$ und es gibt keinen streng einfachen Pfad von einem freien Knoten nach $[w,A]$, der $[w,B]$ nicht enthält

3. $l([w,A]) \leq l$

Wie in der Arbeit von Blum [2] gezeigt wird, tritt bei korrekter Ausführung von Vorwärts- und Rückwärtssuche der zweite Fall nie auf. Die Fälle 1 und 3 können leicht unterschieden werden. In Fall 1 wird das Level entsprechend auf $l+1$ angepaßt und die Kante in die Liste E_K der Kanten auf einem kürzesten Pfad eingefügt. Zusätzlich merken wir uns für jeden Knoten diejenige Kante, über die er zuerst erreicht wurde, in der Liste p. Diese werden wir aber erst bei MDFS benötigen.
Kanten, für die Fall 3 auftritt, gehören nicht zu einem kürzesten Pfad. Die sich somit ergebende MDFS-Vorwärtssuche ist in Algorithmus 5 angegeben. Eine gesonderte Berücksichtigung des Falles $[v,A]$ ist nicht notwendig, da dessen Matchingpartner stets ein undefiniertes Level haben wird.

Algorithmus 5 MBFS Vorwärtssuche

1: **procedure** MBFS VORWÄRTSSUCHE(l)
2:     **for all** $[v,Z] \in \mathcal{L}(l)$ **do**
3:         **for all** $e = ([v,Z],[u,\bar{Z}]) \in E$ **do**
4:             **if** l$([u,\bar{Z}])$ undefiniert oder l$([u,\bar{Z}]) \geq l+1$ **then**
5:                 l$([u,\bar{Z}]) := l+1$
6:                 $\mathcal{L}(l+1) := \mathcal{L}(l+1) \cup \{[u,\bar{Z}]\}$
7:                 $E_K := E_K \cup e$
8:                 **if** $p([u,\bar{Z}])$ undefiniert **then**
9:                     $p([u,\bar{Z}]) := ([v,Z],[u,\bar{Z}])$
10:                 **end if**
11:             **end if**
12:         **end for**
13:     **end for**
14: **end procedure**

4.2.2.2 Rückwärtsmarkierung

Bei der Vorwärtsmarkierung kann man das dem Stengel ”gegenüberliegende” Ende einer Blüte und auf diese Weise die Blüte daran erkennen, dass ein Knoten besucht wird, dessen Symmetriepartner bereits markiert wurde. Abb. 4.3 zeigt ein Beispiel hierzu: Die Suche schreitet auf beiden Seiten de Blüte gleichmäßig voran. In dem Augenblick, in dem die dick gezeichnete Kante untersucht wird (rechtes Bild), kann mit Sicherheit festgestellt werden, dass eine Blüte vorliegen muss.
Diese Identifizierung erfolgt also lange bevor diese Blüte zum Problem werden kann, d.h. bevor eine Kante ausgeschlossen werden muß, da sie einen Kreis ungerader Länge schließt. Dieser “Geschwindigkeitsvorteil” ist zentral für die Korrektheit sowohl des Algorithmus von Blum [2] als auch des Algorithmus von Micali und Varirani [30].

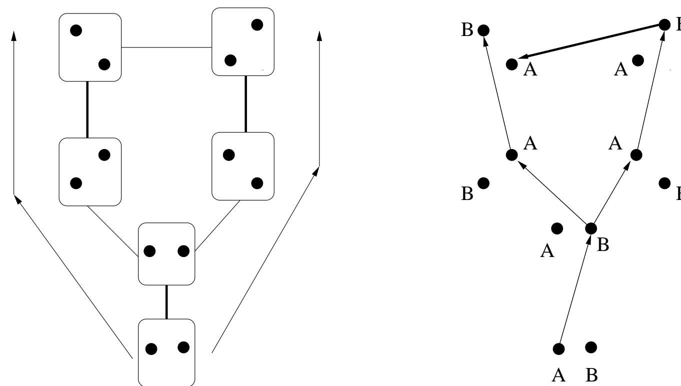

Abbildung 4.3: Identifizieren einer Blüte

Ein Graph kann potentiell eine exponentielle Zahl von Blüten enthalten. Entscheidend für alle Matching-Algorithmen ist es, dass wir nur relativ wenige davon explizit behandeln müssen. Unter einer *identifizierten* Blüte verstehen wir eine Blüte, die durch das gerade beschriebene Verfahren gefunden wurde.

Wurde eine Blüte identifiziert, können alle restlichen Knoten der Blüte unmittelbar markiert werden. Dies geschieht in G_B analog zu der von Micali und Vazirani konzipierten "double depth first search"[40]: Die Grundidee dieses Verfahrens ist es, gleichmäßig schnell auf beiden Seiten der Blüte rückwärts zu laufen. Die Knoten erhalten dabei das Level, das einem Weg "um die Blüte herum" entspricht.
Im Falle einfacher Blüten, die nur aus einem Kreis bestehen, erkennt man die Basis einfach daran, dass sich hier die Rückwärtsmarkierung bezüglich der "linken Hälfte" und die Rückwärtsmarkierung der "rechten Hälfte" begegnen. (Pfeile in Abbildung 4.4).

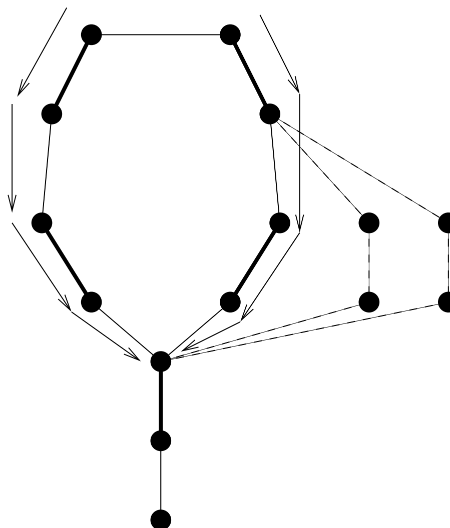

Abbildung 4.4: Gleichzeitige Rückwärtssuche bei DDFS

Im Allgemeinen können Blüten aber auch Seitenäste - sozusagen Blütenblätter - haben, die dieses einfache Schema unmöglich machen.

Um diesen Prozess exakt beschreiben zu können, müssen wir zunächst unsere anschauliche Definitionen von Blüten und Basen durch eine genaue Definition ersetzen, wobei wir hier der Notation von Blum [1] folgen.

Definition 4.1 *Sei $T \subset V$ gegeben, so daß ł(v) für alle $v \in T$ definiert ist. Dann ist $DOM(T)$ die Menge aller Knoten $[u, B]$, für die gilt, dass*

1. *$[u, B]$ liegt auf allen bisher identifizierten kürzesten streng einfachen Pfaden zu jedem Knoten aus T*
2. l$([u, A])$ *wurde noch nicht definiert*
3. l$([u, B])$ *ist maximal unter allen Knoten mit den ersten beiden Eigenschaften*

Wir werden sehen, dass die DOM-Menge eines Knoten stets nur ein Element hat. Nun können wir - quasi rückwirkend - Blüten definieren:

Definition 4.2 *Die Blüte mit Basis v ist die Menge aller Knoten u mit $DOM(u) = v$.*

Man kann leicht überprüfen, dass jede einfache Blüte auch eine Blüte ist.
Die DOM-Menge einer Knotenmenge kann sich im Laufe der Ausführung des Algorithmus ändern. Die obige Definition erfasst stets die größe identifizierte Blüte, zu der ein Knoten gehört. Da der Algorithmus von Blum auf eine explizite Verwaltung verschachtelter Blüten verzichtet, stört uns dies hier nicht weiter. Bei anderen Verfahren, wie z.B. bei dem Algorithmus von Micali und Vazirani[40], werden Blüten geringfügig anders definiert, so dass stets die kleinste Blüte erfasst wird, zu der ein Knoten gehört. Die größeren Blüten werden dann entsprechend rekursiv definiert.
Wenn wir im weiteren anschaulich von "Basis" sprechen, so meinen wir damit die DOM-Menge zur jeweiligen Ausführungszeit.

Um eine Laufzeit von $\mathcal{O}(m)$ zu gewährleisten ist es notwendig, bei der Rückwärtssuche bereits gefundene Blüten zu überspringen, da sonst deren Kanten mehrfach durchlaufen werden müssten. Dass dabei keine Pfade übersehen werden, garantiert Satz 4.3, zu dessen Beweis wir wegen seines Umfangs auf [40] verweisen.

Satz 4.3 (Micali, Vazirani, 1980-1989 [40]) *Jeder kürzeste augmentierende Pfad P zu einem Knoten v, bei dem $l([v, A])$ und $l([v, B])$ bereits gesetzt wurden, muss durch $DOM(v)$ verlaufen.*

D.h., wenn wir auf eine Blüte treffen, können wir die Suche unmittelbar an deren Basis fortsetzen.
Da Blüten aber auch in anderen Blüten enthalten sein können, müssen diese Basen so verwaltet werden, daß alle Knoten einer Blüte eine neue Basis erhalten, sobald diese als Unterblüte einer größeren Blüte erkannt wird. Dazu verwalten wir die Blüten mittels einer UNION-FIND Datenstruktur. Sobald bei der Rückwärtssuche eine Blüte durchquert wird, wird die Menge, die ihre Knoten repräsentiert, mit der Menge der übergeordneten Blüte vereint, so daß wir mittels eines FIND die größte einen Knoten enthaltende Blüte und

somit deren Basis finden können. Die aktuelle Blüte, in der sich ein Knoten u befindet, kann durch FIND(u) und deren Basis durch B(FIND(u)) ermittelt werden.

Die Notwendigkeit eines Übergangs von einem einfachen Zurücklaufen der Kanten der Blüte zu einer speziellen DFS-Variante ergibt sich daraus, dass eine Blüte mehrere "Seitenäste" haben kann (vgl. gestrichelte Kanten in Abb. 4.4) Dementsprechend geht MBFS bei der Rückwärtsmarkierung wie folgt vor: Für alle Kanten $([v,Z],[w,\bar{Z}])$, die in der vorangegangenen Vorwärtsmarkierung mit Level l betrachtet wurden und für die gilt, dass

1. $([v,Z],[w,\bar{Z}]) \in E_M$
2. $l([v,Z])$ und $l([w,Z])$ wurden bereits gesetzt
3. $l([v,Z])$ oder $l([w,Z])$ ist l ,

startet MBFS in den beiden Knoten $[v,Z]$ und $[w,Z]$ parallel zwei Tiefensuchen (DFS), die jeweils die bereits untersuchten Kanten rückwärts verfolgen, indem sie deren Symmetriepartnern folgen. Dabei werden alle Knoten, $[u,X]$, deren Level noch nicht bestimmt wurde, in Level $l([v,Z]) + l([w,Z]) - l([u,\bar{X}]) + 1$ eingefügt. Dies entspricht der Länge eines alternierenden Weges, der von der Basis über $([v,Z],[w,\bar{Z}])$ bzw. $([v,Z],[w,\bar{Z}])$ zu $[u,X]$ führt, d.h. mittels eines "Umlaufs" um die Blüte konstruiert werden kann.

Um zu verhindern, dass die Suche über die Basis hinausläuft, wird das Fortschreiten der beiden Suchen so synchronisiert, dass sie gleichzeitig bei der Basis eintreffen müssen. Seien dazu die sogenannten Köpfe der Suchen, d.h. die jeweils aktuell betrachtete Knoten, o.B.d.A. mit K_l und K_r bezeichnet. Es wird nun erzwungen, dass die beiden Köpfe der Suche sich beim Vorstoß in eine neue Suchtiefe nur um 1 unterscheiden können. Wenn also ein Kopf auf ein niedrigeres Level vorstoßen soll, so muss der andere sich mindestens auf dem gleichen Level befinden.
Wenn also K_l von Level j nach Level $j-1$ wechseln soll, muss K_r auf einem Level $\leq j$ sein. Zur Vereinfachung der Überprüfung (vgl. Algorithmus 6_{70}) verlangen wir umgekehrt für K_r sogar, dass K_l auf einem Level $\leq j-1$ sein muss, bevor K_r nach $j-1$ wechselt, d.h. dass der rechte Kopf der Suche den linken Kopf nie überholen darf.

Algorithmus 6_{70} gibt das DDFS-Verfahren detailliert an. Führt man ihn auf einer einfachen Blüte aus, so laufen die beiden Köpfe von den Enden der Kante, die dem dem Stengel gegenüberliegt, aus gleichmäßig und abwechselnd bis zur Basis, wo sie sich treffen.

In komplexeren Blüten könnte ein Zusammentreffen der beiden Köpfe aber auch zufällig geschehen und ein weiterer Weg nach unten möglich sein (s. Abb. 4.5 Bild A): Werden K_l und K_r identisch, so wird ein Backtrack-Schritt bezüglich des rechten Kopfes K_r durchgeführt (Bild B) und K_l "bekommt" diesen Knoten. Sollte auf diese Weise kein neuer tieferer Knoten gefunden werden, d.h. K_r kehrt - mit gewissen Einschränkungen (s.u.) - zum Anfangspunkt seiner Suche zurück, wird für diesen Knoten ein Backtrack bezüglich des anderen Kopfes K_l durchgeführt. Dann setzt K_r seine Suche an dieser Stelle fort (Bild C). Einen Knoten, bei dem eine solche Übergabe von K_l auf K_r stattgefunden hat, bezeichnen wir als *Engstelle*.

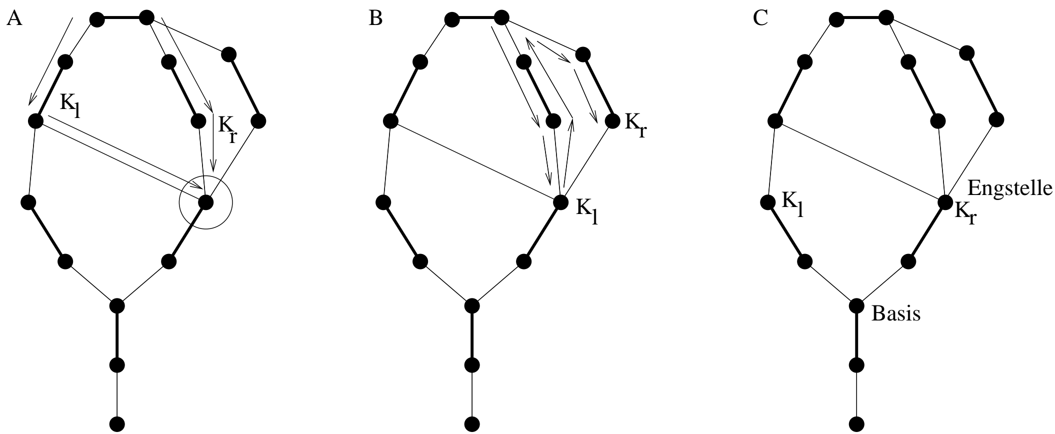

Abbildung 4.5: Vorzeitiger Zusammenstoß der beiden Köpfe der Rückwärtssuche

Bei den Engstellen tritt ein zusätzliches Problem auf: Die Knoten oberhalb der Engstelle wurden nur von K_l abgesucht und es wurde kein Weg weiter nach unten gefunden. K_r würde diese nochmals durchsuchen, da diese noch nicht als von K_R besucht markiert wurden(vgl. [40]). Um sicherzustellen, dass diese Knoten und Kanten nicht mehrfach besucht werden und um somit die Laufzeit innerhalb von $\mathcal{O}(m)$ zu halten, wird diese Engstelle in der Variablen "Sperre" vermerkt und das Backtracking wird nicht durch diesen Knoten hindurch fortgeführt. Abbildung 4.6 zeigt ein Beispiel hierfür: Nachdem der später als "Sperre" bezeichnete Knoten als umgehbar erkannt wurde und K_r diesen zugesprochen bekommen hat, würde K_r im zweiten Bild erneut von diesem aus einen Backtrack-Schritt versuchen. Alle oberhalb gelegenen Pfade (Kasten) können aber nicht zu einem Knoten mit einem niedrigeren Level führen als "Sperre", da ansonsten K_R schon einen anderen Weg gefunden hätte, während "Sperre" K_l zugesprochen war.
Beim linken Suchkopf kann dieses Problem nicht auftreten, da er jeden Knoten als erster zugeordnet bekommt.

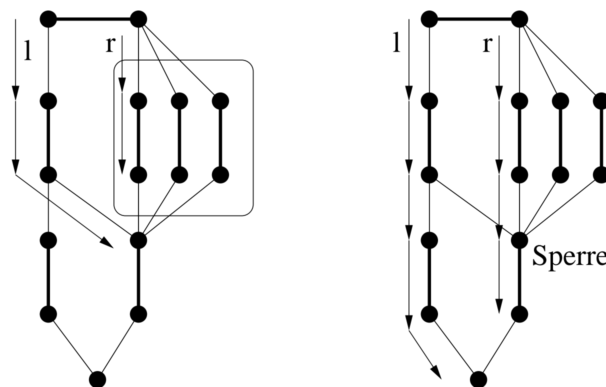

Abbildung 4.6: Blockade der Suche des rechten Kopfes durch eine "Sperre"

Die Rückwärtssuche folgt den Kanten - wie der Name schon sagt - rückwärts. Damit folgt sie gleichzeitig den Symmetriepartnern dieser Kanten vorwärts und fügt diese Partner in die entsprechenden Level ein. Die p-Zeiger für die Symmetriepartner können so eben-

falls bestimmt werden. Indem wir die p Zeiger von den Symmetriepartnern wieder auf die Knoten aus der Vorwärtssuche zurück übertragen, z.B. in der Form $\overline{p(\bar{K}_l)}$, können wir sie zugleich zur Verwaltung der Backtracking-Schritte der DDFS verwenden.
Wir markieren hierbei überquerte Kanten als "bereits bearbeitet", so dass sie bei der nächsten Vorwärtssuche nicht mehr berücksichtigt werden. Dadurch dass auch die zur Basis incidenten Kanten schon als bearbeitet markiert werden, wird verhindert, dass die Basis in einer späteren Vorwärtssuche über diese fälschlich erreicht werden kann.

4.2.2.3 Bemerkungen zur Korrektheit von MBFS

Da die Korrektheit von MBFS in [2] und die der DDFS Subroutine in [40] umfassend dargestellt werden, geben wir an dieser Stelle nur die Lemmata aus [1] wieder, wobei wir an den Stellen Beweisskizzen geben, die für die spätere Beschleunigung relevant sind.

Lemma 4.4 (Blum) *Sei eine nichtleere Teilmenge der Knoten $T \subset V$ gegeben, so dass für alle Knoten $[v, X]$ in dieser Menge ł$([u, X])$ definiert ist. Dann gilt:*
1. $|\text{DOM}(T)| = 1$
2. Sei $\text{DOM}(T) = [u, B]$ *Dann gilt nach der Definition von* ł$([u, A])$ *stets* $\text{DOM}(T) = \text{DOM}([u, B])$.

> **Beweisskizze:** 1. Der Schnitt mehrerer Pfade muss stets eine Teilmenge von einem (weil jedem) der Pfade sein. Da alle Knoten aus $DOM(T)$ auf einem kürzesten Pfad liegen müssen und somit die Level der B-Knoten nicht identisch sein können, erzwingt die Maximalität des Levels die Eindeutigkeit.
> 2. Gilt, da jeder streng einfache Pfad zu Knoten aus T durch $[u, B]$ führen muß. ◊

Lemma 4.5 (Blum)
Die folgenden Invarianten werden bei der Durchführung von MBFS stets eingehalten:
1. Unterfall 2 (vgl. S. 64) in der Vorwärtsmarkierung von MBFS tritt nie auf.
2. Für alle $[u, x], [u, \bar{X}] \in V_B$ mit $l([u, X]) < l([u, \bar{X}])$ gilt: Nach dem Ende des ersten Teils von Runde $l([u, X])$ wurde $l([u, X])$ definiert und alle Kanten, die auf streng kürzesten Wegen zu diesem Knoten liegen, wurden identifiziert. Das gleiche gilt für $l([u, \bar{X}])$ nach Ende des zweiten Teils von Runde l, wobei $l := \frac{1}{2}(l([u, A]) + l([u, B])) - 1$.
3. Wenn in Teil 2 der $l$ten Runde die Knoten $[v, Z]$ und $[w, Z]$ eine Rückwärtssuche auslösen, so galt $l([v, Z] = l([w, Z]) = l$.
4. Sämtliche Level werden unmittelbar korrekt gesetzt.

Invariante 1 garantiert, dass die Vorwärtssuche genauso wie ein Suchschritt des herkömmlichen BFS realisiert werden kann.

4.2.3 Der Übergang von MBFS nach MDFS

Wie wir im nächsten Kapitel sehen werden, findet MDFS in einem gegebenen Graphen eine nicht erweiterbare Menge disjunkter augmentierender Pfade. Abbildung 4.7[71] zeigt,

Algorithmus 6 Double Depth First Search nach [40] angepaßt an [2]

1: **for all** $([v,Z],[w,Z]) | ([v,Z],[w,\bar{Z}]) \in E_M$, $l([v,Z]), l([w,Z])$ sind definiert und eines der beiden ist l **do**
2: $K_l := [v,Z]$
3: $K_r := [w,Z]$
4: UNION(FIND(K_r),FIND(K_l))
5: Sperre:=$[w,Z]$
6: **while** not (l($\bar{K}_l$) $= 0$ or l($\bar{K}_R$) $= 0$) **do** ▷ sonst echter augmentierender Pfad. vgl. [40]
7: **while** (l($\bar{K}_l$) $\geq$ l($\bar{K}_r$)) **do** ▷ Lloop
8: markiere K_l als L und besucht
9: füge $\bar{K}_l$ in Level l($[v,Z]$)+l($[w,Z]$)−l(K_l) ein.
10: **for all** $(\bar{u}, \bar{K}_l) \in E_M$, die nicht besucht wurden **do**
11: markiere $(\bar{u}, \bar{K}_l)$ als besucht
12: u :=B(FIND(u)) ▷ bereits bestimmte Blüten überspringen
13: UNION(FIND(K_l), u)
14: **if** MinB(u) undefiniert **then** ▷ Blüten für MDFS speichern
15: MinB(u) :=FIND(K_l)
16: **end if**
17: **if** $u = K_r$ **then** ▷ K_l und K_r würden identisch
18: $K_r := \overline{p(\bar{K}_r)}$; $p(\bar{u}) := \bar{K}_l$; $K_l := u$
19: goto 30
20: **else if** u nicht besucht **then** ▷ normaler Suchschritt
21: $p(\bar{u}) := \bar{K}_l$
22: $K_l := u$
23: **end if**
24: **end for**
25: **if** $K_l = [v,Z]$ **then** ▷ K_L findet keinen Weg, Basis gefunden
26: B(FIND(K_r)) $:= K_r$
27: HALT
28: **end if**
29: **end while**
30: **while** l($\bar{K}_r$) $>$ l($\bar{K}_l$) **do** ▷ Rloop
31: markiere K_r als R und besucht
32: füge $\bar{K}_r$ in Level l($[v,Z]$)+l($[w,Z]$)−l(K_r) ein.
33: **for all** $(\bar{u}, \bar{K}_r) \in E_M$, die nicht besucht wurden **do**
34: markiere $(\bar{u}, \bar{K}_r)$ als besucht
35: u :=B(FIND(u)) ▷ Blüten überspringen
36: UNION(FIND(K_r), u)
37: **if** MinB(u) undefiniert **then**
38: MinB(u) :=FIND(K_r)
39: **end if**
40: **if** u noch nicht besucht **then** ▷ normaler Suchschritt
41: $p(\bar{u}) := \bar{K}_r$; $K_r := u$
42: goto 30
43: **end if**
44: **end for**
45: **if** $K_r \neq$ Sperre **then**
46: $K_r = \overline{p(\bar{K}_r)}$ ▷ Backtrack bezügl. K_r
47: **else**
48: Sperre:=K_l
49: $K_r := K_l$ ▷ Der rechte Kopf darf den Knoten verwenden
50: $K_l := \overline{p(\bar{K}_l)}$ ▷ Der linke Kopf sucht einen anderen Weg
51: goto 7
52: **end if**
53: **end while**
54: **end while**
55: **end for**

dass es aber nicht hinreichend ist, MDFS nur die Liste der Kanten zu übergeben, die auf einem kürzesten Pfad liegen, da MDFS in dem übergebenen Graphen irgendeinen strikt einfachen Weg bestimmen würde. Dies muss nicht notwendigerweise ein kürzester sein:

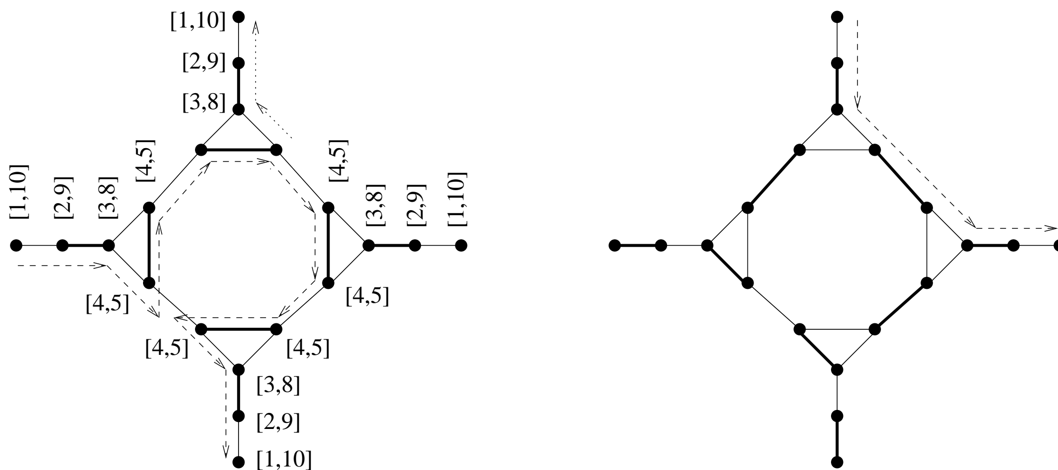

Abbildung 4.7: Ungültiger MDFS-Pfad

Statt den Pfeilen im linken Bild bis zum Ende zu folgen, hätte MDFS auf den durch die gepunkteten Pfeile markierten Pfad wechseln müssen. Dies führt sogar dazu, dass nach der anschließenden Augmentierung ein kürzerer augmentierender Pfad entsteht (Pfeile im rechten Bild). Dies zeigt, dass auf diese Weise der gesamte - an das Verfahren von Hopcroft-Karp angelehnte - Algorithmus zu Fall kommen würde.
Das Beispiel zeigt ebenfalls, daß das zusätzliche Übergeben der Level keine Abhilfe schafft, da der Algorithmus erkennen müßte, wieviele Knoten auf Level 4 besucht werden müssen.

Um diesem Problem zu begegnen, muss daher auch die gefundene Blütenstruktur an MDFS übergeben werden. Wir werden erst im folgenden Kapitel die Strukturen kennenlernen, die MDFS zum Verwalten von Blüten verwendet. Die genauen Details der Übergabe können also erst dort beschrieben werden.
Hier werden wir zuächst zeigen, dass diese Probleme nicht auftreten, wenn MDFS beim Erreichen einer von MBFS gefundenen Blüte die Suche an deren Basis fortgesetzt.
Zunächst kann aus Satz 4.3$_{66}$, der garantiert, dass alle kürzesten strikt einfachen Pfade MBFS-Blüten durch deren Basen betreten oder verlassen müssen, gefolgert werden, dass diese Übergabe keinen zulässigen strikt-einfachen Pfad blockiert.
Dass diese Vorgehensweise Pfade zu goßer Länge verhindert, zeigt

Lemma 4.6 *Enthält die von MBFS bestimmte Kantenmenge einen augmentierenden Pfad P_k, der länger ist, als ein kürzester augmentierender Pfad, so muss P_K eine identifizierte Blüte betreten und wieder verlassen.*

Beweis: Da MBFS nur Kanten aufnimmt, die auf einem kürzesten Pfad zu oder von einem Knoten liegen, muss P_k aus Teilstücken verschiedener

kürzester Pfade $P_1, \ldots, P_r$ bestehen. Dabei müssen zwei aufeinanderfolgende Teilstücke immer mindestens einen Knoten gemeinsam haben. Sei angenommen, dass ein kürzester Pfad die Länge h habe.
Es muss einen Knoten auf P_r geben, der ebenfalls zu einem anderen Pfad $P_i, i < r$ gehört, dessen streng einfacher Abstand zum Anfangspunkt von P_i größer ist als der zum Anfangspunkt von P_r. Ansonsten hätte auch P_k nur die Länge h. Sei v der erste derartige Knoten auf P_r. Könnte man nun P_r von seinem Anfangspunkt P_r^0 bis v folgen, dort auf P_i wechseln und diesen bis zu seinem Endpunkt P_i^h fortsetzen, so hätte man einen augmentierenden Pfad kürzerer Länge als h gefunden (vgl. Abb 4.8).

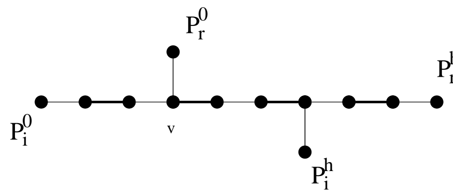

Abbildung 4.8: Hypothetischer längerer augmentierender Pfad

Da dies nicht sein kann, kann der Pfad $P_r|_{P_r^0,v} \circ P_i|_{v,P_i^h}$ nicht streng einfach sein. Nun müssen zwei Fälle unterschieden werden:

1. Der Pfad ist nicht einfach, d.h. es gibt einen gemeinsamen Knoten w von $P_r|_{P_r^0,v}$ und $P_i|_{v,P_i^h}$ mit $w \neq v$. In diesem Fall ist $P_r|_{P_r^0,w} \circ P_i|_{w,P_i^h}$ ein augmentierender Pfad[2], der noch kürzer ist als $P_r|_{P_r^0,v} \circ P_i|_{v,P_i^h}$ und das Argument kann iteriert (mit w als neuem v) angewandt werden. Die Iteration muss abbrechen, da P_r^0 und P_i^h nicht identisch sein können, da P_r^0 von der Form $[x, A]$, P_i^h aber von der Form $[y, B]$ ist.

2. Es gibt einen Knoten $[x, Z]$ auf $P_r|_{P_r^0,v}$, so dass $[x, \bar{Z}]$ auf $P_i|_{v,P_i^h}$ liegt. In diesem Fall liegt v in einer Blüte mit Basis $[x, B]$ und entweder P_i von P_i^h aus bis $[x, B]$ oder P_r von P_r^0 bis $[x, B]$ bildet den Stengel. Es gilt $\mathrm{l}([x, A]) \leq h$, da die gesamte Blüte einschließlich Stengel weniger als h Knoten umfaßt. Daher ist die Blüte von MBFS erkannt worden.

□

4.2.4 MDFS

4.2.4.1 Grundlegende Problemstellung

In diesem Kontext ist es die Aufgabe von MDFS, zu einem von MBFS konstruierten Graphen G^l, der nur aus den Kanten und Knoten besteht, die auf einem kürzesten augmentierenden Pfad liegen, eine nicht erweiterbare Menge disjunkter kürzester augmentierender

[2]Die Pfadstücke passen wegen der Unterteilung in A- und B-Knoten auch bezüglich des Alternierens zusammen.

Pfade zu finden. Die Veränderungen gegenüber herkömmlichem DFS sind dabei notwendig, um das vollständige Traversieren von Kreisen ungerader Länge zu verhindern. Es ergeben sich somit zwei Änderungen gegenüber DFS, wobei die zweite Änderung durch die erste erzwungen wird:

1. Ein Knoten $[v, A]$ darf nicht besucht werden, wenn sein Symmetriepartner $[v, B]$ auf dem aktuellen Pfad vom Startknoten s zum Knoten $[v, A]$, d.h. auf dem DFS-Stack, liegt.
2. Ein derartiger Knoten muss eventuell zu einem späteren Zeitpunkt betrachtet werden, zu dem er von DFS nicht mehr betrachtet worden wäre.

Um ansonsten den Prinzipien von DFS nahe zu kommen und die Notation zu vereinfachen, fügen wir außerdem zwei Knoten s und t hinzu, wobei wir s mit $[x, B]$ und $[x, A]$ mit t für alle freien Knoten x verbinden. Auf diese Weise entspricht die Suche nach einem alternierenden Pfad der Suche nach einem streng einfachen $s - t$ Pfad.

4.2.4.2 Suchphase

MDFS benutzt wie DFS einen Stack K, um den bisherigen Suchpfad zu speichern. Um später augmentierende Pfade rekonstruieren zu können, modifizieren wir dies jedoch so, dass ein POP nicht explizit ausgeführt wird: Wir ersetzen den Stack durch einen Baum.

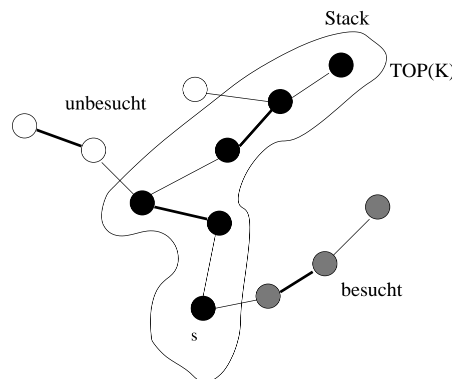

Abbildung 4.9: MDFS-Stack

Im Fall eines PUSHs erhält der aktuelle Knoten einen neuen Sohn und dieser wird der neue aktuelle Knoten. Im Fall eines POPs wird der Zeiger, der den aktuellen Knoten markiert, auf den Vater des aktuellen Knotens gerichtet. Den auf diese Weise entstehenden Baum werden wir “MDFS-Baum” nennen. Der Stack entspricht dann jeweils dem Pfad von der Wurzel zum aktuell betrachteten Knoten (vgl. Abb 4.9).
Treffen wir während der Suche erstmals auf einen Knoten u, so speichern wir die Kante zum davor besuchten Knoten v, also den Vaterknoten von u im DFS-Baum in $p(u)$.

Wir betrachten im Folgenden den wesentlichen Schritt von (M)DFS, in dem MDFS eine neue Kante e betrachtet, die vom obersten Element des Stacks $\text{TOP}(K) = [v, X]$ aus zu einem Knoten $[w, \bar{X}]$ führt. Wir unterscheiden die folgenden Fälle

1. $X = A \rightarrow$ PUSH($[w, \bar{X}]$). ▷ Matchingkante
2. $X = B$
   (a) $[w, A] \in K \rightarrow$ kein PUSH
   (b) $[w, A] \notin K$ aber $[w, B] \in K$
      i. $[w, A]$ war zuvor in $K \rightarrow$ kein PUSH ▷ wie DFS
      ii. $[w, A]$ war zuvor nicht in $K \rightarrow$ kein PUSH, speichern, von wo $[w, A]$ erreicht werden konnte
   (c) $[w, A] \notin K$ und $[w, B] \notin K$
      i. $[w, A]$ war zuvor in $K \rightarrow$ überprüfe, ob von $[w, A]$ aus erreichbare, aber nichtgepushte Knoten erreichbar sind
      ii. $[w, A]$ war zuvor nicht in $K \rightarrow$ PUSH($[w, A]$) ▷ wie DFS

In Fall $X = A$ kann $[w, B]$ noch nicht besucht worden sein, da jeder Pfad zu $[w, B]$ über $[v, A]$ führen muss. Umgekehrt muss jeder Pfad durch $[w, A]$ durch $[v, B]$ fortgesetzt werden. Alle Probleme bezüglich "nicht einfach" oder "nicht streng einfach" müssen also schon bei der Betrachtung von $[v, A]$ aufgetreten sein.
Die einzige Schwierigkeit im Fall $X = B$ ist die Realisierung von Unterfall 2(c)i. Hierzu nehmen wir an, dass es zu jedem Knoten v eine Liste L_v von Knoten gibt, die von diesem aus erreichbar sind, aber wegen Fall 2(b)ii nicht auf den Stack gelegt wurden. Wird ein Knoten v mit $L_v \neq \emptyset$ erreicht, so werden nach und nach - analog zu echten ausgehenden Kanten - die Knoten aus L_v auf den Stack gelegt und die Suche wird von diesen aus fortgesetzt.
Um einen derartigen Vorgang später nochmals erkennen und rekonstruieren zu können, wird der aktuelle Knoten im DFS-Baum durch einen sogenannten "erweiterten" Knoten ersetzt, in dem zusätzlich die Kante gespeichert wird, die diesen Schritt ausgelöst hat. Da im weiteren Verlauf Knoten, die vom Stack gelöscht wurden, im MDFS-Baum verbleiben, bleibt diese Information erhalten.
Zudem fügen wir in den MDFS-Baum eine Kante $([v, B], L_{[w,A]})$ ein. Diese künstlichen Kanten werden *erweiterbare* Kanten genannt.

Zur Vereinfachung der Bedingungen im Programmcode gibt es zusätzlich im Algorithmus von Blum [1] eine Menge L (ohne Index), die diejenigen Knoten $[v, X]$ enthält, deren $L_{[v,X]}$-Menge mindestens einmal nicht leer war.

4.2.4.3 Rekonstruktionsphase

Die Rekonstruktionsphase dient dazu, nach dem Finden eines streng einfachen $s-t$-Pfades diesen Pfad explizit aufzulisten. Dies ist bei allen Kanten, die auf einem DFS entsprechenden Weg in den MDFS-Baum aufgenommen wurden, kein Problem, da in der Liste p zu jedem Knoten die Kante zu demjenigen Vorgänger gespeichert wurde, von dem

aus er zuerst erreicht wurde.
Komplizierter gestaltet sich das Ganze an den Stellen, an denen ein Sprung nach Bedingung 2(c)i durchgeführt wurde. Die Daten zu diesem Sprung sind in dem entsprechenden "erweiterten" Knoten kodiert: Er enthält genau die Kante, von der der Sprung ursprünglich ausging.
Der konkrete Weg zurück zu dieser Kante muss noch gefunden werden. Dass dieser Sprung ausgeführt werden konnte, bedeutet aber, dass MDFS vorher schon einen streng einfachen Pfad gefunden hatte, der durch diesen Sprung lediglich abgekürzt wurde. Um diesen ursprünglichen Pfad zu finden, stehen die jeweiligen Vorgängerknoten wieder über die p-Zeiger zur Verfügung - nur wird jetzt statt nach einem $s-t$-Pfad nach einem Pfad zwischen den Knoten gesucht, die Anfangs- und Endpunkt des Sprunges waren. Wir können also die gleiche Suchstrategie rekursiv auf diesen Teilpfad anwenden.

4.2.4.4 Datenstrukturen

Die im wesentlichen verbleibende Aufgabe ist die Verwaltung der L-Listen der Knoten. Wie aus [1] hervorgeht, kann jede dieser Liste pro Knoten und Zeitpunkt nur einen Eintrag enthalten. Konkret müssen für diese Listen die folgenden Operationen realisiert werden:

1. Nach PUSH($[u, A]$): $L_{[w,A]} := \emptyset$, falls $L_{[w,A]} = [u, A]$
2. Nach POP($[u, B]$): $L_{[w,A]} := [u, A]$, falls push($[u, A]$) nie ausgeführt wurde und MDFS einen Pfad von $[w, A]$ nach $[u, A]$ gefunden hat, der $[u, B]$ nicht enthält.

MDFS muss zum Ausführen der PUSH-Schritte alle Knoten finden, die Eigenschaft 2 erfüllen. Um dies effizient durchzuführen, müssen Knotenmengen, die bereits bearbeitet wurden, gezielt übersprungen werden.
Hierzu werden wir die L-Werte und die "erweiterbaren Kanten" ("extensible edges" [1]) verwenden.

Zur Verwaltung der Mengen mit gleichen L-Werten führen wir die Mengen $D_{[q,A]}$ ein, wobei $[q, A]$ der gemeinsame L-Wert aller Knoten aus $D_{[q,A]}$ ist.
Wird einer der Knoten aus $D_{[q,A]}$ über einen Pfad erreicht, der $[q, B]$ nicht enthält, so kann $[q, A]$ auf den Stack gelegt werden (Fall 2(c)i von MDFS). In diesem Fall wird die Menge $D_{[q,A]}$ eigentlich nicht mehr benötigt.
Ist aber $L_{[q,A]}$ dann wiederum selbst nicht leer, enthält also einen Knoten $[w, A]$ so kann dieser natürlich auch von allen Knoten aus $D_{[q,A]}$ erreicht werden. Es ist also sinnvoll $D_{[q,A]}$ zu erhalten, so dass man sie $D_{[w,A]}$ hinzufügen kann. Daher verwalten wir diese Mengen wie bei MBFS als UNION-FIND Struktur: Wenn beim Identifizieren der zu einer D-Menge gehörigen Knoten eine andere D-Menge angetroffen wird, werden die beiden Mengen vereinigt.

Diese Eigenschaft ermöglicht es zudem, die L Mengen der Knoten effizient zu verwalten. Da die L-Mengen der Knoten einer D-Menge identisch sind, verwaltet man stets für die gesamte D-Menge nur eine einzelne L-Menge. Wir behandeln also die L-Mengen nicht länger als eigenständige Mengen, sondern als Funktion, die zu einem gegebenen Knoten

den entsprechenden Knoten liefert, der in der zugehörigen L-Menge wäre. Diese Funktion wird durch die folgende Definition aus [1] beschrieben:

1. Gegeben $[p, A]$, bestimme mittels FIND $[q, A]$, so dass $[p, A] \in D_{[q,A]}$ und $D_{[q,A]}$ die größte solche Menge ist.

2. Existiert $[q, A]$ nicht, so ist $L_{[p,A]} = \emptyset$.

3. $L_{[p,A]} = \begin{cases} [q, A] & \text{falls PUSH}([q, A])\text{nie ausgeführt wurde} \\ \emptyset & \text{sonst} \end{cases}$

Es ist daher nicht mehr notwendig $L_{[p,A]}$ explizit zu leeren, wenn der enthaltene Knoten auf den Stack gelegt wurde.

Zum konkreten Auffinden der Knoten, die zu einer D-Menge gehören, benutzt MDFS zwei weitere Datenstrukturen:

- Die Mengen $R_{[w,A]}, [w, A] \in G_B$, enthalten zu jedem Knoten, der wegen Fall 2(b)ii$_{74}$ noch nicht auf den Stack gelegt wurde, die Nachbarn, von denen aus er erreicht wurde.

- Die Mengen $E_{[w,A]}$ enthalten alle Nachbarn, von denen aus $[w, A]$ bereits im Rahmen der laufenden Suche erreicht wurde und die nicht zu $R_{[w,A]}$ gehören.

Wird nun ein Knoten $[w, B]$ vom Stack genommen, dessen Symmetriepartner $[w, A]$ noch nicht auf den Stack gelegt wurde, so werden von diesem aus zunächst alle Knoten aus $R_{[w,A]}$ besucht. Danach wird die Suche in der Art einer Breitensuche über Kanten aus $E_{[w,A]}$ fortgesetzt.

An diesem Punkt stellen wir nun den Bezug zur Blütenstruktur in G wieder her, da diese uns später eine anschaulichere Darstellung der Anwendung von Graphenkompression erlauben wird. Der MDFS-Algorithmus selbst betrachtet diese nicht explizit.

Proposition 4.7 *Die Knoten u mit $L_{[u,A]} = [v, A]$ gehören zu einer einfachen Blüte mit Basis v.*

Beweis: Nach Definition von $L_{[u,A]}$ hat MDFS einen einfachen Pfad Q von $[u, A]$ nach $[v, A]$ gefunden, der $[v, B]$ nicht enthält.
Da $[v, A]$ nicht auf den Stack gelegt werden konnte, muss $[v, B]$ zum Zeitpunkt, als $[u, A]$ über Q erreicht wurde, auf dem Stack gelegen haben. Da $[v, B]$ nicht zu Q gehört, muss $[v, B]$ ein Vorgänger von $[u, A]$ gewesen sein.
Wir bezeichnen den Teilpfad des MDFS-Baums von $[v, B]$ zu $[u, A]$ mit P.
Dann ist $P \circ Q$ ein Pfad von $[v, B]$ nach $[v, A]$ und somit ein Kreis ungerader Länge.
Da wir die betrachteten Knoten von einem freien Knoten aus erreicht haben, muss es einen Stengel geben. □

Bemerkung 4.8 *Die Symmetriepartner der Knoten aus $P \circ Q$ bilden ebenfalls einen Pfad von $[v, B]$ nach $[v, A]$ (s. z.B. Abb. 4.10).*

Bei der Rückwärtssuche nach Knoten für die aktuelle D-Menge müssen “Unterblüten”, also Knoten, die bereits zu einer D-Menge gehören, erneut übersprungen werden.
Trifft man dabei von außen, d.h. nicht über die Basis auf eine Unterblüte, kann der Sprung genauso durchgeführt werden, wie dies im Rahmen der Vorwärtssuche geschieht. Trifft man auf einen Knoten $[p, A]$, der zu L gehört, dessen L-Menge also schon einmal nicht leer war, so bestimmt man mittels FIND die Mange $D_{[r,A]}$ mit $[p, A] \in D_{[r,A]}$ und setzt die Suche in $[r, A]$ fort (vgl. [1]).
Dies wird durch Abb. 4.10 und 4.11[78] veranschaulicht: Abb. 4.10 zeigt dabei zunächst die Vorwärtssuche durch die äußere Blüte, um die Reihenfolge der Operationen zu verdeutlichen. Wir nehmen an, dass die Suche zunächst dem rechten Ast der äußeren Blüte folgt (Bild 1). Dabei erreicht sie zunächst die innere Blüte. Wir nehmen an, dass sie diese erst rechts- und dann linksherum durchläuft. Beide Male kann dabei der Knoten $[p, A]$ nicht auf den Stack gelegt werden, da $[p, B]$ auf dem Stapel liegt. Also verläßt die Suche die innere Blüte entlang der äußeren Blüte bis zur Basis der äußeren Blüte, wo der Knoten $[q, A]$ nicht auf den Stapel gelegt werden kann.
Dann kehrt die Suche zurück und $[p, B]$ wird vom Stapel genommen.
Nun findet eine Rückwärtssuche bezüglich der inneren Blüte statt. Entsprechend werden die Knoten $[x, A] \in D_{[p,A]}$ mit $L_{[x,A]} = [p, A]$ identifiziert (Bild 2). Danach folgt die Suche dem linken Ast der äußeren Blüte und erreicht somit die innere Blüte von außen. Für diese Knoten gilt $L_{[x,A]} = [p, A]$. Daher wird die Suche mit einem Sprung zu $[p, A]$ fortgesetzt (Bild 3).
Wieder kann der Knoten $[q, A]$ nicht auf den Stapel gelegt werden. Die Suche kehrt schließlich ganz zurück und der Knoten $[q, B]$ verlässt den Stapel.

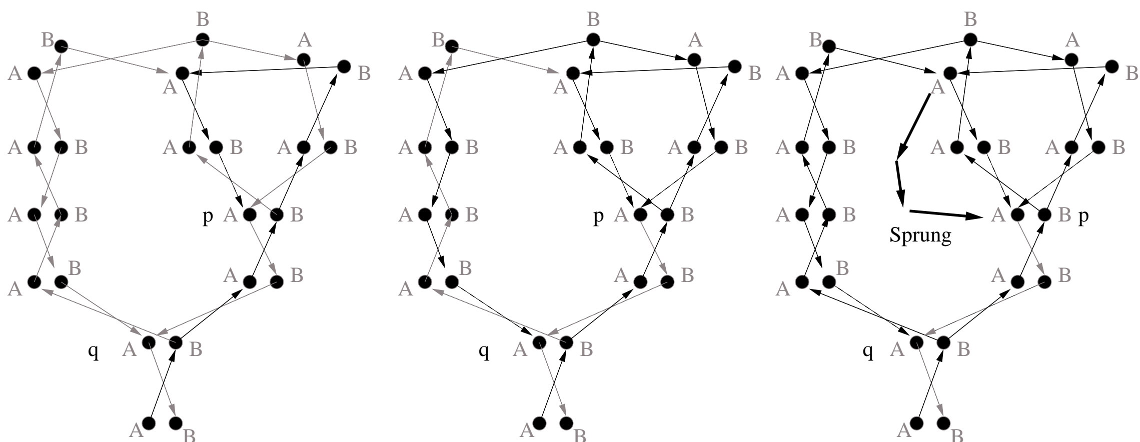

Abbildung 4.10: Vorwärtssuche durch eine Unterblüte

Nun findet die Rückwärtssuche zum Identifizieren der Knoten mit $L_{[x,A]} = [q, A]$ statt (Abb. 4.11) . Wir nehmen an, dass wir dabei zunächst dem linken Ast rückwärts folgen. Wir erreichen die innere Blüte wieder von außen und können mit einem Sprung zum Knoten $[p, A]$ gelangen. Die Menge $D_{[p,A]}$ wird zur Menge $D_{[q,A]}$ hinzugefügt (Bild 2).

Nun erfolgt die Rückwärtssuche durch den rechten Ast. In dieser Richtung können wir die innere Blüte entlang der erweiterbaren Kanten, die in Fall 2(c)i[74] eingefügt wurden, überspringen (Bild 3).

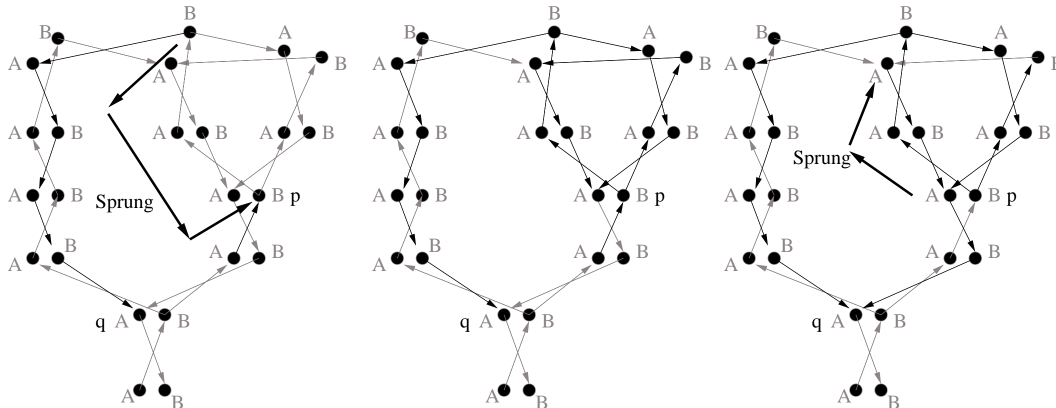

Abbildung 4.11: Rückwärtssuche durch eine Unterblüte

Dabei müssen aber nicht nur die Kanten berücksichtigt werden, für die dieser Sprung tatsächlich vollzogen wurde, sondern auch solche, von denen aus er nicht vollzogen wurde, da L bereits geleert wurde. Ein beispiel zeigt Abbildung 4.12:

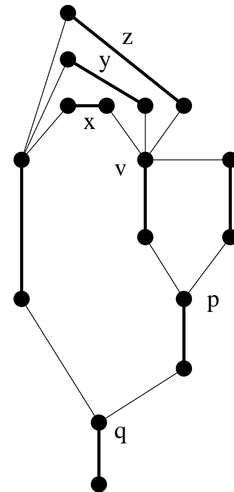

Abbildung 4.12: Unterblüte mit mehreren eingehenden Kanten

Wenn wir annehmen, dass hier wieder der rechte Ast der äußeren Blüte zuerst besucht wurde, so wird die Unterblüte übersprungen, sobald die Suche später dem linken Ast folgt. Nur beim ersten Erreichen von $[v, A]$ o.B.d.A. über den Teilpfad x ist $L_{[v,A]} \neq \emptyset$. Würden wir also nur der hierbei erzeugten erweiterbaren Kante rückwärts folgen, so würden die Knoten auf den Teilpfaden y und z nicht in $D_{[q,A]}$ eingefügt.

Um diese Probleme zu lösen, führen wir eine Funktion $L'_{[v,X]}$ ein, die stets den letzten Wert von $L_{[v,X]}$ zurückliefert, der ungleich $\emptyset$ war. Wir fügen die in [1] beschriebenen erweiterbaren Kanten (vgl. Abb. 4.13) nicht nur für Knoten mit nichtleeren L-Werten,

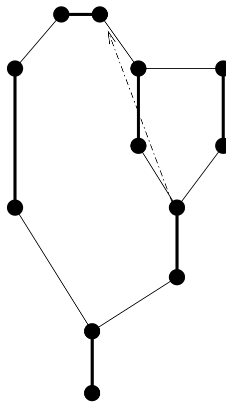

Abbildung 4.13: "erweiterbare" Kante

sondern auch für solche mit nichtleeren L'-Werten ein.
Die L'_u-Funktion und die "erweiterbaren" Kanten sind eine Alternative zum tatsächlichen Schrumpfen der Blüten. Mittels der L_u-Funktion springt man beim Erreichen einer Blüte von außen zu deren Basis und fährt dort mit der Suche fort. Bei Erreichen der Blüte von der Basis aus kann man mittels der erweiterbaren Kanten alle Knoten erreichen, von denen aus (bekannte) Kanten die Blüte erreichen. Damit verhält sich die Blüte wie ein einzelner Knoten.

An dieser Stelle können wir nun darauf eingehen, wie die von MBFS gefundenen Blüten behandelt werden. In MBFS wurde zu jedem Knoten v die Basis MinB(v) der kleinsten Blüte, die diesen Knoten enthalthält, identifiziert. Zusätzlich wurde in den p-Zeigern analog zu denen in MBFS ein Pfad zur Basis dieser Blüte gespeichert.
Wir behandeln diese Blüten nun auf die selbe Weise, wie von MDFS selbst gefundene D-Mengen. Erreichen wir von $[w, B]$ aus einen Knoten $[v, A]$, so betrachten wir $[\text{MinB}(v), A]$ und fügen, wenn $[\text{MinB}(v), B]$ nicht auf dem Stapel liegt, eine erweiterbare Kante $([w, B], [\text{MinB}(v), A])$ ein.
War $[\text{MinB}(v), A]$ noch nie auf dem Stapel und liegt $[\text{MinB}(v), B]$ nicht auf dem Stapel, so setzen wir die Suche unmittelbar in $[\text{MinB}(v), A]$ fort und überprüfen entsprechend auch MinB$([\text{MinB}(v), A])$.

Proposition 4.9 *Durch diese Modifikation wird ein Verhalten gemäß Lemma 4.6*[71] *bewirkt, ohne die Korrektheit des Algorithmus zu beeinträchtigen.*

Beweis: Gemäß Lemma 4.6[71] muss eine von MBFS gefundene Blüte, die nicht über ihre Basis betreten wird, über ihre Basis verlassen werden. Entsprechend werden solche Blüten erst gar nicht durchsucht, sondern die Suche wird unmittelbar an der Basis fortgesetzt. Sind hierbei mehrere Blüten ineinander verschachtelt, so zeigt der MinB-Zeiger der Basis der jeweils inneren Blüte auf die Basis der äußeren Blüte, so dass die Basis der größten von außen betretenen Blüte rekursiv erreicht wird.
Später kann dann der tatsächliche Pfad zum Erreichen der Basis aus den von

MBFS übergebenen p-Zeigern rekonstruiert werden.
Wird die Blüte über die Basis v betreten, so befindet sich $[v, B]$ auf dem Stapel und entsprechend findet kein Sprung statt. □

Bemerkung 4.10 *Wir verwenden "MinB", also Zeiger auf die kleinsten Blüten, die MBFS gefunden hat, damit MDFS keine eingebetten Blüten in MBFS-Blüten rekonstruieren muss. Prinzipell wäre es aber auch möglich, einen direkten Zeiger auf die Basis der jeweils größten Blüte zu übergeben.*

Proposition 4.11 *Die zusätzlichen Operationen zur Verwaltung der MBFS-Blüten lassen sich in $\mathcal{O}(m)$ realisieren.*

Beweis: Die Sprünge, die durch MBFS-Blüten ausgelöst werden, unterscheiden sich nicht von denen, die durch von MDFS selbst gefundene Blüten bewirkt werden. Die Kosten für die Betrachtung der "erweiterbaren" Kanten bei der Rückwärtssuche können jeweils der Kante zugeordnet werden, durch die erweiterbare Kante erzeugt wurde. Da jede Kante nur einmal die Erzeugung einer erweiterbaren Kante auslösen kann, kann sich der Aufwand maximal verdoppeln. Erweiterbare Kanten selbst können keine weiteren erweiterbaren Kanten erzeugen, da sie nur bei der Rückwärtssuche traversiert und nur bei der Vorwärtssuche erzeugt werden. □

Wir geben im Folgenden den Pseudocode aus [1] wieder (Algorithmen 7ff), und hoffen, dass er anhand der gerade gegebenen Übersicht auch in dieser Kürze verständlich ist. Für eine ausführliche Besprechung, sowie für den Beweis, dass MDFS selbst in $\mathcal{O}(m)$ durchgeführt werden kann, verweisen wir auf die Arbeit von Blum [1].

4.2.4.5 Bemerkungen zur Korrektheit von MDFS

In Kapitel 4.2.3$_{69}$ wurde bereits gezeigt, dass jeder streng einfache Pfad, den MDFS findet, zwangsläufig ein kürzester augmentierender Pfad ist. Es verbleibt also zu zeigen, dass MDFS nur solche Pfade findet und auf der anderen Seite die Menge der identifizierten Pfad maximal ist. Wir geben hier wieder nur einen kurzen Überblick und verweisen im Wesentlichen auf [1].

Lemma 4.12 (Blum 1999) *Solange MDFS nur streng einfache Pfade konstruiert, gilt: Nach der Operation* PUSH$([v, A])$*, wobei v zum Matching gehört, folgt stets die Opertaion* PUSH$([w, B])$*, wobei $([v, A], [w, B])$ die zugehörige Matchingkante ist, und diese Operation erzeugt einen streng einfachen Pfad.*

Das folgende Lemma zeigt, dass MDFS diejenigen Knoten, die durch "normale" Tiefensuche erreichbar sind, korrekt behandelt.

Algorithmus 7 MDFS nach [1]

1: **procedure** SEARCH
2: **if** TOP$(K) = t$ **then**
3: Rekonstruiere den gefundenen s-t-Pfad
4: **else**
5: markiere TOP(K) als "gepushed"
6: **for all** $[w, Y] \in \Gamma$(TOP(K)) **do**
7: **if** $Y = B$ **then**
8: PUSH$([w, B])$
9: SEARCH
10: **else**
11: **if** $[w, A] \in K$ **then**
12: $E_{[w,A]} := E_{[w,A]} \cup \{$TOP(K)$\}$
13: **else**
14: **if** $[w, B] \in K$ **then**
15: **if** $[w, A]$ ist markiert als "pushed" **then**
16: $E_{[w,A]} := E_{[w,A]} \cup \{$TOP(K)$\}$
17: **else**
18: $R_{[w,A]} := R_{[w,A]} \cup \{$TOP(K)$\}$
19: **end if**
20: **else**
21: **if** $[w, A]$ ist als "gepushed" markiert **then**
22: **if** $L_{[w,A]} \neq \emptyset$ **then**
23: Ersetzte TOP(K) durch (TOP$(K), [w, A]$), TOP(K)
24: $E_{L_{[w,A]}} := E_{L_{[w,A]}} \cup$ TOP(K) ▷ erweiterbare Kante
25: PUSH$(L_{[w,A]})$
26: SEARCH
27: **else**
28: **if** $L'_{[w,A]} = \emptyset$ **then**
29: $E_{[w,A]} := E_{[w,A]} \cup \{$TOP(K)$\}$
30: **else**
31: $E_{L'_{[w,A]}} := E_{L'_{[w,A]}} \cup$ TOP(K) ▷ erweiterbare Kante
32: **end if**
33: **end if**
34: **else**
35: **if** $[B($MinB$([w, A])), B] \notin K$ **then** ▷ MBFS Blüten überspringen
36: Ersetzte TOP(K) durch $(TOP(K), [w, A])$, TOP(K)
37: $E_{[B(\text{MinB}([w,A])),A]} := E_{[B(\text{MinB}([w,A])),A]} \cup \{[w, A]\}$
38: PUSH$([B($MinB$([w, A])), A])$
39: **else**
40: PUSH$([w, A])$
41: **end if**
42: SEARCH
43: **end if**
44: **end if**
45: **end if**
46: **end if**
47: **end for**
48: $[v, X] :=$ TOP(K)
49: BLFIND$([v, X])$
50: POP
51: **end if**
52: **end procedure**

Algorithmus 8 Routine zum Bestimmen der Level in Blüten [1]

```
 1: procedure BLFIND([v, X])
 2:     if X = B und [v, A] ist nicht als “gepushed” markiert then
 3:         L_act := [v, A]
 4:         D_{L_act} := ∅
 5:         L_def := ∅
 6:         for all [q, B] ∈ R_[v,A] do
 7:             CONSTRL(([q, B], [v, a]), [v, B])
 8:         end for
 9:         while L_def ≠ ∅ do
10:             Wähle [k, A] ∈ L_def
11:             L_def := L_def \ {[k, A]}
12:             for all [q, B] ∈ E_[k,A] do
13:                 CONSTRL(([q, B], [k, A]), [v, B])
14:             end for
15:         end while
16:     end if
17: end procedure
```

Algorithmus 9 Subroutine CONSTRL nach [1]

```
 1: procedure CONSTRL(([q, B], [u, A]), [x, B])
 2:     P_act := ([q, B], [u, A])
 3:     [z, B] := [q, B]
 4:     while [x, B] wurde nicht erreicht do
 5:         for all [y, A] auf dem Pfad rückwärts von [z, B] nach L ∪ {[x, B]} do
 6:             D_{L_act} := D_{L_act} ∪ {[y, A]}
 7:             L := L ∪ {[y, A]}
 8:             P_[y,A] := P_act
 9:             L_def := L_def ∪ {[y, A]}
10:         end for
11:         if [y, A] ∈ L then                                ▷ Sei [y, A] in D_[r,A]
12:             D_{L_act} := D_{L_act} ∪ D_[r,A]
13:             [z, B] := [r, B]
14:         end if
15:     end while
16: end procedure
```

Algorithmus 10 Algorithmus zur Rekonstruktion des $s - t$ Pfades nach [1]

```
 1: procedure RECONSTRPATH(t, s)
 2:     ACTNODE:= t
 3:     while ACTNODE ≠ s do
 4:         if p(ACTNODE) ist nicht expandiert then
 5:             ACTNODE:= p(ACTNODE)     ▷ Bei MBFS- Blüten wird hier das p aus
                    MBFS verwandt.
 6:         else                          ▷ p(ACTNODE) sei (([v, B], [w, A]), ([v, B]))
 7:             RECONSTRQ(ACTNODE, [w, A])
 8:             ACTNODE:=[v, B]
 9:         end if
10:     end while
11: end procedure
```

Algorithmus 11 Subroutine RECONSTRQ [1]

```
procedure RECONSTRQ([u, A], [w, A])
    ANF := [w, A]
    RECONSTRPATH (P^1_ANF, ANF)                 ▷ p(ANF) sei ([P^1_ANF, P^2_ANF])
    while P^2_ANF ≠ [u, A] do
        ANF := P^2_ANF
        RECONSTRPATH(P^1_ANF, ANF)
    end while
end procedure
```

Lemma 4.13 (Blum 1999) *Man betrachte den Zeitpunkt, zu dem ein Knoten $[u, B]$ vom Stack genommen wird und nehme an, dass MDFS bis dahin nur streng einfache Pfade konstruiert hat. Sei des weiteren angenommen, dass es einen Knoten $[x, A]$ gibt, der von $[u, B]$ aus erreicht werden kann, ohne einen Knoten auf dem Stack oder dessen Symmetriepartner zu verwenden. Dann wurde* PUSH$([x, A])$ *vor* POP$([u, B])$ *durchgeführt.*

Lemma 4.14 (Blum 1999) *Man betrachte erneut den Augenblick, in dem ein Knoten $[u, B]$ vom Stack genommen wird, und nehme an, dass MDFS bis zu diesem Zeitpunkt nur streng einfache Pfade gefunden hat. Sei $P = [v, A], Q[w, B]$ ein streng einfacher Pfad, der zum Zeitpunkt von* PUSH$([u, B])$ *zum Stack streng disjunkt war. Wenn die Kanten $([u, B], [v, A])$ und $([w, B], [u, A])$ zum Graphen gehören, dann wurden sämtliche Knoten auf P sowie ihre Symmetriepartner auf den Stack gelegt, bevor $[u, B]$ von diesem genommen wurde.*

Satz 4.15 (Blum 1999) *Während der Ausführung von MDFS sind stets die folgenden Invarianten erfüllt:*

1. MDFS konstruiert nur streng einfache Pfade.
2. $|L_{[w,A]}| \leq 1 \quad \forall [w, A] \in V'$
3. Angenommen der Algorithmus führt die Zuordnung $L_{[w,A]} := [u, A]$ durch, so ist nach einem PUSH$([u, A])$ stets $L_{[w,A]} = L_{[u,A]}$.

Satz 4.16 (Blum)

1. *MDFS konstruiert einen streng einfachen Pfad von s nach t, wenn ein solcher existiert.*
2. *MDFS konstruiert nur streng einfache Pfade.*

Satz 4.17 *MDFS findet im von MBFS erstellten Graphen und unter Berücksichtigung der übergebenen Blüten nur kürzeste Pfade.*

> **Beweis:** Dies folgt aus Lemma 4.6[71] und Proposition 4.9[79] □

Zusammen mit Invariante 1 ist somit gesichert, dass MDFS nur kürzeste streng einfache Pfade identifiziert.

Satz 4.18 *Die Menge der durch wiederholte Anwendung von MDFS identifizierten kürzesten Pfade ist nicht erweiterbar.*

> **Beweis:** Wenn es im verbliebenen Graphen einen Pfad gibt, so findet MDFS diesen gemäß Satz 4.16. Es ist also nur zu zeigen, dass das Entfernen bereits besuchter Knoten keinen Pfad zerstört.
> Dies kann bei Knoten, die den Stack verlassen haben, nicht geschehen, da

ein durch diese Knoten führender Pfad bereits vorher gefunden worden wäre. Knoten auf dem Stack gehören zum Pfad selbst. Ansonsten werden Knoten nur gelöscht, wenn sie zu einer Blüte gehören, deren Basis auf dem augmentierenden Pfad lag. Da die Blüte identifiziert wurde, wurde jeder weitere mögliche Pfad wegen der L-Zeiger durch deren Basis geleitet. Da die Basis gelöscht wurde, kann es keinen weiteren Pfad durch diese Knoten geben. □

4.3 Algorithmische Suche in Cliquen und Sternen

4.3.1 Einleitung

Der zeitaufwändigste Teil der hier betrachteten Matching-Algorithmen ist die Suche nach alternierenden Pfaden. Dies geschieht mittels MBFS und MDFS. Es handelt sich also stets um eine Variante der sogenannten algorithmischen Suche in Graphen (vgl. z.B. [3]), d.h. ausgehend von einer Menge bereits betrachteter Knoten wird versucht, über eine Kante zu einem "neuen" Knoten zu gelangen.
Die algorithmische Suche erzeugt knotendisjunkte Pfade, also Strukturen in der Größenordnung von n, benötigt dafür aber m Schritte. Dies liegt daran, dass alle Kanten des Graphen betrachtet werden, da man nicht weiß, welche der Kanten zu bisher noch nicht untersuchten Knoten führen. Sinnvoll wäre es also, ausgehend vom momentan betrachteten Knoten nur solche Nachbarn zu besuchen, die noch nicht betrachtet wurden.

Im Fall einer bipartiten Clique der Größe $r \times r$ kann jeder Knoten bis zu $r-1$ mal "sinnlos" besucht werden, da dieser Knoten nach dem ersten (sinnvollen) Besuch nochmals von allen anderen $r-1$ Nachbarn aus aufgesucht wird.
Da wir Bicliquen stets in der selben Richtung durchqueren - die Suche nach freien Kanten verläuft stets von B nach A -, werden wir im folgenden von der *Anfrageseite* und der *Antwortseite* sprechen. Dabei bezeichnet die Anfrageseite die Menge der Knoten, von denen aus man andere Knoten erreichen will, und die Antwortseite die erreichten Knoten. In den folgenden Darstellungen ist die "linke" Seite einer Clique die Anfrageseite und die "rechte" Seite die Antwortseite. Später kann sich diese Wahl in Abhängigkeit von den jeweils betrachteten Algorithmen ändern.

Ersetzt man die Kanten der Clique wiederum durch einen "Stern" (vgl. Abb. 2.1[19] und Abb. 4.1[60]), so sind nach wie vor alle Knoten der Antwortseite von allen Knoten der Anfrageseite aus erreichbar. Es wird nun aber nur noch eine "sinnlose" Aktion pro Anfrageknoten vorgenommen: Der Test, dass der "Zentralknoten" bereits abgearbeitet wurde. Somit kann man die unnötigen erneuten Zugriffe vermeiden.
Wir haben in Bemerkung 2.4[20] gesehen, dass wir im Kontext alternierender Pfade diese Ersetzung nicht vornehmen können, da immer nur ein alternierender Pfad durch einen solchen Stern führen kann. Außerdem können wir den Kompressionsalgorithmus nicht nach jeder Augmentierung neu aufrufen, so dass wir auf Veränderungen der Kantenrichtungen keine Rücksicht nehmen können.

Wir wollen daher versuchen, die komprimierte Version des Graphen als zusätzliche In-

formationsquelle zu nutzen, während wir mit dem ursprünglichen Graphen arbeiten. Wir schauen während der Phasen algorithmischer Suche gewissermaßen kurz auf den komprimierten Graphen, um einen “Tipp” zu bekommen, wo wir die Suche am günstigsten fortsetzen können.
Hierzu ist es nicht notwendig, tatsächlich Sterne für Cliquen einzusetzen. Es genügt, dass wir die Cliquen kennen.

Wir fassen alle Knoten auf jeweils einer Seite einer Clique zu einer sogenannten *Nachbarschaftsliste* zusammen. Zu jedem Knoten v speichern wir dann die inzidenten Kanten, die nicht Teil irgendeiner Clique sind, sowie Zeiger auf alle Nachbarschaftslisten von Cliquen, zu deren Knoten v inzident ist. Abstrakt gesehen kann man die Nachbarschaftslisten mit den Zentralknoten der Sterne identifizieren (vgl. Abb. 4.14).

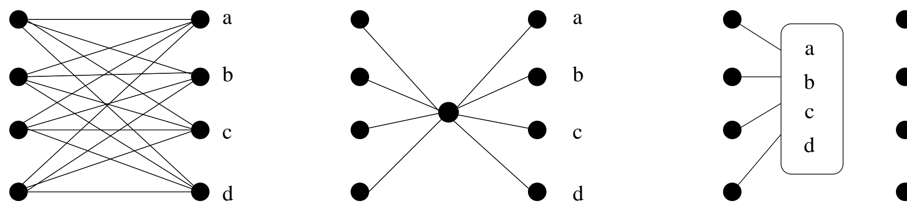

Abbildung 4.14: Nachbarschaftsliste als Ersatz für den zentralen Knoten eines Sterns

4.3.2 Anwendung im bipartiten Fall

Zunächst zeigen wir nun, dass man im bipartiten Fall die Nachbarschaftslisten zum Simulieren der zentralen Knoten verwenden kann. Auf diese Weise ist auf bipartiten Graphen Kompression auch im Kontext alternierender Pfade einsetzbar. Danach gehen wir darauf ein, wie dieses Konzept flexibler gestaltet werden kann, so dass später eine Anwendung im nichtbipartiten Fall möglich wird.

Um diese spätere Flexibilisierung zu ermöglichen, kapseln wir den Mechanismus, der diese “Tipps” liefert, in einer abstrakten Datenstruktur, die wir als *Suchstruktur* bezeichnen. Auf eine Suchstruktur wird mittels der drei folgenden Operationen zugegriffen:

1. SUGGEST(v): Liefert einen möglicherweise unbesuchten Nachbarn von v. Jeder zulässige (s.u.) unbesuchte Knoten wird höchstens $\delta(v)-1$ mal hintereinander nicht vorgeschlagen.

2. DEACTIVATE(v): Sorgt dafür, dass Knoten v nicht bei SUGGEST zurückgeliefert wird.

3. ACTIVATE(v): Erlaubt das Vorschlagen von v.

Immer wenn wir versuchen einen neuen Knoten zu finden, können wir also mittels SUGGEST nach einem Vorschlag fragen. Wenn wir wissen, dass uns ein Knoten momentan

nicht interessiert, können wir der Suchstruktur mit DEACTIVATE mitteilen, dass dieser nicht vorgeschlagen werden soll, und dies ggf. mit ACTIVATE wieder rückgänig machen.

Nun stellen wir ein erstes Modell vor, wie diese Datenstruktur "von innen" aussehen könnte:
Im bipartiten Fall bilden wir mit den Suchstrukturen die Ersetzung durch "Sterne" nach, so wie sie im Algorithmus von Feder und Motwani [11] beschrieben wird. Jede Biclique wird durch eine eigene Nachbarschaftsliste repräsentiert. Diese enthält alle Knoten, die zur Antwortseite der Clique gehören. Die Liste selbst entspricht also dem Zentralknoten des Sterns und die Einträge entsprechen den Kanten.
Wird SUGGEST(v) aufgerufen, so liefert es nach und nach die Knoten aus allen Nachbarschaftslisten von v und dann die zu v adjazenten Knoten u, für die (v, u) zu keiner Clique gehört. Dabei wird jeder besuchte Knoten aus der entsprechenden Nachbarschaftsliste entfernt. Ein Besuch von den anderen Knoten der jeweiligen Anfrageseite aus ist nicht mehr sinnvoll.
Dies entspricht dem "als besucht markieren" der Kante vom Zentralknoten des Sterns zu dem entsprechenden Knoten bei der Ersetzung durch Sterne. Das linke Bild von Abb. 4.15 zeigt die "herkömmliche" Suchstrategie. Nur die Kante vom ersten Knoten nach d wurde deaktiviert. Von allen anderen Knoten auf der Anfrageseite aus wird d nochmals besucht werden.
Das mittlere Bild zeigt die Situation bei Verwendung eines Sterns. Nachdem die Kante vom Zentralknoten zu d entfernt wurde, endet die Suche von jedem anderen Anfrageknoten aus beim Zentralknoten. Den selben Effekt erzielt im dritten Bild das Entfernen von d aus der Nachbarschaftsliste.

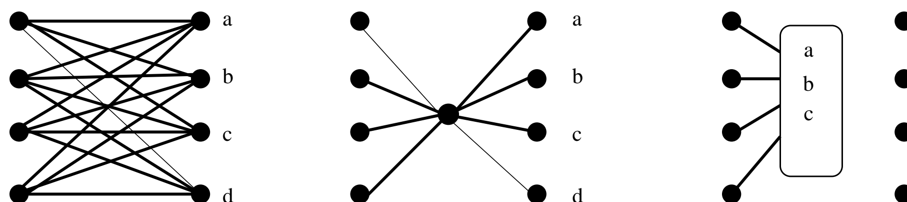

Abbildung 4.15: Entfernen eines Knotens aus der Nachbarschaftsliste

Auf diese Weise gibt es pro Knoten auf der Anfrageseite und Clique nur noch eine unnötige Aktion, nämlich die Überprüfung, dass eine der Nachbarschaftslisten bereits leer ist.
An dieser Stelle könnte man einwenden, dass es möglich wäre, die gesamte Nachbarschaftsliste zu löschen, sobald sie ein einziges Mal von der Anfrageseite aus erreicht wurde. Dies wäre bei einer reinen Erreichbarkeitsfrage tatsächlich möglich. In unserem Fall ist dies leider nicht möglich:
Es ist im Rahmen eines Matching-Algorithmus nur konstant ($\mathcal{O}(1)$) oft möglich, den Kompressionsalgorithmus aufzurufen. Konkret rufen wir ihn daher nur einmal zu Beginn auf und erhalten eine Kompression, in der spätere Änderungen der Kantenrichtungen nicht berücksichtigt werden können.
Schlägt uns SUGGEST(v) später eine Kante zu einem Nachbarknoten vor, so kann es sein, dass wir diese Kante gar nicht benutzen dürfen, da sie nicht in der gewünschten

Richtung passiert werden kann. Dies ist genau dann der Fall, wenn diese Kante zum momentanen Matchingpartner von v führt, da genau diese Kanten entgegengesetzt gerichtet werden.
Da jeder Knoten aber nur einen Matchingpartner hat, kann dieser “Fehler” (wir nennen ihn *Fehlertyp 1*) nur einmal pro Matchingkante und daher insgesamt n mal auftreten.
Betrachten wir also eine Nachbarschaftsliste und enthält diese den momentanen Matchingpartner, so verbleibt dieser nach Abarbeitung der Liste als einziger Knoten noch in der Liste. Wird der nächste Knoten auf der Anfrageseite bearbeitet, wird diese Liste also nochmals aufgerufen und dieser letzte Knoten ggf. besucht. Dies kostet zwei weitere Operationen.

Die Zahl der Operationen pro Clique während einer Suche ist, wenn der gerade beschriebene Fall nicht auftritt, beschränkt durch die Zahl der Knoten auf der Anfrage- und Antwortseite. Summiert man dies über alle Cliquen und Knoten auf, so ergibt sich genau die Gesamtgröße der Cliquenzerlegung $\mathcal{O}\left(m\frac{\log\frac{2n^2}{m}}{\log n}\right)$. Die Fehler vom Fehlertyp 1 können insgesamt $2n$ weitere Operationen kosten.
Somit ergibt sich als Gesamtzahl der Operationen für eine algorithmische Suche: $\mathcal{O}\left(m\frac{\log\frac{2n^2}{m}}{\log n}+2n\right)$.

Wir wollen alle Operationen, die im Rahmen einer algorithmischen Suche auftreten, jeweils in konstanter Zeit durchführen können. Um dies zu realisieren, setzen wir an mehreren Stellen doppelt verkettete Listen ein, da diese Einfügen und Löschen in konstanter Zeit erlauben, und ein gelöschtes Objekt beim Durchlaufen der Liste keine Operationen mehr verursacht.
Um die jeweiligen Objekte nach einem Löschvorgang wieder einfügen zu können, verwalten wir jeweils zusätzlich eine Liste der gelöschten, oder - wie wir sagen werden - deaktivierten Objekte. So kann Aktivieren und Deaktivieren durch ein Verschieben der Objekte zwischen den beiden Listen realisiert werden.

Basierend auf den eben beschriebenen Prinzipien ergibt sich die folgende Konstruktion (vgl. Abb. 4.16):
Zu jeder Clique C speichern wir vier doppelt verkettete Listen von aktivierten bzw. deaktivierten Eingangs- und Ausgangs-Knoten E_C^a, E_C^d, A_C^a und A_C^d. Dies erlaubt es, den jeweiligen Eintrag des Knotens in konstanter Zeit zu aktivieren bzw. deaktivieren. Das Deaktivieren von Ausgangsknoten muss ermöglicht werden, da im nichtbipartiten Fall die Cliquen auch “rückwärts” durchquert werden.
Um die Nachbarschaft eines Knotens v zu verwalten, verfügt jeder Knoten v über zwei doppelt verkettete Listen K_v^a und K_v^d. Diese enthalten Zeiger auf alle Cliquen, zu denen v gehört, sowie auf alle Knoten, zu denen v über normale Kanten adjazent ist. Wiederum enthält hierbei die a-Liste alle aktiven Objekte und die d-Liste die inaktiven.

Hin und wieder wollen wir ausgehend von einem Knoten diesen in einer Clique deaktivieren. Hierzu müssen wir den entsprechenden Eintrag dieses Knotens in den Listen der Clique finden.
Dabei sei jedem Zeiger $k_v \in K_v^a$ bzw. K_v^d auf eine Clique ein Zeiger $\vec{k}_v$ auf den Eintrag des Knotens v in der Liste der Clique bzw. des Knotens zugeordnet. Dabei nehmen

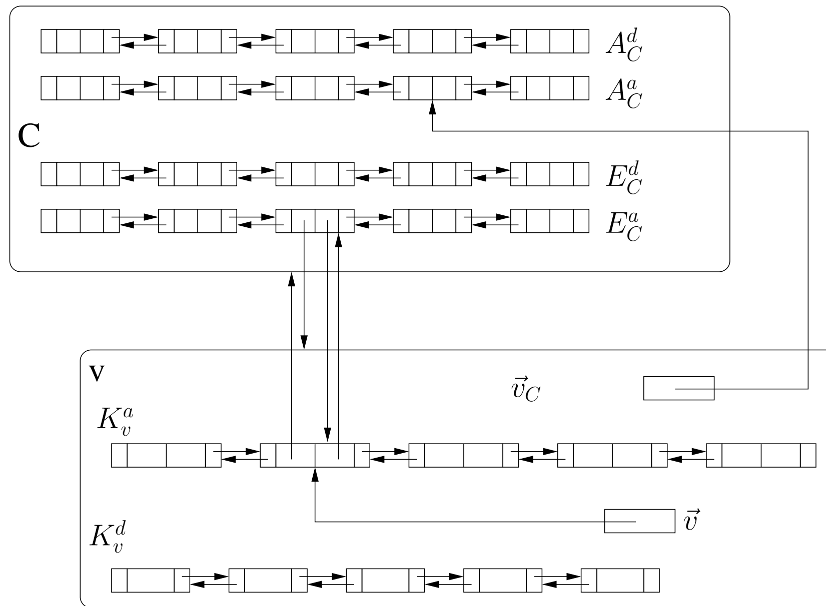

Abbildung 4.16: Datenstruktur zur Cliquenverwaltung

wir an, dass der entsprechende Eintrag bei einem Wechsel aus der a- in die d-Liste und zurück nicht im Speicher verschoben wird. Ansonsten müsste der Zugriff über einen weitern Zwischenzeiger realisiert werden, der bei jedem Wechsel aktualisiert wird (d.h. als sog. “Handle”).
Entsprechend sehen wir auch Zeiger von den Cliqueneinträgen zu den entsprechenden Einträgen in den Listen der Knoten vor: $\vec{e}_C$ zu jedem Eintrag $e_C \in E_C$ bzw. $\vec{a}_C$ zu jedem $a_C \in A_C$.
Zudem seien jedem Knoten v zwei Zeiger $\vec{v}$ und $\vec{v}_C$ zugeordnet. $\vec{v}$ markiert den momentan behandelten Knoten bzw. die momentan behandelte Clique aus K_v. Ist es eine Clique, so vermerkt $\vec{v}_C$ die aktuelle Arbeitsposition innerhalb dieser Clique.
$\vec{v}$ zeigt zu Beginn auf den Anfang von K_v^a. Die Kanten von v zu Knoten, auf die $\vec{v}$ und $\vec{v}_C$ in einer Phase oder einem Schritt schon gezeigt haben, betrachten wir als *bearbeitet* für diese Phase bzw. für diesen Schritt. Das gleiche gilt für Verweise auf Cliquen.

Den Graphen, der sich ergibt, wenn alle gefundenen Cliquen in dieser Weise umgewandelt werden, bezeichnen wir mit G_B^C.
Nun müssen wir noch die drei Zugriffsoperationen SUGGEST, ACTIVATE und DEACTIVATE realisieren: Dies geschieht mittels der Algorithmen 12, 13 und 14. Bei der Angabe von Zeigern unterscheiden wir zur Verbesserung der Lesbarkeit nicht zwischen dem Zeiger selbst und dem Ziel des Zeigers. Die jeweilige Bedeutung ergibt sich stets eindeutig aus dem Kontext.

Satz 4.19 *Auch mit dem Paradigma der augmentierenden Pfade kann im bipartiten Fall eine Laufzeit von* $\mathcal{O}\left(\sqrt{n}m\frac{\log\frac{2n^2}{m}}{\log n}\right)$ *erreicht werden.*

Algorithmus 12 SUGGEST - Routine zum Auffinden neuer Knoten

```
1:  procedure SUGGEST(v)
2:      if v⃗ = Ende(K_v^a) then
3:          Rückgabewert: undefiniert
4:      else
5:          if v⃗ zeigt auf eine Clique C then
6:              if v⃗_C zeigt nicht auf einen Knoten in C then
7:                  v⃗_C := Anfang(A_C^a)
8:              end if
9:              Rückgabewert := Zielvon v⃗_C
10:             if v⃗_C = Ende(A_C^a) then
11:                 v⃗ := next(v⃗)        ▷ next liefert den folgenden Eintrag der verketteten
                        Liste, in der sich das Zielobjekt befindet.
12:                 Exit
13:             end if
14:             v⃗_C := next(v⃗_C)
15:             Exit
16:         else
17:             Rückgabewert := Ziel von v⃗
18:             v⃗ := next(v⃗)
19:             Exit
20:         end if
21:     end if
22: end procedure
```

Beweis:

1. Die Korrektheit des Algorithmus bleibt unangetastet, da durch die Beschleunigung lediglich Schritte vermieden werden, die ohnehin keine Auswirkungen gehabt hätten.

2. Das Alternieren über die in einer Phase gefundenen Wege ist in Zeit $\mathcal{O}(n)$ möglich. Die algorithmische Suche in jeder Phase kostet - wie eben gezeigt - $\mathcal{O}\left(m\frac{\log\frac{2n^2}{m}}{\log n}\right)$ Schritte, sowohl in den MBFS - als auch in den MDFS-Phasen. Es gibt insgesamt höchstens $\sqrt{n}$ BFS und $2\sqrt{n}$ DFS Suchen.

□

Algorithmus 13 ACTIVATE - Routine zum Aktivieren eines Knotens

1: **procedure** ACTIVATE(v)
2: **for all** $k_v \in K_v^d$ **do**
3: entferne das Ziel von $\vec{k}_v$ aus der Liste in $v = A_C^d$-, E_C^d- bzw. K_z^d-Liste, in der es sich befindet
4: füge es ans Ende der entsprechenden A_C^a, E_C^a bzw. K_z^a-Liste ein.
5: **end for**
6: **end procedure**

Algorithmus 14 DEACTIVATE - Routine zum Deaktivieren eines Knotens

1: **procedure** DEACTIVATE(v)
2: **for all** $k_v \in K_v^a$ **do**
3: entferne da Ziel von $\vec{k}_v$ aus der Liste in A_C^a-,E_C^a- bzw. K_z^a-Liste, in der es sich befindet
4: füge es in die entsprechende A_C^d, E_C^d bzw. K_z^d-Liste ein.
5: **end for**
6: **end procedure**

4.4 Die Beschleunigung von MBFS

Im vorangegangenen Kapitel haben wir beschrieben, wie auch im Kontext alternierender Pfade eine Beschleunigung von Matchingalgorithmen für bipartite Graphen erreicht werden kann.
In den folgenden Kapiteln übertragen wir diese Vorgehensweise auf Algorithmen für den nichtbipartiten Fall. Dieser Fall ist wesentlich komplexer, da Knoten gegebenenfalls mehrfach betrachtet werden müssen (vgl. Kapitel 4.2$_{60}$).
Zentral für unsere Analyse ist wieder die Beobachtung, dass die Zahl der tatsächlich "Fortschritt bringenden" Operationen durch $\mathcal{O}(n)$ beschränkt ist. Alle anderen Operationen sind eigentlich "überflüssig". Wir wollen also wiederum versuchen, die Zahl der überflüssigen Operationen durch die Gesamtgröße der Cliquenzerlegung, d.h. durch die Summe der Ordnungen aller Cliquen zuzüglich der sonstigen Kanten, zu beschränken.
Wir wollen nun die im vorangegangenen Kapitel 4.3.1$_{85}$ entwickelten Routinen SUGGEST, ACTIVATE und DEACTIVATE im Kontext von MBFS verwenden.

4.4.1 Beschleunigung der Vorwärtssuche

Während der Vorwärtssuche müssen Kreise ungerader Länge nicht berücksichtigt werden (vgl. Invariante 4.5$_{69}$ von Blum [2]). Da somit die Suche analog zu normalem BFS verläuft, kann jeder Knoten sofort deaktiviert werden, sobald er besucht wurde. Es ergeben sich also hier nur wenige Veränderungen gegenüber dem im Kapitel 4.3.1$_{85}$ beschriebenen Vorgehen: Erreichen wir eine Clique, lassen wir uns von SUGGEST einen Knoten vorschlagen und überprüfen, ob er auch in G_B erreichbar ist. Ist er erreichbar, so betrach-

ten wir die Kante zu ihm mit den gleichen Regeln wie eine “normale Kante”. Danach kann er deaktiviert werden.

Beim Algorithmus von Blum [2] werden aber nicht nur die Kanten von der Suche ausgeschlossen, die bei einer vorangegeangenen Vorwärtssuche bereits betrachtet wurden, sondern auch die, die bei einer vorangegangenen Rückwärtssuche betrachtet wurden. Wir können diese Kanten nach den jeweiligen Rückwärtssuchen aber nicht in komprimierter Form bereitstellen. Wir müssen also einen anderen Weg finden, diese Kanten zu vermeiden.
Dass die bereits bei einer Rückwärtssuche betrachteten Kanten bei der Vorwärtssuche nicht erneut betrachtet werden, hat zwei Effekte:

1. Es wird eine zweite Betrachtung jeder Kante gespart. Diese ist unnötig, da die Kante bereits das Level hat, das ihr die Vorwärtssuche erneut vergeben würde und sich schon in E_K befindet.

2. Es wird verhindert, dass der A-Knoten der Basis einer Blüte ein Level erhält, das auf einem nicht streng einfachen Pfad beruht.

Die erste Eigenschaft beschleunigt die Behandlung dieser Kante maximal um einen Faktor 2, kann also im Sinne einer asymptotischen Laufzeitbetrachtung ignoriert werden. Die zweite Eigenschaft ist für die Korrektheit des Algorithmus von zentraler Bedeutung (vgl [2]).
Wir müssen also sicherstellen, dass bei der Vorwärtssuche niemals eine Kante verwendet wird, die bei einer Rückwärtssuche von einem Knoten $[v, B]$ zu einem Knoten $[u, A]$ geführt hat, für den irgendwann DOM$([v, B]) = \{[u, A]\}$ gegolten hat.

Proposition 4.20 *Für jede Kante mit den gerade beschriebenen Eigenschaften gilt, dass nach der betreffenden Rückwärtssuche MinB*$([v, B]) = [u, A]$ *gilt.*

Beweis:

1. Zum Zeitpunkt der Rückwärtssuche muss DOM$([v, B]) = \emptyset$ gewesen sein, sonst wäre keine $[v, B]$ verlassende Kante betrachtet, sondern sofort ein Sprung nach DOM$([v, B])$ durchgeführt worden.

2. Da die Kante betrachtet wurde, muss nach dieser Rückwärtssuche entweder DOM$([v, B]) = \{[u, A]\}$ oder DOM$([v, B]) =$ DOM$([u, A]) = [w, A]$ gegolten haben.

3. Im Fall DOM$([v, B])$ =DOM$([u, A]) = [w, A]$ kann aber auch später nie mehr DOM$([v, B]) = \{[u, A]\}$ gelten.

4. Damit muss $[u, A]$ der A-Knoten der Basis der ersten Blüte sein, zu der $[v, B]$ gehörte.

□

Wir müssen also beim Betrachten jeder Kante $([v, B], [u, A])$, die von SUGGEST vorgeschlagen wird, prüfen, ob

1. die Kante wirklich in der Richtung $B \rightarrow A$ gerichtet ist.
2. $[u, A] \neq \text{MinB}([v, B])$.

Es müssen nun all die Kanten für die Rückwärtssuche bereit gestellt werden, die von der jeweiligen Runde der Vorwärtssuche gefunden wurden. Da dies $\Omega(m)$ Kanten sein könnten, müssen diese zudem in komprimierter Form an die Rückwärtssuche übergeben werden.
Die Vorwärtssuche bestimmt Baum- und Vorwärtskanten. Bei den Baum- und Vorwärtskanten führen alle Kanten stets von einem Level l zu einem Level $l + 1$.
Somit kann von jedem Knoten mit Level l auf der Anfrageseite einer Clique jeder Knoten mit Level $l + 1$ auf der Antwortseite der Clique erreicht werden, wenn die Kante zwischen den Knoten in G_B die richtige Richtung hat. Wiederum verschieben wir die Berücksichtigung der Kantenrichtung auf einen späteren Zeitpunkt, d.h. in diesem Fall in die Rückwärtssuche und nach MDFS. Den Fall $[u, A] = \text{MinB}([v, B])$ muss nicht berücksichtigt werden, da bei der Rückwärtssuche von MBFS Blüten niemals von der Basis aus durchsucht werden.
ir können diese Knoten mit Level l und Level $l + 1$ alle paarweise miteinander verbinden. Hierbei entstehen wieder vollständige bipartite Teilgraphen, die zudem auch Teilgraphen der ursprünglichen Clique sind.
Eine einzelne Biclique kann also in Subcliquen aufgeteilt werden, die jeweils zu einer Level-Distanz von 1 korrespondieren (vgl. Abbildung 4.17). Da dabei jeder der Knoten zu höchstens einer der Teilcliquen gehört, erhalten wir eine neue Cliquenzerlegung mit gleicher oder kleinerer Gesamtgröße.

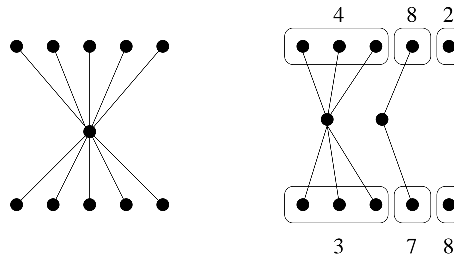

Abbildung 4.17: Aufsplitten von Cliquen

Die zu einer Clique gehörigen Teilcliquen werden als verkettete Liste verwaltet, die im Laufe der Iterationen vergrößert wird und wegen des monotonen Wachsens der Level implizit sortiert ist. Wir bezeichnen zu einer Clique C die Subclique mit Level i auf der Anfrageseite und $i + 1$ an der Antwortseite mit $C_{L(i)}$. Die Anfrageseite nennen wir $C^E_{L(i)}$, die Antwortseite $C^A_{L(i)}$. Die Knoten in $C^A_{L(i)}$ haben entsprechend Level $l + 1$.
Der Wert $C_{l_{max}}$ enthalte das Level der letzten Teilclique $C_{L(C_{l_{max}})}$ in der Liste. Um zu entscheiden, ob eine neue Clique begonnen werden muss, reicht es also aus, den Wert des hinzuzufügenden Knoten mit $C_{l_{max}}$ zu vergleichen.

Bemerkung 4.21 *Meistens wird hier ohnehin nur ein Level vertreten sein, da mit Ausnahme des Matchingpartners des betrachteten Knotens beim ersten Besuch der Anfrageseite alle noch nicht besuchten Knoten auf der Antwortseite das gleiche Level erhalten.*

Die Kanten, die an MDFS übergeben werden sollen und nicht zu einer Clique gehören, werden wie im ursprünglichen Algorithmus in der Menge E_K gesammelt.
Die Vorwärtssuche hat bezüglich Baum- und Vorwärtskanten die folgende Form:

1. Führe die Vorwärtssuche für alle Kanten, die nicht Bestandteil einer Clique sind, genauso durch, wie im originalen MBFS Verfahren.
2. Gehört ein betrachteter Knoten zu einer Clique C, so wird er in $C^E_{L(i)}$, eingetragen.
3. Alle Knoten auf der Antwortseite von C, deren Level noch nicht bestimmt wurde, oder die Level $i+1$ haben, erhalten Level $i+1$ und werden in die Antwortseite von $C_{L(i)}$, $C^A_{L(i)}$ eingetragen.

Die Kanten, die später die Rückwärtssuche auslösen (Micali und Vazirani [40]:Brücken), erscheinen zunächst auch als Vorwärtskanten und werden somit korrekt in E_K bzw. die entsprechende Clique eingefügt. Ihre gesonderte Behandlung beschreiben wir im Rahmen der Rückwärtssuche.
Algorithmus 15 beschreibt die Änderungen an der Vorwärtssuche von MBFS. Operationen bezüglich der Suchstruktur, mit der die Vorwärtssuche arbeitet, markieren wir mit dem Index V. Operationen mit den aufgeteilten Cliquen als Vorbereitung für die Rückwärtssuche sind mit dem Index R versehen. Zu Begin einer Phase, d.h. vor allen Vorwärts- und Rückwärtssuchen werden für alle Knoten $v \in V$ die Operationen ACTIVATE$_V(v)$ und DEACTIVATE$_R(v)$ durchgeführt.

4.4.2 Beschleunigung der Rückwärtssuche

Die Rückwärtssuche verwendet die Kanten aus G_B in umgekehrter Richtung. Entsprechend vertauschen wir bei den von der Vorwärtssuche übergebenen Cliquen Anfrage- und Antwortseite. Dies ist möglich, da wir die zugrundeliegenden Datenstrukturen von vorneherein symmetrisch konzipiert haben (s. Kapitel 4.3.1[85]).
Das wesentliche Problem bei der Rückwärtssuche ist, dass es nicht möglich ist, einen Knoten zu deaktivieren, sobald er einmal gefunden wurde, da ein erneuter Besuch dieses Knotens notwendig sein kann, wenn eine Blüte von außen betreten wird. Ein einfaches Beispiel zeigt die Abbildung 4.18[96]: Der Knoten $[x.B]$ wird sowohl bei der Rückwärtssuche von Kante 1 aus als auch bei der Rückwärtssuche von Kante 2 aus berücksichtigt.
Auf der anderen Seite muss nicht jeder zweite Besuch eines dafür vorgemerkten Knotens im Rahmen des Aufspürens einer neuen Blüte geschehen. Ein Beispiel hierzu gibt Abbildung 4.19[97]. Hier werden mehrere Knoten der gleichen Blüte von der Clique aus erreicht.

Wir wollen eine Clique nach Möglichkeit also nur dann durchqueren, wenn der Knoten auf der anderen Seite zu einer anderen Blüte gehört, als der Knoten, von dem die Suche

Algorithmus 15 Beschleunigte MBFS Vorwärtssuche

```
1: procedure MBFS VORWÄRTSSUCHE(l)
2:     for all [v, Z] ∈ ℒ(l) do
3:         for all Cliquen C, mit [v, z] ∈ E^a_C do
4:             if C_{L_max} ≠ l then
5:                 Erzeuge eine neue leere Teilclique und nenne die zugehörigen Mengen
                       C^E_{L(l)} und C^A_{L(l)}
6:                 C_{L_max} := l
7:             end if
8:             ACTIVATE_R([v, Z])
9:         end for
10:        while ([u, Z̄] := SUGGEST_V([v, Z])) definiert do
11:            if ([u, Z̄], [v, Z]) ∈ E(G_B) und (Z = A oder [u, A] ≠ MinB([v, B])) then
12:                DEACTIVATE_V([u, Z̄])
13:                if l([u, Z̄]) undefiniert oder l([u, Z̄]) = l + 1 then
14:                    l([u, Z̄]) := l + 1
15:                    ℒ(l + 1) := ℒ(l + 1) ∪ [u, Z̄]
16:                    if ([v, Z], [u, Z̄]) war eine normale Kante then
17:                        E_K := E_K ∪ e
18:                    else            ▷ Sei C die Clique zu der ([v, Z], [u, Z̄]) gehört
19:                        C^E_{L(l)} := C^E_{L(l)} ∪ {[v, Z]}
20:                        C^A_{L(l)} := C^A_{L(l)} ∪ {[u, Z̄}
21:                    end if
22:                end if
23:            end if
24:        end while
25:    end for
26: end procedure
```

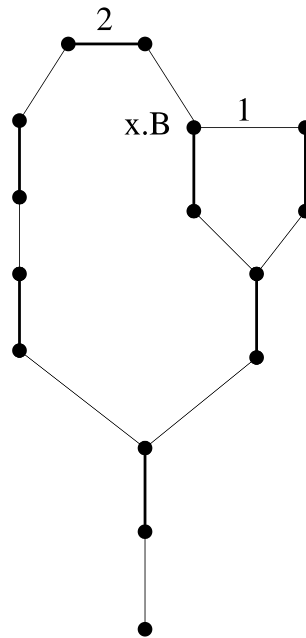

Abbildung 4.18: Erneuter Besuch eines Knotens bei der Rückwärtssuche

ausgeht. Gehören mehrere Knoten auf der Antwortseite zur selben Clique, so soll also nur einer von diesen besucht werden. Man könnte also die Menge dieser Knoten durch einen einzelnen Repräsentanten ersetzen.
Hierbei tritt allerdings das Problem auf, dass dieser Repräsentant wegen der vorhandenen Matchingkanten ggf. von einem Knoten der Anfrageseite aus nicht erreicht werden kann. Ein Beispiel zeigt Abb. 4.2097: Wird die Clique vom mittleren Knoten auf der linken Seite aus erreicht und ist der mittlere Knoten rechts der einzige Repräsentant, so wird die Blüte (durch Ellipse angedeutet) rechts nicht betreten, obwohl im nicht komprimierten Graphen Kanten zu den Knoten rechts oben und unten vorhanden waren. Da jeder Knoten aber nur einen Matchingpartner haben kann, gleichen wir dies dadurch aus, dass wir einfach zwei Repräsentanten pro Blüte in einer Clique verwalten.

Leider ist es aus Zeitgründen nicht möglich, die Information, welche Knoten zu welchen Blüten gehören, bei jedem UNION-Schritt auf alle Cliquen zu übertragen. Es kann also vorkommen, dass wir erst beim tatsächlichen Betrachten der Knoten, die zu einer Clique gehören, feststellen, dass diese bereits die gleiche Menge repräsentieren. Für diese Clique findet der UNION-Schritt also sozusagen verspätet statt.
Um dies besser erfassen zu können, bilden wir das UNION-FIND-Mengensystem nochmals für jede Clique nach. Dies wird es uns erlauben, den Zeitaufwand der jeweiligen Operationen UNION-Aufrufen in dem zur Clique gehörenden Mengensystem zuzuordnen.
Der MBFS-Algorithmus verwendet zur Verwaltung der Blüte die Menge M der D-Mengen. Wir erzeugen nun für jede bipartite Clique $C = (A_C, B_C)$ eine weitere Instanz des UNION-FIND-Problems mit den Antwortseiten-Mengen M_C=$\{\{v\}|v \in A_C\}$. Wir indizieren im Folgenden die UNION und FIND Anweisungen um zu verdeutlichen, auf welches der Systeme sich die Operationen beziehen.

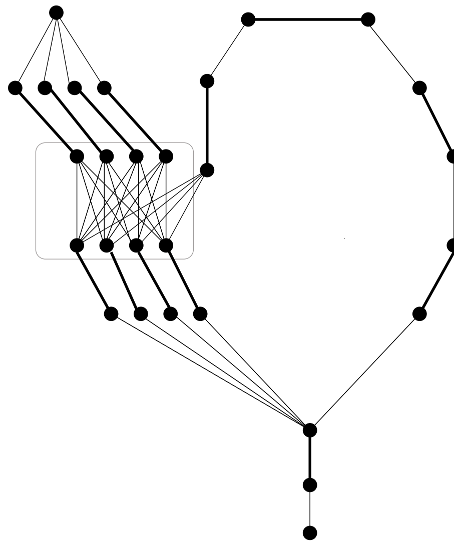

Abbildung 4.19: Clique in der Seite einer Blüte

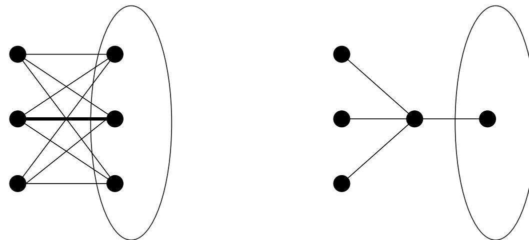

Abbildung 4.20: Blockierter Repräsentant

Zusätzlich versehen wir jeden Knoten aus $b \in B_C$ mit einem Zeiger $\gamma(b, C)$, der auf diejenige Menge in M_C zeigen soll, zu der b gehört. Da ein Knoten nicht zu beiden Seiten einer Clique gehören kann, sind diese Zeiger zu Beginn undefiniert, d.h. sie zeigen auf einen Platzhalter.

Nehmen wir an, wir suchen bei der Rückwärtssuche von einem Knoten b aus, der in M in der Menge S_b enthalten ist, nach Vorgängerknoten.
Normale Kanten werden wie bei MBFS behandelt. Wir betrachten alle Cliquen, die b auf der Anfrageseite enthalten, und bei diesen nach und nach die Mengen auf der jeweiligen Antwortseite. Sei A_C eine solche Menge und v einer ihrer Repräsentanten, der in M zur Menge S_v gehört. Nun können verschiedene Fälle auftreten:

1. Die Kante (b, v) gehört zum Matching: In diesem Fall tun wir nichts und betrachten - sofern vorhanden - den zweiten Repräsentanten von A_C.

2. $\gamma(b) = A_C$: Wir tun nichts.

3. $\gamma(b) \neq A_C$: Wir vereinigen $\gamma(b)$ und A_C und behalten von den möglichen 4 Repräsentanten auf der Antwortseite 2. Sollte $\gamma(b)$ noch undefiniert sein, setzen wir

$\gamma(b)$ auf A_C. Wir rufen UNION$_M$ für S_b und S_v auf.
SUGGEST' liefert v als nächsten Nachbarknoten zurück und MBFS springt nach DOM(v).

Proposition 4.22 FIND$_{M_C}(a)$ = FIND$_{M_C}(b)$ $\Rightarrow$ FIND$_M(a)$ = FIND$_M(b)$. *Die Umkehrung muss nicht gelten.*

Beweis: Nach einem UNION-Aufruf für M_C wird stets danach ein UNION in M in Bezug auf die beiden auslösenden Knoten ausgeführt. Es kann aber sein, dass ein UNION in M im Rahmen der Betrachtung einer anderen Clique oder Kanten durchgeführt wird. In diesem Fall wird M_C nicht verändert. □

Abbildung 4.21 veranschaulicht dies nochmals: Wir betrachten ausgehend von Knoten b, der zur Blüte A gehört, die andere Seite der Clique.

- Knoten x bildet noch allein eine Menge und ist somit auch ihr einziger Repräsentant. x gehört zur Blüte B. Also wird B mit A vereinigt und $\{x\}$ entsprechend mit $\gamma(b)$, also mit $\{y\}$.

- Für $\{y\}$ geschieht nichts.

- $\{z\}$ gehört bereits zu A, aber da dies an anderer Stelle bewirkt wurde, "wissen" wir dies innerhalb dieser Clique noch nicht. Wir betrachten also z und vereinigen $\{z\}$ und $\{x, y\}$ (natürlich nicht A mit A).

- Am Ende betrachten wir die Menge $\{u, v, w\}$ mit den Repräsentanten u und v. Diese Menge ist zwischenzeitlich zusammen mit B nach A übergegangen und wird ebenso mit $\{x, y\}$ vereinigt, wobei wir z.B. x und u als Repräsentanten behalten können.

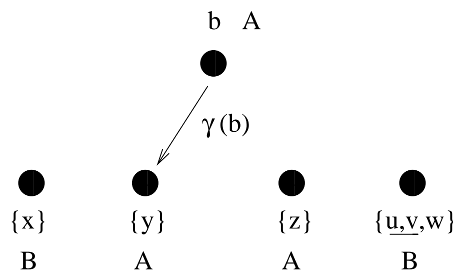

Abbildung 4.21: Situation in einer Clique vor einer Rückwärtssuche

Ebenso wie bei der Vorwärtssuche werden danach nahezu alle Knoten vereinigt sein. Es kann aber wiederum ein Knoten übrigbleiben, bei dem die Kantenrichtung nicht stimmte. Um diese Operationen zu realisieren, ersetzen wir bei der Rückwärtssuche die

SUGGEST-Routine durch eine neue Routine SUGGEST', die wir in Algorithmus 16[111] wiedergeben. Algorithmus 17[112] beschreibt wiederum das gesamte Vorgehen.
Die Korrektheit des Verfahrens wird durch dieses Vorgehen nicht beeinträchtigt, da nur solche Kanten ausgespart werden, die einen Sprung zu einer bereits im Rahmen der gleichen DDFS besuchten Basis auslösen würden.
Am Backtracking muss nichts geändert werden, da die bei DDFS implizit erzeugten DFS-Bäume nur $\mathcal{O}(n)$ Kanten haben.

Das Auslösen der Rückwärtssuche am Ende der Vorwärtssuche kann in analoger Weise behandelt werden. Einzelkanten können wieder normal verarbeitet werden. Bei Cliquen wird wiederum ein Union-Find-Problem für jede Clique gelöst:
Sobald im Verlaufe von MBFS einem Knoten ein Level zugeordnet wird, wird dieser Knoten für die Rückwärts-Suche aktiviert. Am Ende jeder Vorwärtssuche suchen wir mit SUGGEST' von den Knoten in Level l aus nach Nachbarn. Eine Rückwärtssuche wird genau dann gestartet, wenn die Knoten zu verschiedenen Blüten gehören, ihre DOM-Werte also unterschiedlich sind.

4.4.3 Laufzeitanalyse

Wir betrachten erneut Vorwärts- und Rückwärtssuche getrennt:

Lemma 4.23 *Die Anzahl der benötigten Operationen bei allen MBFS-Vorwärtssuchen einer Phase kann durch die Gesamtgröße der Cliquenzerlegung beschränkt werden.*

Beweis:
Die Zahl der Operationen für die Suche selbst kann ebenfalls mit der Methode beschränkt werden, die wir beim Beweis von Satz 4.19[89] verwendet habn:
Immer wenn SUGGEST eine Kante $([v, B], [u, A])$ vorschlägt, trifft mindestens einer der folgenden Punkte zu

1. Die Kante ist eine Matchingkante und von $[u, A]$ nach $[v, B]$ gerichtet.
2. MinB$([v, B]) = [u, A]$.
3. Die "normale" Kante $([v, B], [u, A])$ wird im komprimierten Graphen als bearbeitet angesehen.
4. $[u, A]$ wird in der Nachbarschaftsliste einer Clique als bearbeitet angesehen.
5. Eine Clique, die $[u, A]$ enthält, wird in der Nachbarschaft von $[v, B]$ als bearbeitet angesehen.

Die ersten beiden Fälle kommen im Rahmen eines Aufrufes von MBFS jeweils maximal n mal vor. Die gesamte Häufigkeit der Fälle 3,4 und 5 zusammen ist durch die Gesamtgröße der Cliquenzerlegung beschränkt:

Fall 3 kann höchstens so oft auftreten, wie es Kanten gibt, die zu keiner Clique gehören.
Die Häufigkeit von Fall 4 entspricht der Summe aller Knotenzahlen auf den Antwortseiten der Cliquen.
Die Häufigkeit von Fall 5 entspricht der Summe aller Knotenzahlen den Anfrageseiten der Cliquen.

Es verbleibt, die Komplexität des Cliquen-Aufteilungsprozesses zu betrachten.

Proposition 4.24
1. Die Zahl der Teilcliquen und die Größe der zu ihrer Verwaltung benötigten Datenstrukturen kann linear durch die Zahl der zur jeweiligen Clique gehörenden Knoten begrenzt werden.
2. Der Cliquen-Aufteilungsalgorithmus benötigt $\mathcal{O}(1)$ Operationen für jeden dieser Knoten.
3. Die Summe der Gesamtgrößen der an alle Rückrunden übergebenen Cliquen ist durch m^* beschränkt.
4. Die Cliquen können ihre Größe in weiteren Runden nicht mehr ändern.

Beweis:

1. Die Teilcliquen einer Clique werden als verkettete Liste verwaltet. Es werden nur Cliquen hinzugefügt, die mindestens zwei Knoten enthalten. Daraus folgt die Begrenzung unmittelbar.

2. Ein Knoten kann in konstanter Zeit in die passende Teilclique einsortiert werden, da die Level bei der Vorwärtssuche monoton wachsen und somit stets nur entschieden werden muss, ob der Knoten in die zuletzt benutzte Teilclique aufgenommen werden soll, oder ob eine neue Teilclique begonnen werden muss.

3. Dies ist gegeben, da die Cliquen das Ergebnis der Aufteilung größerer Cliquen sind. Jeder Knoten kommt in genau eine der Teilcliquen. Die Summe der Knotenzahlen der entstehenden Teilcliquen ist daher stets geringer oder gleich der Knotenzahl der ursprünglichen Clique.

4. Die von der Vorwärtssuche identifizierten Cliquen sind durch das jeweilige Level definiert und daher nach Abschluss der Runde unveränderlich. Die Rückwärtssuche verwendet nur von der Vorwärtssuche übergebene Kanten.

$\square$

Es ist also möglich, die entsprechenden Datenstrukturen in $\mathcal{O}(m^*)$ zu erzeugen und zu verwalten. $\square_{\text{L4.23}}$

Lemma 4.25 *Die Anzahl der benötigten Operationen bei allen MBFS-Rückwärtssuchen einer Phase kann durch die Gesamtgröße der Cliquenzerlegung beschränkt werden.*

Beweis: Wir werden die Kosten für die jeweiligen Operationen den Knoten der Eingangsseite und der Clique zuordnen, wobei jeder Kante nur eine konstante Zahl von Operationen und jeder Clique maximal ein konstantes Vielfaches der Zahl ihrer Ausgangsknoten an Operationen zugeordnet wird. Eine explizite Zuordnung zu Knoten der Ausgangsseite ist, wie gleich ersichtlich wird, nicht möglich.

Wenn SUGGEST' ausgehend von einem Knoten b einen Nachbarknoten vorschlagen soll, können die folgenden fünf Fälle eintreten:

1. Die momentan betrachtete Clique ist abgearbeitet und der Verweis von b zu dieser wird deaktiviert.
2. Die nächste Menge in der Nachbarschaftsliste ist $\gamma(C, b)$.
3. Die nächste Menge hat nur einen Repräsentanten und dieser ist der Matchingpartner von b.
4. $\gamma(C, b)$ zeigt noch auf nichts und wird auf diese nächste Menge gesetzt.
5. Die nächste Menge kann mit $\gamma(C, b)$ vereinigt werden.

Die ersten vier Fälle können pro Knoten auf der Anfrageseite jeweils nur einmal auftreten. Im fünften Fall findet eine UNION-Operation statt. Dies ist höchstens so oft möglich, wie es zu Beginn Mengen auf der Antwortseite der Clique gab. Da jeder Knoten auf der Antwortseite in genau eine Menge umgewandelt wurde, entspricht dies genau der Zahl dieser Knoten.
Summiert man dies über alle Cliquen auf, so kann die Zahl der Aufrufe von SUGGEST' durch das Fünffache der Gesamtgröße der Cliquenzerlegung beschränkt werden.

Beobachtung 4.26 *Alle Operationen von MBFS können einem SUGGEST' Aufruf so zugeordnet werden, dass jedem SUGGEST'-Aufruf nur eine endliche Zahl von Operationen zugeordnet wird.*

Das Backtracking während DDFS findet auf den beiden zuvor konstruierten DFS Suchbäumen statt, die entsprechend nur linear in ihrer Knotenzahl viele Kanten haben. Bei den folgenden Runden werden die Knoten, die zu Suchbäumen einer vorangegangenen Runde gehören, dann bereits vollständig übersprungen.
Da sich die Zahl der SUGGEST-Aufrufe durch die Summe der Zahl nicht von der Kompression betroffener Kanten und der Gesamtgröße der Cliquenzerlegung beschränken läßt, folgt die Aussage des Lemmas. $\square_{\text{L4.25}}$

4.5 Die Beschleunigung von MDFS

4.5.1 Aufbereitung der Ergebnisse von MBFS für MDFS

Wir erhalten von MBFS eine komprimierte Repräsentation der Kanten, die auf einem kürzesten Pfad liegen. Diesen komprimperten Graphen werden wir ebenfalls mit G_B^C be-

zeichnen: Die von der Vorwärtssuche von MBFS gefundenen Kanten werden bereits in komprimierter Form an die Rückwärtssuche von MBFS übergeben und können daher ohne weitere Veränderung an MDFS weitergeleitet werden. Bei der Rückwärtssuche von MBFS werden nur Kanten betrachtet, deren Symmetriepartner bereits bei der Vorwärtssuche berücksichtigt wurden (s.u.).
Dieser komprimierte Graph soll wieder dazu dienen, Kanten zu identifizieren, über die die Suche fortgesetzt werden soll.
MDFS bräuchte zum Arbeiten nun den Teilgraphen von G_B, der nur aus den Kanten besteht, die auf einem kürzesten Pfad liegen. Diesen zu erstellen würde aber zu lange dauern. Wir arbeiten also weiterhin mit dem vollständigen Graphen G_B.Wir verwenden aber die von MBFS übergebene komprimierte Darstellung der Cliquen und Kanten, die zu einem kürzesten Pfad gehören, um Kanten zu unbesuchten Knoten zu finden. Diese Vorgehensweise liefert genau die benötigten Kanten, da genau die Kanten, die nicht auf einem kürzesten Weg liegen, in der komprimierten Version fehlen und daher nie vorgeschlagen werden.
Wie in [2] beobachtet müssen die bei der Rückwärtssuche traversierten Kanten zunächst nicht gesondert in die Kantenmenge E_K eingefügt werden, da ihre jeweiligen Symmetriepartner im Rahmen einer Vorwärtssuche eingefügt wurden. Der Symmetriepartner jeder Kante auf einem kürzesten augmentierenden Pfad liegt ebenfalls auf einem kürzesten Pfad. Daher können nach Ablauf von MBFS diese Symmetriepartner nachträglich eingefügt werden. Es ist also nicht notwendig, Rückwärtskanten während der Rückwärtssuche von MBFS gesondert zu speichern.

Es tritt nun das Problem auf, dass die Symmetriepartner im Fall nichtsymmetrischer Kompression durch verschiedene Cliquen repräsentiert werden. Würde man aus diesen Cliquen diese einzelnen Kanten herausgreifen, so könnte dies die Zahl der Kanten, die nicht zu einer Clique gehören, soweit erhöhen, dass die Laufzeitreduktion nicht mehr garantiert ist.
Daher konstruieren wir zu jeder einzufügenden Clique C die symmetrische Clique $\bar{C}$ und fügen diese hinzu. Dies könnte dazu führen, dass eine Kante zweimal repräsentiert wird, aber dies kann die Laufzeit maximal verdoppeln, hat also keine Auswirkungen auf die asymptotische Komplexität (vgl. Beobachtung 5.14[121]).

Wie in Abschnitt 4.2.3[69] gezeigt, müssen mehr Daten an MDFS übergeben werden als nur die Level und Kanten. Die Größe dieser Informationen ist aber durch $\mathcal{O}(n)$ beschränkt, da pro Knoten lediglich ein Zeiger auf den Vorgänger und ein Zeiger auf die Wurzel der kleinsten Blüte, in der dieser Knoten liegt, übergeben werden muss.

4.5.2 Beschleunigung von Vorwärts- und Rückwärtssuche von MDFS

Wir werden bei MDFS analog zu MBFS von Vorwärts und Rückwärtssuche sprechen, wobei wir mit der Rückwärtssuche den Teil bezeichnen, in dem die D-Mengen bestimmt werden.
Auf die Rekonstruktionsphase müssen wir hier nicht eingehen, da ihre Laufzeit ohnehin

durch die Zahl der Baumkanten des MDFS-Baums und somit durch $\mathcal{O}(n)$ beschränkt ist. Es gibt drei Stellen, an denen Knoten bezüglich ihrer ausgehenden Kanten gescannt werden und an denen die Kompression ansetzen muss (die Nummerierung bezieht sich auf die Algorithmen 18_{113} und 19_{114}):

Stelle A) Die Wahl des nächsten Nachbarn bei der Vorwärtssuche in Zeile 7 in Algorithmus 18_{113}

Stelle B) Die Rückwärtssuche über die schwachen Rückwärtskanten in Zeile 6 in Algorithmus 19_{114}

Stelle C) Die Rückwärtssuche über die anderen Rückwärtskanten in Zeile 12 in Algorithmus 19_{114}

Wiederum ist es nicht einfach möglich, einen bereits besuchten Knoten unmittelbar zu deaktivieren, da in allen drei Fällen die Blütenstruktur eine Rolle spielt. Erreichen wir einen Knoten u, von dem aus ein bisher aus Symmetriegründen ausgelassener Knoten erreicht werden kann ($L(u) \neq \emptyset$), so müssen wir diesen erneut betrachten, um zum Knoten aus $L(u)$ zu springen.

Die Datenstrukturen von MDFS, die diese Sprünge koordinieren, sind die D-Mengen. Bei diesen handelt es sich wie bei der Blütenverwaltung in MBFS um Union-Find-Mengensysteme. Entsprechend können wir an diesen Stellen wieder Aufrufe der Funktion SUGGEST' einsetzen, wie wir sie für MBFS entwickelt haben.
Vier der besonderen Eigenschaften von MDFS müssen wir dabei berücksichtigen:

1. Ein Knoten, dessen Symmetriepartner bereits auf dem Stapel liegt, darf nicht ebenfalls hinzugefügt werden (Stelle A).

2. Wenn man bei der Vorwärtssuche eine identifizierte Blüte von außen erreicht, wird deren Basis auf den Stapel gelegt (Stelle A).

3. Die von MBFS übergebenen Cliquen müssen berücksichtigt werden (Stelle A).

4. Wenn man bei der Rückwärtssuche eine identifizierte Blüte erreicht, wird die Suche unmittelbar an deren Basis fortgesetzt oder die erweiterbaren Kanten müssen betrachtet werden. (Stellen B und C).

Punkt 1 läßt sich wie folgt realisieren: Sobald ein Knoten $[u, X]$ auf den Stapel gelegt wird, wird sein Symmetriepartner $[u, \bar{X}]$ deaktiviert. Sobald $[u, X]$ vom Stapel genommen wird, wird $[u, \bar{X}]$ wieder reaktiviert. Da dies pro Knoten nur einmal passieren kann und zum Deaktivieren und Aktivieren eine Knotens nur die zu ihm inzidenten Cliquen betroffen sind, erhöht dies die Laufzeit nur um einen weiteren Summanden von höchstens $\mathcal{O}(m^\star)$.
Punkt 2 benötigt eine effiziente Verwaltung der Blüten. Zudem dürfen wir nur zu solchen Basen springen, von denen aus noch neue Knoten erreicht werden können. Dies kann in ähnlicher Weise wie bei MBFS durch eine Übertragung des Union-Find-Problems auf

einzelne Cliquen realisiert werden: Auf normalen Kanten, einschließlich solcher, die wegen L'-Werten erweiterbare Kanten erzeugen, verwenden wir MDFS wie in Kapitel 4.2.4[72] beschrieben.
Treffen wir auf eine Clique, so wollen wir nur noch zu denjenigen Knoten übergehen, deren L und L' Werte noch leer sind (d.h unbesuchten Knoten) und zu solchen, deren L bzw. L' Knoten bei der momentanen Suche noch nicht angetroffen wurden.
Hierzu ersetzen wir auf den Antwortseiten der Cliquen die Knoten durch Mengen und beginnen auch hier mit den Mengen, die jeweils genau einen Knoten der Antwortseite enthalten. Stellen wir fest, dass zwei Mengen Teilmengen der gleichen D-Menge sind, oder fassen wir bei Rückwärtssuchen Mengen zusammen, so behalten wir pro D-Menge wieder zwei Repräsentanten (vgl. z.B. Abb. 4.21[98]).

Im Gegensatz zu MBFS sind wir damit aber noch nicht fertig, da wir noch die erweiterbaren Kanten verwalten müssen, um Blüten später auch rückwärts durchqueren zu können. Da es möglich wäre, dass es $\Omega(n^2)$ erweiterbare Kanten gibt, deren "auslösende" Kanten in einer Clique zusammengefaßt waren (vgl. Abb. 4.22), müssen diese nun ebenfalls anders verwaltet werden. Hierzu verändern wir die Konstruktion der erweiterbaren Kan-

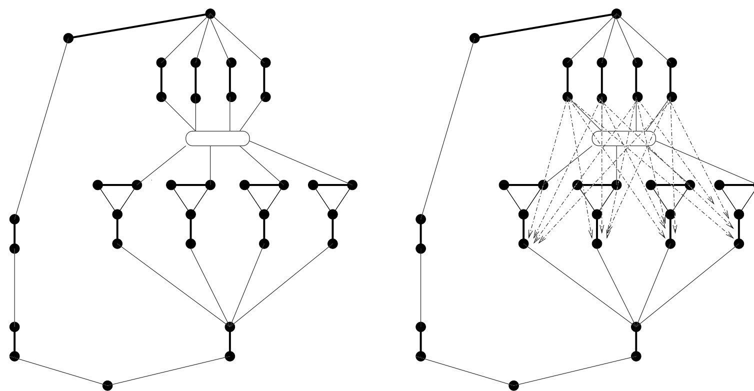

Abbildung 4.22: Quadratische Zahl erweiterbarer Kanten

ten: Beim MDFS zeigen die erweiterbaren Kanten von der Basis der Blüte zu demjenigen Knoten, von dem aus die Blüte von außen erreicht wurde. Dies hat den Vorteil, dass ein alternierender Pfad diese Kanten problemlos passieren kann, da die Kante aus der Blüte heraus stets eine freie Kante ist und die erweiterbare Kante daher korrekt gerichtet eingefügt werden kann (vgl. Abb. 4.23 linkes Bild).
Um die Zahl der erweiterbaren Kanten gering zu halten und stattdessen die bereits in komprimierter Form vorliegenden Kanten nutzen zu können, "verkürzen" wir die erweiterbaren Kanten nun so, dass sie von der Basis aus nicht mehr auf den ersten Knoten außerhalb der Blüte zeigen, sondern auf den letzten Knoten innerhalb (vgl. Abb. 4.23 rechtes Bild). Des weiteren fügen wir pro Menge auf der Antwortseite der Clique nur einmal eine erweiterbare Kante ein. Dies ist hinreichend dafür, dass bei der Rückwärtssuche alle Knoten auf der Anfrageseite berücksichtigt werden und stellt sicher, dass die Gesamtzahl eingefügter erweiterbarer Kanten durch die Gesamtgröße der Cliquenzerlegung beschränkt bleibt.
Die Kante vom Knoten außerhalb der Blüte zu diesem Knoten wird neu in die E-Menge

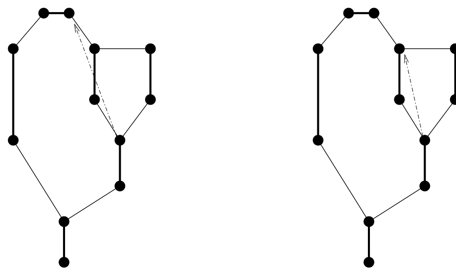

Abbildung 4.23: Änderung der erweiterbaren Kanten

eingefügt.
Dafür müssen wir beim Verfolgen einer erweiterbaren Kante einen Sprung machen, bei dem wir von einem Knoten $[u, A]$ zu einem Knoten $[v, A]$ springen, obwohl es normalerweise zwischen diesen keine Kante geben kann. Am Ziel setzen wir die Rückwärtssuche dann wie unter Punkt 4 beschrieben fort.
Für die spätere Kostenermittlung assoziieren wir zu der erweiterbaren Kante denjenigen SUGGEST'-Aufruf, deren Betrachtung das Einsetzen der erweiterbaren Kante ausgelöst hat.

Punkt 3 wird analog zum unbeschleunigten MDFS so behandelt, dass die MBFS-Blüten, wenn sie von außen erreicht werden, genauso behandelt werden, wie von MDFS selbst gefundene Blüten. Insbesondere werden bei einer Vorwärtssuche von einer Clique aus Ausgänge, die auf die gleiche MBFS-Blüte zeigen, vereinigt. Erweiterbare Kanten werden entsprechend eingefügt.

Bei **Punkt 4** müssen die benutzen Kanten rückwärts verfolgt werden, um die D-Mengen zu bestimmen. D.h. wir verfolgen die Cliquen rückwärts und vertauschen Anfrage- und Antwortseiten. Da wieder Unterblüten übersprungen werden müssen, werden nun die ehemaligen Anfrageseiten als UNION-FIND-Problem behandelt.
Im Gegensatz zu MBFS konnten bei MDFS für diese Rückwärtssuche keine Teilcliquen in den ursprünglichen Cliquen identifiziert werden. Wir werden aber sehen, dass die Tatsache, dass alle Kanten auf einem kürzesten Pfad liegen, dafür sorgt, dass die von MBFS vorgegebenen Cliquen ohnehin nicht weiter zerlegt werden müssen.
Zudem verwendet MDFS für die Rückwärtssuche die Kantenlisten E und R, um die wir uns in besonderer Weise kümmern müssen.

Die im Algorithmus von Blum [1] für diese Rückwärtssuche verwendeten Kantenlisten E und R (vgl. 4.2.472) werden in jedem Schritt des Algorithmus verändert. Selbst wenn man die Zahl der tatsächlichen Kantenoperationen durch Zusammenfassen von Cliquen verringern kann, so bleibt doch die Zahl der Operationen auf diesen Kantenlisten unverändert. Dies würde eine Laufzeit in der Größenordnung von $\Omega(m)$ implizieren.
Ziel der folgenden Lemmata ist daher zu zeigen, dass auf eine explizite Verwaltung dieser Listen verzichtet werden kann. Wir werden zeigen, dass unser Vorgehen dennoch alle Schritte ausführt, die bei Verwendung der Listen aufgetreten wären. Somit bleibt die Korrektheit des Algorithmus unangetastet.

Beobachtung 4.27 *Alle Operationen bei der Vorwärtssuche von MDFS, bei denen kein*

neuer Knoten auf den Stack gelegt wird, verändern lediglich die $D-, E-$ oder R-Mengen.

Zunächst zeigen wir, dass man auf die E- und R-Mengen vollständig verzichten kann. Danach demonstrieren wir, wie man die D-Mengen analog zur Blüten-Verwaltung von MBFS handhaben kann.

Lemma 4.28 *Es ist möglich, mit einem zusätzlichen Aufwand von $\mathcal{O}(n)$ zusätzlichen Operationen auf die E-Mengen zu verzichten.*

Beweis:

Zunächst geben wir eine Charakterisierung der E Kanten, die die Baum- und Rückwärtskanten einschließt und auf den Zeitpunkt beschränkt ist, zu dem die E-Kanten für die Rückwärtssuche benötigt werden.

Lemma 4.29 *$([v, B], [q, A])$ ist bei MDFS zu dem Zeitpunkt, wenn $[q, A]$ vom Stapel genommen wird, in $E_{[q,A]}$, falls*

1. $[v, B]$ bereits den Stapel verlassen hat,

2. $([v, B], [q, A])$ keine Baumkante ist

3. und $([v, B], [q, A])$ keine Erzeugung einer erweiterbaren Kante ausgelöst hat.

> **Beweis:** Die Bedingung ist hinreichend: Für alle Knoten $[v, B]$, die bereits den Stapel verlassen haben, wurde $([v, B], [q, A])$ schon betrachtet, da alle von $[v, B]$ ausgehenden Kanten betrachtet wurden. Außer Baum- und Rückwärtskanten und Kanten, die erweiterbare Kanten erzeugt haben, wurden alle Kanten in die E-Mengen der inzidenten Knoten aufgenommen, also auch in $E_[q, A]$. Da $[q, A]$ auf dem Stapel gelegen hat, kann es sich bei keiner dieser Kanten um schwache Rückwärtskanten handeln.
> Die Bedingung ist notwendig: Von Kanten, die noch nie auf dem Stapel lagen, können keine Kanten ausgehen, die zu einer E-Menge gehören, da diese Kanten nie betrachtet wurden.
> Es müssen also nur noch die Knoten betrachtet werden, die momentan auf dem Stapel liegen. Die von ihnen ausgehenden Kanten wurden ggf. teilweise betrachtet. Es könnte also noch nicht betrachtete Kanten geben, die von anderen, weiter unten auf dem Stapel liegenden Knoten ausgehen. Diese Kanten würden aber einen kürzeren Weg zum momentanen Knoten erlauben und sind daher wegen der Korrektheit von MBFS unmöglich.
> □

In unserer modifizierten Form des MDFS benötigen wir neben den Kanten aus den E-Mengen zudem auch die Kanten, die das Einfügen einer erweiterbaren Kante bewirkt haben, da wir die erweiterbaren Kanten wie Kanten der E-Mengen handhaben.

Betrachtet man also jede Kante, die Bedingung 1 erfüllt, als mögliche E-Kante und stellt erst durch nachträgliches Untersuchen fest, dass es sich doch um eine Baumkante handelt, so kann man auf diese Weise pro Baumkante nur einen Fehlversuch erhalten. Die Anzahl der Baumkanten und somit die dieser Fehlversuche ist aber durch O(n) beschränkt.

Als nächstes geben wir eine analoge Charakterisierung der R-Kanten.

Lemma 4.30 *Zu dem Zeitpunkt, zu dem ein Knoten $[u, B]$ vom Stapel entfernt und Knoten $[u, A]$ noch nicht auf den Stapel gelegt wurde, sind alle Kanten, die $[u, A]$ von Knoten $[v, B]$ aus erreichen, die nach $[u, B]$ auf den Stapel gelegt wurden, in $R_{[u,A]}$.*

> **Beweis:** Alle Kanten zu diesem Zeitpunkt bereits betrachteten Kanten, die zu $[u, A]$ führen, müssen schwache Rückwärtskanten sein, da anderenfalls $[u, A]$ bereits auf den Stapel gelegt worden wäre. Alle Knoten, die nach $[u, B]$ auf den Stapel gelegt wurden, sind bereits vollständig abgearbeitet. Kanten, die von Knoten ausgehen, die noch nie betrachtet wurden, sind nach Definition nicht in R-Mengen. Es verbleiben also nur noch Kanten von noch auf dem Stapel befindlichen, aber noch nicht völlig abgearbeiteten Knoten. Da diese sich unterhalb von $[u, B]$ auf dem Stapel befinden, können Kanten von diesen zu $[u, A]$ noch nicht betrachtet worden sein, da $[u, A]$ ansonsten auf den Stapel gelegt worden wäre. □

Korollar 4.31 *Zu dem Zeitpunkt, zu dem ein Knoten $[u, B]$ vom Stapel entfernt wurde und Knoten $[u, A]$ noch nicht auf den Stapel gelegt wurde, sind genau die Kanten, die $[u, A]$ von Knoten $[v, B]$ aus erreichen, die den Stapel bereits verlassen haben, in $R_{[u,A]}$.*

> **Beweis:** Da $[u, B]$ vom Stapel genommen wird, müssen alle Knoten, die nach $[u, B]$ auf den Stapel gelegt wurden, diesen bereits verlassen haben. Andererseits kann $[u, A]$ nicht von vom Stapel genommenen Knoten aus erreichbar sein, die vor $[u, B]$ auf den Stapel gelegt wurden, da diese bereits vollständig gescannt wurden und daher $[u, A]$ selbst hätte auf den Stapel gelegt werden müssen. Die Argumentation zu auf dem Stapel befindlichen Knoten ist die gleiche wie im vorangegangenen Lemma. □

Mittels dieser Überlegungen ist es also möglich, die durch die E und R- Datenstrukturen erfassen Kanten nachträglich zu bestimmen und somit auf eine beständige Aktualisierung dieser Mengen während der Ausführung des Algorithmus zu verzichten. $\square_{\text{L4.28}}$

Wir verwenden für die “simulierte” Verwaltung der E und R Mengen eine Kopie der komprimierten Version des Graphen G_B^C. Diese Kopie markieren wir in Algorithmus 18[113] mit dem Index E. Alle Knoten beginnen in deaktiviertem Zustand.
Sobald ein Knoten vom Stapel genommen wird, wird er in diesem komprimierten Graphen aktiviert. Dies ist gemäß Lemma 4.28 und Korollar 4.31 hinreichend, damit genau die Knoten, die von Lemma 4.29 erfaßt werden, zu dem Zeitpunkt aktiviert sind, an dem sie für die Rückwärtssuche über die E- und R-Kanten benötigt werden. SUGGEST’ wird höchstens n Baumkanten irrtümlich als E-Kanten vorschlagen, die mit eben diesem Zeitaufwand erkannt werden können.

Nun müssen wir noch auf die erweiterbaren Kanten eingehen. Bei normalem MBFS konnten erweiterbare Kanten wie normale Kanten in der E-Menge benutzt werden. Dies ist nun nicht mehr möglich.
Erreichen wir die Basis einer Blüte während der Rückwärtssuche über die Matchingkante, so springen wir nach und nach zu den Knoten, auf die die erweiterbaren Kanten zeigen. Hier folgen wir dann wieder den Kanten aus den jeweiligen E-Mengen. Dies ist möglich, da die entsprechenden Kanten in den E Mengen nun auch zu den Knoten führen, von denen aus ein Sprung durchgeführt wurde. Der Rückweg über eine erweiterbare Kante im Sinne des ursprünglichen MDFS wurde also in zwei Schritte aufgeteilt. Auf diese Weise werden aber immer noch alle Knoten identifiziert, die in die entsprechende D-Menge eingefügt werden müssen.
Die notwendigen Modifikationen an MDFS beschreibt Algorithmus 7$_{81}$. Dass dies genügt um die Laufzeitschranke einzuhalten, zeigen wir in Lemma 4.34$_{109}$.

In der Rekonstruktionsphase kann auf eine Beschleunigung verzichtet werden, da alle Operationen entweder mit den n Kanten des MDFS Baumes und den höchstens n Basen von Blüten arbeiten. Die Wege durch MBFS-Blüten können anhand der von MBFS übergebenen Zeiger rekonstruiert werden.
Die Korrektheit dieses Vorgehens ergibt sich daraus, dass der Ablauf von MDFS nicht verändert wird, so dass sich die Korrektheitsbeweise aus [1] unmittelbar übertragen. Dass die Laufzeitschranke eingehalten wird, zeigen wir in Lemma 4.33.

4.5.3 Laufzeitanalyse

Die durch MBFS übergebenen Blüten verhalten sich bezüglich der Analyse wie von MDFS selbst bestimmte, so dass wir sie in dieser Analyse nicht gesondert berücksichtigen müssen.

Lemma 4.32 *Die Zahl der Operationen bei allen MDFS-Vorwärtssuchen einer Phase ist durch $\mathcal{O}(m^*)$ beschränkt.*

Beweis: Mit Ausnahme der Deaktivierung der jeweiligen Symmetriepartner und mit Ausnahme der Fälle, in denen eine Blüte übersprungen werden muss, verhält sich MDFS wie DFS. Die Aktivierung und Deaktivierung als Symmetriepartner findet pro Knoten höchstens einmal statt und ist somit durch die Gesamtgröße der Cliquenzerlegung beschränkt.
Die Analyse des Überspringens von Unterblüten ist ähnlich zu der Analyse bei der Rückwärtssuche von MBFS:

Lemma 4.33 *Die zum Durchqueren einer Clique benötigten Operationen lassen sich durch die Zahl der Ein- und Ausgangsknoten der entsprechenden Clique beschränken.*

Beweis: Bei einem Aufruf von SUGGEST'(b) im Rahmen der Vorwärtssuche können folgende fünf Fälle eintreten:

1. Die momentan betrachtete Nachbarschaftsliste ist abgearbeitet.

2. Die nächste Menge in der Nachbarschaftsliste hat als einzigen Repräsentanten den Matchingpartner von b.

3. Die nächste Menge in der Nachbarschaftsliste ist $\gamma(b)$.

4. $\gamma(b)$ erhält erstmals einen Wert.

5. Die nächste Menge in der Nachbarschaftsliste ist ungleich $\gamma(b)$.

Die ersten vier Fälle können nur einmal pro Knoten auf der Anfrageseite vorkommen. Diese Kosten werden dem Anfrageknoten zugerechnet.
Treffen wir auf eine andere Menge, so findet in jedem Fall eine UNION-Operation statt. Die Zahl dieser Operationen ist durch die Zahl der dazu zur Verfügung stehenden Mengen begrenzt. Diese entspricht aber der Zahl der Anfrage- und Antwortknoten: In den Fällen, in denen die Eingangsknoten bereits einen nichtleeren L oder L' Wert gehabt hätten, wäre der Sprung bereits von diesem Knoten aus ausgeführt worden. Daher können keine weiteren D-Mengen in das Mengensystem der Clique aufgenommen werden. □

Die Kosten für das Einfügen einer erweiterbaren Kante ordnen wir dem SUGGEST'-Aufruf zu, die dieses Einfügen bewirkt hat. Da nur SUGGEST'-Aufrufe, bei denen eine UNION-Operation durchgeführt wird, erweiterbare Kanten erzeugen können, ist der Aufwand hierfür durch die Zahl der Knoten der Cliquen abschätzbar. Erweiterbare Kanten selbst können keine weiteren erweiterbaren Kanten erzeugen, da sie bei der Vorwärtssuche nicht benutzt werden. Summieren über alle Cliquen liefert damit das gewünschte Ergebnis. $\square_{\text{L4.32}}$

Lemma 4.34 *Alle Rückwärtssuchen einer Phase können in $\mathcal{O}(m^*)$ durchgeführt werden.*

Beweis: Solange keine erweiterbaren Kanten besucht oder Blüten übersprungen werden, entspricht die Rückwärtssuche einer normalen algorithmischen Suche und die Beschleunigung funktioniert entsprechend. Das Überspringen von Blüten zur Wurzel hin entspricht dem gleichen Vorgang bei MBFS und bei der Vorwärtssuche bei MDFS. Die Analyse ist daher identisch.
Treffen wir auf einen Knoten, von dem erweiterbare Kanten ausgehen, also auf die Basis einer Blüte, springen wir zu dem Knoten, auf den diese Kante zeigt. Die Kosten für den Sprung weisen wir wiederum dem SUGGEST'-Aufruf zu, die die erweiterte Kante erzeugt hat. Bei einem SUGGEST'-Aufruf, der eine erweiterbare Kante bewirkt, muss auf jeden Fall eine UNION-Operation stattgefunden haben. Die Zahl der UNION-Operationen ist aber durch die Gesamtzahl aller Knoten in Cliquen und damit durch die Gesamtgröße der Cliquenzerlegung beschränkt. □

4.6 Gesamtergebnis

Durch Kombination der beschleunigten Versionen von MBFS und MDFS und deren Verwendung in einem Algorithmus gemäß dem Satz von Hopcroft und Karp [21] - wie von Blum [1] beschrieben - erhält man somit den

Satz 4.35 *Bei der Anwendung des Graphenkompressionsverfahrens nach Feder und Motwani [11] auf den Algorithmus von Blum[1][2] kann auch unter Verzicht auf eine symmetrische Kompression ein maximales Matching in einem Graphen in Zeit* $\mathcal{O}(\sqrt{n}m\frac{\log\frac{2n^2}{m}}{\log n})$ *gefunden werden.*

Algorithmus 16 Änderung der SUGGEST Routine zur Anpassung an das Union-Find-Problem

1: **procedure** SUGGEST′(b)
2:   **if** $\vec{b} = \text{Ende}(K_b^a)$ **then**
3:     Rückgabe: undefiniert
4:   **else**
5:     **if** $\vec{b}$ zeigt auf eine Clique C **then**
6:       **if** $\vec{b}_C$ zeigt nicht auf einen Knoten in C **then**
7:         $\vec{b}_C :=$ Anfang von A_C^a
8:       **end if**
9:       **if** $\vec{b}_C = \text{Ende}(A_C^a)$ **then**
10:         $\vec{b} := \text{next}(\vec{b})$
11:         goto 2
12:       **end if**
13:       **if** $\gamma(b, C)$ undefiniert **then**
14:         $\gamma(b, C) := FIND_{M_C}(\vec{b}_C)$
15:         Rückgabewert := $\vec{b}_C$
16:         $\vec{b}_C := \text{next}(\vec{b}_C)$
17:         Exit
18:       **else**
19:         $x := \text{FIND}_{M_C}(\gamma(b, C))$ ▷ Zu welcher Menge gehört der Ausgangsknoten
20:         $y := \text{FIND}_{M_C}(\vec{b}_C)$
21:         **if** $y \neq x$ **then**
22:           $x := UNION_{M_C}(x, y)$
23:           Rückgabewert := $\vec{b}_C$
24:           $\vec{b}_C := \text{next}(\vec{b}_C)$
25:           **if** x hat schon zwei Repräsentanten in M_C **then**
26:             verschiebe $\vec{b}_C$ aus A_C^a nach A_C^d.
27:           **else**
28:             $\vec{b}_C$ wird neuer Repräsentant von X in M_C.
29:           **end if**
30:           Exit
31:         **else**
32:           **if** x hat schon zwei Repräsentanten in M_C **then**
33:             verschiebe $\vec{b}_C$ aus A_C^a nach A_C^d.
34:           **else**
35:             $\vec{b}_C$ wird neuer Repräsentant von X in M_C.
36:           **end if**
37:           $\vec{b}_C := \text{next}(\vec{b}_C)$
38:           goto 2
39:         **end if**
40:       **end if**
41:     **else** ▷ nicht von der Kompression betroffen → keine Modifikation
42:       Rückgabewert := Knoten am anderen Ende der Kante
43:       $\vec{b} := \text{next}(\vec{b})$
44:       Exit
45:     **end if**
46:   **end if**
47: **end procedure**

Algorithmus 17 DDFS nach [40] angepasst an [2] mit Kompression

1: Vertausche bei den von der Vorwärtssuche übergebenen Cliquen Anfrage- und Antwortseite.
2: **for all** $[v, Z] \in \mathcal{L}(l)$ **do**
3: **while** $[w, \bar{Z}] :=$ SUGGEST$'_R([v, Z])$ definiert **do**
4: **if** $([v, Z], [w, \bar{Z}])$ ist eine normale Kante **then**
5: $E_K := E_K \cup \{([v, Z], [w, \bar{Z}])\}$
6: **else** ▷ Sei C die Clique, die $([v, Z], [w, \bar{Z}])$ enthält
7: $C^E_{L(l)} := C^E_{L(l)} \cup \{[v, Z]\}$
8: $C^A_{L(l)} := C^A_{L(l)} \cup \{[w, \bar{Z}]\}$
9: **end if**
10: $K_l := [v, Z]$; $K_r := [w, Z]$
11: UNION$_M$(FIND(K_r), FIND(K_l))
12: barrier:=$[w, Z]$
13: **while** not (l$(\bar{K}_l) = 0$ or l$(\bar{K}_R) = 0$) **do** ▷ sonst echter augmentierender Pfad. vgl. [40]
14: **while** (l$(\bar{K}_l) \geq$ l$(\bar{K}_r)$) **do** ▷ Lloop
15: markiere K_l als L und besucht
16: füge $\bar{K}_l$ in Level l$([v, Z])$+l$([w, Z])-$l(K_l) ein.
17: **while** $(u :=$ SUGGEST$'_R(K_l))$ definiert **do**
18: $u :=$B(FIND(u)); UNION(FIND$(K_l), u)$ ▷ bereits bestimmte Blüten überspringen
19: **if** MinB(u) undefiniert **then** ▷ Blüten für MDFS speichern
20: MinB(u) :=FIND(K_l)
21: **end if**
22: **if** $u = K_r$ **then** ▷ K_l und K_r würden identisch
23: $K_r := \overline{p(\bar{K}_R)}$; $p(\bar{u}) = \bar{K}_l$; $K_l := u$
24: goto 34
25: **else if** u nicht besucht **then** ▷ normaler Suchschritt
26: $p(\bar{u}) := \bar{K}_l$; $K_l := u$
27: **end if**
28: **end while**
29: **if** $K_l = [v, Z]$ **then** ▷ Basis gefunden, immer vom Typ $[X, A]$
30: B(FIND$(K_r)) := K_r$
31: HALT
32: **end if**
33: **end while**
34: **while** l$(\bar{K}_r) >$ l$(\bar{K}_l)$ **do** ▷ Rloop
35: markiere K_r als R und besucht
36: füge $\bar{K}_l$ in Level l$([v, Z])$+l$([w, Z])-$l(K_l) ein.
37: **while** $(u :=$ SUGGEST$'_R(K_r))$ definiert **do**
38: $u := B$(FIND(u));UNION(FIND$(K_r), u)$ ▷ Blüten überspringen
39: **if** MinB(u) undefiniert **then**
40: MinB(u) :=FIND(K_r)
41: **end if**
42: **if** u noch nicht besucht **then** ▷ normaler Suchschritt
43: $p(\bar{u}) := \bar{K}_r$; $K_r := u$
44: goto 34
45: **end if**
46: **end while**
47: **if** $K_r \neq$ barrier **then**
48: $K_r = \overline{p(\bar{K}_r)}$ ▷ Backtrack-Schritt bezügl. K_r
49: **else**
50: barrier:=K_l
51: $K_r := K_l$ ▷ Der rechte Kopf darf den Knoten verwenden
52: $K_l := \overline{p(\bar{K}_l)}$ ▷ der linke Kopf sucht einen anderen Weg
53: goto 14
54: **end if**
55: **end while**
56: **end while**
57: **end while**
58: **end for**

Algorithmus 18 MDFS nach [1] mit Kompression

```
1:  procedure SEARCH
2:      if TOP(K) = t then
3:          Rekonstruiere den gefundenen s-t-Pfad
4:      else
5:          markiere TOP(K) als "gepushed"
6:          DEACTIVATE_D(TOP(K)); DEACTIVATE_D(‾TOP(K)‾)
7:          while [w,Y] := SUGGEST'_D(TOP(K)) definiert do        ▷ Vorwärtssuche
8:              if Y = B then
9:                  PUSH([w,B])
10:                 SEARCH
11:             else
12:                 if [w,A] ist als "gepushed" markiert then
13:                     if L_[w,A] ≠ ∅ then
14:                         Ersetzte TOP(K) durch (TOP(K), [w,A]), TOP(K)
15:                         PUSH(L_[w,A])
16:                         Erzeuge die erweiterbare Kante (L_[w,A], [w,A]).
17:                         L'_[w,A] := L_[w,A]
18:                         L_[w,A] := ∅
19:                         SEARCH
20:                     else
21:                         if L'_[w,A] ≠ ∅ then
22:                             Erzeuge die erweiterbare Kante (L'_[w,A], [w,A])
23:                         end if
24:                     end if
25:                 else
26:                     if [B(MinB([w,A])), B] ∉ K then        ▷ MBFS Blüten überspringen
27:                         Ersetzte TOP(K) durch (TOP(K), [w,A]), TOP(K)
28:                         Erzeuge die erweiterb. Kante ([B(MinB([w,A])), A], [w,A])
29:                         PUSH([B(MinB([w,A])), A])
30:                     else
31:                         PUSH([w,A])
32:                     end if
33:                     SEARCH
34:                 end if
35:             end if
36:         end while
37:         [v,X] := TOP(K)
38:         BLFIND(‾[v,X]‾)
39:         if ‾TOP(K)‾ war noch nicht auf dem Stapel then
40:             ACTIVATE_D(‾TOP(K)‾)
41:         end if
42:         ACTIVATE_E(TOP(K))
43:         POP
44:     end if
45: end procedure
```

Algorithmus 19 Levelbestimmung in Blüten mit Kompression

```
1: procedure BLFIND'([v, X])
2:     if X = B und [v, A] ist nicht als “gepushed” markiert then
3:         L_act := [v, A]
4:         D_{L_act} := ∅
5:         L_def := ∅
6:         while [q, B] := SUGGEST'_E[v, A] definiert do
7:             CONSTRL(([q, B], [v, a]), [v, B])
8:         end while
9:         while L_def ≠ ∅ do
10:            Wähle [k, A] ∈ L_def
11:            L_def := L_def \ {[k, A]}
12:            while [q, B] := SUGGEST'_E[k, A] definiert do
13:                CONSTRL(([q, B], [k, a]), [v, B])
14:            end while
15:        end while
16:    end if
17: end procedure
```

Algorithmus 20 Subroutine CONSTRL nach [1] mit Kompression

```
1: procedure CONSTRL(([q, B], [u, A]), [x, B])
2:     P_act := ([q, B], [u, A])
3:     [z, B] := [q, B]
4:     while [x, B] wurde nicht erreicht do
5:         [y, A] := [z, B]
6:         for all [y, A] auf dem Pfad rückwärts von [z, B] nach L ∪ {[x, B]} do
7:             D_{L_act} := D_{L_act} ∪ {[y, A]}
8:             L := L ∪ {[y, A]}
9:             P_{[y,A]} := P_act
10:            L_def := L_def ∪ {[y, A]})
11:            for all erweiterbaren Kanten ([y, A], [h, A]) do
12:                L_def := L_def ∪ {[h, A]})
13:            end for
14:        end for
15:        if [y, A] ∈ L then
16:            UNION_E(D_{L_act}, D_{[r,A]})
17:            UNION_D(D_{L_act}, D_{[r,A]})
18:            [z, B] := [r, B]
19:        end if
20:    end while
21: end procedure
```

Kapitel 5

Anmerkungen und Erweiterungen

5.1 Realisierung der Union-Find Algorithmen

Sowohl beim Algorithmus von Blum [2] als auch bei dem von Micaeli und Vazirani [30] wird angegeben, daß das Union-Find Problem in $\mathcal{O}(m+n)$ lösbar sei, wobei hier m die Zahl der Such- und n die Zahl der Vereinigungsschritte meint. In beiden Fällen baut dies auf dem "incremental tree union"-Algorithmus von Gabov und Tarjan [20] auf. Bei diesem wird ein Maschinenmodell vorausgesetzt, das es unterstützt, Operanden der Länge $\log\log n$ in konstanter Zeit zu verarbeiten. Verwendet man dieses Modell nicht, gelten wieder die Zeitschranken für das Zeiger-Modell und die Algorithmen für das UNION-FIND-Problem verlangsamen sich auf $\mathcal{O}((m+n) * (A^{-1}(m,n)))$, wobei $A^{-1}(m,n)$ die inverse Ackermannfunktion bezeichnet (vgl. z.B. [38]).

5.2 Beschleunigung von MBFS durch explizite Kompression

Das Hauptproblem bei der Übertragung der Graphenkompression auf den nichtbipartiten Fall ergibt sich aus der Tatsache, dass mehrere Pfade den Ersatzgraphen passieren können müssen. Dieses Problem tritt bei reinem MBFS nicht auf. Hier kommt es lediglich auf die Existenz eines Pfades an, so dass die Betrachtung mehrerer Pfade gar nicht notwendig ist. Naheliegend wäre es also, den Graphen aus Abbildung 5.1 Bild b) für die Clique aus Bild a) einzusetzen. Diese eine eingefügte Matchingkante würde es einem augmentierenden Pfad erlauben, die Clique zu durchqueren.
Diese Ersetzung durch einen einzelnen Stern genügt aber nicht: Wie Abbildung 5.2 zeigt, kann auf diese Weise ein augmentierender Pfad zerstört werden, da eine neue Blüte entsteht.
Ein kürzester augmentierender Pfad muss eine solche Clique also zweimal passieren können. Entsprechend verwendet man den Graphen aus Abbildung 5.1 Bild c) als Substitutionsgraphen für eine jeweilige Clique.
Voraussetzung hierbei ist - wie bei der Verwendung von CMA-Graphen auf Basis von

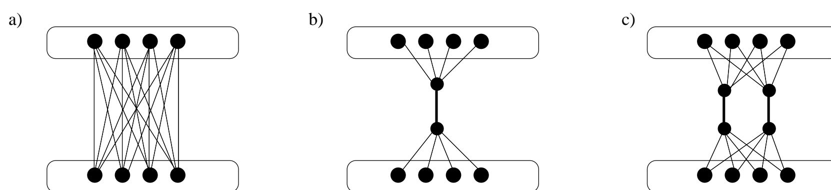

Abbildung 5.1: Cliquenersetzung bei expliziter Kompression für BFS

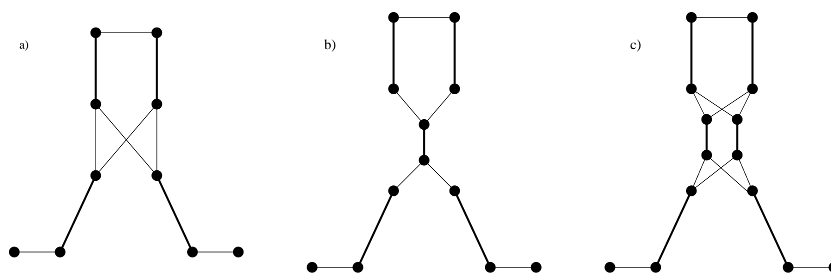

Abbildung 5.2: Künstliche Blüte

Superkonzentratoren - dass symmetrische Kompression vorliegt, so dass wiederum echte Bicliquen im nichtbipartiten Graphen identifiziert werden müssen.

Damit alle Pfade bei dieser Ersetzung gleichmäßig verlängert werden, werden einzelne Kanten als $(1,1)$ Cliquen behandelt. Einzelne Kanten werden also durch einen Pfad der Länge 3 aus zwei freien Kanten und einer Matchingkante ersetzt, so dass sich jeder streng einfache Pfad im Graphen um das Dreifache verlängert.

Auch hier kann bei der Kompression auf Matchingkanten keine Rücksicht genommen werden, da eine häufigere Anwendung des Kompressionsalgorithmus zuviel Zeit in Anspruch nehmen würde. Die Matchingkanten müssen also wieder nachträglich berücksichtigt werden. Waren im Ausgangsgraphen die Knoten a und b gematcht, so führt man nun zusätzlich einen Pfad der Länge 3 zwischen diesen ein, wobei die beiden äußeren Kanten als gematcht markiert werden. Somit werden diese Kanten ebenfalls durch einen dreifach längeren Pfad ersetzt. Wir bezeichnen den Graphen, der durch diese Ersetzungen aus dem ursprünglichen Graphen G hervorgeht, mit G'.

Abb. 5.3 gibt ein Beispiel für die somit durchgeführte Transformation.

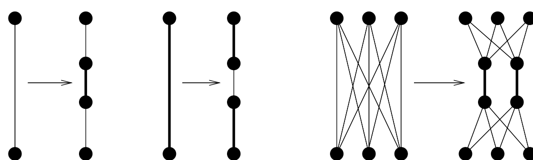

Abbildung 5.3: Ersetzung von Kanten und Cliquen

Lemma 5.1 *Wenn es in G einen einfachen alternierenden Weg der Länge l zwischen zwei Knoten u und v gibt, so gibt es in G' einen der Länge $3l$.*

Beweis: Um dieses Lemma zu zeigen, benötigen wir die folgende Beobachtung:

Beobachtung 5.2 *Ein kürzester augmentierender Pfad kann eine Clique nur einmal in jeder Richtung, d.h. insgesamt höchstens zweimal passieren.*

> **Beweis:** Passiert er die Clique zweimal in der gleichen Richtung, so hätte er beim ersten Durchgang die Clique bereits über den Knoten verlassen können, den er beim zweiten Durchgang benutzt hat. Dann gäbe es aber einen kürzeren augmentierenden Pfad. □

Solange der Pfad einfache Kanten benutzt, können diese durch den substituierten Weg der Länge 3 ersetzt werden. Wenn er eine Clique passiert, wird jeweils einer der beiden Ersatzgraphen traversiert, wobei wiederum eine Kante durch drei ersetzt wird. Da der Pfad jede Clique nur zweimal traversieren kann, reicht ein Ersatzgraph mit zwei möglichen Pfaden aus. $\square_{\text{L5.1}}$

Lemma 5.3 *Wenn es in G' einen alternierenden einfachen Weg der Länge $3l$ zwischen zwei Knoten gibt, so gibt es in G einen der Länge l.*

> **Beweis:** Jeder der Pfade der Länge 3, die für eine Kante eingesetzt wurden hat keinerlei Abzweigung, er muss also von dem einen Knoten der ursprünglichen Kante bis zum anderen Knoten durchlaufen werden. Entsprechend kann dies als Überqueren eben dieser Kante interpretiert werden.
> Dieses Konzept wird nur an einer Stelle durchbrochen, nämlich durch die doppelte Repräsentation von Matchingkanten in G'. Ein augmentierender Pfad, der sowohl den für die Matchingkante eingesetzten Pfad als auch die Repräsentation dieser Kante im Rahmen der Cliquenersetzung benutzt, kann nicht nach G zurückübertragen werden.
> Der Teilpfad durch die Clique hat aber an beiden Enden freie Kanten. Als Teilpfad eines alternierenden Pfades müssten die an beiden Enden jeweils folgenden Kanten also zum Matching gehören. Dies ist aber nur für die Kanten des hinzugefügten Pfades der Fall, so dass der einzige alternierende Pfad, der durch diese Kante führen kann, ein kurzer Kreis ist, der eben diese Kanten umfasst (vgl. Abb 5.4) und keine freien Knoten enthält.

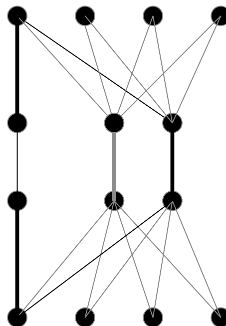

Abbildung 5.4: Einzig möglicher Pfad durch eine doppelt repräsentierte Kante

> Eine Verwendung dieser ungültigen Kanten ist also im Rahmen eines alternierenden Pfades zwischen zwei freien Knoten nicht möglich. □

Lemma 5.4 *Die Levelberechnung durch MBFS kann bei vorliegender symmetrischer Kompression mit Hilfe der obigen Transformation in Zeit $\mathcal{O}(m^*)$ durchgeführt werden.*

Beweis: Die Substitution der Kanten durch Pfade und die Rekonstruktion der Wege kann $\mathcal{O}(|E(G')|)$ erfolgen.
Wir vergleichen nun die Zahl der Kanten in G' mit der Zahl der Kanten, die ein Graph G'' hat, der aus G bei Substitution der Cliquen durch Sterne - wie bei Feder und Motwani [11] beschrieben - entsteht. Die Kantenzahl von G'' liegt entsprechend in $\mathcal{O}(m^*)$.

1. Jede Kante, die zu keiner Clique gehört, wurde durch drei Kanten ersetzt.

2. Jeder Kante, die einen Knoten w mit dem Zentrum eines Sternes verbindet, können wir die zwei Kanten zuordnen, die w in G' mit den beiden zentralen Kanten unseres Ersatzgraphen verbinden.

3. Da alle Cliquen, die wir ersetzten, auf beiden Seiten mindestens 2 Kanten haben, können wir die beiden zentralen Kanten unseres Ersatzgraphen beliebig zwei anderen Kanten der Clique zuordnen.

Somit kann G' maximal dreimal soviele Kanten haben wie G''. Der MBFS Algorithmus erhält also einen Graphen mit $\mathcal{O}(3m^*)$ Kanten und benötigt darauf eine in der Kantenzahl lineare Laufzeit. □

5.3 Verkleinerung der CMA-Graphen

Die Größe der CMA-Graphen kann durch die folgende Konstruktion weiter verringert werden:

Definition 5.5 *Ein (n, o, k)-Superkonzentrator ist ein gerichteter azyklischer Graph mit n Eingängen, $o < n$ Ausgängen und höchstens nk Kanten, so dass es für zwei beliebige Mengen von $r \leq o$ Eingängen und r Ausgängen stets r knotendisjunkte Pfade gibt, die diese paarweise miteinander verbinden.*

Proposition 5.6 *Ein (n, o, k)-Superkonzentrator kann durch Kombination eines (n, o, k) Konzentrators mit einem (o, k) Superkonzentrator erstellt werden, indem die o Ausgänge des Konzentrators mit den o Eingängen des Superkonzentrators verbunden werden.*

Beweis: Jede Teilmenge der Eingänge mit $r \leq o$ Elementen kann vom Konzentrator über disjunkte Wege zu r Eingängen des Superkonzentrators verbunden werden. Dieser kann sie dann den korrekten Ausgängen zuordnen. □

Wenn wir annehmen, dass der Konzentrator die Zahl der Kanten um einen Faktor Θ reduziert, so muss man, um die $n^{1-\delta}$ Eingänge bis auf $\frac{\log n}{\log \frac{2n^2}{m}}$ zu reduzieren, entsprechend h Konzentratoren mit

$$n^{1-\delta}\Theta^h = \frac{\log n}{\log \frac{2n^2}{m}} \Leftrightarrow h = \left(\frac{\log \frac{\log n}{\log \frac{2n^2}{m} n^{1-\delta}}}{\log \Theta} \right)$$

hintereinander verbinden.
Auf diese Weise muss man eine $n^{1-\delta}\frac{\log n}{\log \frac{2n^2}{m}}$ Kanten enthaltende Clique nicht mehr durch einen $88n^{1-\delta}$ großen CMA-Graphen ersetzen, sondern erreicht das Gewünschte bereits mit höchstens

$$5n^{1-\delta}\sum_{i=0}^{h}\Theta^i + 88\frac{\log n}{\log \frac{2n^2}{m}}$$

Kanten. Dies kann mit der vollständigen geometrische Reihe abgeschätzt werden:

$$\leq \frac{5}{1-\Theta}n^{1-\delta} + 88\frac{\log n}{\log \frac{2n^2}{m}}.$$

Beobachtung 5.7 *Ein (n, o, k) Superkonzentrator kann ebenso in einen CMA-Graphen umgewandelt werden, wie ein (n, k)-Superkonzentrator.*

5.4 Suchstrukturen für Graphen

Die in Kapitel 4.3.185 eingeführte Idee, die Suche in einem Graphen zu organisieren, indem eine Routine Kanten vorschlägt, die betrachtet werden sollen, motiviert die folgende Definition:

Definition 5.8 *Eine Suchstruktur S_G für einen Graphen $G = (V, E)$ ist eine Abbildung S von der Potenzmenge von $V \times V$ nach $V \times V$.*

Eine Suchstruktur schlägt in Abhängigkeit von den bereits vorgeschlagenen Knotenpaaren ein weiteres Paar zur Fortsetzung der Suche vor.

Definition 5.9 *Eine Suchstruktur heißt vollständig, wenn $S(V')$ für alle $V' \subsetneq V \times V$ definiert ist und $S(V') \notin V'$ gilt.*

D.h. es müssen so lange "neue" Paare vorgeschlagen werden, bis alle vorgeschlagen wurden.

Eine derartige Struktur kann nur dann sinnvoll eingesetzt werden, wenn zur Lösung einiger Teilprobleme nicht immer alle Kanten betrachtet werden müssen. Dies ist im Matching-Kontext häufig der Fall.

Es gibt drei Typen von Fehler-Fällen , aufgrund derer ein von der Suchstruktur vorgeschlagenes Knotenpaar (u, v) nicht zur Fortsetzung algorithmischer Suche verwendet werden kann:

1. $(u, v) \notin E$
2. u wurde noch nicht besucht
3. v wurde bereits besucht

Definition 5.10 *Eine Suchstruktur für einen Graphen $G = (V, E)$ heißt (t_1, t_2, t_3) Suchstruktur, wenn die Zahl der Typ i-fehler in $\mathcal{O}(t_i)$ liegt.*

Proposition 5.11 *Algorithmische Suche auf einem Graphen $G = (V; E)$ kann mit einer (t_1, t_2, t_3)- Suchstruktur in Zeit $\mathcal{O}(n + s + \sum_{i=1}^{3} t_i)$ durchgeführt werden; dabei ist s die Zeit, die zur Verwaltung der Suchstruktur selbst benötigt wird.*

> **Beweis:** Die Zahl der Operationen, bei der der Suchbaum vergrößert wird, ist durch $n = |V|$ beschränkt. Die nicht erfolgreichen Operationen fallen in eine der oben beschriebenen Fehlerklassen. □

Definition 5.12 *Wir sagen, dass eine Suchstruktur* effizient *ist, wenn $s \in \mathcal{O}(n + t_1 + t_2 + t_3)$, d.h. wenn der Aufwand zur Verwaltung der Suchstruktur vom Aufwand der Suche selbst dominiert wird.*

Da die Algorithmen, die wir verwenden, dem BFS oder DFS-Schema folgen, ist durch diese Verfahren der "Schwanzknoten" der jeweils vorzuschlagenden Kanten eindeutig gegeben und bereits besucht - entsprechend kann es hier nie Typ 2 Fehler geben.
Wir verwenden daher Suchstrukturen nicht in der obigen funktionalen Schreibweise, sondern stattdessen den in Kapitel 4.3.1[85] beschriebenen abstrakten Datentyp mit den Funktionen SUGGEST, ACTIVATE und DEACTIVATE.

Die folgende Suchstruktur ist eine naheliegende Verallgemeinerung der in Kapitel 4.3.1[85] vorgestellten Variante: Sei H^* die transitive Hülle eines azyklischen Graphen H. Dann läßt sich zu jedem Graphen G mit $G \subset H^*$ in folgender Weise eine $(0, 0, f_3)$-Suchstruktur für G gewinnen:

1. Entferne alle Knoten aus H, von denen aus man keinen Knoten in G erreichen kann.
2. Entferne alle Knoten aus H, die nicht von einem Knoten in G erreicht werden können.
3. Wenn ein Nachbar eines Knotens aus G gesucht wird (SUGGEST), gib nach und nach alle von diesem Knoten in H aus erreichbaren Knoten in G an.
4. Wenn ein Knoten aus G gefunden oder deaktiviert wurde, lösche alle zu diesem Knoten führenden Kanten.
5. Sobald ein Knoten in H Ausgangsgrad 0 hat, lösche ebenfalls alle zu diesem führenden Kanten.
6. Wenn ein Knoten aktiviert wird, füge alle Kanten und Knoten hinzu, von denen aus er erreichbar ist.

Hierbei nehmen wir an, dass jeder Knoten im Laufe der Suche höchstens einmal aktiviert bzw. deaktiviert wird.

Definition 5.13 *Eine so konstruierte Suchstruktur werden wir als "graphbasiert" bezeichnen.*

Bemerkung 5.14
1. Es beeinträchtigt die Funktionsweise einer graphbasierten Suchstruktur nicht, wenn ein Knoten über mehrere verschiedene Wege erreicht werden kann.
2. Sind alle Knoten aus G in H über Pfade endlicher Länge zu erreichen, so ist die Suchstruktur eine effiziente Suchstruktur.

Beobachtung 5.15 *Ersetzt man nach dem Verfahren von Feder und Motwani [11] in einem von Partition B nach Partition A orientierten bipartiten Graphen G Cliquen durch gerichtete Sterne, so gilt für den resultierenden Graphen H, dass $G \subset H^*$.*
Die aus H auf die oben beschriebene Weise erzeugte Suchstruktur verhält sich genau so, wie die Suchstruktur aus Kapitel 4.3.1$_{85}$.

Beobachtung 5.16 *Es gibt Suchstrukturen, die eine bessere Beschleunigung erreichen, als das Verfahren von Feder und Motwani [11].*

Das einfachste Beispiel ist die "optimale" Suchstruktur: Diese besteht aus einer Abbildung, die jeder Teilmenge S von V eine Kante zuordnet, die über den Schnitt $(S, V \backslash S)$ führt. In jedem Schritt schlägt sie also eine Kante von einem besuchten zu einem noch nicht besuchten Knoten vor. Den Wechsel zwischen verschiedenen Teilmengen kann man über Zeiger in konstanter Zeit realisieren. Dies ist eine $(0, 0, 0)$-Suchstruktur.
Das explizite Erstellen dieser Abbildung in Form einer Tabelle kostet allerdings $\Omega(2^n)$ Schritte, was diese Suchstruktur nur nützlich werden läßt, wenn man wesentlich mehr algorithmische Suchen in dem Graphen durchführt, als dies im Matching-Kontext notwendig ist.

Teil II

Bipartite Graphen mit zahlreichen Matchings

Kapitel 6

Einleitung

Mit den im vorangegangenen Kapitel beschriebenen Verfahren können Matchings in nichtbipartiten Graphen genauso schnell gefunden werden, wie dies im bipartiten Fall möglich war. Beim Versuch noch schnellere Algorithmen zu entwickeln, konzentrieren wir uns daher wieder auf die bipartiten Graphen, da deren einfachere Struktur (keine sog. Blüten - vgl. Teil 1) die Komplexität der Aufgabe wesentlich reduziert.

Wir betrachten zunächst einen noch spezielleren Fall, den der sogenannten "regulären" bipartiten Graphen. Reguläre Graphen sind Graphen, in denen alle Knoten den gleichen Grad haben.
Vereinigt man r paarweise disjunkte perfekte Matchings in einem Graphen, so erhält jeder Knoten des Graphen von jedem Matching genau eine Kante. Es entsteht also ein regulärer Graph, in dem jeder Knoten den Grad r hat. Diese Graphen werden als "r-regulär" bezeichnet. Umgekehrt kann man zeigen, dass jeder r-reguläre bipartite Graph ein perfektes Matching hat (s. Lemma 7.13[136]), sich sogar stets in r perfekte Matchings zerlegen lässt [32]. Wegen dieses Zusammenhangs spielten die regulären Graphen bei der Entwicklung von Matching-Algorithmen stets eine Sonderrolle.

Cole, Ost und Schirra [6] ist es 1999 gelungen, einen Algorithmus zum Finden perfekter Matchings in regulären Graphen zu entwickeln, der nur lineare Laufzeit in der Zahl der Kanten, also Zeit $\mathcal{O}(m)$ benötigt.
Wir wollen nun versuchen, diesen Algorithmus auf weitere Typen von bipartiten Graphen zu übertragen: Wenn man zu einem r-regulären Graphen weitere Kanten hinzufügt, werden die r bereits enthaltenen Matchings nicht zerstört. Betrachtet man einen Graphen G, der aus einem regulären Graphen durch das Hinzufügen von Kanten entstanden ist, so bilden diese Matchings zusammen einen sogenannten "*spannenden*" regulären Teilgraphen, d.h. einen regulären Teilgraphen, der alle Knoten des ursprünglichen Graphen enthält. Damit ist jedes perfekte Matching in einem solchen Teilgraphen automatisch auch ein perfektes Matching in G.
Könnte man einen solchen Teilgraphen identifizieren, so könnte man mit dem Algorithmus von Cole, Ost und Schirra [6] in diesem Teilgraphen - und somit in G - ein perfektes Matching in Linearzeit konstruieren.

Wir stellen in diesem Teil der Arbeit einen Algorithmus vor, der sich an der eben beschriebenen Idee orientiert und auf Graphen mit ausreichend großen regulären Teilgraphen ein besseres Laufzeitverhalten zeigt als der beste Algorithmus für den allgemeinen bipartiten Fall:

Satz 6.1 *In einem bipartiten Graphen mit $2n$ Knoten und m Kanten, der $\sigma^3\sqrt{n}$ paarweise disjunkte perfekte Matchings enthält, kann ein perfektes Matching in Zeit $\mathcal{O}\left(\frac{\sqrt{n}m}{\sigma}\right)$ gefunden werden.*

Kapitel 7

Beschreibung des Algorithmus

7.1 Übersicht

Die naheliegendste Idee zur Übertragung des Algorithmus von Cole, Ost und Schirra für reguläre Graphen [6] auf allgemeinere bipartite Graphen wäre, im gegebenen Graphen einen regulären spannenden Teilgraphen zu suchen. Ein perfektes Matching in diesem Teilgraphen könnte in Linearzeit bestimmt werden und wäre, da der Teilgraph spannend ist, auch ein perfektes Matching im ursprünglichen Graphen. Leider dauert es mit allen uns bekannten Verfahren genauso lange, einen solchen Teilgraphen zu finden, wie es dauern würde, direkt ein Matching zu bestimmen.

Daher müssen wir mit einer Näherungslösung arbeiten. Diese erhalten wir, indem wir einen eigentlich zu langsamen Algorithmus zum Finden eines solchen Teilgraphen vorzeitig abbrechen. Der resultierende Teilgraph H ist somit nicht ganz regulär und eventuell nicht spannend.
Um dennoch den Algorithmus von Cole, Ost und Schirra [6] anwenden zu können, müssen wir diesen Teilgraph regulär machen. Dies ist innerhalb des Graphen G nicht einfach möglich. Daher betrachten wir den spannenden Teilgraphen H nun vorübergehend als eigenständigen Graphen, in dem ein Matching bestimmt werden soll.
Um diesen nichtregulären Graphen H dennoch regulär zu machen, ist es notwendig, zu H einige Knoten und Kanten hinzuzufügen. Auf diese Weise entsteht ein neuer regulärer Graph R, von dem H nun wieder ein Teilgraph ist.
Bevor wir überlegen, wie diese Vergrößerung vorgenommen wird, betrachten wir kurz die weitere Verwendung dieses Graphen, da sich daraus die genaue Vorgehensweise bei der Vergrößerung ergibt:
In dem durch die Vergrößerung von H entstandenen regulären Graphen R kann mit dem Algorithmus von Cole Ost und Schirra [6] ein Matching M in Linearzeit bestimmt werden (vgl. Abb. 7.1).

Das Matching M von R kann nun aber auch Kanten enthalten, die nicht zum Teilgraphen H von R gehören. Wir können aber nur Kanten aus H in den ursprünglichen Graphen G zurück übertragen, da nur diese Kanten von R aus G stammen.
Wenn wir das Matching also in den ursprünglichen Graphen G zurück übertragen wol-

len, so müssen wir diese *falschen Matchingkanten* weglassen. Damit bleiben diejenigen Knoten in H - und somit später auch in G - unüberdeckt, deren Matchingpartner zu $V(R)\backslash V(H)$ gehören. D.h. wir müssen dann nochmals auf anderem Wege, z.B. mittels augmentierender Pfade, versuchen, das Matching zu vergrößern. Wir müssen daher dafür sorgen, dass nur ein geringer Anteil der Kanten, die zu H hinzugefügt werden, um R zu bilden, in das Matching aufgenommen wird. Wir werden im Folgenden die Kanten, die zu R aber nicht zu H gehören, als *falsche Kanten* bezeichnen. Das Ziel unserer weiteren Überlegungen ist es, die Konstruktion des Graphen R aus dem Graphen H so vorzunehmen, dass möglichst wenige der falschen Kanten zu falschen Matchingkanten werden (vgl. Abb 7.1).

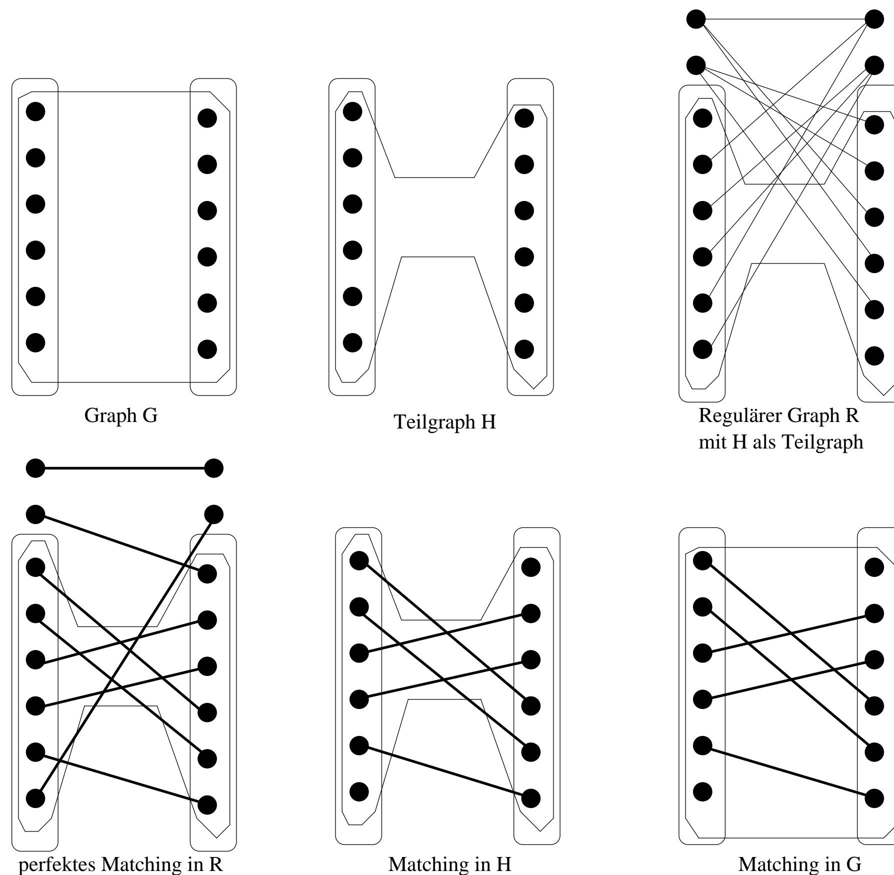

Abbildung 7.1: Veranschaulichung der Vorgehensweise

Von den jeweils zu einem Knoten inzidenten Kanten kann jeweils nur eine in ein Matching aufgenommen werden. Von vielen falschen Kanten, die zum selben Knoten inzident sind, kann also nur eine einzige eine falsche Matchingkante werden. Die Zahl der möglichen falschen Matchingkanten ist also begrenzt durch die Zahl der Knoten, zu denen falsche Kanten inzident sind.
Da alle Knoten in R den gleichen Grad r' haben sollen, ist die Zahl der Knoten, zu denen falsche Kanten inzident sind, umso geringer, je mehr Knoten es gibt, zu denen ausschließlich falsche Kanten führen. Wir fügen also neue Knoten ein, von denen jeder zu r' falschen

Kanten inzident ist.
Das garantierbare Verhältnis zwischen falschen Kanten und falschen Matchingkanten ist somit gleich dem Grad r' der zusätzlichen Knoten.
Je höher dieser Grad r' aber sein soll, desto höher muss auch der Grad der Knoten in dem fast regulären Subgraphen H sein, da nachher alle Knoten den gleichen Grad haben müssen. Um den Algorithmus anwenden zu können, muss der Ausgangsgraph also einen nahezu regulären Teilgraphen von möglichst hohem Grad enthalten.
Um eine erste einfache Formulierung des Sachverhaltes zu ermöglichen, verschärfen wir diese Forderung hier zur Anforderung, dass der Graph sogar einen wirklichen regulären Teilgraphen von hohem Grad enthält. Dies ist bei bipartiten Graphen genau dann der Fall, wenn er viele disjunkte perfekte Matchings enthält. Verallgemeinerungen werden wir im Anschluss betrachten.

Zusammenfassend geht unser Algorithmus wie folgt vor, um ein perfektes Matching in einem Graphen G zu finden:

1. (*Teil 1*) Finde einen annähernd regulären und möglichst spannenden Teilgraph H von G

2. Betrachte H als eigenständigen Graphen

3. (*Teil 2*) Ergänze H zu einem regulären Graphen $R \supset H$

4. Finde in R ein perfektes Matching M

5. Interpretiere die Kanten aus M, die auch zu H und somit zu G gehören, als Matching M' in G

6. (*Teil 3*) Vergrößere M' zu einem perfekten Matching

Das wichtigste Objekt unserer Betrachtungen in diesem Kapitel wird also der "annähernd reguläre" und "möglichst spannende Teilgraph" H sein. Wir müssen daher exakt festlegen, was "annähernd regulärer spannender Teilgraph" bedeuten soll. Wir verwenden hierzu die Zahl der Kanten d, die zusätzlich zu den Knoten inzident sein müssten, damit der Teilgraph r-regulär und spannend wäre:

Definition 7.1 *Ein Teilgraph H eines bipartiten Graphen $G = (V, E)$ heißt (r, d)-regulär, wenn er die folgenden zwei Bedingungen erfüllt:*

$$\begin{aligned} \delta_H(v) &\leq r \ \forall v \in V \\ \sum_{v \in V} (r - \delta_H(v)) &\leq d \end{aligned} \tag{7.1}$$

Hierbei bezeichnet $\delta_H(v)$ den Grad des Knotens v innerhalb des Teilgraphen H. Wir nehmen zur Vereinfachung einiger Rechnungen an, dass im betrachteten Graphen $G = (V, E) = (A \dot{\cup} B, E)$ mit $|A| = |B| = n$ gilt, der Graph also insgesamt $|V| = 2n$ Knoten hat. Ein perfektes Matching hat demnach die Größe n.

7.2 Finden eines nahezu regulären Subgraphen

Satz 7.2 *In einem bipartiten Graphen G mit $2n$ Knoten und m Kanten, der einen r-regulären Teilgraphen hat, kann ein (r, d)-regulärer Teilgraph mit $d = 2\left(\frac{n}{c}\right)^2$ in Zeit $\mathcal{O}(cm)$ gefunden werden.*

Beweis:

Zunächst beschreiben wir einen auf Netzwerkflüssen basierenden Algorithmus zum Finden eines spannenden regulären Subgraphen. Danach zeigen wir, dass man diesen Algorithmus nach wenigen Phasen abbrechen kann und so einen (r, d)-regulären Teilgraph erhält.

Transformation in ein Flussproblem Wir haben in Kapitel 1.4.2[13] gesehen, wie man Flussalgorithmen einsetzen kann, um Matchings in bipartiten Graphen zu finden: Wir erzeugen ein Flussnetzwerk, indem wir

1. eine Quelle und eine Senke hinzufügen,
2. alle Kanten von Partition B nach Partition A richten,
3. die Quelle mit allen Knoten aus B verbinden,
4. alle Knoten aus A mit der Senke verbinden
5. und alle Kapazitäten auf 1 setzen.

Wir haben dabei beobachtet, dass bei einem maximalen Fluss die flusstragenden Kanten zwischen den beiden Partitionen ein maximales Matching bilden. Ist dieses perfekt, so kann es auch als ein 1-regulärer spannender Teilgraph interpretiert werden.

Um einen r-regulären spannenden Teilgraphen zu finden, gehen wir vollkommen analog vor:
Alle Kanten werden wiederum von B nach A gerichtet und erhalten die Kapazität 1. Wir fügen einen Quellknoten s und einen Senkenknoten t hinzu und verbinden s jeweils über eine Kante der Kapazität r mit allen Knoten b aus B und alle Knoten a aus A mit Kanten der Kapazität r mit t (vgl. Abb. 7.2). Das auf diese Weise erzeugte Netzwerk bezeichnen wir mit N_G.

Lemma 7.3 *Hat G einen r-regulären spannenden Teilgraphen, so hat der maximale Fluss im Netzwerk N_G die Größe rn.*

> **Beweis:** Setze den Fluß durch jede Kante in N_G, die zum regulären Teilgraphen gehört, auf 1 und den Fluß der zur Quelle oder Senke inzidenten Kanten auf r. Da die aus der Quelle führenden Kanten damit gesättigt sind, kann es keinen größeren Fluß geben. In jeden Knoten ausser s und t fließen genau r

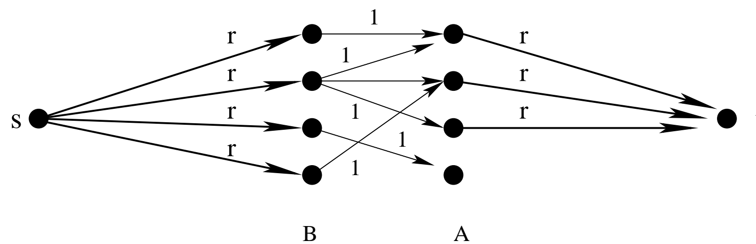

Abbildung 7.2: Das Finden eines regulären Teilgraphen als Flussproblem

Flusseinheiten hinein und ebensoviele wieder aus ihm heraus, da r Kanten mit Flusswert 1 zu ihm inzident sind. □

Lemma 7.4 *Ist g ein maximaler ganzzahliger Fluss mit Größe rn in Netzwerk N_G, so bilden die Kanten zwischen A und B mit positivem Flusswert einen r-regulären Teilgraphen.*

Beweis: Bei Flussstärke rn müssen alle Kanten zwischen s und B bzw. zwischen A und t gesättigt sein, da jede der n Kanten die Kapazität r hat. Insbesondere muss also jeder Knoten v von genau r Flusseinheiten passiert werden. Da aber alle ausgehenden Kanten aus den Knoten in B bzw. eingehenden Kanten der Knoten in A die ganzzahlige Kapazität 1 haben, gibt es auf diesen Kanten nur die Flusswerte 0 und 1. Es gibt also - abgesehen von der Verbindung zur Quelle bzw. Senke - zu jedem Knoten genau r inzidente Kanten mit Flusswert 1. Die anderen Kanten haben Flusswert 0. Es gibt also zu jedem Knoten in $V(G) = V(N)\backslash\{s,t\}$ genau r inzidente Kanten mit positivem Flusswert. □

Um in einem Graphen G, der einen r-regulären Teilgraphen hat, die Kanten dieses Teilgraphen zu identifizieren, müssen wir also einen maximalen ganzzahligen Fluss im zugehörigen Netzwerk N_G konstruieren. Dies ist mit dem Algorithmus von Dinic möglich:

Satz 7.5 ([35]) *Sind bei einem Netzwerk N alle Kapazitäten ganzzahlig, so findet der Algorithmus von Dinic stets ganzzahlige Flüsse.*

Beweisskizze: Der Algorithmus startet mit ganzzahligen Werten

$$f(e) = 0 \quad \forall e \in E.$$

Wir zeigen, dass die Werte bei jeder Augmentierung ganzzahlig bleiben:
Ist der gegebene Fluss ganzzahlig, so ergeben sich alle Kapazitäten des residualen Netzwerks durch Addition und Subtraktion aus den ganzzahligen Kapazitäten von N und den ganzzahligen Flusswerten. Dann ist aber auch der Betrag ganzzahlig, um den der Fluss entlang eines augmentierenden Pfades vergrößert wird. Damit ist der Fluss auch nach einem Augmentierungsschritt noch ganzzahlig. Die Aussage folgt somit per Induktion über die Zahl der Augmentierungsschritte. ◊

Der allgemeine Netzwerkfluss-Algorithmus von Dinic [8] hat eine Laufzeit von $\mathcal{O}(n^2m)$ und ist somit viel zu langsam, um zum Finden eines Matchings eingesetzt werden zu können. Er basiert - wie in Kapitel $1.4.2_{13}$ beschrieben - darauf, bis zu n mal nicht erweiterbare Mengen augmentierender Pfade zu finden und dann über diese zu augmentieren.
Eine solche Menge in allgemeinen Netzwerken zu finden, kann bis zu nm Operationen kosten. In einem Einheits-Kapazitäts-Netzwerk, in dem alle Kapazitäten 0 oder 1 sind, benötigt jede Phase des Algorithmus von Dinic nur lineare Zeit $\mathcal{O}(m)$, da jede in einem augmentierenden Pfad verwendete Kante sofort vollständig blockiert ist und nicht mehrfach verwandt werden kann (vgl. [35]).
Daher können wir eine erste Verbesserung erzielen, indem wir das Netzwerk N_G wie folgt in ein Einheits-Kapazitäts-Netzwerk umwandeln:
Wir ersetzen jede Kante (s, b), bzw. (a, t) mit Kapazität r durch r parallele Kanten mit Kapazität 1. Wir bezeichnen jede dieser Mengen paralleler Kanten als *Multikante*. Es werden damit $2n(r-1)$ Kanten hinzugefügt, was in $\mathcal{O}(m)$ liegt, da der ursprüngliche Graph nach Voraussetzung einen r-regulären Teilgraphen mit nr Kanten enthält, m also größer als nr ist. Somit könnten wir in diesem modifizierten Netzwerk N'_G einen maximalen Fluss in Zeit $\mathcal{O}(nm)$ bestimmen. Dies ist aber immer noch zu langsam.

Finden eines nahezu maximalen Flusses: Wollen wir den Algorithmus von Dinic im Rahmen eines Algorithmus einsetzen, der Matchings schneller bestimmt, als dies bisher möglich war, so muss dies weniger als $\mathcal{O}(\sqrt{n}m)$ Zeit in Anspruch nehmen. Wir müssen also mit weniger als $\sqrt{n}$ Phasen auskommen. Im Folgenden untersuchen wir was passiert, wenn man den Algorithmus vorzeitig abbricht.
An dieser Stelle erinnern wir nochmals an die Definition des residualen Netzwerks aus Kapitel $1.4.2_{13}$: Die Knotenmenge des residualen Netzwerks besteht aus allen Knoten des Graphen. Die Kantenmenge enthält die Kanten, über die noch Fluss geschickt werden kann, und die flusstragenden Kanten mit umgekehrter Richtung, da der Fluss auf diesen Kanten reduziert werden kann, also gewissermaßen Fluss zurückgeschickt werden könnte.
Es ist genau dann noch möglich den Fluss weiter zu vergrößern, wenn es im residualen Netzwerk noch einen gerichteten Pfad von der Quelle zur Senke gibt. Der Abstand eines Knotens zur Quelle s im residualen Netzwerk entspricht der Länge eines kürzesten Pfades, über den noch Fluss zu dem Knoten geschickt werden kann. Entsprechend ist der Abstand von s zu t im residualen Netzwerk gleich der Länge eines kürzesten augmentierenden Pfades.
Bei Anwendung des Algorithmus von Dinic vergrößert sich nach jeder Phase die Länge eines kürzesten augmentierenden Pfades um 1. Es nimmt also auch der Abstand zwischen s und t in jeder Phase um 1 zu.
Wir wenden nun den Algorithmus von Dinic auf das gegebene Netzwerk an und brechen ihn nach $4c + 1 \leq \sqrt{n}$ Phasen ab (Die Wahl von $4c + 1$ begründen wir im nächsten Kapitel.).
Mit Glück ist der Algorithmus in dieser Zahl von Phasen fertig und wir haben einen maximalen Fluss gefunden. Sollte dies nicht der Fall sein, wissen wir zumindestens, dass die

verbleibenden augmentierenden Pfade mindestens die Länge $4c + 1 \leq \sqrt{n}$ haben. Dieses Vorgehen garantiert uns also das folgende Ergebnis:

Lemma 7.6 *In einem Einheits-Kapazitäts-Netzwerk, in dem alle Kapazitäten 0 oder 1 sind, können in $\mathcal{O}(cm)$ entweder ein maximaler Fluss oder ein Fluß und eine Knotennummerierung $l : V \to \mathbb{Z}^+$ gefunden werden, so daß $l(s) = 0, l(t) \geq 4c + 1$ und $l(j) \leq l(i) + 1$ für jede Kante (i, j) des residualen Netzwerks gilt.*

> **Beweis:** Jede Phase des Algorithmus von Dinic [8] verändert das jeweils gegebene Matching so, dass die Länge eines kürzesten augmentierenden Pfades um mindestens 1 zunimmt (zu Details siehe Kapitel 1.4.2$_{13}$). Nach $4c+1$ Phasen hat also ein kürzester Pfad in dem so konstruierten Matching mindestens die Länge $4c + 1$. Sollte bereits vorher keine Vergrößerung des Flusses mehr möglich sein, so wurde bereits ein maximaler Fluss und somit ein r-regulärer Teilgraph gefunden.
> Die Knotennummerierung l ergibt sich als die Länge eines kürzesten alternierenden Pfades von s zu den jeweiligen Knoten bzw. als Abstand der Knoten von s im residualen Netzwerk. Diese Abstände werden vom Algorithmus von Dinic in jeder Phase bestimmt. Die Eigenschaft, dass $l(j) \leq l(i) + 1$ für jede Kante (i, j) gilt, ist ebenfalls erfüllt: $l(j)$ entspricht dem Abstand von s zu j und die Länge des kürzesten Pfades von s zu j kann höchstens $l(i) + 1$ betragen, da es einen Pfad der Länge $l(i) + 1$ über i gibt. Jede Phase nimmt in einem Einheits-Kapazitäts-Netzwerk $\mathcal{O}(m)$ Schritte in Anspruch, so dass insgesamt $\mathcal{O}((4c + 1)m) = \mathcal{O}(cm)$ Schritte benötigt werden. □

Existenz eines Schnitts beschränkter Größe: Nun müssen wir zeigen, dass der gefundene Fluss trotz des vorzeitigen Abbruchs des Algorithmus schon nahezu ein maximaler Fluss ist.
Die Kapazität $K(C)$ eines $s - t$-Schnittes C im residualen Netzwerk ist stets eine obere Schranke für die maximale weitere Vergrößerung des vorliegenden Flusses (vgl. z.B. [35]). Sie gibt also auch an, um wieviel ein maximaler Fluss größer sein kann als der momentan betrachtete Fluss. Da wir nach Konstruktion wissen, dass ein maximaler Fluss die Größe rn hat, wissen wir also, dass der Gesamtfluß größer als $rn - K(C)$ sein muss, wobei C ein beliebiger $s - t$-Schnitt im residualen Netzwerk ist.
Wir werden nun im residualen Netzwerk die Existenz eines gerichteten Schnitts der Kapazität $\left(\frac{n}{c}\right)^2$ nachweisen.

Wir bezeichnen alle Knoten v mit $l(v) = i$ als das $i$te Level. Kanten im residualen Netzwerk, die zu einem höheren Level führen, können wegen der Nebenbedingungen für die Knotennummern immer nur zum $(i + 1)$ten Level führen. Ein Schnitt wird also z.B. gebildet durch alle Knoten, die zu den Leveln 0 bis j gehören. Über den Schnitt führen entsprechend genau die Kanten, die vom j-ten zum $(j + 1)$-ten Level führen (vgl. Abb 7.3).

Lemma 7.7 *Es gibt zwei aufeinanderfolgende Level $i, i+1$ mit $l(s) < i$ und $i+1 < l(t)$, die beide weniger als $\frac{n}{c}$ Knoten enthalten.*

Beweis: Der Graph wird durch die Knotennummern in $4c + 2$ Level zerlegt. Zwischen Level 0 und Level $l(t) - 1$ einschließlich befinden sich also $4c$ Level. Diese lassen sich zu $2c$ Paaren von Leveln zusammenfassen. Da der Graph insgesamt nur $2n$ Knoten enthält, muss es ein Paar geben, das weniger als $\frac{2n}{2c}$ Knoten enthält (vgl. Abb 7.3). Dies bedeutet aber auch, dass jedes der beiden Level weniger als $\frac{n}{c}$ Knoten enthält. □

Seien diese beiden Level mit i und $i + 1$ bezeichnet.

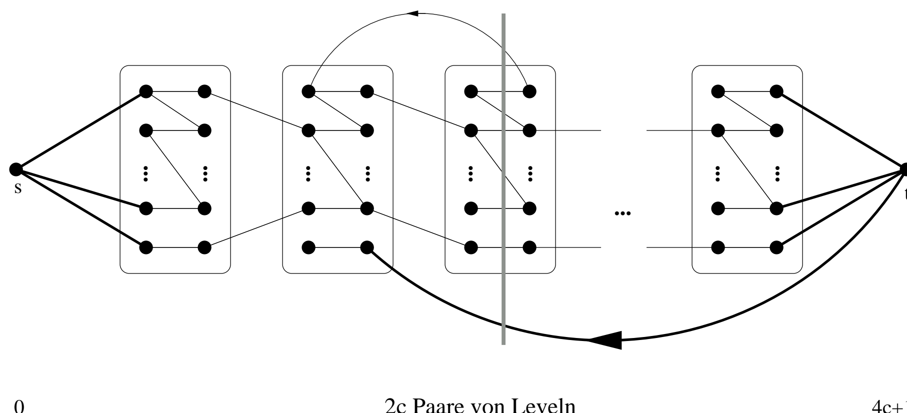

Abbildung 7.3: Schnitt im residualen Netzwerk(Multikanten sind dick dargestellt)

Lemma 7.8 *Es führt keine Multikante von Level $i + 1$ nach Level i.*

Beweis: Alle Multikanten sind mit s oder t verbunden. Nach Konstruktion können aber s und t weder zu Level i noch zu Level $i + 1$ gehören. □

Lemma 7.9 *Der Schnitt im residualen Netzwerk zwischen Level $i + 1$ und Level i hat höchstens Kapazität $\left(\frac{n}{c}\right)^2$.*

Beweis: Die Zahl der Kanten zwischen den beiden Leveln kann höchstens $\left(\frac{n}{c}\right)^2$ betragen, da maximal alle Knoten beider Level miteinander verbunden sein können. Jede dieser Kanten hat Kapazität höchstens 1, da alle Multikanten zu s oder t inzident sind, diese aber nicht zu den beiden Leveln gehören können. □

Aus den obigen Ausführungen folgt unmittelbar:

Lemma 7.10 *Der gefundene Fluss g hat mindestens die Größe $\left(\frac{1}{2}\sum_{v \in V} r\right) - \left(\frac{n}{c}\right)^2$.*

Da alle Kapazitäten 0 oder 1 und somit ganzzahlig sind, liefert der Algorithmus von Dinic einen ganzzahligen Fluss. Alle Flusswerte sind also 0 oder 1.
Zum Beweis von Satz 7.2[130] bleibt zu zeigen, dass die flusstragenden Kanten in der gewünschten Weise einen nahezu regulären, nahezu spannenden Teilgraphen bilden:

Lemma 7.11 *Der Subgraph H mit $(b,a) \in H :\Leftrightarrow g((b,a)) = 1$ ist ein (r,d)-regulärer Teilgraph von G mit $d \leq 2\left(\frac{n}{c}\right)^2$.*

Beweis: Die Flusserhaltungsbedingung - hier bereits spezialisiert auf den vorliegenden Fall -

$$\forall b \in B: \quad \sum_{(s,b)\in E} g((s,b)) = \sum_{(b,a)\in E} g((b,a))$$

liefert wegen der Ganzzahligkeit des gefundenen Flusses:

$$\forall b \in B: \quad r = \sum_{(s,b)\in E} c((s,b)) \geq \sum_{(s,b)\in E} g((s,b)) = \sum_{(b,a)\in E, g((b,a))=1} 1 = \delta_H(b).$$

Das gleiche gilt analog für die Knoten aus A.
Außerdem gilt wegen $\sum_{a\in A} \delta_H(a) = \sum_{b\in B} \delta_H(b) = \frac{1}{2}\sum_{v\in V} \delta_H(v)$

$$\frac{1}{2}\sum_{v\in V} \delta_H(v) = \sum_{(a,b)\in E, a\in A, B\in B} g((a,b)) \geq nr - \left(\frac{n}{c}\right)^2.$$

Multiplikation mit 2 liefert

$$\sum_{v\in V} \delta_H(v) \geq 2nr - 2\left(\frac{n}{c}\right)^2$$

und wegen $|V| = 2n$ folgt

$$\sum_{v\in V} \delta_H(v) \geq \left(\sum_{v\in V} r\right) - 2\left(\frac{n}{c}\right)^2 \Leftrightarrow \sum_{v\in V}(r - \delta_H(v)) \leq 2\left(\frac{n}{c}\right)^2.$$

$\square_{S7.2}$ $\square_{S7.2}$

7.3 Verwenden eines bekannten annähernd regulären Teilgraphen

Im vorangegangenen Abschnitt haben wir beschrieben, wie man in einem bipartiten Graphen G, der einen r-regulären spannenden Teilgraphen enthält, die Kanten eines (r,d)-regulären Teilgraphen H mit $d = 2\left(\frac{n}{c}\right)^2$ in Zeit $\mathcal{O}(cm)$ bestimmen kann. In diesem Abschnitt zeigen wir, wie man diese zusätzliche Information über den Graphen G verwenden kann, um in G ein perfektes Matching zu bestimmen.

Lemma 7.12 *Sei $G = (V,E)$ ein bipartiter Graph mit $|V| = 2n$ Knoten und $|E| = m$ Kanten. Sei H ein bekannter (r,d)-regulärer Subgraph von G. Dann kann man ein maximales Matching von G in Zeit $\mathcal{O}\left(\left(1+\frac{d}{r}\right)m\right)$ finden.*

Beweis:

Ergänzung zu einem regulären Graphen: Wir betrachten im folgenden den Teilgraphen H von G als eigenständigen Graphen. Wir ergänzen H zu einem r-regulären bipartiten Multigraphen $R \supset H$ (vgl Abb. 7.1128). Hierzu fügen wir zu A eine Menge A' und zu B eine Menge B' von jeweils $\left\lceil \frac{d}{2r} \right\rceil$ Knoten hinzu.
Danach verbinden wir beliebig Knoten aus $A \backslash A'$ mit Knoten aus B', solange jeweils beide Knoten einen kleineren Grad als r haben. Wenn dies nicht mehr möglich ist, haben alle Knoten aus $A \backslash A'$ Grad r, da diesen Knoten insgesamt $\frac{d}{2}$ Kanten fehlten, in B' aber $r \left\lceil \frac{d}{2r} \right\rceil$ Kanten enden können. Das gleiche führen wir analog für $B \backslash B'$ und A' durch. Danach haben alle Knoten in $A \cup B \backslash (A' \cup B')$ Grad r. Da der Graph bipartit ist, gilt $\sum_{a \in A \cup A'} \delta(a) = \sum_{b \in B \cup B'} \delta(b)$. Da andererseits aber auch $|A| = |B|$ sowie $|A'| = |B'|$ gilt, folgt

$$\sum_{a \in A'} \delta(a) = \left(\sum_{a \in A' \cup A} \delta(a) \right) - r|A| = \left(\sum_{b \in B' \cup B} \delta(b) \right) - r|B| = \sum_{b \in B'} \delta(b),$$

d.h. den Knoten in A' fehlen genausoviele Kanten wie denen aus B'. Entsprechend können wir $r|A'| - \sum_{a \in A'} \delta(a)$ Kanten zwischen A' und B' beliebig einfügen, so dass der Graph insgesamt regulär wird.
Es kann sein, dass in einem der beiden Schritte mehrere Kanten zwischen zwei Knoten eingeführt werden müssen, also ein Multigraph entsteht.

Konstruktion des Matchings: In dem so konstruierten regulären Graphen R gibt es ein perfektes Matching M:

Lemma 7.13 (König 1916 [25]) *Jeder reguläre bipartite Graph hat ein perfektes Matching.*

Zum Beweis siehe Lemma 7.14138.
Ein solches Matching kann nun mit dem Algorithmus von Cole, Ost und Schirra [6] in Linearzeit gefunden werden. Hierzu bedarf es noch einer Überprüfung, dass der Algorithmus auch auf Multigraphen angewandt werden kann. Dies zeigen wir in Kapitel 7.5.

Übertragung auf den Ausgangsgraphen: Nun entfernen wir aus M alle Kanten, die zu einem Knoten aus A' oder B' inzident sind. Da zu jedem Knoten aus $A' \cup B'$ höchstens eine Matchingkante inzident sein kann, werden höchstens $|A' \cup B'| = 2 \left\lceil \frac{d}{2r} \right\rceil$ Kanten aus M gelöscht. Die verbleibenden Kanten aus M gehören alle zum Graphen H, der nach Konstruktion sowohl ein Teilgraph von R als auch ein Teilgraph von G ist.
Interpretiert man diese Kanten als Matching im ursprünglichen Graphen G, so erhält man ein Matching der Größe mindestens $n - 2 \left\lceil \frac{d}{2r} \right\rceil$ in G. In diesem Graphen muss man also, um das gegebene Matching zu einem maximalen zu vergrößern, höchstens $\mathcal{O}(d/r)$ mal mit Tiefensuche nach einem augmentierenden Pfad suchen und über diesen augmentieren. Die dafür benötigte Laufzeit ist entsprechend in $\mathcal{O}(md/r)$. $\square_{\text{L7.12}}$

7.4 Ausbalancieren der beiden Teile

Wir betrachten einen bipartiten Graphen G mit r perfekten Matchings. In den beiden vorangegangenen Kapiteln haben wir gesehen, dass man in einem solchen Graphen

1. einen $\left(r, 2\left(\frac{n}{c}\right)^2\right)$-regulären Teilgraphen in Zeit $\mathcal{O}(cm)$ finden kann und
2. ein perfektes Matching in Zeit $\mathcal{O}\left(\left(1+\frac{d}{r}\right)m\right)$ finden kann, wenn man einen (r,d)-regulären Teilgraphen kennt.

Wir setzen entsprechend $d := 2\left(\frac{n}{c}\right)^2$ und führen die beiden in den vorangegangenen Abschnitten beschriebenen Verfahren hintereinander aus. Die Konstante 2 können wir im Rahmen der $\mathcal{O}$-Notation vernachlässigen. Wir erhalten eine Gesamtlaufzeit von $\mathcal{O}(cm + m\frac{n^2}{c^2 r})$.
‚Da der erste Summand mit c wächst und der zweite mit c fällt, wird die asymptotische Gesamtlaufzeit minimal, wenn beide gleich sind. Wir erhalten den optimalen Wert für c also als Lösung von

$$c = \frac{n^2}{c^2 r} \Leftrightarrow c^3 = \frac{n^2}{r}.$$

Dies führt zu $c = \sqrt[3]{\frac{n^2}{r}}$. Damit liegt die Gesamtlaufzeit in $\mathcal{O}\left(m\sqrt[3]{\frac{n^2}{r}}\right)$.
Um eine bessere Vergleichbarkeit mit der besten Schranke im allgemeinen Fall zu erlauben, isolieren wir noch den Faktor $\sqrt{n}$ und erhalten $\mathcal{O}\left(\frac{mn^{\frac{1}{2}}n^{\frac{1}{6}}}{\sqrt[3]{r}}\right) = \mathcal{O}\left(\frac{m\sqrt{n}}{\sqrt[3]{\frac{r}{\sqrt{n}}}}\right)$.
Mit $\sigma := \sqrt[3]{\frac{r}{\sqrt{n}}} \Leftrightarrow r = \sigma^3\sqrt{n}$ ergibt sich somit der zu Beginn dieses Teils angekündigte
Satz 6.1:
In einem bipartiten Graphen mit $2n$ Knoten und m Kanten, der $\sigma^3\sqrt{n}$ disjunkte perfekte Matchings enthält, kann ein perfektes Matching in Zeit $\mathcal{O}\left(\frac{\sqrt{n}m}{\sigma}\right)$ gefunden werden.
Unser Verfahren ist asymptotisch schneller als der $\mathcal{O}(\sqrt{n}m)$ Algorithmus von Hopcroft und Karp, wenn σ mit n wächst, also nicht konstant ist:

$$\sigma \in \omega(1) \Leftrightarrow \sqrt[3]{\frac{r}{\sqrt{n}}} \in \omega(1) \Leftrightarrow r \in \omega(\sqrt{n}).$$

D.h. die Zahl der enthaltenen Matchings r muss schneller wachsen als $\sqrt{n}$. Beschränken wir uns beispielsweise auf die Klasse der Graphen mit mehr als $\log^3 n\sqrt{n}$ Matchings, so hat unser Algorithmus eine asymptotische Laufzeit in $\mathcal{O}\left(\frac{\sqrt{n}m}{\log n}\right)$.

7.5 Der Algorithmus von Cole, Ost und Schirra

Wir müssen nun noch zeigen, dass der Algorithmus von Cole, Ost und Schirra auch auf Multigraphen anwendbar ist. Wir tun dies etwas ausführlicher, da wir Details des Algorithmus auch im nächsten Kapitel benötigen werden.

Die Grundidee des Algorithmus läßt sich mit dem folgenden Lemma erfassen, das man als Verallgemeinerung von Lemma 7.13 ansehen kann:

Lemma 7.14 *Sei $G = (V, E)$ ein bipartiter Graph und $f : E \to \mathbb{R}_0^+$ eine Gewichtsfunktion. f erfülle die Bedingung, dass die Summe der Werte der zu einem Knoten inzidenten Kanten stets gleich ist: $\sum_{e \ni v} f(e) = r \quad \forall v \in V$. Dann hat G ein perfektes Matching.*

Beweis: Der Beweis baut auf dem Theorem von Hall auf:

Satz 7.15 (Hall 1935 [19]) *Sei $G = (A \dot{\cup} B, E)$ ein bipartiter Graph. G hat genau dann ein Matching, das A überdeckt, wenn für alle Teilmengen X von A gilt, dass die Nachbarschaft von X mindestens genauso groß ist, wie X, also $|\Gamma(X)| \geq |X| \quad \forall X \subset A$.*

Die Notwendigkeit der Bedingung ist unmittelbar einsichtig. Der Beweis, dass diese Bedingung hinreichend ist, findet sich z.B in [28].
Der Satz gilt aus Symmetriegründen natürlich ebenso für die Partition B.
Sei X eine beliebige Teilmenge einer der beiden Partitionen. Die Nachbarschaft $\Gamma(X)$ von X muss alle Kanten “empfangen”, die von X ausgehen. Diese Kanten haben zusammen einen Wert von $r|X|$. Da in jedem Knoten in $\Gamma(X)$ nur Kanten mit Gesamtwert r enden können, können dies also nicht weniger Knoten als $|X|$ sein:

$$r|X| = \sum_{x \in X, y \in \Gamma(x)} f((x, y)) \leq \sum_{y \in \Gamma(X), z \in \Gamma(y)} f((z, y)) = r|\Gamma(X)|$$

$$\Leftrightarrow |X| \leq |\Gamma(X)|$$

Es gilt also für alle Teilmengen X, dass $|X| \leq |\Gamma(X)|$. Damit ist die Voraussetzung des Satzes von Hall erfüllt. □

Diese Überlegungen gelten natürlich auch, wenn man die Kanten ignoriert, die den Wert Null haben. Dies tun wir expliziter:

Definition 7.16 *Seien $G = (V, E)$ ein Graph und $f : E \to \mathbb{R}_0^+$ eine Bewertung der Kanten. Dann definieren wir $G_+ := (V, E_+)$ mit der Menge der positiv bewerteten Kanten $E_+ := \{e \in E \mid f(e) > 0\}$ als den* bezüglich f positiv bewerteten Teilgraphen *von G. Ist die Funktion f kanonisch gegeben, so verzichten wir auf deren explizite Erwähnung.*

Korollar 7.17 *Sei $G = (V, E)$ ein bipartiter Graph und $f : E \to \mathbb{R}_0^+$ eine (Gewichts-) Funktion, die die folgende Bedingung erfüllt: $\sum_{v \in e} f(e) = r \quad \forall v \in V$. Dann hat G_+ ein perfektes Matching.*

Beweis: Der Beweis verläuft analog zu dem von Lemma 7.14, da die Kanten mit Gewicht 0 in der Rechnung keine Rolle spielen. □

Gelingt es uns, zu einem gegebenen Graphen eine Kantenbewertung zu finden, die $\sum_{v \in e} f(e) = r \quad \forall v \in V$ erfüllt und bei der viele Kanten den Wert Null haben, so bilden die Kanten mit positivem Wert einen Teilgraphen, der ein perfektes Matching enthält, aber weniger Kanten hat. Auf diesem Teilgraphen werden die meisten Matching-Algorithmen schneller ablaufen als auf dem Originalgraphen. Im Idealfall, wenn nur n Kanten einen positiven Wert haben, bildet dieser Teilgraph bereits ein Matching.
Um einen solchen Graphen zu konstruieren, gehen wir wie folgt vor: Wir beginnen mit einer Bewertung, die $\sum_{v \in e} f(e) = r \quad \forall v \in V$ erfüllt.Wir versuchen, diese schrittweise so zu ändern, dass mehr und mehr Kanten den Wert 0 bekommen, wobei $\sum_{v \in e} f(e) = r \quad \forall v \in V$ als Invariante erhalten bleibt.
In einem r-regulären Graphen R können wir als Anfangsbewertung $f(e) = 1 \quad \forall e \in E$ wählen und so $\sum_{x \in e} f(e) = r \quad \forall v \in V$ erhalten. Es gilt entsprechend $R_+ = R$, d.h. der positiv bewertete Teilgraph ist gleich dem ganzen Graphen.
Man will nun durch geschicktes Umverteilen der Kantenwerte versuchen, die Zahl der Kanten in R_+ immer weiter zu reduzieren, ohne die Bedingung $\sum_{v \in e} f(e) = r \quad \forall v \in V$ zu verletzen. Diese garantiert dann weiterhin die Existenz mindestens eines Matchings.

Wählt man einen Kreis $K = (k_1, \ldots, k_{n+1} = k_1)$ mit Kanten $E_k := (e_i | e_i = (k_i, k_{i+1}))$ in diesem Graphen, so hat dieser wegen der Bipartitheit gerade Länge. Senkt man nun den Wert jeder Kante auf K mit geradem Index um jeweils c und erhöht man den Wert der Kanten mit ungeradem Index um c, so bleibt die Bedingung $\sum_{v \in e} f(e) = r \quad \forall v \in V$ erhalten. Wählt man c nun so, dass einer der Kantenwerte 0 wird, also als das Minimum der Werte aller Kanten mit geradem Index $c := \min_{2|i} e_i$, so hat der Graph R_+ nach dieser Umverteilung mindestens eine Kante weniger[1].

$$f(e) \quad := \quad \begin{cases} f(e) & \text{falls } e \notin E_k \\ f(e) - \min_{2|i} e_i & \text{falls } e = e_i, 2 \mid i \\ f(e) + \min_{2|i} e_i & \text{falls } e = e_i, 2 \not| \, i \end{cases} \tag{7.2}$$

Alternativ kann man natürlich auch die Werte der Kanten mit geraden Indizes erhöhen und die anderen Werte senken:

$$f(e) \quad := \quad \begin{cases} f(e) & \text{falls } e \notin E_k \\ f(e) - \min_{2 \not| i} e_i & \text{falls } e = e_i, 2 \not| \, i \\ f(e) + \min_{2 \not| i} e_i & \text{falls } e = e_i, 2 \mid i \end{cases} \tag{7.3}$$

Dieses Verfahren kann man nun so oft anwenden, bis es keinen Kreis mehr in dem Graphen gibt. In diesem Fall bilden die Kanten aber bereits ein perfektes Matching:

Proposition 7.18 *Sei G ein kreisfreier, bipartiter Graph mit einer kantenbewertungsfunktion $f : E \to \mathbb{R}_{\geq 0}$, der $\sum_{v \in e} f(e) = r \quad \forall v \in V$ erfüllt und in dem alle Kanten positiven Wert haben. Dann besteht G nur aus isolierten Kanten.*

Beweis durch Induktion:
Induktionsanfang: Die Aussage gilt für einen Graphen, der nur aus zwei Knoten mit

[1] Wir schreiben $2|i$ für 2 “teilt” i.

einer Kante dazwischen besteht (K_2).
Induktionsschritt: Alle bipartiten Graphen $G = (A \dot{\cup} B, W)$, die $\sum_{v \in e} f(e) = r \quad \forall v \in V$ erfüllen, müssen eine gerade Zahl von Knoten haben, da

$$|A|r = \sum_{a \in A} \sum_{a \in e} f(e) = \sum_{b \in B} \sum_{b \in e} f(e) = |B|r,$$

also $|A| = |B|$ gilt. Wir zeigen also den Schritt von n nach $n+2$. G habe also $n+2$ Knoten. Da G kreisfrei ist, aber mindestens eine Kante enthält, ist G ein nichttrivialer Wald. Jeder Baum, der mindestens eine Kante hat, besitzt ein Blatt, also einen Knoten vom Grad 1. Also enthält G mindestens einen Knoten vom Grad 1.
Da zu diesem Knoten nur eine Kante inzident ist, muss er mit seinem einzigen Nachbarn über eine Kante e mit $f(e) = r$ verbunden sein. Dann hat aber auch der Nachbar Grad eins, da keine weitere Kante mit positivem Wert mit ihm verbunden sein kann. Das Entfernen der beiden Knoten samt der Kante e führt zu einem Graphen mit n Knoten, der die Induktionsvoraussetzung erfüllt. □

Dieses Vorgehen alleine würde nun aber wesentlich zuviel Zeit kosten: Um diesen Zustand zu erreichen, müssen mindestens $m - n + 1$ Kanten entfernt werden. Im ungünstigsten Fall haben alle dazu betrachteten Kreise jeweils eine Länge in der Größenordnung von n und die minimale Bewertung kommt nur bei einer Kante vor. In diesem Fall wird nur eine dieser Kanten entfernt. D.h. pro entfernter Kante würden bis zu n Operationen benötigt, was zu $\Omega(mn)$ Schritten führen würde.

In einem ersten Schritt wird diese Reduktion der Kantenzahl daher in gezielterer Weise eingesetzt:
Die Grundidee ist es, stets nur Kreise mit Kanten gleichen Wertes zu betrachten, so dass die Werte der Hälfte der zugehörigen Kanten auf Null gesetzt werden können, während sich die Werte der anderen Hälfte der Kanten verdoppeln. Findet man keine Kreise mehr, so prüft man, ob die Kanten, die nun den doppelten Wert haben, selbst Kreise bilden. Auf diese wendet man das Verfahren erneut an und erhält Kanten mit dem vierfachen Wert usw.

Formaler gesprochen arbeitet man also mit einer Familie von Graphen $G_0 = (V, E_0), \ldots, G_{\lceil \log r \rceil} = (V, E_{\lceil \log r \rceil})$. Der erste Graph G_0 enthält die einfach bewerteten Kanten, der zweite Graph G_1 die doppelt bewerteten, der dritte Graph G_2 die vierfach bewerteten usw.
Die Kantenwerte ergeben sich jetzt implizit daraus, zu welchem der Graphen die Kante gehört. Innerhalb jedes G_i erhält jede Kante den Wert 1, so dass wir die Summe der Werte aller zu einem Knoten inzidenter Kanten durch seinen Grad abkürzen können: $\sum_{v \in e, e \in E_i} f(e) = \delta_{G_i}(v)$.
Anstatt der einzelnen Kanten werden nun also die Graphen unterschiedlich gewichtet:
Graph G_i erhält die Bewertung 2^i.
Die Kanten sollen so auf die Graphen verteilt werden, dass die Invariante $\sum_{i=0}^{\lceil \log r \rceil} 2^i \delta_{G_i}(v) = r \;\; \forall v \in V$ erfüllt ist.

Zu Beginn setzen wir $G_0 = G$, alle anderen $G_j, j > 0$ enthalten keine Kanten.

Name	Zahl der Kanten	Gewichtung der Kanten
G_0	rn	1
G_1	0	2
G_2	0	4
$\vdots$	$\vdots$	$\vdots$
$G_{\lceil \log r \rceil}$	0	$2^{\lceil \log r \rceil}$

Diese Wahl erfüllt offensichtlich die Invariante, da G r-regulär war.
Man versucht nun diese Graphen so zu verändern, dass alle Graphen G_i kreisfrei werden, ohne die Invariante zu zerstören.
Man sucht also in G_0 einen Kreis. Ist ein solcher gefunden, so verschiebt man jede zweite Kante dieses Kreises nach G_1 und löscht die anderen. Da die Kanten in G_1 in der Endsumme doppelt so stark gewertet werden, wie die Kanten aus G_0, bleibt die Gleichung $\sum_{i=0}^{\lceil \log r \rceil} 2^i \delta_{G_i}(v) = r \ \ \forall v \in V$ nach jeder Behandlung eines Kreises gültig.
Nehmen wir beispielsweise an, wir hätten unmittelbar am Anfang in G_0 einen Kreis der Länge k gefunden. Dies hätte auf die Verteilung von Kanten auf Teilgraphen die folgende Auswirkung: Aus G_0 werden k Kanten entfernt. Jede zweite der entfernten Kanten wird in G_1 übertragen, wo sie doppelt gewertet wird.

Name	Zahl der Kanten	Gewichtung der Kanten
G_0	$rn - k$	1
G_1	$\frac{k}{2}$	2
G_2	0	4
$\vdots$	$\vdots$	$\vdots$
$G_{\lceil \log r \rceil}$	0	$2^{\lceil \log r \rceil}$

Gibt es in einem G_i keinen Kreis mehr, so fährt man in gleicher Weise mit G_{i+1} fort.

Beobachtung 7.19 *Das Verfahren terminiert nach $\lceil \log_r \rceil$ Phasen, da höchstens die Hälfte der zu einem Knoten in G_i inzidenten Kanten nach G_{i+1} übernommen wird.*

Nun müssen wir noch analysieren, wie lange eine einzelne Phase dauert. In einer einzelnen Phase werden

1. solange Kreise gewählt, bis dies nicht mehr möglich ist,
2. und von jedem Kreis die Hälfte der Kanten gelöscht.

Schritt 1 entspricht dem Bestimmen einer Zerlegung des Graphen in Kreise und einen kreisfreien Restgraphen.

Proposition 7.20 *In einem Multigraphen G mit n Knoten und m Kanten kann in Zeit $\mathcal{O}(m)$ eine Kreiszerlegung gefunden werden.*

Beweis: Es sei zunächst angemerkt, daß ein Kreis im obigen Sinne auch aus zwei parallelen Kanten bestehen kann. Dies verändert das Verhalten des Algorithmus nicht.
Man führt DFS von einem beliebigen Knoten aus durch. Erreicht man dabei einen Knoten v, der bereits besucht wurde, so hat man einen Kreis gefunden, den man vollständig aus dem Graphen entfernt. Nun setzt man die Suche in v so fort, als ob dieser Kreis nicht existiert hätte. Ist DFS beendet, so wiederholt man - so oft wie notwendig - die Suche von noch nicht vollkommen abgearbeiteten Knoten aus. Da DFS jede Kante nur einmal besucht, folgt die Laufzeitschranke.
Sei nun angenommen, der Graph, der am Ende verbleibt, enthielte noch einen Kreis K. Da jede Kante des Graphen betrachtet wurde, wurde auch eine Kante aus diesem Kreis betrachtet. Sei e die erste Kante aus dem Kreis, die von DFS besucht wurde. Man betrachte den Zeitpunkt, zu dem DFS zum Anfangsknoten von e über e zurückkehrt. Da e danach noch im Graphen ist, wird kein Kreis gefunden, der durch e führt. Sei e_i die erste Kante auf dem Kreis, die nach Betrachtung von e und vor der Rückkehr über e nicht besucht wurde. Da e selbst besucht wurde, der Anfangsknoten von e aber nicht über K erreicht wurde, muss es eine solche Kante geben.
Nun können wir drei Fälle unterscheiden:

1. War der Endpunkt noch nie auf dem Stapel, so hätte die Kante e_i überquert werden können. Dies steht im Widerspruch dazu, dass e_i nicht überquert wurde.

2. War das Ende von e_i zum Zeitpunkt, zu dem e_i betrachtet worden wäre, auf dem DFS-Stapel, so wäre ein Kreis K' gefunden worden, der unter anderem auch e_i enthalten hätte. Dann wäre e_i mit den anderen Kanten von K' entfernt worden. Dies steht im Widerspruch dazu, dass e_i zu K gehört.

3. Hätte der Endpunkt von e_i den Stapel bereits verlassen, so wäre e_i andersherum betrachtet worden. Dann wäre e_i aber entfernt worden oder der Anfangspunkt von e_i hätte den Stapel ebenfalls verlassen, so dass bereits die in K vorangehende Kante schon nicht mehr betrachtet worden wäre. Dies steht im Widerspruch zu Wahl von e_i.

□

Der zweite Schritt jeder Phase, das Löschen jeweils jeder zweiten Kante aus den Kreisen, ist offensichtlich in Linearzeit $\mathcal{O}(m)$ möglich.

Es bleibt die Laufzeiten der einzelnen Phasen aufzusummieren. In der $i$ten Phase kann der entsprechende Graph G_i höchstens $\frac{m}{2^i}$ Kanten enthalten, da jeweils höchstens die Hälfte der Kanten auf den nächsten Graphen übertragen wird.

$$\sum_{i=0}^{\log r} \mathcal{O}\left(\frac{m}{2^i}\right) = \mathcal{O}\left(m \sum_{i=0}^{\log r} \left(\frac{1}{2^i}\right)\right) \leq \mathcal{O}(m * 2) = \mathcal{O}(m)$$

Proposition 7.21 *Der Graph* $G^* := (V, \bigcup_{i=0}^{\lceil \log r \rceil} E(G_i))$ *enthält ein perfektes Matching und höchstens* $(n-1)\log r$ *Kanten.*

Beweis:
1. Wir übertragen die Bewertung von den Graphen zurück auf die Kanten von G. Mit $f(e) := 2^i$ für $e \in G_i$ erhält man eine Bewertung, die die Voraussetzung von Lemma 7.14138 $\sum_{v \in e} f(e) = r \quad \forall v \in V$ erfüllt. Somit hat der Graph ein perfektes Matching.
2. Da jeder der G_i kreisfrei ist, kann jeder davon höchstens $n-1$ Kanten enthalten (vgl. [22]). Da es $\log r$ dieser Graphen gibt, kann ihre Vereinigung höchstens $(n-1)\log r$ Kanten enthalten. □

Zusammenfassend können wir also feststellen, dass wir in Linearzeit einen spannenden Teilgraphen von G finden können, der höchstens $n \log r \leq n \log n$ Kanten hat, aber dennoch ein perfektes Matching enthält.

In diesem Graphen kann ein Matching mittels des $\mathcal{O}(\sqrt{n}m)$-Algorithmus von Hopcroft und Karp [21] mit $m = n \log n$ in Zeit $\mathcal{O}(\sqrt{n}n \log n)$ gefunden werden.
Dies ist für unsere Zwecke ausreichend: Schätzen wir die Zahl der Kanten im ursprünglichen Graphen nach unten durch die Zahl der für $\sigma^3\sqrt{n}$ Matchings benötigten $\sigma^3\sqrt{n}n$ Kanten ab, so ergibt sich für die Bestimmung des annähernd regulären Teilgraphen bereits eine Laufzeitgrenze von $\mathcal{O}\left(\frac{\sqrt{n}}{\sigma}\sigma^3\sqrt{n}n\right) = \mathcal{O}(n^2\sigma^2)$. D.h. die Zeit von $\mathcal{O}(\sqrt{n}n \log n)$ wird durch diese bereits dominiert.
Der Algorithmus von Cole, Ost und Schirra[6] setzt nun in noch geschickterer Weise die Kantenreduktion über Kreise fort. Dies ist für unsere Zwecke aber nicht mehr gewinnbringend.

Bemerkung 7.22 *Diese erste Phase findet sich bereits in einer früheren Arbeit von Cole und Hopcroft [5].*

Zur Vervollständigung unseres Algorithmus bleibt noch zu zeigen, dass die beschriebenen Verfahren auch für Multigraphen funktionieren.
Hierzu zeigen wir, dass Kreise bzw. Kreiszerlegungen in Multigraphen auf die gleiche Weise und mit vergleichbarem Aufwand gefunden werden können wie in einfachen Graphen:
Wir setzen in jede Kante des Multigraphen einen Knoten vom Grad 2 ein. Hierdurch wird aus einem Multigraphen mit m Kanten ein einfacher Graph mit $2m$ Kanten. In diesem können wir Kreise und Kreiszerlegungen wie oben beschrieben finden. Entfernt man aus den so gefundenen Kreisen die zusätzlichen Knoten, erhält man Kreise im ursprünglichen Multigraphen. Ein solcher Kreis kann mehrere Kanten einer Mehrfachkante enthalten. Die Laufzeit verdoppelt sich hierdurch maximal, bleibt also in $\mathcal{O}(m)$.

Kapitel 8

Anmerkungen und Erweiterungen

8.1 Anmerkungen

8.1.1 Der bestmögliche Fall

Die bestmögliche Verbesserung der Laufzeit gegenüber dem Algorithmus von Hopcroft und Karp [21] tritt ein, wenn der Graph möglichst viele Matchings enthält. Ein bipartiter Graph kann maximal n disjunkte Matchings enthalten. D.h asymptotisch können im besten Fall kn Matchings enthalten sein, wobei $k \leq 1$ eine Konstante ist.

Korollar 8.1 *Sei $0 \leq k \leq 1$ eine Konstante. In einem bipartiten Graphen, der kn disjunkte perfekte Matchings enthält, kann ein perfektes Matching in Zeit $\mathcal{O}(\sqrt[3]{n}m)$ gefunden werden.*

Beweis: Wir haben in Satz 6.1 126 gezeigt, dass man in einem Graphen mit $\sigma^3\sqrt{n}$ perfekten disjunkten Matchings ein perfektes Matching in Zeit $\mathcal{O}\left(\frac{\sqrt{n}m}{\sigma}\right)$ finden kann. Entsprechend müssen wir zur Bestimmung von σ die Terme $\sigma^3\sqrt{n}$ und kn gleichsetzen.
Für positive Zahlen k, n, σ gilt

$$kn = \sqrt{n}\sigma^3 \Leftrightarrow k^2n^2 = n\sigma^6 \Leftrightarrow k^2n = \sigma^6 \Leftrightarrow \sqrt[6]{k^2n} = \sigma.$$

Bei Einsetzen in $\mathcal{O}(\sqrt{n}m/\sigma)$ verschwindet die Konstante k und man erhält $\mathcal{O}(\sqrt{n}m/\sqrt[6]{n}) = \mathcal{O}(\sqrt[3]{n}m)$. □

8.1.2 Parameterwahl

Bisher sind wir stets davon ausgegangen, dass die Größe des enthaltenen Subgraphen von vorneherein bekannt ist. Dies ist in der Praxis meistens nicht gegeben. Es ist aber möglich, die Größe durch binäre Suche zu bestimmen. Es sei angenommen, der Graph enthalte einen $\sqrt{n}(\sigma^*)^3$-regulären Subgraphen, mit einem unbekannten Wert σ^*. Wir wenden

nun den ersten Teil des Algorithmus jeweils mit angenommenen Werten $\sigma_0, \sigma_1, \sigma_2, \ldots$, an, bis ein geeigneter Wert gefunden wurde. Da der Knotengrad eines vollständigen regulären Subgraphen nach oben durch den Minimalgrad des Graphen beschränkt ist, gilt $(\sigma^*)^3\sqrt{n} \leq \delta_{min}(G)$. Also wissen wir, dass σ^* im Intervall $\left[1, \sqrt[3]{\frac{\delta_{min}(G)}{\sqrt{n}}}\right]$ liegen muss. Daher setzen wir $\sigma_0 := \sqrt[3]{\frac{\delta_{min}(G)}{\sqrt{n}}}$.
Im weiteren Verlauf wählen wir jeweils $\sigma_i := \frac{1}{2}\sigma_{i-1}$ für alle $i \geq 1$. Wir brechen die Versuche mit verschiedenen σ_i ab, sobald erstmals ein Fluss der Größe $\sigma_i^3\sqrt{n}n - \sigma_i^2 n$ gefunden wird. Dies ist für jedes $\sigma' \leq \sigma^*$ der Fall. Sei σ_z dasjenige σ_i für das der Abbruch erfolgt ist.

Beobachtung 8.2 $\sigma_z > \frac{1}{2}\sigma^*$

Beweis: Wäre dies nicht der Fall, so wäre $\sigma_{z-1} < \sigma^*$ und das Verfahren wäre eine Iteration früher abgebrochen worden. □

Lemma 8.3 *Durch die binäre Suche nach σ verändert sich die asymtotische Laufzeit nicht.*

Beweis: Ein Probelauf mit einem konkreten σ_x kostet jeweils $\mathcal{O}\left(m\frac{\sqrt{n}}{\sigma_x}\right)$ Schritte. Die Laufzeit der gesamten Suche ist wegen $\frac{1}{2}\sigma^* < \sigma_z = \frac{\sigma_0}{2^z}$ beschränkt durch

$$
\begin{aligned}
\mathcal{O}\left(\sum_{i=0}^{z}\frac{\sqrt{n}m}{\frac{\sigma_0}{2^i}}\right) &\subseteq \mathcal{O}\left(\sqrt{n}m\sum_{i=0}^{z}\frac{2^i}{\sigma_z 2^z}\right) \\
&\subseteq \mathcal{O}\left(\sqrt{n}m\sum_{i=0}^{z}\frac{2^{i-z}}{\frac{1}{2}\sigma^*}\right) \\
&\subseteq \mathcal{O}\left(\frac{\sqrt{n}m}{\frac{1}{2}\sigma^*}\sum_{i=0}^{z}\left(\frac{1}{2}\right)^{z-i}\right) \\
&\subseteq \mathcal{O}\left(\frac{\sqrt{n}m}{\sigma^*}\underbrace{\sum_{i=0}^{z}\left(\frac{1}{2}\right)^{i}}_{\leq 2}\right) \\
&= \mathcal{O}\left(\frac{\sqrt{n}m}{\sigma^*}\right).
\end{aligned}
$$

□

8.1.3 Vereinfachung der Implementierung

Der Übersichtlichkeit halber haben wir bei der Beschreibung des Algorithmus einige kleine - für die Theorie unerhebliche - Verbesserungsmöglichkeiten ausgelassen. Für eine

konkrete Implementierung des Algorithmus könnte man die folgenden, weniger aufwendigen Wege wählen:

Lemma 8.4 *Es ist nicht notwendig, den Graphen zu einem vollständigen Graphen zu ergänzen.*

Beweis: Dies wird durch zwei Fakten ermöglicht: Erstens werden wir zeigen, dass bereits der nicht erweiterte (r, d)-reguläre Subgraph ein Matching ausreichender Größe enthält und dass sich diese Eigenschaft auf (r, d)-reguläre Multigraphen überträgt.
Zweitens genügt es, die erste Phase des Algorithmus von Cole und Hopcroft[5] anzuwenden und in dem so konstruierten Graphen mit $\mathcal{O}(n \log n)$ Kanten über augmentierende Pfade ein Matching zu bestimmen. Dieser Algorithmus setzt nicht zwingend einen regulären Graphen voraus, sondern lediglich einen solchen, bei dem die Existenz eines ausreichend großen Matchings durch die Gradbedingung garantiert wird. Die Zeit, die dies im Vergleich zum Algorithmus von Cole, Ost und Schirra [6] zusätzlich beansprucht, wird von der Laufzeit der anderen Phasen unseres Algorithmus dominiert (vgl. Kapitel 7.5137).
Zum Beweis benötigen wir das folgende Lemma aus [28]:

Lemma 8.5 *Sei $G = (A\dot{\cup}B, E)$ ein bipartiter Graph mit $2n$ Knoten und $|A| = |B|$. Dann gilt für die Größe eines maximalen Matchings $\nu(G) = n - \max_{V' \subset A}(|V'| - |\Gamma(V')|)$.*

Beweis:
"$\leq$" ist trivial.
"$\geq$": Man fügt jeweils $\max_{V' \subset A}(|V'| - |\Gamma(V')|)$ Knoten zu beiden Partitionen hinzu und verbindet diese mit allen Knoten der anderen Partition. Den so entstehenden Graphen bezeichnen wir mit G'. Die hinzugefügten Knoten werden wir als "neue" Knoten bezeichnen, die bereits vorhandenen als "alte" Knoten. Entsprechend bezeichnen wir die Nachbarschaft eines Knotens v bezüglich alter Knoten mit $\Gamma_G(v)$ und bezüglich aller Knoten mit $\Gamma_{G'}$. Auf G' kann nun der Satz von Hall angewendet werden:

Satz 7.15, Hall (1935, S. 138) *In einem bipartiten Graphen $G' = (A\dot{\cup}B, E)$ gibt es genau dann ein Matching, das A überdeckt, wenn $|\Gamma_{G'}(X)| \geq |X|$ für alle $X \subseteq A$.*

Sei X eine Teilmenge von A.

1. Enthält X einen der neuen Knoten, so ist die Nachbarschaft $\Gamma_{G'}(X)$ die gesamte andere Partition und daher sicher größer als $|X|$.

2. Enthält X nur "alte" Knoten, so besteht seine Nachbarschaft $\Gamma_{G'}(X)$ aus den alten Nachbarn $\Gamma_G(X)$ und den $\max_{V' \subset A} |V'| - |\Gamma_G(V')|$ neuen Nachbarn. Nun gilt aber

$$|X| = |\Gamma_G(x)| + (|X| - |\Gamma_G(X)|) \leq |\Gamma_G(x)| + \max_{V' \subset A}(|V'| - |\Gamma_G(V')|) = |\Gamma_{G'}(X)|.$$

Damit ist die Voraussetzung des Satzes von Hall erfüllt und der neue Graph hat ein perfektes Matching.

Betrachtet man ein perfektes Matching im neuen Graphen und entfernt die neuen Knoten wieder, so verlieren maximal $\max_{V' \subset A}(|V'| - |\Gamma_G(V')|)$ Knoten aus einer Partition ihren Nachbarn. Es gibt also im ursprünglichen Graphen ein Matching der Größe $n - \max_{V' \subset A}(|V'| - |\Gamma(V')|)$. $\square_{\text{L8.5}}$

Damit erhalten wir:

Lemma 8.6 *Jeder (r, d)-reguläre bipartite Graph mit $2n$ Knoten hat ein Matching der Größe $n - \frac{d}{2r}$.*

Beweis: Sei $V' \subset A$ eine Menge, für die $|V'| - |\Gamma(V')|$ maximal ist. Die Zahl der Kanten, die von V' ausgehen, ist mindestens $r|V'| - \frac{d}{2}$. Jeder Knoten in $\Gamma(V')$ kann aber höchstens r dieser Kanten "empfangen".
Damit ist

$$|\Gamma(V')| \geq \frac{r|V'| - \frac{d}{2}}{r} = \frac{r|V'|}{r} - \frac{d}{2r} = |V'| - \frac{d}{2r}.$$

Somit ist

$$\max_{V' \subset A}(|V'| - |\Gamma_G(V')|) \leq \frac{d}{2r}$$

und es gilt entsprechend

$$n - \max_{V' \subset A}(|V'| - |\Gamma_G(V')|) \geq n - \frac{d}{2r}.$$

$\square$

Auch der in der ersten Phase des Algorithmus von Cole und Hopcroft[5] bzw. Cole, Ost und Schirra [6] bestimmte Multigraph ist immer noch (r, d)-regulär. Daher kann Lemma 8.6 auch auf diesen angewendet werden.

Bezüglich Punkt zwei beobachten wir, dass nach Ablauf der ersten Phase des Algorithmus von Cole und Hopcroft [5], die Gradbedinung aus Lemma 8.6 für den entstehenden Multigraphen immer noch erfüllt ist. Das bedeutet aber, dass auch in diesem Multigraphen die Nachbarschaft einer Knotenmenge $|X|$ immer noch die Größe mindestens $|X| - \frac{d}{2r}$ hat. Diese Eigenschaft überträgt sich von dem Multigraphen auf den zugrundeliegenden einfachen Graphen. D.h. auch in diesem einfachen Graphen existiert ein Matching der Größe mindestens $n - \frac{d}{2r}$.

Die Laufzeit des ersten und des letzten Teils als Funktion von n kann - wie in Kapitel 7.5137 gezeigt - nur durch $\mathcal{O}(n^2\sigma^2)$ abgeschätzt werden. Wendet man den gemeinsamen ersten Schritt der Algorithmen von Cole und Hopcroft bzw. Cole, Ost und Schirra unmittelbar auf den vom Flussalgorithmus gelieferten (r, d)-regulären Graphen an, so reduziert dies in $\mathcal{O}(m)$ die Zahl der Kanten auf $n \log n$. In dem zugrundeliegenden einfachen Graphen kann man nun in mit Hilfe des Algorithmus von Hopcroft und Karp $\mathcal{O}(\sqrt{n} n \log n)$ ein maximales Matching bestimmen (vgl. Kapitel 7.5). Dies ist stets schneller, als die oben angegebene Laufzeitschranke für den ersten Teil unseres Algorithmus.

$\square_{\text{L8.4}}$

8.2 Verallgemeinerungen

Satz 6.1126 lässt sich noch etwas allgemeiner fassen: Der Algorithmus arbeitet mit annähernd regulären Teilgraphen, also kann man auf die Existenz eines "echten" regulären Teilgraphen verzichten.

Satz 8.7 *Sei G ein bipartiter Graph, der einen $(\sigma^3\sqrt{n}, d)$-regulären Teilgraphen enthält. Dabei sei $d \in \mathcal{O}(n\sigma^2)$. Dann lässt sich ein maximales Matching in G in Zeit $\mathcal{O}\left(\frac{m\sqrt{n}}{\sigma}\right)$ finden.*

> **Beweis:** Man wandelt den Graphen in der gleichen Weise in ein Flussnetzwerk um, wie im vorangegangenen Abschnitt, wobei angestrebt wird, $\sigma^3\sqrt{n}$ Flusseinheiten durch jeden Knoten zu leiten. Dann erlaubt das so gebildete Netzwerk einen Gesamtfluss von mindestens $n\sigma^3\sqrt{n} - \mathcal{O}(n\sigma^2)$, da man diesen Fluss bereits erzielt, wenn man für jede Kante des nahezu regulären Graphen einen Flusswert von 1 wählt. Der nach dem ersten Teil garantierbare Schnitt im residualen Netzwerk hat eine Kapazität von höchstens $\left(\frac{n}{\frac{\sqrt{n}}{\sigma}}\right)^2 = n\sigma^2$. Da dem Fluss somit höchstens $\mathcal{O}(n\sigma^2)$ Einheiten zu einer Größe von $n\sigma^3\sqrt{n} - \mathcal{O}(n\sigma^2)$ fehlen und $\mathcal{O}(n\sigma^2 + n\sigma^2) = \mathcal{O}(n\sigma^2)$ gilt, hat der Fluss wie im ursprünglichen Algorithmus eine Größe von $n\sigma^3\sqrt{n} - \mathcal{O}(n\sigma^2)$. Das Verfahren kann also wie in Kapitel 7127 fortgesetzt werden. □

Nun wollen wir versuchen, die Klasse der Graphen, für die der Algorithmus funktioniert, noch weiter zu vergrößern. Um das Verfahren auf Graphen anzuwenden, die nicht die benötigte Zahl von Matchings enthalten, muss man über einzelne Kanten mehr Fluss schicken, um die fehlenden Kanten auszugleichen. Hierzu versuchen wir, Kanten zuzulassen, die nicht Einheitskapazität haben. Dabei treten drei Probleme auf:

1. Das Netzwerk in der ersten Phase ist kein Einheits-Kapazitäts-Netzwerk mehr, so dass eine größere Laufzeit zur Bestimmung des Flusses benötigt wird.

2. Die Zahl von Kanten, die im zweiten Schritt betrachtet werden müssen, ist größer als die Zahl m der Kanten im ursprünglichen Graphen.

3. Es könnten Kanten mit zu großer Kapazität bzw. zu viele Kanten über den Schnitt führen, so dass in der zweiten Phase zu viele "falsche" Kanten ins Matching aufgenommen werden. Dann wird entsprechend mehr Zeit benötigt, um das Matching in der dritten Phase mittels DFS zu maximieren.

Zu Problem 1:

Beobachtung 8.8 *Bei der Flussbestimmung im ersten Teil werden nur $\mathcal{O}\left(\frac{\sqrt{n}}{\sigma}\right)$ Phasen des Algorithmus von Dinic durchgeführt. Es werden also nur augmentierende Pfade der Länge $\mathcal{O}\left(\frac{\sqrt{n}}{\sigma}\right)$ betrachtet.*

Wir wissen also, dass jede Flusseinheit nur $\mathcal{O}\left(\frac{\sqrt{n}}{\sigma}\right)$ Kanten überquert. Also ist die Zahl der “wichtigen” Schritte gewissermaßen durch das Produkt aus Flussgröße und diesem Wert beschränkt. Wir zeigen dies allgemeiner für eine beliebige Anzahl von Phasen l.

Lemma 8.9 *Sei $G = (V, E, c)$ ein Netzwerk mit ganzzahligen Kapazitäten, in dem es einen s, t-Schnitt der Größe γ gibt. Dann können l Phasen des Algorithmus von Dinic [8] in Zeit $\mathcal{O}(l(m + \gamma))$ durchgeführt werden.*

Beweis: Der BFS-Anteil einer Phase ist unabhängig von der konkreten Kapazität einer Kante, da nur die Entscheidung getroffen werden muss, ob die Kante noch einen weiteren (Rück-) Fluss zulässt oder nicht. Jede BFS-Phase dauert also $\mathcal{O}(m)$. Alle l BFS-Anteile lassen sich also in Zeit $\mathcal{O}(lm)$ durchführen.

Bei der Analyse des DSF-Teils werden wir versuchen, die Kosten einzelnen Kanten zuzuweisen. Hierzu definieren wir:

Definition 8.10 *Unter einer* Kantenoperation *verstehen wir den Vorgang, dass die Suche eine Kante überquert, d.h. dass ein Knoten neu auf den DFS-Stapel gelegt wird. Die zugehörige Kante verbindet das zuvor oberste Element des Stapels mit dem neuen obersten Element.*

Die Kantenoperationen des DFS-Anteils teilen wir für jede Phase in zwei Gruppen ein: Erfolgreiche und nicht erfolgreiche Operationen. Eine Kantenoperation ist erfolgreich, wenn in der konkreten Phase der Flusswert der betroffenen Kante geändert wird und der Pfad, über den diese Änderung erfolgt, nach dieser Kantenoperation und bevor die Suche über die Kante zurückkehrt gefunden wurde. Ansonsten ist die Kantenoperation nicht erfolgreich.

Proposition 8.11 *Die Zahl der nicht erfolgreichen Operationen pro Phase ist durch m beschränkt.*

> **Beweis:** Wird über die entsprechende Kante in dieser Phase kein weiterer Fluss geschickt, so ist von ihrem Ende aus t nicht mehr erreichbar. Somit wird die Kante für die aktuelle Suche ausgeschlossen, sobald der von ihr aus erreichbare Teilgraph einmal durchsucht wurde. Während dieser Durchsuchung wird diese Kante nicht betrachtet, da ihre Knoten auf dem Stack liegen. Es kann also pro Phase und Kante nur eine nicht erfolgreiche Kantenoperation geben. □

Proposition 8.12 *Auf eine erfolgreiche Kantenoperation erfolgt stets das Identifizieren eines augmentierenden Pfades, bevor eine weitere Kantenoperation mit dieser Kante vorgenommen wird.*

Beweis durch Widerspruch: Es sei angenommen, es wurde kein neuer Pfad gefunden.
Fall 1: Die nächste Operation mit dieser Kante ist erfolgreich. Dann gibt es einen Weg vom Ende der Kante zu t, der schon nach der aktuellen Operation hätte gefunden werden

müssen.
Fall 2: Die nächste Operation mit dieser Kante ist nicht erfolgreich. Dann gibt es keinen solchen Weg. Da zwischenzeitlich keine Augmentierung statt gefunden hat, war ein Pfad über die betrachtete Kante schon zuvor nicht möglich und die betrachtete Operation ist nicht erfolgreich. □

Proposition 8.13 *Die Gesamtzahl erfolgreicher Kantenoperationen ist durch $l\gamma$ beschränkt.*

> **Beweis:** Jeder erfolgreichen Kantenoperation lässt sich gemäß der vorangegangenen Proposition ein augmentierender Pfad zuordnen, über den der Fluss geändert wird und auf dem diese Kante liegt. Jeder augmentierende Pfad hat Länge $\leq l$, so dass er höchstens l Kantenoperationen zugeordnet werden kann. Des weiteren wird der Fluss wegen der Ganzzahligkeit der Kapazitäten über jeden augmentierenden Pfad um mindestens 1 vergrößert. Da der Fluss insgesamt nur die Größe γ erreichen kann, kann es nur γ Pfade geben, über die augmentiert wird. □

Die Gesamtzeit ergibt sich also als $\mathcal{O}(2ml + l\gamma)$. Damit ist das Lemma bewiesen. □L8.9

Zu Problem 2: Nun zeigen wir, dass sich die Größe r des angestrebten Knotengrades im zweiten Schritt, d.h. bei der Reduktion auf einen Graphen mit $n \log n$ Kanten, nur logarithmisch auf die Laufzeit auswirkt.
Hierzu benötigen wir die folgende Eigenschaft der betrachteten Multigraphen: Fasst man die parallelen Kanten des hier betrachteten Multigraphen als Mehrfachkanten auf, so gibt es genau m Mehrfachkanten. Dies ergibt sich unmittelbar aus der Konstruktion, da wir lediglich die Kapazität existierender Kanten erhöht haben.
Wenn wir aber jeweils eine Mehrfachkante betrachten, so wissen wir, wieviele Kreise sich mit den Kanten dieser Mehrfachkante bilden lassen, ohne diese Kreise explizit zu suchen.

Proposition 8.14 *In einem r-regulären Multigraphen mit n Knoten und m Mehrfachkanten kann ein Teilgraph mit $\mathcal{O}(n \log r)$ Kanten, der ein perfektes Matching enthält, in Zeit $\mathcal{O}(m \log r)$ gefunden werden.*

> **Beweis:** Der Beweis baut auf der Beschreibung des Algorithmus in Kapitel 7.5[137] auf. Dort haben wir gesehen, dass Multigraphen beim Finden von Kreisen in der gleichen Weise behandelt werden können wie einfache Graphen.
> Nun nutzen wir aus, dass wir direkt ausrechnen können, wieviele Kreise sich mit den Kanten einer Mehrfachkante bilden lassen. Wir können ausrechnen, wieviele 2-fach, 4-fach usw. gewichtete Kanten sich ergeben, wenn man die Gewichtungen auf diesen Kreisen umverteilt.
>
> Wir zerlegen den Graphen bereits vor dem Suchen von Kreisen in unterschiedlich gewertete Teilgraphen G_i. Dabei erhalten alle Kanten des i ten Graphen wiederum das Gewicht 2^i. Sei $f(e)$ die Zahl der Kanten der Mehrfachkante e. Wir wollen nun in die Graphen G_i so Kanten e_i einfügen, dass

$f(e) = \sum_{i=0}^{\lceil \log f(e) \rceil} \begin{cases} 2^i & \text{falls} \quad e_i \in G_i \\ 0 & \text{sonst} \end{cases}$.
Damit gibt es die Kante e_i genau dann, wenn die Binärdarstellung von $f(e)$ an der i-ten Position von rechts den Wert 1 hat.
Somit gilt wiederum für alle $v \in V$, dass $\sum_{i=0}^{\lceil \log r \rceil} \delta_{G_i}(v) 2^i = \sum_{(v,u) \in E} f((v,u))$. Insbesondere ist nun jeder G_i ein normaler Graph mit höchstens m einfachen Kanten.

Mit dieser Menge von Graphen können wir nun wieder gemäß dem Algorithmus aus Kapitel 7.5[137] weiterarbeiten: In jedem G_i - für sich allein betrachtet - können wir also in $\mathcal{O}(m)$ eine Kreiszerlegung finden, jede zweite Kante eines Kreises entfernen und die jeweils anderen Kanten in den nächst höherbewerteten Graphen verschieben.
Dies reicht aber noch nicht zum Nachweis der Laufzeitschranke, da die G_i nicht unahängig voneinander betrachtet werden können. Es könnten bis zu $\log rm$ weitere Kanten aus G_i nach G_{i+1} verschoben worden sein, so dass G_{i+1} zum Zeitpunkt seiner Betrachtung deutlich mehr als m Kanten enthält. Es bleibt also zu zeigen, dass im Laufe des Algorithmus nicht zuviele neue Kanten aus einem G_i in einen G_{i+1} übertragen werden.
Im ungünstigsten Fall waren zu Beginn alle G_j mit $j \leq i$ vollständige Graphen und bei der Betrachtung von G_i werden alle Kanten nach G_{i+1} übertragen. Der Gesamtwert aller Kanten in $G_0 \cdots G_i$ ist beschränkt durch $m \sum_{j=0}^{i} 2^j < m 2^{i+1}$. Das heißt, dass in G_{i+1}, in dem jede Kante den Wert 2^{i+1} hat, höchstens m zusätzliche Kanten eingefügt werden.

Die Zahl der Kanten in jedem G_i ist also stets durch $2m$ beschränkt, d.h. eine Kreiszerlegung kann immer noch in Zeit $\mathcal{O}(m)$ erstellt werden. Da es insgesamt $\log r$ Graphen G_i gibt, ist die Gesamtlaufzeit dieser Variante des 2. Teils des Algorithmus in $\mathcal{O}(m \log r)$.
Jeder der $\log r$ Graphen ist wiederum kreisfrei, er enthält also höchstens $n-1$ Kanten. Damit ist die Kanzenzahl durch $\mathcal{O}(n \log r)$ begrenzt.
Analog zu Lemma 7.14[138] muß auch hier die Vereinigung dieser Graphen wieder ein perfektes Matching enthalten. □

Proposition 8.15 *In einem r-regulären Multigraphen mit n Knoten und $m \geq n\sqrt{n}$ Mehrfachkanten kann ein perfektes Matching in Zeit $\mathcal{O}(m \log r)$ gefunden werden.*

Beweis: Man reduziert zunächst in Zeit $\mathcal{O}(m \log r)$ die Kantenzahl auf $n \log r$. Dann findet man mit dem Algorithmus von Hopcroft und Karp [21] in diesem Graph ein perfektes Matching in Zeit $\mathcal{O}(\sqrt{n}(n \log r))$. Für $m \geq n\sqrt{n}$ wird dies von $\mathcal{O}(m \log r)$ dominiert. □

Zu Problem 3: Die aus den vorangegangenen Überlegungen folgende Idee, allen nicht mit s oder t verbunden Kanten eine sehr große Kapazität geben zu wollen, scheitert daran, dass die Größe des Schnittes im residualen Netzwerk, die wir zur Abschätzung der Laufzeit von Teil 3 benötigen, dabei proportional zunimmt. Wir können also nicht allzuviele

Kanten mit hoher Kapazität zulassen.
Unter dieser Voraussetzung erhält man eine noch allgemeinere Form von Satz 6.1126:

Satz 8.16 *Sei G ein bipartiter Graph, der wie in Kapitel 1.4.2**13* *in ein Flussnetzwerk umgewandelt wird. Die Kapazitäten seien dabei wie folgt gewählt:*

1. *Die Kanten von s nach B und von A nach t haben alle Kapazität $\sigma^3\sqrt{n}$.*
2. *Bis zu χ Kanten haben eine beliebige Kapazität.*
3. *Alle übrigen Kanten haben Kapazität 1.*

Wenn ein maximaler Fluss in diesem Graphen die Größe $\sigma^3 n\sqrt{n} - \mathcal{O}(\sigma^2 n)$ hat, dann kann ein maximales Matching in G in Zeit $\mathcal{O}(\frac{\sqrt{n}m\chi}{\sigma} + n^2\sigma^2\chi)$ gefunden werden.

Beweis: Wie in Lemma 8.9149 gezeigt, können $\frac{(\chi+2)\sqrt{n}}{\sigma} + 1$ Phasen des Algorithmus von Dinic in Zeit $\mathcal{O}(\frac{\chi\sqrt{n}}{\sigma}(m + n\sqrt{n}\sigma^3)) = \mathcal{O}(\sqrt{n}m\chi/\sigma + n^2\sigma^2\chi)$ durchgeführt werden. Danach gibt es $\frac{(\chi+2)\sqrt{n}}{\sigma} + 2$ Level, die man in s, t und $\sqrt{n}/\sigma$ aufeinanderfolgende Gruppen von je $\chi+2$ Leveln zerlegen kann. Nach dem Schubfachprinzip muss es also $\chi + 2$ aufeinanderfolgende Level geben, die zusammen höchstens $\sqrt{n}\sigma$ Knoten enthalten, so dass jedes einzelne davon ebenfalls höchstens $\sqrt{n}\sigma$ Knoten enthalten kann.
Diese $\chi + 2$ Level enthalten einen Schnitt der Kapazität höchstens $n\sigma^2$:
Wie beim Beweis von Satz 6.1126 bilden die Kanten zwischen zwei aufeinanderfolgenden Leveln einen gerichteten Schnitt. Zwischen den ingesamt $\chi+2$ betrachteten Leveln gibt es also ingesamt $\chi + 1$ derartige Schnitte.
Da es nur χ Kanten mit Kapazität größer als 1 gibt, die nicht mit s oder t verbunden sind, besteht einer der $\chi + 1$ Schnitte nur aus Kanten der Kapazität 1.
Da es zwischen zwei Leveln nur höchstens $(\sqrt{n}\sigma)^2$ Kanten geben kann hat, dieser Schnitt eine Kapazität kleiner gleich $n\sigma^2$.

Der Algorithmus verfährt im Folgenden analog zu Satz 6.1126. Teil zwei kann gemäß Proposition 8.14150 in $\mathcal{O}(m\log(\sigma^3\sqrt{n})) \subset \mathcal{O}(m\log n)$ durchgeführt werden. Nach Teil zwei bleiben wieder bis zu $\frac{n\sigma^2}{\sqrt{n}\sigma^3} = \frac{\sqrt{n}}{\sigma}$ Knoten unüberdeckt. Hierdurch dauert der dritte Teil, also die Bestimmung der noch notwendigen $\frac{\sqrt{n}}{\sigma}$ augmentierenden Pfade, wieder $\frac{\sqrt{n}m}{\sigma}$.
Insgesamt ergibt sich also eine Laufzeit von

$$\mathcal{O}\left(\underbrace{\frac{\sqrt{n}m\chi}{\sigma} + n^2\sigma^2\chi}_{\text{Teil1}} + \underbrace{m\log n}_{\text{Teil2}} + \underbrace{\frac{\sqrt{n}m}{\sigma}}_{\text{Teil3}}\right).$$

Das entspricht für $\frac{\sqrt{n}}{\sigma} \geq \log n$ der Voraussetzung des Satzes. Dies ist aber wegen $\sigma^3 \leq \sqrt{n}$ (vgl. Abschnitt 8.1.1144) für ausreichend große Werte von n der Fall. □

8.3 Anwendung von Graphenkompression

Die in Teil 1 dieser Arbeit vorgestellten Verfahren zur Graphenkompression lassen sich mit den Ergebnissen aus Teil 2 kombinieren. Da unser Algorithmus weniger Zeit benötigt, als der von Hopcroft und Karp [21], müssen wir, wenn wir die Gesamtlaufzeit nicht verschlechtern wollen, auch die Kompressionsverfahren entsprechend beschleunigen. Dies ist aber leicht möglich, da die Laufzeit der Kompressionsverfahren unmittelbar von der Kantenzahl der Graphen abhängt, so dass wir lediglich die betrachteten Graphklassen weiter einschränken müssen.
Unser oben vorgestellter Algorithmus besteht im wesentlichen aus drei Teilen:

1. Wir finden einen annähernd regulären Teilgraphen, indem wir in einem geeignet konstruierten Netzwerk mit Hilfe des Algorithmus von Dinic[8] einen nahezu maximalen Fluss finden. Dies benötigt Zeit $\mathcal{O}(cm)$ Schritte.

2. Wir erweitern den gefundenen Teilgraphen mit zusätzlichen Knoten und Kanten zu einem regulären Graphen. Wir wenden den Algorithmus von Cole, Ost und Schirra [6] auf diesen regulären Graphen an und finden in dieser Erweiterung ein perfektes Matching in Zeit $\mathcal{O}(m)$.

3. Wir entfernen die Matchingkanten des erweiterten Graphen, die nicht zum ursprünglichen Graphen gehören. Danach erweitern wir das verbleibende Matching mittels alternierender Tiefensuche zu einem maximalen Matching. Dies benötigt Zeit $\mathcal{O}(m\frac{n^2}{c^2r})$, wobei r der Grad des regulären Graphen war.

In Kapitel 7.4137 haben wir den Parameter c so gewählt, dass der erste und der dritte Teil die gleiche Laufzeit haben. Wir werden nun beschreiben, wie eine Beschleunigung des dritten Teils durch Graphenkompression erfolgen kann, und dann diese Verbesserung durch eine andere Wahl von c teilweise auf den ersten Teil übertragen. Danach werden wir diskutieren, wie auch der erste Teil alleine beschleunigt werden kann. Der Aufwand des zweiten Teils wird von dem der beiden anderen dominiert. Er kann zudem schneller ausgeführt werden als jede Variante des Graphenkompressionsalgorithmus. Wir werden ihn daher im weiteren nicht betrachten.

8.3.1 Beschleunigung des dritten Teils

Im dritten Teil des Algorithmus werden mittels alternierendem DFS augmentierende Pfade gesucht, um die in den vorangegangenen Teilen übrig gebliebenen freien Knoten zu überdecken. Dies entspricht dem zweiten Teil des Algorithmus von Hopcroft und Karp [2]. Die Beschleunigung des Algorithmus von Hopcroft und Karp[21] durch Kompression wurde bereits von Feder und Motwani[11] in ihrer Arbeit demonstriert und entspricht der in Kapitel 2.1.222 beschriebenen Vorgehensweise: Durch die Kompression kann DFS einen alternierenden Baum in Zeit $\mathcal{O}\left(m\frac{\log\frac{2n^2}{m}}{\delta\log n}\right)$ konstruieren. Um einen augmentierenden Pfad zu finden und so zwei unüberdeckte Knoten neu zu überdecken, werden also

ebenfalls $\mathcal{O}\left(m\frac{\log\frac{2n^2}{m}}{\delta\log n}\right)$ Operationen benötigt.
Wir können diese Kompression also unverändert anwenden und erhalten somit für den dritten Teil des Algorithmus eine Gesamtlaufzeit von $\mathcal{O}\left(m\frac{n^2}{c^2r}\frac{\log\frac{2n^2}{m}}{\delta\log n}\right)$.
Die Gesamtlaufzeit des ersten und dritten Teils zusammen ist somit

$$\mathcal{O}\left(cm+\frac{m\log\frac{2n^2}{m}}{\delta\log n}\frac{n^2}{c^2r}\right).$$

Das Minimum wird wiederum angenommen, wenn beide Teile die gleiche Laufzeit haben. Es muss also gelten:

$$\begin{aligned} cm &= \frac{m\log\frac{2n^2}{m}}{\delta\log n}\frac{n^2}{c^2r} \\ \Leftrightarrow c^3r &= \frac{\log\frac{2n^2}{m}}{\delta\log n}n^2 \end{aligned}$$

Wir können nun dieses Ergebnis wahlweise einsetzen, um eine weitere Beschleunigung zu erhalten (Fall 1) oder um die Zahl der behandelbaren Graphen zu verändern (Fall 2). Man kann natürlich beides kombinieren, aber die Extremfälle erscheinen hier am interessantesten.

Fall 1: Bei Wahl von $r := \sqrt{n}\sigma^3$ erhält man

$$c=\sqrt[3]{\frac{\log\frac{2n^2}{m}}{\delta\log n}\frac{n^{\frac{3}{2}}}{\sigma^3}}=\frac{\sqrt{n}}{\sigma}\sqrt[3]{\frac{\log\frac{2n^2}{m}}{\delta\log n}}$$

Damit wird bei einer Gesamtlaufzeit von $\mathcal{O}(cm)=\mathcal{O}\left(\frac{\sqrt{n}m}{\sigma}\sqrt[3]{\frac{\log\frac{2n^2}{m}}{\delta\log n}}\right)$ das Ergebnis aus Satz 6.1126 übertroffen.

Fall 2: Bei konstanter Wahl von $c := \sqrt{n}/\sigma$ ergibt sich für r:

$$r=\frac{\log\frac{2n^2}{m}}{\delta\log n}\frac{n^2\sigma^3}{n^{\frac{3}{2}}}=\sqrt{n}\sigma^3\frac{\log\frac{2n^2}{m}}{\delta\log n},$$

so dass weniger Matchings im Graphen benötigt werden.

Nun müssen wir noch den Aufwand der Kompression mit $\mathcal{O}(mn^\delta\log^2 n)$ berücksichtigen. Variiert man die Konstante δ, so ändern sich lediglich die Kompressionslaufzeit $\mathcal{O}(mn^\delta\log^2 n)$ und - über die Bedingung $m > n^{2-\delta}$ - die Klasse der behandelbaren Graphen. Die asymptotische Kompressionsgüte $\frac{\log\frac{2n^2}{m}}{\delta\log n}$ ändert sich nicht, da δ hier als Faktor

auftritt, also in der $\mathcal{O}$-Notation verschwindet.
Feder und Motwani verlangen für ihren Algorithmus $m > n^{2-\delta}$ mit $\delta < \frac{1}{2}$. Wir werden δ nun so klein wählen, das die für die Kompression benötigte Zeit die Laufzeit des restlichen Algorithmus nicht übertrifft. Wir beschränken uns also auf Graphen mit noch mehr Kanten.
Im ersten Fall also für $r := \sqrt{n}\sigma^3$ muss

$$\underbrace{n^\delta \log^2 n}_{\text{Kompression}} = n^{\delta + \frac{\log\log^2 n}{\log n}} \leq n^{\frac{1}{2} - \frac{\log\sigma}{\log n} + \frac{\log \sqrt[3]{\frac{\log \frac{2n^2}{m}}{\log n}}}{\log n}} = \underbrace{\sqrt{n}/\sigma \sqrt[3]{\frac{\log \frac{2n^2}{m}}{\delta \log n}}}_{restlicher\,Algorithmus}$$

gelten. Wegen der Monotonie des Logarithmus muss also gelten:

$$\begin{aligned} \delta + \frac{\log\log^2 n}{\log n} &\leq \frac{1}{2} - \frac{\log\sigma}{\log n} + \frac{\log \sqrt[3]{\frac{\log \frac{2n^2}{m}}{\log n}}}{\log n} \\ \Leftrightarrow \delta &\leq \frac{1}{2} - \frac{\log\sigma}{\log n} - \frac{\log\log^2 n}{\log n} + \frac{\log \sqrt[3]{\frac{\log \frac{2n^2}{m}}{\log n}}}{\log n} \end{aligned}$$

Um in der gewünschten Zeit komprimierbar zu sein, muss G gemäß Kapitel 2.1.2[22] mindestens $n^{2-\delta}$ Kanten enthalten. Setzt man die obige Grenze für δ ein, und berücksichtigt $n^{\frac{\log x}{\log n}} = x$, so erhält man als benötigte Kantenzahl $(n^{1,5}) \frac{\sigma \log^2 n}{\sqrt[3]{\frac{\log \frac{2n^2}{m}}{\log n}}}$.
Im zweiten Fall $c := \sqrt{n}/\sigma$ ergibt sich bei analoger Rechnung: $\delta \leq \frac{1}{2} - \frac{\log\sigma}{\log n} - \frac{\log\log^2 n}{\log n}$, der Graph muss also lediglich $n^{1,5}\sigma \log^2 n$ Kanten enthalten.

8.3.2 Beschleunigung des ersten Teils

Im vorangegangenen Abschnitt haben wir gezeigt, dass man den dritten Teil des Algorithmus um $\frac{\log \frac{2n^2}{m}}{\log n}$ beschleunigen kann. Wir haben dann einen Teil dieser Beschleunigung durch neue Wahl des Parameters c, der den Übergang zwischen den verschiedenen Teilen regelt, an den ersten Teil abgegeben, um den gesamten Algorithmus zu verbessern.
In diesem Abschnitt wollen wir zeigen, wie man auch den ersten Teil beschleunigen kann, um auf diesen Ausgleich über c verzichten zu können.
Die Beschleunigung des ersten Teils erscheint auf den ersten Blick einfacher, da nun Flüsse gefunden werden müssen und nicht mehr knotendisjunkte Pfade. Bei der Beschleunigung des Algorithmus von Dinic[8] nach Feder und Motwani [11] erlaubte dies das einfache Ersetzen von Bicliquen durch Sterne (vgl. Kapitel 2.1.1[18]). Da jetzt aber mehr als eine Flusseinheit einen Knoten passieren kann, haben wir zwar noch Einheitskapazitäten auf den Kanten, aber nicht mehr auf den Knoten.
Eine Kante eines Sterns kann nun mehrere Kanten des alten Graphen repräsentieren, von

denen auch mehrere einen Fluß tragen können. Eine solche Kante muß also eine höhere Kapazität als 1 haben.
Dies ist aber nicht möglich, da diese Kanten hoher Kapazität im Fall überlappender Cliquen lange Pfade bilden können (vgl. Abb 8.1).

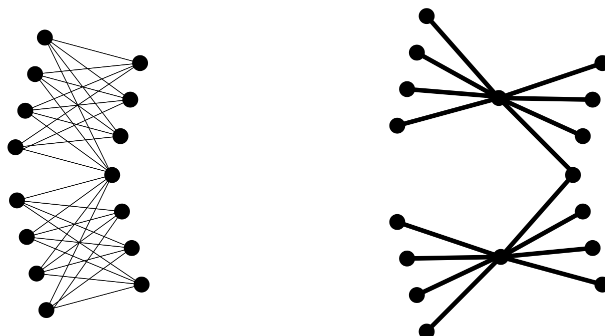

Abbildung 8.1: Überlappende Cliquen

Werden diese Pfade zu lang, funktionieren unsere Abschätzungen für die Größe des gefundenen Flusses nicht mehr (vgl. z.B. Lemma 8.16[152]), da wir davon ausgegangen sind, dass diese Pfade relativ kurz sind oder dass alle Kanten Kapazität 1 haben.
Wenn wir diese Art von Kompression also verwenden wollen, so müssen wir sicherstellen, dass wir die gefundene Lösung stets auf ein Einheits-Kapazitäts-Netzwerk zurück übertragen können. Diese Überprüfung wird zusätzliche Zeit in Anspruch nehmen.

Wir führen zunächst für jede Clique $C = (B\dot{\cup}A)$ eine Ersetzung durch, wie in Abb. 8.1 dargestellt, wobei die Kante von jedem Knoten b aus B zum "Zentrum" des Sterns Kapazität $|A|$ erhält und jede Kante zu einem Knoten aus A entsprechend Kapazität $|B|$. Somit kann jede Kante den gesamten Fluss all der Kanten des ursprünglichen Graphen tragen, die sie repräsentiert. Wir tun also genau das, was wir gemäß der Argumentation von eben nicht tun dürfen. Wir müssen also zusätzliche Bedingungen einführen, um die Korrektheit unseres Vorgehens dennoch sicherzustellen.
Im Gegensatz zum bisherigen Modell, in dem jeder Knoten Ein- oder Ausgangsgrad 1 hatte, können Knoten nun von mehr als einer Einheit passiert werden. Dies kann dazu führen, dass im komprimierten Graphen mehr als eine Einheit von einem Knoten aus $v \in B$ zu einem Knoten $u \in A$ geschickt wird. Dies ist aber im unkomprimierten Graphen nicht möglich (vgl. Abb 8.2).

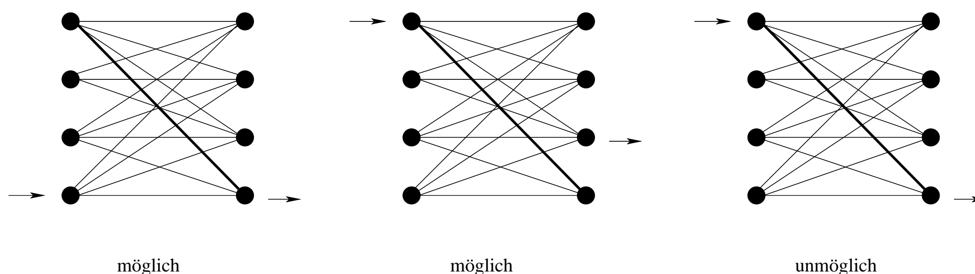

Abbildung 8.2: Flüsse im Einheits-Kanten- aber nicht Einheits-Knoten-Fall

Es muss nun also sichergestellt werden, dass von jedem Knoten v auf der Eingangsseite einer Clique zu jedem Knoten u auf der Ausgangsseite höchstens eine Flusseinheit fließt. Dies können wir überprüfen, indem wir die entsprechenden Flüsse auch im unkomprimierten Graphen eintragen und entsprechend dort wieder nachsehen, ob eine konkrete Flusseinheit auch im unkomprimierten Graphen den gleichen Weg nehmen könnte. Grundlage für unser Vorgehen ist die folgende Beobachtung:

Beobachtung 8.17 *Wenn es im unkomprimierten Graphen möglich ist, Fluss von einem Knoten aus B zu einem Knoten aus A zu schicken, so ist es auch in der komprimierten Repräsentation möglich, aber nicht zwangsläufig umgekehrt.*

Wir wissen also umgekehrt, dass es sicher keine Möglichkeit gibt, in der unkomprimierten Version eine Flusseinheit über die entsprechende Verbindung zu schicken, wenn dies in der komprimierten bereits nicht möglich war. Entsprechend prüfen wir für jeden Versuch zunächst, ob es in der komprimierten Darstellung möglich ist, und prüfen dann erst, ob es tatsächlich möglich ist.
Es können also bei jedem Versuch, eine weitere Möglichkeit zum Erweitern eines Flusses zu finden, drei Fälle auftreten:

1. Es ist bereits in der komprimierten Darstellung nicht möglich, über diese Kante einen Fluss zu schicken. Dies ist insbesondere der Fall, wenn der Endknoten der Kante bereits als “abgearbeitet” markiert wurde.

2. Es ist in der komprimierten Version möglich, aber in der ursprünglichen Version nicht möglich, einen Fluss über die Kante zu schicken.

3. Es ist in beiden Versionen möglich, einen Fluss über diese Kanten zu schicken.

Im ersten und im letzten Fall verhält sich unser Algorithmus genau wie der von Feder und Motwani [11]. Pro Kante kommt lediglich noch eine konstante Zahl von Operationen für die Durchführung des Tests hinzu. Rechnen wir nur diese beiden Fälle zusammen, so kann also jede Phase des Algorithmus von Dinic in $\mathcal{O}(m\frac{\log\frac{2n^2}{m}}{\log n})$ Schritten durchgeführt werden.
Wir müssen nun abschätzen, wieviel Aufwand Fall 2 verursacht. Hierzu betrachten wir nochmals genau, wann dieser Fall eintritt (vgl. Abb 8.3): Wir betrachten eine Biclique mit vier Knoten. Wird diese komprimiert, so wird sie entsprechend durch einen Stern mit vier “Strahlen” ersetzt. Jede Kante vom oder zum Zentrum des Sterns hat Kapazität zwei, da jeder der Knoten innerhalb der Clique zwei Flusseinheiten abgeben oder erhalten könnte.
Wir nehmen nun an, in einer früheren Phase wurde Fluss über die Kante $(B1, A1)$ geschickt. Diese Situation ist in Teil b) von Abb. 8.3 dargestellt.
In der komprimierten Repräsentation ergibt sich entsprechend ein Fluss der Größe 1 von $B1$ über das Zentrum des Sterns nach $A1$. Die Kanten von $B1$ zum Zentrum haben aber dennoch noch die Restkapazität 1, da es noch möglich wäre, eine Flusseinheit von $B2$ nach $A1$ und eine von $B1$ nach $A2$ zu schicken.

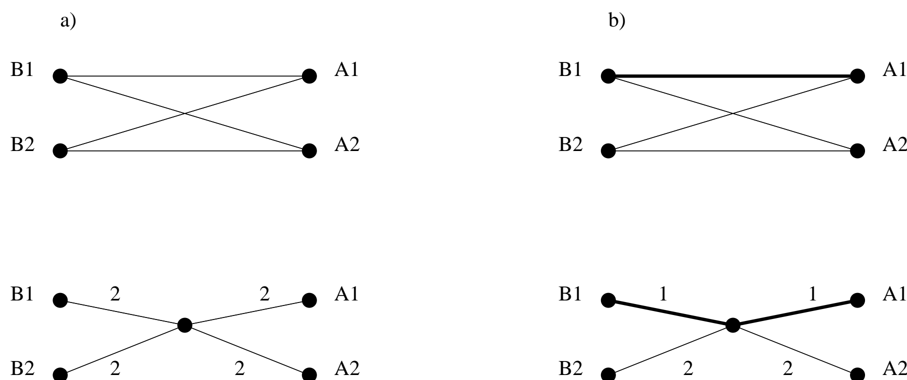

Abbildung 8.3: Beispiel zu Fall 2

Ein Algorithmus, der nur mit der komprimierten Darstellung arbeitet, wird also eine weitere Möglichkeit sehen, Fluss von $B1$ nach $A1$ zu schicken. An dieser Stelle überprüfen wir nun, ob dies auch im unkomprimierten Graphen möglich ist. In diesem Beispiel ist es nach Konstruktion natürlich nicht möglich und wir müssen den Vorgang abbrechen.
Im ungünstigsten Fall haben wir nun zwei Schritte umsonst gemacht: Den Schritt von $B1$ zum Zentralknoten des Sterns und die Untersuchung der Kante vom Zentralknoten nach $A1$. Die Laufzeit beider Schritte ist von der Gesamtgröße der Eingabe unabhängig. Jedes Auftreten von Fall 2 benötigt somit eine konstante Zahl von Operationen. Es genügt also abzuschätzen, wie oft dieser Fall auftritt.
Da Fall 2 genau dann auftritt, wenn durch die entsprechende Kante (im Beispiel $(B1,A1)$) schon ein Fluss fließt, und jede dieser Kanten pro Phase nur einmal betrachtet wird, kann die Häufigkeit des Auftretens von Fall 2 durch die Zahl der flusstragenden Kanten beschränkt werden. Wenn wir also deren Zahl per Voraussetzung begrenzen, erhalten wir den

Satz 8.18 *Sei G ein bipartiter Graph mit mehr als $n^{1,5}\frac{\sigma\log^2 n}{\sqrt[3]{\frac{\log\frac{2n^2}{m}}{\log n}}}$ Kanten, der $\sigma^3\sqrt{n}$ disjunkte perfekte Matchings enthält, wobei*

$$\sigma^3\sqrt{n} \leq \frac{m}{n}\frac{\log\frac{2n^2}{m}}{\log n}.$$

Dann kann ein perfektes Matching in G in Zeit $\mathcal{O}\left(\sqrt{n}m\frac{\log\frac{2n^2}{m}}{\sigma\log n} + n^2\sigma^2\right)$ gefunden werden.

Beweis: Der dritte Teil kann wie im vorangegangenen Kapitel beschrieben durch das Kompressionsverfahren von Feder und Motwani [11] um den Faktor $\frac{\log\frac{2n^2}{m}}{\sigma\log n}$ beschleunigt werden.
Wir verzichten nun darauf, diese Veränderung über die Modifikation des Parameters c an den ersten Teil weiterzugeben, sondern beschleunigen diesen wie eben beschrieben. Wir wiederholen zunächst die Beobachtungen des Beweises von Lemma 8.9149 (zu Details siehe dort):

1. Es kann pro Phase und Kante nur eine Kantenoperation geben, die nicht zu einer Flussänderung führt, da die entsprechende Kante in dieser Phase ansonsten nicht mehr betrachtet wird.

2. Die Zahl der Kantenoperationen, die im Folgenden zu einer Flussänderung auf dieser Kante führen, ist durch $\mathcal{O}((\sqrt{n}/\sigma)(n^{1,5}\sigma^3)) \leq \mathcal{O}(n^2\sigma^2)$ beschränkt, da der Gesamtfluss $n^{1,5}\sigma^3$ nicht übersteigt und jede neue Flusseinheit ihren Weg über höchstens $\sqrt{n}/\sigma$ Kanten nimmt.

Analog zum Beweis von Lemma 8.9[149] ergibt sich also, dass alle Operationen, die nichts mit Fall 2 zu tun haben, in Zeit $\mathcal{O}(\sqrt{n}m\frac{\log\frac{2n^2}{m}}{\sigma\log n})$ realisiert werden können.
Nun muss noch der oben beschriebene Fall 2, in dem eine Kantenoperation ggf. unmöglich ist, berücksichtigt werden: Wir haben eben gesehen, dass die Zahl der auf diese Weise pro Phase zusätzlich durchgeführten Operationen durch ein konstantes Vielfaches der Anzahl der flusstragenden Kanten beschränkt werden kann. In der Voraussetzung haben wir die Zahl der perfekten Matchings auf $\frac{m}{n}\frac{\log\frac{2n^2}{m}}{\log n}$ beschränkt. Jedes Matching besteht aus n Kanten, so dass wir beim Versuch, einen höchstens nahezu $\frac{m}{n}\frac{\log\frac{2n^2}{m}}{\log n}$-regulären Graphen zu erzeugen einen Fluss von höchstens $m\frac{\log\frac{2n^2}{m}}{\log n}$ Einheiten durch das Netzwerk schicken. Jede Flusseinheit nimmt den Weg s, B, A, t, verwendet also nur eine Kante, die von B nach A führt. Die Zahl der wegen Fall 2 abgebrochenen Kantenoperationen beträgt also höchstens $m\frac{\log\frac{2n^2}{m}}{\log n}$. Bei insgesamt $\frac{\sqrt{n}}{\sigma}$ Phasen ergibt sich also für Fall 2 eine zusätzliche Laufzeit von $\mathcal{O}(\sqrt{n}m\frac{\log\frac{2n^2}{m}}{\sigma\log n})$. □

Damit können beide langsamen Phasen beschleunigt werden.

Teil III

Zusammenfassung und Verzeichnisse

Kapitel 9

Zusammenfassung

Das Matchingproblem kann als eine eine der klassischen Fragestellungen der Graphentheorie bezeichnet werden. Die momentan schnellsten deterministische Algorithmen für dieses Problem sind der Algorithmus von Micali und Vazirani (1980-1994)[30] und der Algorithmus von Blum (1990-1999)[1][2]. Beide basieren auf dem Algorithmus von Hopcroft und Karp (1973)[21], dem schnellsten Algorithmus für den wichtigsten Spezialfall, den der bipartiten Graphen. All diese Algorithmen haben eine Laufzeit von $\mathcal{O}(\sqrt{n}m)$, wobei n die Zahl der Knoten des Graphen und m die Zahl der Kanten ist. Sowohl die Algorithmen für den bipartiten, als auch die für den nichtbipartiten Fall können für dichte Graphen mit dem Kompressionsverfahren von Feder und Motwani[11] von 1991 noch weiter beschleunigt werden [13][17].

Im ersten Teil dieser Arbeit zeigen wir verschiedene Möglichkeiten auf, wie sich das Kompressionsverfahren von Feder und Motwani [11] auf nichtbipartite Matchingprobleme übertragen läßt.
Das Kompressionsverfahren basiert auf der Idee, vollständige bipartite Teilgraphen - sogenannte Bicliquen - durch geeignete Graphen mit weniger Kanten zu ersetzen. Hierzu werden bisher sternförmige Graphen eingesetzt, die sich nur im Kontext von Erreichbarkeitsfragen und Flussalgorithmen äquivalent zu den ersetzten Cliquen verhalten. Matching-Probleme müssen daher zunächst in Flussprobleme umgewandelt werden. Dies führt im nichtbipartiten Fall zu sehr komplexen Algorithmen, da hier sogenannte schiefsymmetrische Flüsse behandelt werden müssen.
Wir vermeiden diese Umwandlung in ein schiefsymmetrisches Flussproblem und arbeiten direkt mit alternierenden Pfaden.
Wir konstruieren einen neuen Typ von Ersatzgraphen für Bicliquen. Dieser basiert auf sogenannten Superkonzentratoren [34]. Dies sind Graphen, die wenige Kanten haben und dennoch eine hohe Anzahl knotendisjunkter Pfade zulassen. Wir verändern diese Superkonzentratoren so, dass viele knotendisjunkte alternierende Pfade durch sie hindurchführen.
Dann verringern wir die Kantenzahl im betrachteten Graphen, indem wir möglichst viele Bicliquen durch diese kleineren Graphen ersetzen. Da die Ersatzgraphen sich ähnlich verhalten, wie die ersetzten Bicliquen, können wir auf den so entstehenden Graphen einen be-

liebigen anderen Matchingalgorithmus anwenden. Die Laufzeit dieses Algorithmus verringert sich aufgrund der nun geringeren Kantenzahl deutlich gegenüber einer direkten Anwendung des selben Algorithmus auf den ursprünglichen Graphen.
Auf diese Weise erhalten wir den folgenden Satz:

Satz 3.15$_{47}$: *Zu einem gegebenen nichtbipartiten Graphen $G = (V, E)$ mit $m := |E|$ und $n := |V|$ läßt sich in Zeit $\mathcal{O}(mn^\delta \log^2 n)$ ein Graph G_S mit $|E(G_S)| = \mathcal{O}\left(m \frac{\log \frac{2n^2}{m}}{\delta \log n}\right)$ Kanten konstruieren, so dass aus einem maximalen Matching M in G_S in Zeit $\mathcal{O}(m)$ ein maximales Matching M' in G berechnet werden kann.*

Die Verwendung von Ersatzgraphen setzt voraus, dass bei der Suche nach den Bicliquen zusätzliche Nebenbedingungen berücksichtigt werden. Zudem haben die Ersatzgraphen eine Kantenzahl, die zwar asymptotisch klein ist, aber so große Konstanten enthält, dass eine signifikante Beschleunigung erst bei sehr großen Graphen eintritt.
Wir stellen daher ein zweites Verfahren vor, das anstatt der expliziten Substitution von Bicliquen durch Ersatzgraphen einen abstrakten Datentyp verwendet, der genau in den zeitkritischen Phasen eingesetzt wird und das Verhalten der Ersatzgraphen nachbildet. Dies erlaubt es uns, die auftretenden Konstanten deutlich zu verringern und mit einer allgemeineren Form der Graphenkompression zu arbeiten. Der eingesetzte Matching-Algorithmus muss dafür an diese Vorgehensweise angepasst werden. Dies zeigen wir anhand des Matching-Algorithmus von Blum [1][2].

Im zweiten Teil der Arbeit wenden wir uns dem bipartiten Fall zu. Für den Spezialfall regulärer bipartiter Graphen - also bipartiter Graphen bei denen jeder Knoten den gleichen Grad hat - haben Cole, Ost und Schirra [6] 2001 einen Algorithmus mit linearer Laufzeit entwickelt.
Wir demonstrieren, wie sich dieser Algorithmus als Subroutine einsetzen läßt, um schnellere Algorithmen für eine größere Klasse von Graphen zu erhalten. Enthält ein Graph einen regulären Graphen als Teilgraphen, so kann man in diesem Teilgraphen mit dem Algorithmus von Cole, Ost und Schirra [6] ein perfektes Matching schnell finden. Ist der Teilgraph spannend, umfasst also alle Knoten, so ist dies auch ein perfektes Matching im gesamten Graphen.
Damit dieses Vorgehen funktioniert, muss man jedoch den regulären Teilgraphen explizit kennen. Dies ist normalerweise nicht der Fall. Die Suche nach einem solchen Teilgraphen ist zudem mindestens ebenso aufwändig wie das Lösen des Matchingproblems selbst.
Unser Verfahren sucht daher einen nahezu regulären Teilgraphen, erweitert diesen zu einem regulären Graphen und wendet dann auf diesen regulären Graphen den Algorithmus von Cole, Ost und Schirra [6] an. Damit die Zahl der bei der Erweiterung hinzugefügten Kanten im Verhältnis nicht zu groß wird, müssen wir voraussetzen, dass der nahezu reguläre Teilgraph selbst eine hohe Kantenzahl hat.
Einen regulären Teilgraphen eines bipartiten Graphen mit einer hohen Kantenzahl kann man als Vereinigung von kantendisjunkten perfekten Matchings interpretieren. Insgesamt erhalten wir somit das folgende Ergebnis:

Satz 6.1$_{126}$**:** *In einem bipartiten Graphen, der* $\sigma^3\sqrt{n}$ *disjunkte perfekte Matchings enthält, kann ein perfektes Matching in Zeit* $\mathcal{O}\left(\frac{\sqrt{n}m}{\sigma}\right)$ *gefunden werden.*

Damit identifizieren wir eine weitere große Teilklasse der bipartiten Graphen, für die ein deutlich effizienterer Algorithmus als der von Hopcroft und Karp[21] möglich ist.
Wir stellen mehrere Möglichkeiten vor, wie sich diese Klasse noch leicht erweitern läßt.
Abschließend zeigen wir, dass auch dieses Verfahren in vielen Fällen mit den Mitteln der Graphenkompression [11] beschleunigt werden kann.

Verzeichnis der Definitionen, Sätze und Lemmata

Satz 1.1 Berge 1952 6
Satz 1.2 Hopcroft, Karp 1972 7
Satz 1.3 Micali, Vazirani 1980 7
Satz 1.4 Feder, Motwani 1980-1994 7
Lemma 1.5 8
Satz 1.7 Hopcroft, Karp 9
Satz 1.8 Hopcroft, Karp 1973 10
Lemma 1.9 11
Lemma 1.10 12
Satz 1.12 13
Definition 1.13 13
Definition 2.1 18
Lemma 2.3 18
Definition 2.5 21
Definition 2.6 21
Lemma 2.8 21
Definition 2.10 [11] 22
Satz 2.11 Feder, Motwani[11] 22
Lemma 2.12 Feder, Motwani[11] 23
Definition 2.13 24
Definition 2.14 24
Definition 2.15 25
Definition 2.16 25
Definition 2.17 25
Definition 2.18 25
Lemma 2.19 25
Satz 2.21 Feder, Motwani 1991[11] 26
Lemma 2.22 27
Satz 2.23 Feder, Motwani 1991[11] 27
Satz 2.26 29

Definition 2.27 30
Definition 2.29 31
Lemma 2.30 32
Lemma 2.31 Blossom Expansion Lemma,[9],[41] 32
Definition 2.32 36
Lemma 2.33 [1] 36
Definition 2.34 36
Definition 2.35 37
Definition 2.36 37
Definition 3.1 39
Lemma 3.2 Goldberg, Karzanov 2003[17] 39
Lemma 3.3 40
Definition 3.5 CMA-Graph 42
Definition 3.7 42
Definition 3.8 43
Lemma 3.9 44
Definition 3.10 Defizit 45
Lemma 3.11 45
Lemma 3.12 45
Lemma 3.13 46
Satz 3.15 47
Lemma 3.17 48
Lemma 3.18 49
Lemma 3.19 49
Definition 3.20 50
Definition 3.21 50
Lemma 3.23 51
Definition 3.24 [15] 53
Definition 3.25 [15] 53
Satz 3.26 [15] 53
Lemma 3.28 [15] 54
Lemma 3.30 55
Definition 4.1 66

Definition 4.2 66
Satz 4.3 Micali, Vazirani, 1980-1989 [40] 66
Lemma 4.4 Blum 69
Lemma 4.5 Blum 69
Lemma 4.6 71
Lemma 4.12 Blum 1999 80
Lemma 4.13 Blum 1999 84
Lemma 4.14 Blum 1999 84
Satz 4.15 Blum 1999 84
Satz 4.16 Blum 84
Satz 4.17 84
Satz 4.18 84
Satz 4.19 89
Lemma 4.23 99
Lemma 4.25 100
Lemma 4.28 106
Lemma 4.29 106
Lemma 4.30 107
Lemma 4.32 108
Lemma 4.33 108
Lemma 4.34 109
Satz 4.35 110
Lemma 5.1 116
Lemma 5.3 117
Lemma 5.4 118
Definition 5.5 118
Definition 5.8 119
Definition 5.9 119
Definition 5.10 120
Definition 5.12 120
Definition 5.13 121
Satz 6.1 126
Definition 7.1 129

Satz 7.2 130
Lemma 7.3 130
Lemma 7.4 131
Satz 7.5 [35] 131
Lemma 7.6 133
Lemma 7.7 133
Lemma 7.8 134
Lemma 7.9 134
Lemma 7.10 134
Lemma 7.11 135
Lemma 7.12 135
Lemma 7.13 König 1916 [25] 136
Lemma 7.14 138
Satz 7.15 Hall 1935 [19] 138
Definition 7.16 138
Lemma 8.3 145
Lemma 8.4 146
Lemma 8.5 146
Lemma 8.6 147
Satz 8.7 148
Lemma 8.9 149
Definition 8.10 149
Satz 8.16 152
Satz 8.18 158

Abbildungsverzeichnis

1.1 Paarbildung bei Tanzveranstaltung 1

1.2 Zeichnerische Darstellung eines ungerichteten (a) und eines gerichteten Graphen (b) 2

1.3 a) Ausgangsgraph, b) nicht erweiterbares Matching, c) maximales Matching 3

1.4 Augmentierender Pfad / Augmentierung 4

1.5 Augmentierender Pfad 6

1.6 Blüte 7

1.7 Umwandlung in ein Erreichbarkeitsproblem 11

1.8 Umwandlung in ein Flussproblem 13

2.1 Ersetzen einer Clique durch einen Stern im Flussfall 19

2.2 Nachbarschaftsbaum von Knoten u 27

2.3 Kreis ungerader Länge und augmentierende Pfade 30

2.4 Schrumpfen einer Blüte 31

2.5 Beispiel zum Beweis von Lemma 2.31 Teil 1 33

2.6 Beispiel zum Beweis von Lemma 2.31 Teil 2 33

2.7 Beispiel zum Beweis von Lemma 2.31 Teil 3 34

2.8 Übertragen eines augmentierenden Pfades von G/B nach G 34

2.9 Umwandlung in ein bipartites Problem 35

2.10 Besuch von $[c, A]$ und $[c, B]$ 36

3.1 Cliquen in G und G_B 41

3.2 Ersetzen der Kanten einer Clique 41

3.3 Abstraktes Beispiel eines CMA-Graphen 42

3.4 Einsetzen eines CMA-Graphen 43

3.5 Korrespondierende Matchings in G und $G_{C_S/C}$ 44

3.6 Grobstruktur eines Superkonzentrators 51
3.7 CMA Graph 55

4.1 Beispiel für eine Tiefensuche ohne und mit Kompression 60
4.2 Überspringen eines Levels 63
4.3 Identifizieren einer Blüte 65
4.4 Gleichzeitige Rückwärtssuche bei DDFS 65
4.5 Vorzeitiger Zusammenstoß der beiden Köpfe der Rückwärtssuche 68
4.6 Blockade der Suche des rechten Kopfes durch eine “Sperre” 68
4.7 Ungültiger MDFS-Pfad 71
4.8 Hypothetischer längerer augmentierender Pfad 72
4.9 MDFS-Stack 73
4.10 Vorwärtssuche durch eine Unterblüte 77
4.11 Rückwärtssuche durch eine Unterblüte 78
4.12 Unterblüte mit mehreren eingehenden Kanten 78
4.13 “erweiterbare” Kante 79
4.14 Nachbarschaftsliste als Ersatz für den zentralen Knoten eines Sterns . . . 86
4.15 Entfernen eines Knotens aus der Nachbarschaftsliste 87
4.16 Datenstruktur zur Cliquenverwaltung 89
4.17 Aufsplitten von Cliquen 93
4.18 Erneuter Besuch eines Knotens bei der Rückwärtssuche 96
4.19 Clique in der Seite einer Blüte 97
4.20 Blockierter Repräsentant 97
4.21 Situation in einer Clique vor einer Rückwärtssuche 98
4.22 Quadratische Zahl erweiterbarer Kanten 104
4.23 Änderung der erweiterbaren Kanten 105

5.1 Cliquenersetzung bei expliziter Kompression für BFS 116
5.2 Künstliche Blüte 116
5.3 Ersetzung von Kanten und Cliquen 116
5.4 Einzig möglicher Pfad durch eine doppelt repräsentierte Kante 117

7.1 Veranschaulichung der Vorgehensweise 128
7.2 Das Finden eines regulären Teilgraphen als Flussproblem 131

7.3 Schnitt im residualen Netzwerk(Multikanten sind dick dargestellt) 134

8.1 Überlappende Cliquen 156
8.2 Flüsse im Einheits-Kanten- aber nicht Einheits-Knoten-Fall 156
8.3 Beispiel zu Fall 2 158

Liste der Algorithmen

1 Cliquenabspaltung nach [11] 26
2 Cliquenpartitionierung nach [11] 28
3 MBFS und MDFS[1] - Überblick 61
4 Realisierung des Algorithmus von Hopcroft und Karp im nichtbipartiten Fall unter Verwendung von MBFS und MDFS gemäß [1] 62
5 MBFS Vorwärtssuche 64
6 Double Depth First Search nach [40] angepaßt an [2] 70
7 MDFS nach [1] 81
8 Routine zum Bestimmen der Level in Blüten [1] 82
9 Subroutine CONSTRL nach [1] 82
10 Algorithmus zur Rekonstruktion des $s - t$ Pfades nach [1] 83
11 Subroutine RECONSTRQ [1] 83
12 SUGGEST - Routine zum Auffinden neuer Knoten 90
13 ACTIVATE - Routine zum Aktivieren eines Knotens 91
14 DEACTIVATE - Routine zum Deaktivieren eines Knotens 91
15 Beschleunigte MBFS Vorwärtssuche 95
16 Änderung der SUGGEST Routine zur Anpassung an das Union-Find-Problem 111
17 DDFS nach [40] angepasst an [2] mit Kompression 112
18 MDFS nach [1] mit Kompression 113
19 Levelbestimmung in Blüten mit Kompression 114
20 Subroutine CONSTRL nach [1] mit Kompression 114

Literaturverzeichnis

[1] Norbert Blum. Maximum matching in general graphs without explicit consideration of blossoms. Technical Report 85231-CS, University of Bonn, Department of Computer Science, September 10 2001.

[2] Norbert Blum. A simplified realization of the Hopcroft-Karp approach to maximum matching in general graphs. Technical Report 85232-CS, University of Bonn, Department of Computer Science, September 10 2001.

[3] Norbert Blum. *Theoretische Informatik*. Oldenbourg Verlag, 2001.

[4] N. Christophides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, grad. School of Ind. Admin, CMU, 1976. report 388.

[5] R. Cole and J. Hopcroft. On edge coloring bipartite graphs. *Siam J. Comput.*, 11, 1982.

[6] R. Cole, K. Ost, and S. Schirra. Edge-coloring bipartite multigraphs in $O(E \log D)$ time. *Combinatorica*, 21:5–12, 2001.

[7] Elias Dahlhaus and Marek Karpinski. On the computational complexity of matching on chordal and strongly chordal graphs. Technical report, Institute of Computer Science V, University of Bonn, February 23 1994.

[8] E.A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Sovi. Math. Dokl.*, 11:1277–1280, 1970. Zitiert nach [22].

[9] J. Edmonds. Maximum matching and a polyhedron with 0, 1 vertices. *Journal of Research National Bureau of Standards*, 69B:125–130, 1965.

[10] S. Even and O. Kariv. An $O(n^{2.5})$ algorithm for maximum matching in general graphs. In *16th Annual Symposium on Foundations of Computer Science*, pages 100–112, The University of California, Berkeley, 13–15 October 1975. IEEE.

[11] Tomás Feder and Rajeev Motwani. Clique partitions, graph compression and speeding-up algorithms. In *STOC proceedings*, pages 123–133, 1991.

[12] Tomás Feder and Rajeev Motwani. Clique partitions, graph compression and speeding-up algorithms. *J. Comput. Syst. Sci.*, 51(2):261–272, 1995.

[13] Freymuth-Paeger and Jungnickel. Balanced network flows. VIII. A revised theory of phase-ordered algorithms and the $o(\sqrt{n}mlog(n^2/m)/logn)$ bound for the non-bipartite cardinality matching problem. *Networks: An International Journal*, 41, 2003.

[14] Christian Freymuth-Paeger and Dieter Jungnickel. Balanced network flows i-viii. *Networks*, 33.1,37.4,39.1,41, 1999-2003.

[15] O. Gabber and Z. Galil. Explicit constructions of linear size concentrators and superconcentrators. In *20th Annual Symposium on Foundations of Computer Science (FOCS '79)*, pages 364–370, Long Beach, Ca., USA, October 1979. IEEE Computer Society Press.

[16] Harold N. Gabow, Haim Kaplan, and Robert E. Tarjan. Unique maximum matching algorithms. *J. Algorithms*, 40(2):159–183, 2001.

[17] Andrew V. Goldberg and Alexander V. Karzanov. Maximum skew-symmetric flows and matchings. In *Mathematical Programming*, pages 537–568, 2004.

[18] A.V. Goldberg and A.V. Karzanov. Maximum skew-symmetric flows. In *ESA 95*, 1995. zug. LNCS 979.

[19] P. Hall. On represantatives of subsets. *Journal of the London Matheamatical Society*, 10:26–30, 1935.

[20] R. E. Tarjan H.N.Gabov. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci*, 30:209–221, 1985.

[21] J. E. Hopcroft and R. M. Karp. A $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *Siam J. Comput.*, 2, 1973.

[22] D. Jungnickel. *Graphen, Netzwerke und Algorithmen*. BI-Wissenschaftsverlag, Mannheim, 1989.

[23] Marek Karpinski and Wojciech Rytter. *Fast Parallel Algorithms for Graph Matching Problems*. Oxford University Press, 1998.

[24] Alexander V. Karzanov and Andrew V. Goldberg. Maximum skew-symmetric flows and their applications to B-matchings, April 26 1999.

[25] Denes König. *Theorie der Graphen*. Chelsea Publishing Company, 1935. Nachdruck der AMS.

[26] Martin Löhnertz. Theorie und Praxis der automatischen Stundenplanerstellung, 1999.

[27] Martin Löhnertz. Finding one of many disjoint perfect matchings in a bipartite graph. In *Proceedings of GT04, Paris*, 2004.

[28] László Lovász and Michael D. Plummer. *Matching Theory*. Akadémiai Kiadó, North-Holland Publishing, 1986.

[29] Meena Mahajan and Kasturi R. Varadarajan. A new NC-algorithm for finding a perfect matching in bipartite planar and small genus graphs (extended abstract). In *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 351–357. ACM Press, 2000.

[30] S. Micali and V. Vazirani. An algorithm for maximum matching in general graphs. In *Proceedings of the 21st IEEE Annual Symposium on the Foundations of Computer Science*, pages 17–27, 1980.

[31] N.Alon and M.Capalbo. Smaller explicit superconcentrators. *Internet Mathematics*, 1.2:151–163, 2003.

[32] J. Peterson. Die Theorie der regulären Graphen. *Acta Math.*, 15:193–220, 1891. zitiert nach [28].

[33] M. S. Pinsker. On the complexity of a concentrator. *Proc. 7th int. teletraffic cong.*, 1973.

[34] N. Pippenger. Superconcentrators. *SIAM Journal of Computing*, 6(2):298–304, June 1977.

[35] James B. Orlin Ravindra K. Ahuja, Thomas L. Magnati. *Network flows*. Prentice Hall, New Jersey, 1993.

[36] Uwe Schöning. *Perlen der theoretischen Informatik*. BI-Wissenschaftsverlag, 1995.

[37] A. Schrijver. *Combinatorial Optimization*. Springer, 2003.

[38] R. L. Rivest T. H. Corman, C. E. Leiserson. *Introduction to Algorithms*. MIT Press, 1989.

[39] Leslie G. Valiant. On non-linear lower bounds in computational complexity. In ACM, editor, *Conference record of Seventh Annual ACM Symposium on Theory of Computing: papers presented at the Symposium, Albuquerque, New Mexico, May 5–May 7, 1975*, pages 45–53, New York, NY, USA, 1975. ACM Press.

[40] Vijay V. Vazirani. A theory of alternating paths ans blossoms for proving correctness of the $O(\sqrt{V}E)$ general graph maximum matching algorithm. *Combinatorica*, 14:71–109, 1994.

[41] Christoph Moll Winfried Hochstättler, Stephan Kromberg. A simple proof of the blossom expansion lemma. Technical report, Universität Köln, 1992.