technische universität
dortmund

Ein Branch-and-Cut Ansatz für das Maximum c-planare Subgraphen Problem

Mathias Jansen

Algorithm Engineering Report
TR07-1-010
November 2007
ISSN 1864-4503

Fakultät für Informatik
Algorithm Engineering (LS 11)
44221 Dortmund / Germany
http://ls11-www.cs.uni-dortmund.de/

UNIVERSITÄT DORTMUND
FACHBEREICH INFORMATIK

Diplomarbeit

Ein Branch-and-Cut Ansatz für das Maximum C-planare Subgraphen Problem

Mathias Jansen
26. November 2007

INTERNE BERICHTE
INTERNAL REPORTS

Diplomarbeit am
Fachbereich Informatik
der Universität Dortmund

Betreuer:
Prof. Dr. Petra Mutzel
Dipl. Inform. Karsten Klein

Kurzzusammenfassung

Gegenstand dieser Diplomarbeit ist ein Problem aus dem Forschungsbereich *Automatisches Zeichnen von Graphen und Diagrammen*. Eine der Hauptaufgaben dieses Forschungsbereichs liegt in der Entwicklung von automatischen Zeichen- und Layout-Algorithmen, um Graphen und Diagramme auf möglichst übersichtliche und gut nachvollziehbare Weise in der Ebene einzubetten und zu zeichnen. Mittels eines mächtigeren Graphenmodells, den sogenannten Clustergraphen, können zusätzlich hierarchische Inklusionsstrukturen modelliert werden. In vielen praktischen Anwendungsgebieten, z.B. Software-Engineering, Bio-Informatik, Prozess-Management, VLSI-Design, wird dies häufig zur Repräsentation bestimmter Informationen genutzt und stellt somit eine wünschenswerte Anforderung dar. Eine geringe Anzahl von Kantenkreuzungen in einer Zeichnung ist dabei ein wichtiges ästhetisches Güte-Kriterium. Die Eigenschaft eines Graphen, ohne Kantenkreuzungen in der Ebene gezeichnet werden zu können (Planarität), lässt sich in ähnlicher Weise auch für Clustergraphen definieren (C-Planarität). Es existieren Linearzeit-Algorithmen zum Testen eines Graphen auf Planarität. Im Falle von Clustergraphen ist die Komplexität des C-Planaritäts-Problems hingegen noch unbekannt und ein seit langem offenes Problem.
In dieser Arbeit wird ein praktischer Lösungsansatz vorgestellt, einen beliebigen Clustergraphen auf C-Planarität zu testen. Dazu wird ein allgemeineres Problem betracht, das NP-schwierige *Maximum C-planare Subgraphen Problem (MCPSP)*, das nach einem C-planaren Subgraphen fragt, der maximal viele der Kanten des gegebenen Clustergraphen enthält. Eine optimale Lösung für MCPSP beantwortet implizit auch die Frage, ob der gegebene Clustergraph C-planar ist.
Es wird ein Modell für das MCPSP entwickelt, das als ILP formuliert werden kann. Für dieses wird ein Branch-and-Cut Algorithmus implementiert zur optimalen Lösung dieses Problems. Anhand eines Benchmark-Sets von Clustergraphen wird der Algoritmus experimentell untersucht und die Ergebnisse ausgewertet und interpretiert. Die Analyse zielt zum Einen auf die Praxistauglichkeit des Algorithmus ab, zum Anderen wird das Optimierungs-Verhalten genauer untersucht, um tiefere Einsichten in die Schwierigkeit des Problems zu erhalten.

Abstract

This work deals with an algorithmic problem called *Maximum c-planar subgraph problem (MCPSP)* that can be sorted into the research area *Graph Drawing*. Graph Drawing considers the development of efficient algorithms for determining „good“ drawings and layouts of graphs in the plane with the purpose of giving an appropriate visualization of the information it models. A more powerful class of graphs are so called *clustered graphs* which can be used to model hierarchical inclusion structures, which is needed in many practical application areas. Considering clustered graphs the *planarity* property is extended to *cluster-planarity (c-planarity)*. The algorithmic complexity of the problem to determine, if a given clustered graph is c-planar, is unknown, whereas testing planarity of classical graphs is in P. However, the problem is well studied and efficient algorithms have been developed for specific classes of clustered graphs, e.g. cluster-connected ones.
MCPSP is a more general problem that asks for a c-planar subgraph with a maximum number of edges. An optimal solution to this problem also yields an answer to the question, whether the given clustered graph is c-planar or not. In this work a practical approach for solving MCPSP to optimality, and thus solving the c-planarity problem for arbitrary clustered graphs, is developed. Therefore we give an ILP formulation for this problem, which is based on the theorem that completely connected clustered graphs are c-planar iff the underlying graph is planar. The ILP is solved and examined by means of a Branch-and-Cut algorithm that has been implemented for this purpose. Branch-and-Cut algorithms have shown out to be a very appropriate and promising approach in exact combinatorial optimization. According to a set of benchmark clustered graphs the algorithm is tested and analyzed considering practical usability etc. Non-cluster-connected clustered graphs are examined extensively, since no efficient algorithm for testing c-planarity for those clustered graphs could have been developed.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Gegenstand dieser Diplomarbeit ist eine praktische, experimentelle Aufgabenstellung, die sich in das Forschungsgebiet „Automatisches Zeichnen von Graphen und Diagrammen“ einordnen lässt. Graphen modellieren Beziehungen zwischen Objekten. Allgemein spricht man bei den Objekten eines Graphen von *Knoten* und bei den Beziehungen zwischen Objekten von *Kanten*. In erster Linie handelt es sich bei einem *Graphen* um eine abstrakte Datenstruktur, die in einer Vielzahl von Algorithmen auftaucht. Zum Einen lassen sich viele algorithmische Probleme auf natürliche Weise als graphentheoretisches Problem beschreiben, zum Anderen tauchen Graphen häufig in Subroutinen auf, oder werden zur effizienten Verwaltung von Daten verwendet. Die Graphentheorie ist ein sehr umfassend studiertes Gebiet in der Mathematik und der Informatik. Es existiert eine Fülle von Veröffentlichungen zu diesem Thema.
Durch Graphen modellierte Zusammenhänge sind sehr eingängig und gut nachvollziehbar. Daher sind diese auch in vielen verschiedenen praktischen Anwendungsgebieten fernab der Informatik ein beliebtes und geeignetes Mittel zur Repräsentation bestimmter Informationen und Zusammenhänge, die zum besseren Verständnis für den menschlichen Betrachter graphisch visualisiert werden. Anwendungsgebiete sind unter anderem:

- Bio-Chemie und Bio-Informatik. Graphen und Diagramme tauchen hier zum Beispiel zur Visualisierung metabolischer Reaktionswege auf. Abbildung 1.1 [1] zeigt ein Beispiel.

- VLSI-Design. Bauteile und die verbindenden Leiterbahnen implizieren einen Graphen.

- Software-Engineering. In diesem Bereich tauchen Graphen und Diagramme unmittelbar auf (UML-Diagramme, Klassendiagramme, Sequenzdiagramme etc.). Dynamische Abläufe in einem Softwaretool können ebenfalls schematisch als Diagramm bzw. Graph dargestellt werden (siehe Abbildung 1.2 [2]).

- Prozess-Management. Eine Vielzahl von Geschäftsprozessen und -abläufen, wie zum Beispiel Kaufabwicklungen, Produktionsabläufe, Kommunikationswege zwischen Ab-

[1] http://www2.ufp.pt/ pedros/bq/integration.htm
[2] http://www.perlcabal.org/ audreyt/svk-overview.png

teilungen eines Unternehmens, werden schematisch als Diagramme mit Entitäten und Verbindungslinien visualisiert (siehe Abbildung 1.3 [3]).

- Datenbank-Entwurf. Bei in diesem Bereich geläufigen Diagrammen, wie zum Beispiel *(Structured) Entity-Relationship-Modelle*, handelt es sich auch um Graphen. Die Visualisierung solcher Diagramme ist eine obligatorische Anforderung.

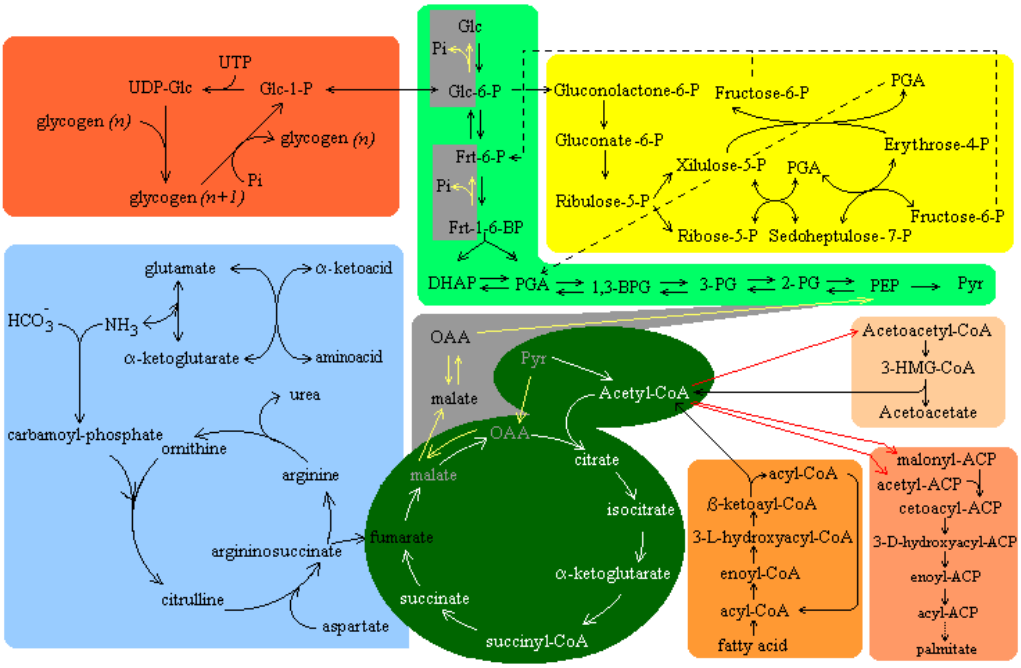


Abbildung 1.1: Ein metabolischer Reaktionsweg

Ist das Erstellen eines Diagramms oder Graphen eine eher selten auftretende Aufgabe und besitzen diese eine übersichtliche Größe, so reicht es häufig sicherlich aus, diese per Hand oder mittels eines geeigneten Grafikprogramms zu zeichnen. Dabei muss dann natürlich eine gute Zeichnung intuitiv und durch „Ausprobieren“ erzielt werden. Wenn das Erstellen derartiger Zeichnungen allerdings eine regelmäßig anfallende Anforderung ist, oder die zu zeichnenden Diagramme aus vielen Knoten und Kanten bestehen, ist ein unterstützendes Softwaretool hilfreich. In diesem Zusammenhang stellt sich zunächst die unmittelbare Frage: wie muss oder sollte ein Graph *gezeichnet* werden, damit die durch diesen repräsentierten Informationen und Zusammenhänge möglichst „gut“ visualisiert werden, also für den menschlichen Betrachter eingängig und leicht nachzuvollziehen sind? Mit dieser Frage und der Entwicklung algorithmischer Lösungen für die in diesem Zusammenhang auftauchenden Probleme und Fragestellungen beschäftigt sich das Forschungsgebiet *Automatisches Zeichnen von Graphen.*

[3] http://www.netcons.net/_itil/service_management.jpeg

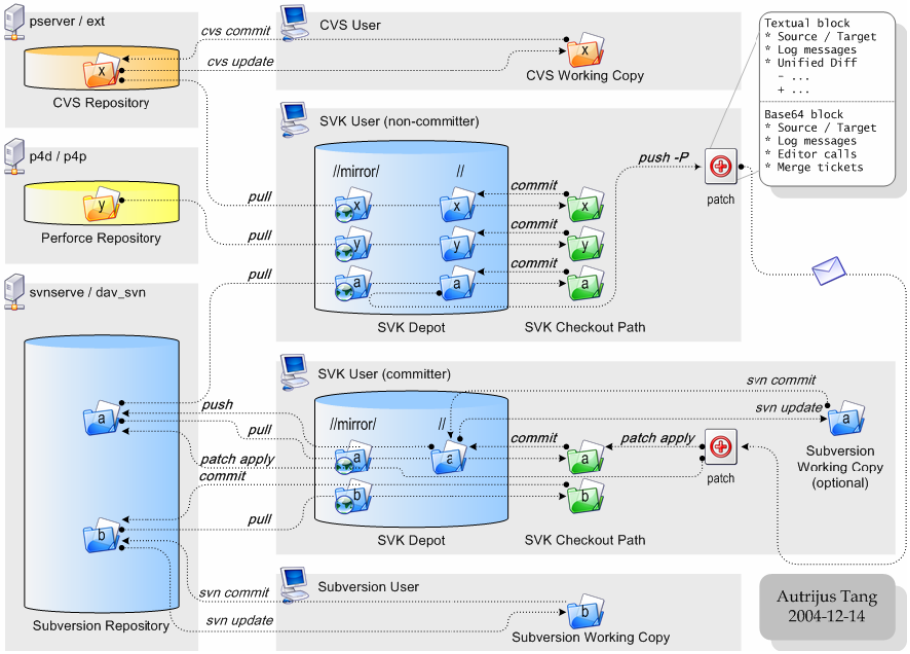


Abbildung 1.2: Schematischer Aufbau des SVK (ein verteiltes Version Control Tool).

1.1 Automatisches Zeichnen von Graphen

Eine eindeutige und allgemeingültige Antwort auf die Frage, wie eine „gute“ Zeichnung eines Graphen auszusehen hat, gibt es natürlich nicht, da die Klassifizierung in „gute“ und „schlechte“ Zeichnungen stark von der konkreten Anwendung abhängig ist, aus der die Graphen stammen. Außerdem unterliegen Zeichnungen je nach Kontext bestimmten *semantischen Restriktionen*, sodass zum Beispiel ein bestimmtes Layout erfüllt werden muss, damit eine Zeichnung überhaupt ‚korrekt“ ist. Es existieren allerdings auch *ästhetische* Kriterien, die allgemein darauf abzielen, das Verständnis und die Eingängigkeit der Zeichnung für den Betrachter zu fördern [Pur97, HMM00, PMCC01, Pur02, WPCM02]. Um diese allgemein formulieren zu können, reduzieren wir im Folgenden die Vielzahl denkbarer, grafischer Visualisierungsmöglichkeiten von Diagrammen und Graphen auf den einfachen Fall, dass die Objekte, die Knoten eines Diagramms oder Graphen durch einfache Punkte, und die Beziehungen zwischen diesen, die Kanten, durch Verbindungslinien bzw. Polygonzüge dargestellt werden. Wir beschränken uns außerdem auf den Fall, dass die Diagramme bzw. Graphen im zweidimensionalen Raum, also in der Ebene gezeichnet werden sollen. Im Folgenden wird zudem lediglich der Begriff *Graph* verwendet. Einige ästhetische Kriterien sind im Folgenden aufgelistet und duch die schematischen Zeichnungen in Abbildung 1.4 veranschaulicht, bei denen es sich jeweils um denselben Graphen handelt:

- Kantenlänge. Durch kurze Kantenlängen können die Beziehungen der Objekte zueinander von einem menschlichen Betrachter in der Regel einfacher nachvollzogen werden (Abbildung 1.4 (d)).
- Minimierung der Kantenknicke. Eine hohe Anzahl an Kantenknicken beeinflusst die Übersicht in einer Zeichnung meist negativ (Abbildung 1.4 (c)).

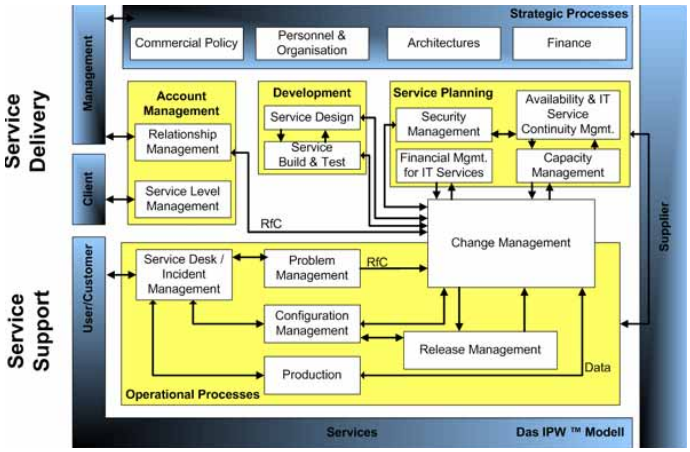


Abbildung 1.3: Service-Management nach ITIL (IT Infrastructure Library) der Firma NetConsult

- Minimierung der Kantenkreuzungen. Je weniger Kanten sich in einer Zeichung überkreuzen, desto leichter ist diese im Allgemeinen nachzuvollziehen (Abbildung 1.4 (d)). Jede Kreuzung „unterbricht“ den Verlauf einer Kante und macht es schwieriger, dem Kantenverlauf zu folgen (Abbildung 1.4 (a), (b), (c)).

- Minimierung der benötigten Zeichenfläche. „Unnötig“ große Abstände der Graphenelemente zueinander können sich negativ auf die Güte der Zeichnung auswirken, da dies meist auch zu längeren Kanten führt, die vom Betrachter verfolgt werden müssen. Insgesamt wirkt eine eher kompaktere Zeichnung, in der zueinander in Beziehung stehende Elemente nahe beisammen liegen, meist übersichtlicher (Abbildung 1.4 (d)).

- Kantenaustrittswinkel. Viele spitze Kantenwinkel oder vom Knoten ausgehend parallel verlaufende Kanten wirken sich in der Regel eher negativ auf das Verständnis der Zeichnung aus (Abbildung 1.4 (a)).

- Ausnutzung von Symmetrie-Eigenschaften. Existieren in einem Graphen bestimmte Symmetrien, so sollten diese möglichst auch durch die Zeichnung bzw. das Layout wiedergespiegelt werden.

Eine gute Zeichnung des Graphen aus Abbildung 1.4 ist also offensichtlich (d), was in erster Linie daran liegt, dass dieser keine Kantenkreuzungen beinhaltet. Im direkten Vergleich zu den Zeichnungen (a) und (c) gewinnt man sogar auf den ersten Blick den Eindruck, als enthielte dieser weniger Kanten, was allerdings nicht der Fall ist. In (b) sind die Knoten zwar symmetrisch angeordnet. Die Kantenkreuzungen machen die Zeichnung dennoch „unschön“.

Die „gleichzeitige“ Optimierung der vorgestellten ästhetischen Kriterien ist im Allgemeinen nicht möglich. Allein die Berechnung einer Zeichnung, in der nur eines der Kriterien optimiert ist, stellt meist keine triviale Aufgabe dar. So sind zum Beispiel die algorithmischen Probleme der *Kantenkreuzungsminimierung* und der *Kantenknickminimierung* NP-schwierig [GJ83], [GT94].

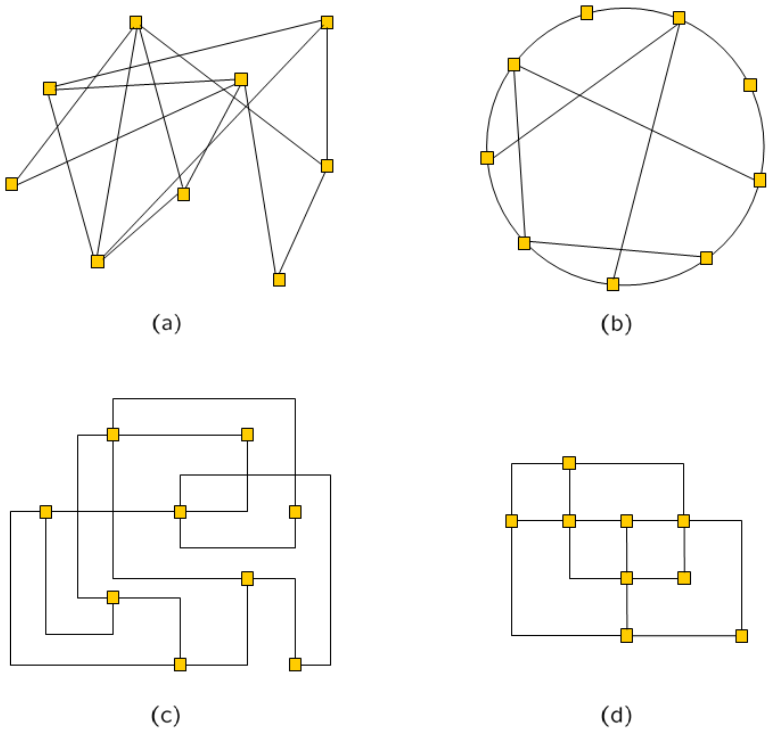


Abbildung 1.4: Verschiedene Zeichnungen desselben Graphen

1.2 Erweiterte Graphmodelle

Durch zunehmende Anforderungen und den immer stärker werdenen Bedarf an Modellierungs- und Visualisierungsmöglichkeiten bestimmter Informationen, reichen in vielen praktischen Anwendungsgebieten, wie z.B. den bereits genannten, klassische Graphen oft nicht aus, um den gestellten Anforderungen gerecht zu werden und die geforderten Informationen und Zusammenhänge zu modellieren. Das betrifft insbesondere, wie zuvor schon kurz angemerkt, bestimmte *semantisch bedingte Kriterien*, die im Kontext der Anwendung durch eine Zeichnung erfüllt sein müssen. Es existieren in diesem Zusammenhang eine Reihe mächtigerer Graphentypen, mittels derer komplexere Informationen und Zusammenhänge modelliert werden können. Einige prominente Beispiele sind:

- *Hypergraphen*
- *Compound-Graphen*
- *Higraphs*
- *Cigraphs*

Hypergraphen

In Hypergraphen [Ber85] sind Kanten nicht ausschließlich auf zwei Knoten beschränkt, sondern können eine beliebige Knotenmenge umfassen. Sie werden daher als *Hyperkanten* be-

zeichnet. Geeignete Visualisierungsstile sind jedoch nicht leicht zu entwickeln. Dies betrifft vor allem die geeignete Zeichnung der Hyperkanten. Es existieren verschiedene Ansätze diese zu visualisieren, zum Beispiel durch Umrandung der zur Hyperkante gehörenden Knoten, oder durch „verzweigende“ Verbindungslinien. Abbildung 1.5 (a) zeigt diese beiden Stile. Selbst für kleine Hypergraphen wird es für den menschlichen Betrachter allerdings meist schon schwierig, die modellierten Zusammenhänge nachzuvollziehen. Auch in Bezug auf automatische Layout- und Zeichenalgorithmen stellen Hypergraphen ein schwieriges Problem dar.

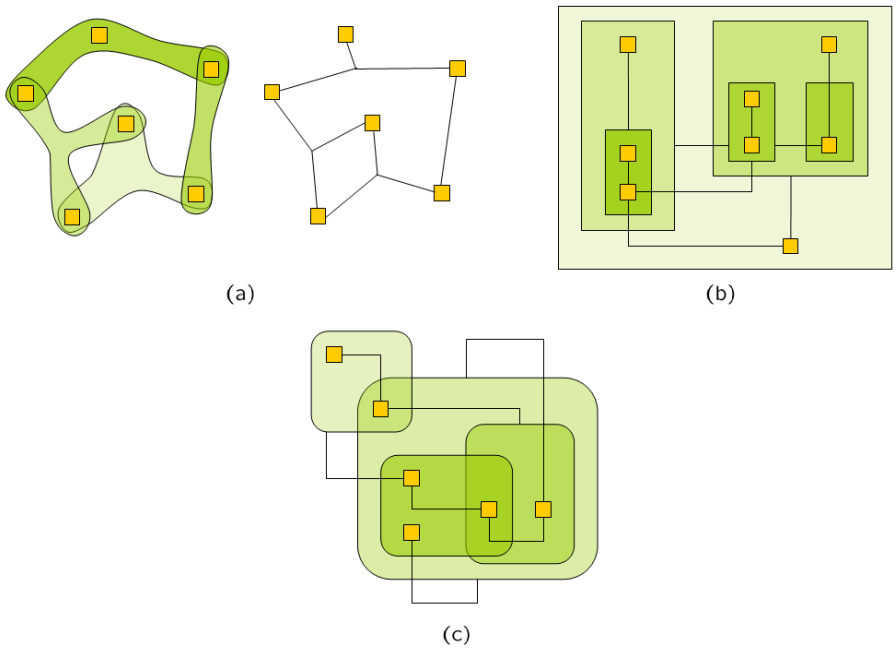


Abbildung 1.5: Zeichnungen von Graphen unterschiedlicher Graphmodelle. (a) Hypergraph, (b) Compound-Graph, (c) Higraph

Compound-Graphen

Compound-Graphen [For02] bestehen aus einem klassischen Graphen, zusammen mit einer Inklusionsstruktur, die auf den Knoten des Graphen definiert ist. In einer Zeichnung eines Compound-Graphen können die Inklusionsstrukturen durch ineinander verschachtelte umschließende Flächen repräsentiert werden, die jeweils genau die Knoten enthalten, die semantisch „zusammen gehören“. Die *Compounds* eines solchen Graphen werden dabei ebenfalls als Knoten angesehen, sodass Kanten zwischen beliebigen Compounds verlaufen dürfen. Abbildung 1.5 (b) zeigt einen einfachen Compound-Graphen.

Higraphen

Higraphen [Har95] stellen im Prinzip eine zusätzliche Erweiterung von Compound-Graphen dar. Im Gegensatz zu diesen, erlauben Higraphen auch eine Überschneidung

von Inklusionsstrukturen. Durch Higraphen können daher zum Beispiel *StateChart*-Diagramme modelliert werden. Ein Beispiel dazu findet sich in Abbildung 1.5 (c).

Cigraphen

Cigraphen [LE96] sind den Compound-Graphen ebenfalls sehr ähnlich. In ihrer grafischen Visualisierung unterscheiden sich diese im Prinzip nicht. Die Idee hinter Cigraphen besteht darin, den Graphen explizit und formal in Subgraphen zu unterteilen, sodass zum Beispiel verschiedene Layout-Algorithmen auf verschiedene Subgraphen des Cigraphen angewendet werden können. Dies erfordert unter anderem eine gesonderte Behandlung derjenigen Kanten, die zwischen verschiedenen Compounds bzw. Subgraphen verlaufen.

Automatisches Zeichnen erweiterter Graphmodelle

Diese vorgestellten erweiterten Graphmodelle bieten alle einen hohen Level an Abstraktion und eignen sich in vielen Bereichen um die geforderten Informationen darzustellen. Automatische Layout-Algorithmen für diese Graphmodelle zu entwickeln, ist hingegen ein schwieriges Unterfangen, zum Einen algorithmisch aufgrund der erhöhten Komplexität, die sich durch die erweiterten strukturellen Eigenschaften dieser Graphentypen ergibt, zum Anderen auch aufgrund der Tatsache, dass es nicht leicht ist, überhaupt geeignete Visualisierungs-Stile zu entwerfen (siehe Hypergraphen). Ein hohes Maß an Abstraktion muss also meist durch einen Mangel an anschaulichen Visualisierungsmöglichkeiten sowie dem Fehlen guter automatischer Layout- und Zeichenalgorithmen erkauft werden. Für die vorgestellten Graphmodelle existieren nur verhältnismäßig wenige Layout- und Zeichenalgorithmen, wie zum Beispiel [BM99].

Einen guten Kompromiss stellen hierbei die sogenannten *Clustergraphen* dar. Mittels dieser ist es möglich, Inklusions- bzw. Hierarchiestrukturen auf der Menge der Knoten des Graphen zu modellieren. Anders als bei den vorgestellten Compound-Graphen werden die Inklusionsstrukturen, Cluster genannt, hier allerdings nicht als ineinander verschachtelte Knoten behandelt, sondern durch die Cluster lediglich eine (semantische) Zusammengehörigkeit impliziert. Daher verlaufen Kanten nach wie vor nur zwischen Knoten des „eigentlichen“ Graphen. Diese Inklusionsstrukturen werden im Allgemeinen durch sich nicht überschneidende, ineinander verschachtelte geschlossene Flächen repräsentiert, die die zugehörigen Knoten beinhalten. Diese Art der Informationsrepräsentation findet verbreitete Anwendung und ist häufig vom Grad der sich dadurch ergebenden Modellierungsmöglichkeiten ausreichend. Bei den Beispiel-Graphen aus den Abbildungen 1.1, 1.3 und 1.2 handelt es sich um Clustergraphen.

1.3 Problemstellung

Wie bereits in Bezug auf die klassischen Graphen motiviert, stellt die Minimierung der Kantenkreuzungen ein wichtiges Kriterium für eine gut nachvollziehbare und übersichtliche Zeichnung dar. Dies gilt natürlich auch in Bezug auf Clustergraphen. In diesem Zusammenhang ist die Frage interessant, ob ein gegebener Graph bzw. Clustergraph kreuzungsfrei in der Ebene gezeichnet werden kann. Diese Eigenschaft eines Graphen nennt

sich *Planarität* bzw. *Cluster-Planarität* in Bezug auf Clustergraphen. Für klassische Graphen existieren effiziente Algorithmen, die einen Graphen auf diese Eigenschaft hin testen. Bei Clustergraphen handelt es sich hingegen um ein schwieriges algorithmisches Problem. Es ist bisher trotz intensiven Studiums noch nicht gelungen, einen effizienten Algorithmus zu entwickeln, der einen beliebigen Clustergraphen auf Cluster-Planarität testet.

Im Rahmen dieser Arbeit wird eine Variante dieses Problems betrachtet, das *Maximum c-planare Subgraphen Problem (MCPSP)*, und ein Modell zur optimalen Lösung für dieses entworfen. Das Problem besteht darin, für einen gegebenen Clustergraphen einen Cluster-planaren Teilgraphen zu berechnen, der maximal viele Kanten des gegebenen Clustergraphen enthält. Eine optimale Lösung für dieses Problem impliziert nun auch gleichzeitig eine Antwort auf die Frage, ob der gegebene Clustergraph Cluster-planar ist. Das entwickelte Modell kann als ganzzahliges lineares Programm (ILP) formuliert werden. In dieser Diplomarbeit wird nun ein praktischer, experimenteller Ansatz verfolgt, indem ein Branch-and-Cut Algorithmus implementiert wird, mittels dessen sich das MCPSP anhand der ILP-Formulierung optimal lösen lässt. Der implementierte Algorithmus wird ausgiebig auf seine Performance untersucht, insbesondere auch in Hinblick auf seine Praxistauglichkeit.

Eines der Schwerpunktgebiete am Lehrstuhl 11 für Algorithm Engineering der Universität Dortmund ist die Entwicklung und Implementierung von Algorithmen für Anwendungsprobleme. Dies beinhaltet insbesondere auch Graphenlayoutprobleme. Es existiert eine Bibliothek für Graphen-Algorithmen, das *Open Graph Drawing Framework (OGDF)*, das ursprünglich aus der *AGD* [GJK+01] hervorgegangen ist. Neue zu implementierende Algorithmen aus diesem Anwendungsgebiet werden in der Regel in die OGDF integriert, so auch der im Rahmen dieser Diplomarbeit implementierte Branch-and-Cut Algorithmus.

Im folgenden Kapitel werden zunächst eine Reihe benötigter Grundbegriffe definiert und eine formale Charakterisierung von Clustergraphen und Cluster-Planarität sowie alle in diesem Zusammenhang notwendigen Kenntnisse vermittelt. Anschließend wird dann die zugrunde liegende Problemstellung formalisiert.

Kapitel 2

Grundlagen

Dieses Kapitel dient dazu, dem Leser alle nötigen Kenntnisse zu vermitteln, die er zum Verständnis des in der Diplomarbeit behandelten Themas benötigt. Es werden weiterhin vorangegangene Arbeiten und theoretische Erkenntnisse und Ergebnisse zu der betrachteten Problemstellung vorgestellt. In Abschnitt 2.1 werden zunächst einige grundsätzliche Definitionen und Eigenschaften in Bezug auf allgemeine Graphen gegeben, sowie der in diesem Zusammenhang wichtige Begriff der Planarität. Die Klasse der Clustergraphen wird in Abschnitt 2.2 definiert, sowie der Begriff der „Cluster-Planarität“ der die Grapheneigenschaft „Planarität“ auf Clustergraphen erweitert. In Abschnitt 2.3 wird eine kurze Übersicht über bisherige Arbeiten bezüglich des Testens eines Clustergraphen auf C-Planarität gegeben.

2.1 Graphen

Definition 2.1.1. ***Graph.*** *Ein Graph G ist ein 2-Tupel $G = (V, E)$ zweier disjunkter Mengen V und E mit $E \subseteq V \times V$. Die Elemente $v \in V$ bezeichnet man als* Knoten*, die Knotenpaare $e = (v, w)$ mit $e \in E,\ v, w \in V$ als* Kanten*. Graphen bzw. dessen Kanten können* gerichtet *oder* ungerichtet *sein. Ungerichtete Kanten werden durch $e = \{v, w\}$, gerichtete Kanten durch $e = (v, w)$ gekennzeichnet.*

Im gerichteten Fall nennt man die beiden zu einer Kante $e = (v, w)$ gehörenden Knoten auch *Anfangs-* bzw. *Endpunkt* der Kante. Dadurch bekommt diese also eine Richtung. Im ungerichteten Fall spricht man lediglich von Endpunkten. Falls nicht explizit anders angegeben werden im Folgenden ausschließlich ungerichtete Graphen betrachtet. Die Anzahl der Knoten, also die Kardinalität der Menge $|V|$ eines Graphen $G = (V, E)$, wird in der Literatur meist mit n bezeichnet, die Kardinalität der Kantenmenge $|E|$ mit m. Bei den beiden folgenden Definitionen handelt es sich lediglich um formale Begriffsbildungen.

Definition 2.1.2. ***Adjazent.*** *Zwei Knoten $v, w \in V,\ v \neq w$ eines Graphen $G = (V, E)$ heißen* adjazent *zueinander, genau dann wenn eine Kante $e = \{v, w\} \in E$ existiert. Zwei Kanten $e, f \in E$ sind adjazent zueinander, genau dann wenn $e \cap f \neq \emptyset$ und $|e \cap f| \neq 2$ gilt, e und f also genau einen gemeinsamen Endpunkt haben.*

Definition 2.1.3. ***Inzident.*** *Gegeben sei ein Graph $G = (V, E)$. Eine Kante $e = \{v, w\}, e \in E, v, w \in V$ ist zu einem Knoten $u \in V$* inzident*, genau dann wenn gilt $u = v \vee u = w$.*

Definition 2.1.4. ***(Induzierter) Subgraph.*** *Gegeben sei ein Graph $G = (V, E)$. Ein* Subgraph *von G ist ein Graph $G' = (V', E')$ mit $V' \subseteq V$ und $E' \subseteq \{\{v, w\} \mid v, w \in V',\ \{v, w\} \in E\}$. Gilt statt $E' \subseteq \{\{v, w\} \mid v, w \in V',\ \{v, w\} \in E\}$ die stärkere Bedingung $E' = \{\{v, w\} \mid v, w \in V',\ \{v, w\} \in E\}$, so spricht man bei $G' = (V', E')$ von dem* (durch V') induzierten Subgraphen *von G.*

Ein Subgraph wird in der Literatur häufig auch als *Teilgraph* bezeichnet.

Definition 2.1.5. ***Bipartiter Graph.*** *Ein Graph $G = (V, E)$ heißt* bipartit*, falls es eine Partition der Knoten in V in zwei disjunkte Mengen V' und $V \backslash V'$ mit $V' \subset V$ gibt, sodass gilt:*

$$\forall\, e = \{v, w\} \in E: \quad v \in V' \wedge w \in V \backslash V' \quad \vee \quad v \in V \backslash V' \wedge w \in V'$$

Graphen können je nach Struktur der Knoten- und Kantenmenge eine Vielzahl unterschiedlicher Eigenschaften besitzen. So verbietet die allgemeine Definition eines Graphen aus 2.1.1 zum Beispiel nicht, dass manche Kanten „mehrmals“ im Graphen enthalten sind. In diesem Fall spricht man von einem Multigraphen.

Definition 2.1.6. ***Multigraph.*** *Gegeben sei ein Graph $G = (V, E)$. Seien $v, w \in V$ zwei Knoten von G. Gibt es paarweise verschiedene Kanten $e_1, \ldots, e_k$ in E, $k \in \mathbb{N}$, mit $e_i = \{v, w\}\ \forall\, i \in \{1, \ldots, k\}$, so bezeichnet man $\{e_1, \ldots, e_k\}$ als* Multikante*. Ein Graph G, der Multikanten enthält, heißt* Multigraph*.*

Definition 2.1.7. ***Schleife.*** *Existiert in einem Graphen $G = (V, E)$ eine Kante $e = \{v, w\},\ v, w \in V$ mit $v = w$, so bezeichnet man e als* Schleife*.*

Im Rahmen dieser Diplomarbeit beschränken wir uns grundsätzlich auf schleifenfreie Graphen, die keine Multikanten besitzen. Es folgen einige weitere grundlegende Begriffe und Definitionen im Zusammenhang mit Graphen, die auch zum Verständnis bestimmter in folgenden Kapiteln beschriebener Sachverhalte notwendig sind.

Definition 2.1.8. ***Pfad, Kreis.*** *Gegeben sei ein ungerichteter Graph $G = (V, E)$. Ein* Pfad *π in G der Länge k, $k \in \mathbb{N}$ ist definiert als eine Folge $(v_0, v_1, \ldots, v_k)$, $v_i \in V$, $i \in \{0, \ldots, k\}$, mit $\{v_i, v_{i+1}\} \in E$, $i \in \{0, \ldots, k-1\}$. Sind alle Knoten des Pfades paarweise verschieden, so ist π ein* einfacher *Pfad. Gilt $v_0 = v_k$ so wird π* Kreis *in G genannt.*

Definition 2.1.9. ***Azyklisch.*** *Ein Graph $G = (E, V)$, der keine Kreise enthält, heißt* azyklisch*.*

Definition 2.1.10. ***(k)-Zusammenhang.*** *Gegeben sei ein Graph $G = (V, E)$. G heißt* zusammenhängend*, wenn für je zwei beliebige Knoten $v, w \in V, v \neq w$ ein Pfad zwischen v und w in G existiert. G heißt* k-zusammenhängend *für ein $k > 1, k \in \mathbb{N}_0$, falls $|V| > k$ und $G' = (V \backslash V', E)$ zusammenhängend ist für alle $V' \subseteq V$ mit $|V'| < k$.*

Im Falle $k = 1$ spricht man also einfach von einem zusammenhängenden Graphen. Im Falle $k = 2$ nennt man G auch *biconnected* und die einelementigen Mengen V' *Schnittknoten.*

Ist ein Graph G k-zusammenhängend mit $k > 1$, so ist G per Definition auch $(k-1)$-zusammenhängend.

Wir nennen zwei Knoten $v, w \in V$ eines Graphen $G = (V, E)$ *erreichbar*, falls ein Pfad π zwischen v und w in G existiert. Bezüglich der Erreichbarkeit von Knoten lässt sich nun eine Äquivalenzrelation $\rightleftharpoons$ auf der Menge der Knoten V definieren:

$$v \rightleftharpoons w \quad \Leftrightarrow \quad \exists \pi = (v, u_1, \ldots, u_k, w),\ v, w, u_i \in V,\ i \in \{1, \ldots, k\},\ k \in \mathbb{N}$$

Definition 2.1.11. ***Zusammenhangskomponente.*** *Gegeben sei ein Graph $G = (V, E)$. Seien $V_1, \ldots, V_k$ mit $k \in \{1, \ldots, |V|\}$ die Äquivalenzklassen bezüglich der Erreichbarkeitsrelation $\rightleftharpoons$ und sei für ein $i \in \{1, \ldots, k\}$ die Kantenmenge E_i definiert als $E_i = \{\{v, w\} | v, w \in V_i\}$. Dann bezeichnet man den durch V_i induzierten Subgraphen $G_i = (V_i, E_i)$, $i \in \{1, \ldots, k\}$ als* Zusammenhangskomponente *von G. Die Mengen V_i, $i \in \{1, \ldots, k\}$ sind per Definition disjunkt und es gilt $\bigcup_{1 \leq i \leq k} V_i = V$.*

Eine besondere Klasse von Graphen sind die sogenannten *Bäume*. Diese sind wie folgt charakterisiert:

Definition 2.1.12. ***Baum.*** *Ein Graph $G = (V, E)$ heißt* Baum*, wenn er folgende Eigenschaften hat:*

- *G ist zusammenhängend.*
- *$|E| = |V| - 1$.*

Definiert man einen ausgezeichneten Knoten $v_r \in V$, so lässt sich den Knoten von G eine eindeutige Tiefe *bzw. ein* Level *zuordnen, wenn der Graph von v_r startend mittels Breitensuche durchlaufen wird. v_r wird dann die* Wurzel *des Baumes genannt.*

Durch die Wurzel erhält ein Baum eine gerichtete Struktur. Auch wenn der gegebene Graph G ungerichtet ist, bezeichnet man die Kanten von G, falls G ein gewurzelter Baum ist, häufig trotzdem als „von oben nach unten gerichtet“. Die Kanten verlaufen nur zwischen benachbarten Leveln. Liegt bezüglich einer Kante $e = \{v, w\}$ der Knoten v höher als w, so nennt man v *Vorgänger* oder *Vater* von w. Andersherum ist dann w ein *Nachfolger* bzw. ein *Kind* von v. Knoten, die keine Kinder haben, heißen *Blätter*.

Definition 2.1.13. ***Spannbaum.*** *Gegeben sei ein Graph $G = (V, E)$. Ein Subgraph $T = (V, E')$, $E' \subseteq E$ heißt* Spannbaum *von G, falls T ein Baum ist.*

Definition 2.1.14. ***Minimaler Spannbaum.*** *Gegegeben sei ein Graph $G = (V, E)$ und eine Funktion $w : E \mapsto \mathbb{R}^+$, die jeder Kante $e \in E$ einen positiven, reellen Wert zuweist. Sei $E' \subseteq E$, dann sind die Kosten von E' bezüglich w definiert als: $w(E') = \sum_{e \in E'} w(e)$. Damit sind die Kosten $w(T)$ für einen Spannbaum $T = (V, E')$ von G definiert als $w(E')$. Ein Spannbaum $T_m = (V, E_m)$ von G heißt nun* minimaler Spannbaum von G *falls gilt:*

$$w(T_m) \leq w(T) \qquad \forall \quad T \text{ Spannbaum von } G$$

2.1.1 Planarität

Im vorangegangenen Kapitel wurde motiviert, dass eine kreuzungsfreie Zeichung eines Graphen, bzw. eine Zeichnung mit möglichst wenigen Kantenkreuzungen, von einem menschlichen Betrachter einfacher nachzuvollziehen ist. Die Eigenschaft eines Graphen, kreuzungsfrei in der Ebene gezeichnet werden zu können, nennt sich *Planarität.*

Definition 2.1.15. ***Planarität.*** *Ein Graph $G = (V, E)$ heißt* planar *genau dann wenn er ohne Kantenkreuzungen in der Ebene gezeichnet werden kann.*

Kuratowski gab 1933 eine interessante Charakterisierung planarer Graphen. Er zeigte, dass die kleinsten nicht-planaren Graphen der K_5 (der vollständige Graph mit 5 Knoten) und der $K_{3,3}$ (der vollständige bipartite Graph mit je 3 Knoten pro Partition) sind, und dass jeder nicht-planare Graph mindestens eine K_5- oder $K_{3,3}$-*Subdivision* als Subgraph enthält. Eine *Subdivision* eines Graphen G, geht aus diesem dadurch hervor, dass die Kanten des Graphen durch einfache Pfade ersetzt werden. Anders ausgedrückt geht eine Subdivision durch beliebig häufiges (endliches) *Splitten* der Kanten von G hervor. Unter dem Splitten einer Kante $e = \{v, w\}$ versteht man dabei die Ersetzung der Kante $e = \{v, w\}$ durch einen Pfad (e_1, e_2) mit $e_1 = \{v, u\}$, $e_2 = \{u, w\}$, wobei u ein neuer Knoten ist. Eine *Kuratowski-Subdivision* bezüglich eines Graphen $G = (V, E)$ ist nun wie folgt definiert:

Definition 2.1.16. ***Kuratowski-Subdivision.*** *Gegeben sei ein Graph $G = (V, E)$. Existiert ein Subgraph $G' = (V', E')$ in G, sodass dieser einer Subdivision des K_5 oder $K_{3,3}$ entspricht, so bezeichnet man G' als* Kuratowski-Subdivision.

Theorem 2.1.1. ***Satz von Kuratowski*** *Ein Graph $G = (V, E)$ ist genau dann planar, wenn er keine K_5- und keine $K_{3,3}$-Subdivision als Subgraph enthält.*

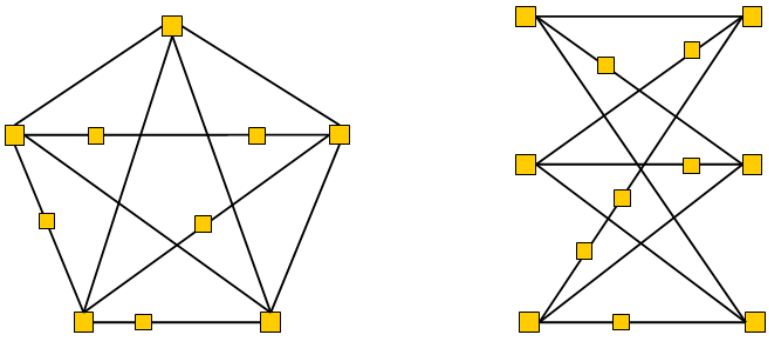


Abbildung 2.1: K_5 und $K_{3,3}$ Subdivisions

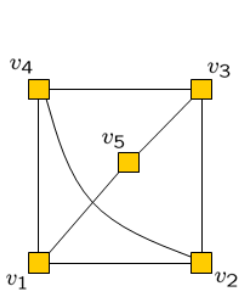
Ein graphisches Beispiel zu K_5- und $K_{3,3}$-Subdivisions findet sich in Abbildung 2.1. Die Zeichungen sind so dargestellt, dass die Strukturen von K_5 und $K_{3,3}$ gut zu erkennen sind. Es gilt insbesondere auch, dass eine Kuratowksi-Subdivision durch Löschung einer beliebigen Kante planar wird.

Diese Charakterisierung planarer Graphen ist zum Testen eines Graphen auf Planarität allerdings nicht von großem Nutzen. Kuratowski-Subdivisions werden allerdings für das

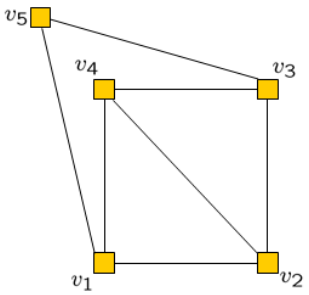
in dieser Diplomarbeit behandelte Problem noch von großer Bedeutung sein. Eine andere Charakterisierung planarer Graphen erfolgt über den Begriff der *Einbettung* eines Graphen.

Definition 2.1.17. ***Kombinatorische Einbettung.*** *Gegeben sei ein Graph $G = (V, E)$ Eine* kombinatorische Einbettung *von G ist definiert durch eine feste zyklische Reihenfolge der zu den Knoten inzidenten Kanten.*

Abbildung 2.2 zeigt zwei verschiedene Einbettungen desselben Graphen. Die beiden Tabellen zeigen jeweils die fixen, zyklischen Adjazenzlisten der Knoten v_1 und v_3 (jeweils entgegen der Uhrzeigerrichtung). Die Einbettung links impliziert eine Zeichnung, in der sich eine Kantenkreuzung nicht vermeiden lässt. In der Einbettung rechts haben die beiden Adjazenzlisten eine andere zyklische Reihenfolge, sodass in einer Zeichnung, die dieser Einbettung zugrunde liegt, der Knoten v_5 nun „außen“ liegt. Dadurch kann der Graph ohne Kantenkreuzung gezeichnet werden.



Knoten	Adjazenzliste
v_1	v_2, v_5, v_4
v_3	v_4, v_5, v_2



Knoten	Adjazenzliste
v_1	v_2, v_4, v_5
v_3	v_5, v_4, v_2

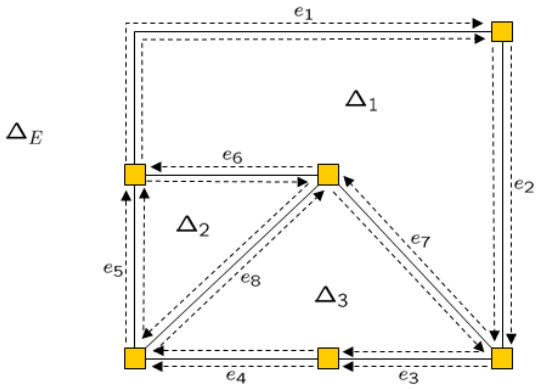
Abbildung 2.2: Einbettungen eines Graphen

Die fixe, zyklische Reihenfolge der Kanten in den Adjanzenzlisten definiert die Topologie des Graphen in der Ebene. Durch die Definition der kombinatorischen Einbettungen ist somit eine Äquivalenzrelation auf der Menge aller möglichen Zeichnungen definiert. Dabei werden zwei Zeichnungen als kombinatorisch äqivalent angesehen, wenn diesen dieselbe kombinatorische Einbettung zugrunde liegt. Um die Eigenschaft der Planarität zu charakterisieren, ist es somit nicht erforderlich, sich auf konkrete Zeichnungen zu beziehen.

Durch die Einbettung eines Graphen werden außerdem geschlossene Flächen (*Faces*) definiert, die sich anhand einer Zeichnung, die dieser Einbettung zugrunde liegt, ergeben. Ein Beispiel dazu ist in Abbildung 2.3 gegeben. Δ_E ist dabei das äußere oder *externe* Face, durch das letztendlich die Topologie des Graphen eindeutig bestimmt ist. Dadurch lässt sich nun auch formal eine *planare Einbettung* definieren.

Definition 2.1.18. ***Planare Einbettung.*** *Eine* planare Einbettung *eines Graphen $G = (V, E)$ ist definiert als eine kombinatorische Einbettung von G zusammen mit der Wahl eines externen Faces.*

Definition 2.1.19. ***Planarität.*** *Ein Graph $G = (V, E)$ ist genau dann planar, wenn eine planare Einbettung für G existiert.*



Faces	definierende Kantenfolge
Δ_1	e_1, e_2, e_7, e_6
Δ_2	e_5, e_6, e_8
Δ_3	e_8, e_7, e_3, e_4
Δ_E	e_5, e_1, e_2, e_3, e_4

Abbildung 2.3: Faces definiert durch die Einbettung

Das Testen eines Graphen auf Planarität ist kein triviales Problem. Vielmehr handelt es sich bei den existierenden Algorithmen um komplizierte und ausgeklügelte Verfahren. Ein Graph kann exponentiell viele Einbettungen besitzen, sodass eine Enumeration aller Einbettungen und Testen, ob es sich jeweils um eine planare Einbettung handelt, kein effizientes Vorgehen ist. Eine gegebene Einbettung auf Planarität zu überprüfen, ist jedoch einfach. Euler definierte und bewies bereits 1758 die nach ihm benannte *Eulersche Polyeder-Formel* für planare Graphen.

Theorem 2.1.2. ***Eulersche Polyeder-Formel.*** *Gegeben sei ein planarer Graph $G = (V, E)$. Betrachte eine planare Einbettung $\mathcal{E}$ durch die f Faces definiert sind. Dann gilt:*

$$|V| - |E| + f = 2$$

Eine Einbettung lässt sich demnach also auf Planarität testen, indem man die durch diese definierten Faces zählt und die Gleichung überprüft. Als Folgerung aus dieser Formel ergibt sich außerdem eine obere Schranke für die Anzahl der Kanten in einem planaren Graph.

Theorem 2.1.3. *Gegeben sei ein Graph $G = (V, E)$. Ist G planar, so gilt:*

$$G \text{ planar } \Rightarrow |E| \leq 3|V| - 6$$

Der erste Planaritätstest wurde 1961 von Auslander und Parter entwickelt [AP61], der quadratische Laufzeit hat. Es folgten eine Reihe weiterer Algorithmen, Korrekturen und Verbesserungen. 2004 wurde dann von Boyer und Myrvold ein Linearzeit-Planaritätstest vorgestellt [BM04], der nach Angaben der Autoren vergleichsweise „einfach“ nachzuvollziehen und zu implementieren sei. Dieser Algorithmus ist auch im Rahmen einer vorangegangenen Diplomarbeit [Sch07] am Lehrstuhl 11 der Universität Dortmund implementiert und in die Algorithmen-Bibliothek OGDF integriert worden.

2.1.2 Planarisierung

Planarisierungs-Verfahren (siehe zum Beispiel [MW98]) werden verwendet, um für nichtplanare Graphen eine Einbettung zu berechnen, die möglichst wenige Kantenkreuzungen impliziert. Das Problem der *Kreuzungsminimierung* ist jedoch NP-schwierig, wie bereits in der Einleitung kurz erwähnt. Das grundsätzliche Prinzip ist wie folgt:

1. Zunächst wird durch Löschung möglichst weniger Kanten ein planarer Subgraph berechnet.
   (a) Anschließend werden die gelöschten Kanten iterativ wieder in den Graphen eingefügt. Dabei wird jeweils versucht, die nächste Kante so einzufügen (einzubetten), dass dadurch möglichst wenig neue Kantenkreuzungen entstehen.
   (b) Falls eine eingebettete Kante neue Kantenkreuzungen induziert, werden diese durch *Dummy-Knoten* ersetzt, so dass der resultierende Graph wieder planar eingebettet ist und als Ausgangspunkt für die nächste Iteration verwendet werden kann.
2. Sind alle Kanten eingebettet, kann der Graph dann durch einen Layout-Algorithmus gezeichnet werden.
3. Zum Schluss werden alle Dummy-Knoten wieder entfernt und durch Kantenkreuzungen ersetzt.

2.2 Clustergraphen

Im Folgenden wird das bereits informell beschriebene erweiterte Graphmodell *Clustergraph* vorgestellt und formal definiert. Durch diese Graphen können Inklusionsstrukturen modelliert werden, die durch ineinander verschachtelte *Cluster* repräsentiert werden. Somit kann zum Beispiel für bestimmte Knoten, die aus semantischen, technischen, oder je nach Anwendung unterschiedlichsten Gründen „zusammen gehören“ , deren Verbindung bzw. Zusammengehörigkeit dadurch modelliert werden, dass diese zum selben Cluster gehören. Die Cluster lassen sich ineinander schachteln, so dass auf diese Weise eine hierarchische Struktur auf den Knoten des Graphen entsteht, die durch einen Baum repräsentiert werden kann. Für den weiteren Verlauf der Arbeit werden in Bezug auf Clustergraphen die Notationen aus [EFC95] verwendet.

2.2.1 Definition Clustergraph

Ein Clustergraph $C = (G, T)$ besteht aus einem Graphen $G = (V, E)$ und einem gewurzelten Baum T, dessen Blätter exakt den Knoten aus V entsprechen. Man bezeichnet G als den *zugrunde liegenden Graphen* und T als *Inklusionsbaum* von C. Jeder innere Knoten ν von T repräsentiert einen Cluster. Die Blätter des an ν gewurzelten Teilbaums von T entsprechen exakt denjenigen Knoten aus V, die in ν enthalten sind. Wir bezeichnen die entsprechende Knotenmenge eines Clusters ν mit $V(\nu)$. T beschreibt eine Inklusionsstruktur zwischen den einzelnen Clustern von C. Ist ν' ein Nachfolger von ν in T, so wird ν' *Subcluster* von ν genannt, und ν *Supercluster* von ν'. Der durch $V(\nu)$ induzierte Subgraph von G wird mit $G(\nu)$ gekennzeichnet.

Abbildung 2.4 (a) zeigt eine Zeichung eines Clustergraphen. Die Cluster wurden hier jeweils durch einfache, geschlossene Flächen gekennzeichnet. Es ist zu erkennen, dass der Cluster ν_3 „innerhalb“ des Clusters ν_2 liegt. Die durch die Zeichnung repräsentierte Clusterstruktur ist in (b) durch den zugehörigen Inklusionsbaum dargestellt. Die Wurzel des Inklusionsbaums, der *root-Cluster*, korrespondiert zu dem Supercluster, der den gesamten Graphen beinhaltet. Der Knoten ν_3 des Inklusionsbaums ist ein Nachfolger des Knotens

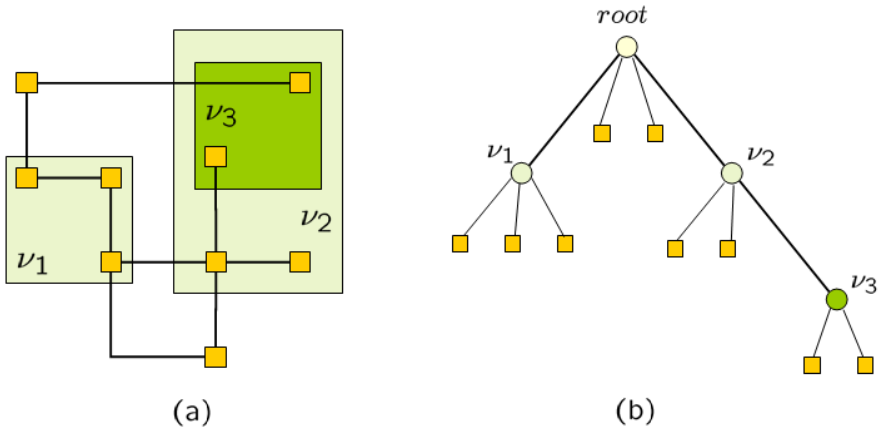


Abbildung 2.4: (a) Zeichnung eines Clustergraphen (b) zugehöriger Inklusionsbaum

ν_2, was per Definition die in (a) geschachtelt gezeichnete Inklusion von $V(\nu_3)$ in ν_2 repräsentiert. Das heißt, die Knoten in $V(\nu_3)$ liegen sowohl in ν_3 als auch in ν_2.

Zeichnung eines Clustergraphen

Die in Abbildung 2.4 dargestellte Zeichnung des Clustergraphen ist intuitiv und eingängig. Die Inklusionsstrukturen bzw. Cluster sind aufgrund deren Visualisierung durch die rechteckigen Flächen klar zu erkennen. Feng, Eades und Cohen haben in [EFC95] formalisiert, was eine konsistente Zeichnung eines Clustergraphen ist.

Definition 2.2.1. *[EFC95]* ***Zeichnung Clustergraph.*** *In einer Zeichnung eines Clustergraphen $C = (G = (V, E), T)$ korrespondiert zu jedem Knoten ν von T eine einfache, geschlossene Region R. Diese „umrandet“ in der Zeichnung den durch $V(\nu)$ induzierten Subgraphen. R wird dabei durch eine einfache geschlossene Kurve repräsentiert, sodass folgende Bedingungen erfüllt sind:*

- *die Regionen aller Subcluster des zu R gehörenden Clusters ν liegen vollständig im Inneren von R.*
- *die Regionen aller anderen Cluster liegen vollständig außerhalb von R.*
- *gibt es eine Kante $e = \{v, w\}$ mit $v, w \in V(\nu)$, so verläuft e vollständig innerhalb von R, kreuzt also nicht die Kurve, die die Grenze von R definiert.*

Es ist leicht zu sehen, dass die Zeichnung in Abbildung 2.4 die obigen Bedingungen erfüllt, und somit eine konsistente Zeichnung des Clustergraphen darstellt.

Aus derselben Motivation heraus wie bei allgemeinen Graphen, ist in Bezug auf die grafische Visualisierung eines Clustergraphen *Planarität* von besonderer Bedeutung. Die Charakterisierung von Planarität kann jedoch nicht unmittelbar von den allgemeinen Graphen auf Clustergraphen übertragen werden. Wohingegen jedoch für die in Abschnitt 1.2 vorgestellten, erweiterten Graphklassen bislang noch kein Planaritätsbegriff definiert wurde,

stellten Feng, Eades und Cohen in ihrem Paper für die Klasse der Clustergraphen eine Charakterisierung von Planarität vor, die sogenannte *Cluster-Planarität (C-Planarität).*

Definition 2.2.2. *[EFC95]* ***C-planare Zeichnung Clustergraph.*** *Eine Zeichnung eines Clustergraphen, die den Anforderungen aus Definition 2.2.1 genügt, heißt* Cluster-planar (C-planar)*, falls diese keine Kreuzungen zwischen Kanten und keine Kreuzungen zwischen einer Kante und einer Region enthält.*

Eine Kante-Region-Kreuzung besteht genau dann, wenn eine Kante die begrenzende Kurve einer Region mehr als einmal durchkreuzt. Eine C-planare Zeichnung eines Clustergraphen impliziert natürlich auch zwangsläufig eine planare Zeichnung des zugrunde liegenden Graphen. Eine C-planare Einbettung definiert wie bei klassischen Graphen eine Menge möglicher C-planarer Zeichnungen, die dieselbe Topologie besitzen.

Definition 2.2.3. ***C-planare Einbettung*** *Eine* C-planare Einbettung *eines Clustergraphen $C = (G, T)$ besteht aus einer planaren Einbettung $\mathcal{E}$ des zugrunde liegenden Graphen, sowie einem fixen, externen Face, derart, dass für alle Knoten ν in T gilt: die Knoten $V \backslash V(\nu)$ liegen in einer planaren Zeichnung gemäß $\mathcal{E}$ alle im externen Face des durch $V(\nu)$ induzierten Graphen $G(\nu)$.*

In ihrem Paper [EFC95] geben Feng, Eades und Cohen eine Charakterisierung von C-planaren Clustergraphen anhand der beiden folgenden Theoreme an.

Definition 2.2.4. ***Cluster-Zusammenhang*** *Ein Clustergraph $C = (G, T)$ heißt* Cluster-zusammenhängend (C-zusammenhängend)*, genau dann wenn:*

$$\forall\ \nu \in T : G(\nu) \text{ ist zusammenhängend}$$

Ist ein Cluster-induzierter Graph $G(\nu)$ nicht zusammenhängend, so besteht $G(\nu)$ aus k Zusammenhangskomponenten $\nu^1, \ldots, \nu^k$, $k \in \{1, \ldots, |V|\}$. Die ν^i, $i \in \{1, \ldots, k\}$ werden Chunks *des Clusters ν genannt.*

Theorem 2.2.1. *[EFC95] Ein C-zusammenhängender Clustergraph $C = (G, T)$ ist C-planar, genau dann wenn G planar ist, und eine planare Zeichnung $\mathcal{D}$ von G existiert, sodass für alle $\nu \in T$ gilt: alle Knoten und Kanten von $G \backslash G(\nu)$ liegen im externen Face der Zeichnung von $G(\nu)$.*

Theorem 2.2.1 ist beschränkt auf C-zusammenhängende Clustergraphen. Das nächste Theorem stellt einen Zusammenhang zu nicht zwangsläufig C-zusammenhängenden Clustergraphen dar.

Theorem 2.2.2. *[EFC95] Ein Clustergraph $C = (G, T)$ ist C-planar, genau dann wenn er ein Subgraph eines C-zusammenhängenden Clustergraphen ist.*

Komplexität des Cluster-Planaritätsproblems

Das Testen eines Clustergraphen auf C-Planarität ist keine einfache Erweiterung des Testens auf Planarität. Abbildung 2.5 zeigt einen Graphen (a), der offensichtlich im klassischen Sinne planar ist. Definiert man eine Clusterstruktur, wie durch den Inklusionsbaum in (b) beschrieben, sodass also jedes der „Dreiecke“ zu einem eigenen Cluster gehört, so gilt

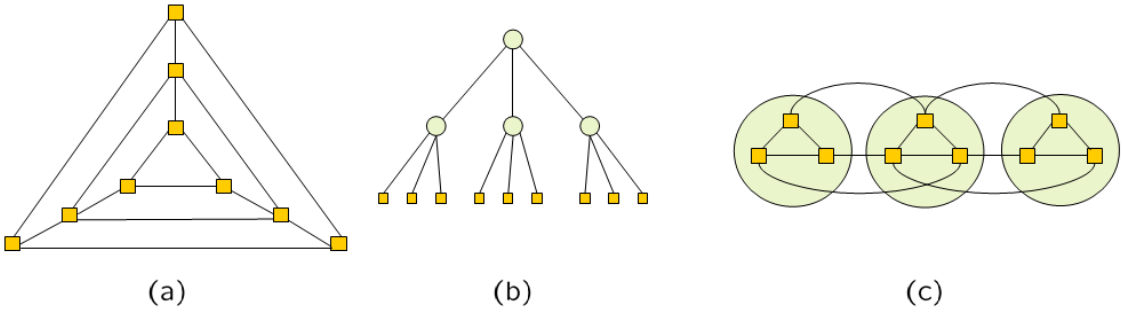


Abbildung 2.5: Ein planarer aber nicht C-planarer Graph

zwar, dass jeder Cluster-induzierte Subgraph planar ist, als auch der resultierende Graph, wenn man die drei Cluster-induzierten Subgraphen jeweils zu einem Knoten kollabiert. Der Clustergraph ist dennoch nicht C-planar, wie in (c) ersichtlich. Dieses Standard-Beispiel ist aus [EFC95] entnommen.

Die Einfachheit von Clustergraphen, bezogen auf ihre Beschreibung, schlägt sich nicht in der Entwicklung von Algorithmen zum Testen eines Clustergraphen auf C-Planarität nieder. Im Gegenteil, die über dem zugrunde liegenden Graphen G definierte Clusterstruktur T macht das Problem schwierig. Das Problem selbst ist gut studiert. Es ist bereits für eine Vielzahl bestimmter struktureller Klassen von Clustergraphen gezeigt worden, dass sie effizient auf C-Planarität getestet werden können. Trotz der diesbezüglichen Erfahrungen und Fortschritte ist es dennoch bislang noch nicht gelungen, einen effizienten Algorithmus für den Cluster-Planaritätstest für beliebige Clustergraphen, also insbesondere auch nicht zwangsläufig C-zusammenhängende Clustergraphen, zu entwickeln. Ebenso steht aber auch ein NP-Vollständigkeitsbeweis noch aus. Die Komplexität dieses Problems ist daher unbekannt und ein noch offenes Problem. Die verschiedenen Klassen von Clustergraphen, für die gezeigt werden konnte, dass sie effizient auf C-Planarität testbar sind, werden im folgenden Abschnitt vorgestellt.

2.3 Bisherige Resultate

2.3.1 C-zusammenhängende Clustergraphen

Der erste effiziente C-Planaritätstest für eine Unterklasse von Clustergraphen wurde von Feng, Eades und Cohen entwickelt und in [EFC95] vorgestellt. Sie zeigen, dass C-zusammenhängende Clustergraphen (siehe Definition 2.2.4) in quadratischer Zeit, bezogen auf die Anzahl der Knoten des zugrunde liegenden Graphen, auf C-Planarität getestet werden können. Sie geben außerdem eine Erweiterung ihres Algorithmus an, anhand der im positiven Fall auch eine C-planare Einbettung berechnet wird. Eine solche Einbettung kann dann von einem Layout-Algorithmus benutzt werden, um den Clustergraphen C-planar zu zeichnen.

Um eine planare Einbettung für G des C-zusammenhängenden Clustergraphen $C = (G, T)$ zu finden, die den Anforderungen aus Theorem 2.2.1 genügt, wird der Inklusionsbaum

bottom-up durchlaufen. Dadurch werden immer zuerst die inkludierten Cluster betrachtet, bevor deren Supercluster betrachtet wird. An jedem Cluster ν wird nach planaren Einbettungen für den durch ν induzierten Subgraphen $G(\nu)$ gesucht, die die Bedingungen aus 2.2.1 erfüllen. Das Ausschließen bestimmter Einbettungen für einen Subgraphen $G(\nu)$ schränkt natürlich automatisch auch die Menge der möglichen Einbettungen für die durch die Supercluster induzierten Subgraphen ein. Kann am Ende der Traversierung des Inklusionsbaums eine c-planare Einbettung für den root-Cluster ermittelt werden, so ist der Clustergraph C-planar. Der Algorithmus benutzt eine effiziente Datenstruktur, die *PQ-Bäume* [BL76]. Dabei handelt es sich in erster Linie um eine Datenstruktur zur Speicherung und Verwaltung von Permutationen, die hierbei aber benutzt wird, um die Menge der noch möglichen planaren Einbettungen effizient zu verwalten.

Dahlhaus entwarf 1998 einen Linearzeit-Algorithmus zum Testen von C-zusammenhängenden Clustergraphen auf C-Planarität [Dah98], basierend auf Graphgrammatiken.

2.3.2 Fast C-zusammenhängende Clustergraphen

Gutwenger et.al. erweiterten 2002 die Menge der effizient lösbaren Clustergraphen um die Klasse der *fast C-zusammenhängenden* Clustergraphen [GJL$^+$02]. Dabei handelt es sich um eine Obermenge der C-zusammenhängenden Clustergraphen in folgender Weise:

Definition 2.3.1. ***Fast C-zusammenhängender Clustergraph*** *Ein Clustergraph $C = (G, T)$ heißt* fast C-zusammenhängend *falls einer der folgenden Fälle vorliegt:*

- *alle nicht zusammenhängenden Cluster-induzierten Subgraphen $G(\nu)$ liegen in T auf demselben Pfad von der Wurzel aus. Oder andersherum ausgedrückt: es existiert ein Pfad $\pi = (\nu_r, \nu_1, \nu_2, \ldots, \nu_k)$ in T mit ν_r root-Cluster, sodass gilt:*

$$G(\nu) \text{ nicht zusammenhängend} \quad \Rightarrow \quad \nu \in \pi$$

- *für alle nicht zusammenhängenden Cluster-induzierten Subgraphen $G(\nu)$ gilt, dass sowohl der durch den Vaterknoten μ von ν induzierte Subgraph $G(\mu)$, als auch alle durch die Geschwisterknoten von μ induzierten Subgraphen zusammenhängend sind.*

Abbildung 2.6 zeigt die Inklusionsbäume zweier Clustergraphen. (a) erfüllt die erste Bedingung, da alle nicht-zusammenhängenden Cluster auf demselben Pfad von der Wurzel aus startend liegen. (b) erfüllt die zweite Bedingung, da der Vaterknoten und dessen Geschwisterknoten des einzigen nicht-zusammenhängenden Clusters zusammenhängend sind.

In ihrem Paper [GJL$^+$02] geben Gutwenger et. al. einen effizienten Algorithmus an, der einen fast vollständig zusammenhängenden Clustergraphen auf C-Planarität testet.

2.3.3 Kreise in Clusterkreisen

In [CBPP04] betrachten Cortese et. al. eine spezielle Klasse von Clustergraphen, deren zugrunde liegender Graph ein Kreis ist. Als erstes zeigen sie, dass sich Clustergraphen, die aus drei sich auf demselben Level befindenden Clustern bestehen und deren zugrunde liegender Graph ein Kreis ist, in Linearzeit auf C-Planarität testen lassen. Sie formulieren

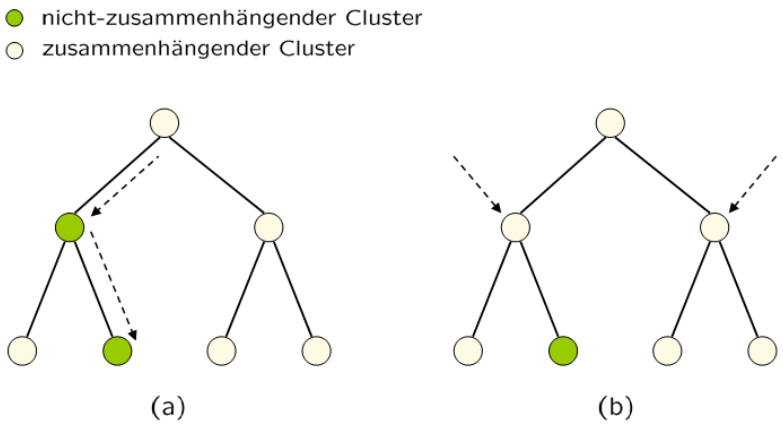


Abbildung 2.6: Beispiel für einen *fast C-zusammenhängenden* Clustergraphen. (a) alle nicht-zusammenhängenden Cluster liegen auf demselben Pfad von der Wurzel des Inklusionsbaumes aus. (b) Der Supercluster des nicht-zusammenhängenden Clusters und dessen Geschwister sind zusammenhängend

außerdem eine Verallgemeinerung dieser Klasse, bei der die Clusterstrutur auch tiefer sein kann. Dabei müssen die Cluster-induzierten Graphen auf jedem Level ebenfalls einen Kreis induzieren, wenn man diese jeweils zu einem Knoten kollabiert. Auch für diese wird ein effizienter C-Planaritätstest angegeben. Die Clustergraphen, die in diese Klasse fallen, können dabei je nach Wahl der Cluster sowohl C-zusammenhängend als auch hochgradig nicht-C-zusammenhängend sein.

2.3.4 Extroverte Clustergraphen

2005 führten Goodrich et. al. eine neue Klasse von Clustergraphen ein, die *extroverten* Clustergraphen [GLS05], und zeigen per Konstruktion durch Entwicklung eines Algorithmus, dass diese in $O(n^3)$ auf C-Planarität getestet werden können.

Gegeben sei ein Clustergraph $C = (G, T)$. Eine Kante $e = \{v, w\}$ heißt *extrovert*, falls $v \in V(\nu)$ und $w \notin V(\nu)$ gilt, bezüglich eines Clusters ν. Ein Chunk ν^i eines nicht-zusammenhängenden Clusters $\nu = (\nu^1, \ldots, \nu^k)$ heißt extrovert, falls der Supercluster μ von ν zusammenhängend ist, und mindestens eine extroverte Kante von ν^i aus dem Supercluster μ „heraus führt“. Ein nicht-zusammenhängender Cluster $\nu = (\nu^1, \ldots, \nu^k)$ heißt extrovert, falls jeder Chunk ν^i, $i \in \{1, \ldots, k\}$ extrovert ist. Der Clustergraph $C = (G, T)$ heißt extrovert, falls alle nicht-zusammenhängenden Cluster extrovert sind.

Die Definition eines extroverten Clustergraphen ist recht unanschaulich, weshalb dies hier noch kurz anhand eines graphischen Beispiels veranschaulicht wird. Abbildung 2.7 zeigt die Zeichnung eines extroverten Clustergraphen. Die Kanten $e_1, \ldots, e_4$ sind extroverte Kanten der Cluster ν_1 und ν_2, da der Supercluster μ zusammenhängend ist und die Kanten $e_1, \ldots, e_4$ aus dem Supercluster heraus führen.

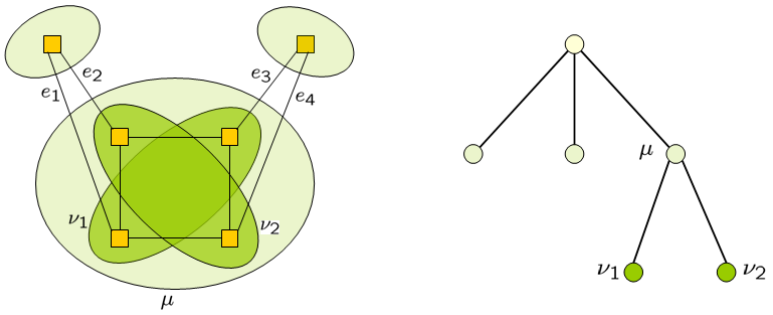


Abbildung 2.7: Beispiel für einen *extroverten* Clustergraphen. Die beiden nichtzusammenhängenden Cluster ν_1 und ν_2 sind extrovert. Deren Supercluster μ is zusammenhängend.

2.3.5 Vollständig zusammenhängende Clustergraphen

Eine weitere Klasse effizient testbarer Clustergraphen sind *vollständig zusammenhängende* Clustergraphen, die 2003 in [CW03] von Cornelsen und Wagner vorgestellt wurden. Dabei handelt es sich um Clustergraphen, bei denen zusätzlich zu den Cluster-induzierten Subgraphen auch jeweils die durch Entfernung eines Cluster-induzierten Subgraphen entstehenden Komplement- bzw. Restgraphen zusammenhängend sind. Formal sind vollständig zusammenhängende Clustergraphen wie folgt definiert:

Definition 2.3.2. *[CW03]* ***Vollständiger Zusammenhang.*** *Gegeben sie ein Clustergraph $C = (G, T)$. Sei ν ein Cluster aus T und $G(\nu)$ der durch ν induzierte Subgraph von C und $G(V \backslash V(\nu))$ der durch Entfernung von $G(\nu)$ induzierte Komplementgraph. C heißt* vollständig zusammenhängend*, wenn für jeden Cluster ν von C die durch diesen induzierten Subgraphen $G(\nu)$ und $G(V \backslash V(\nu))$ zusammenhängend sind.*

Zur Veranschaulichung betrachte man Abbildung 2.8. (a) zeigt den aus den drei Clustern ν_1, ν_2, ν_3 bestehenden Clustergraphen aus Abbildung 2.4. Dieser ist offensichtlich C-zusammenhängend, da die durch ν_1, ν_2 und ν_3 induzierten Subgraphen jeweils zusammenhängend sind. Die durch Entfernung eines Cluster-induzierten Subgraphen $G(\nu_i)$ induzierten Komplementgraphen sind ebenfalls zusammenhängend, wie aus der Abbildung ersichtlich. Demnach ist der in (a) dargestellte Clustergraph vollständig zusammenhängend. (b) zeigt den durch Entfernung der Kante e modifizierten Clustergraphen. Dieser ist nicht vollständig zusammenhängend. Zum Einen ist der Cluster ν_3 durch die entfernte Kante e nicht mehr zusammenhängend, zum Anderen ist auch der durch die Entfernung des Cluster-induzierten Graphen $G(\nu_1)$ induzierte Komplementgraph nicht zusammenhängend.

In ihrem Paper [CW03] stellen Cornelsen und Wagner eine interessante und erstaunliche Eigenschaft vollständig zusammenhängender Clustergraphen heraus.

Theorem 2.3.1. *[CW03] Ein vollständig zusammenhängender Clustergraph $C = (G, T)$ ist genau dann C-planar, wenn der zugrunde liegende Graph G planar ist.*

Theorem 2.3.2. *[CW03] Jeder C-planare Clustergraph ist ein Subgraph eines C-planaren vollständig zusammenhängenden Clustergraphen.*

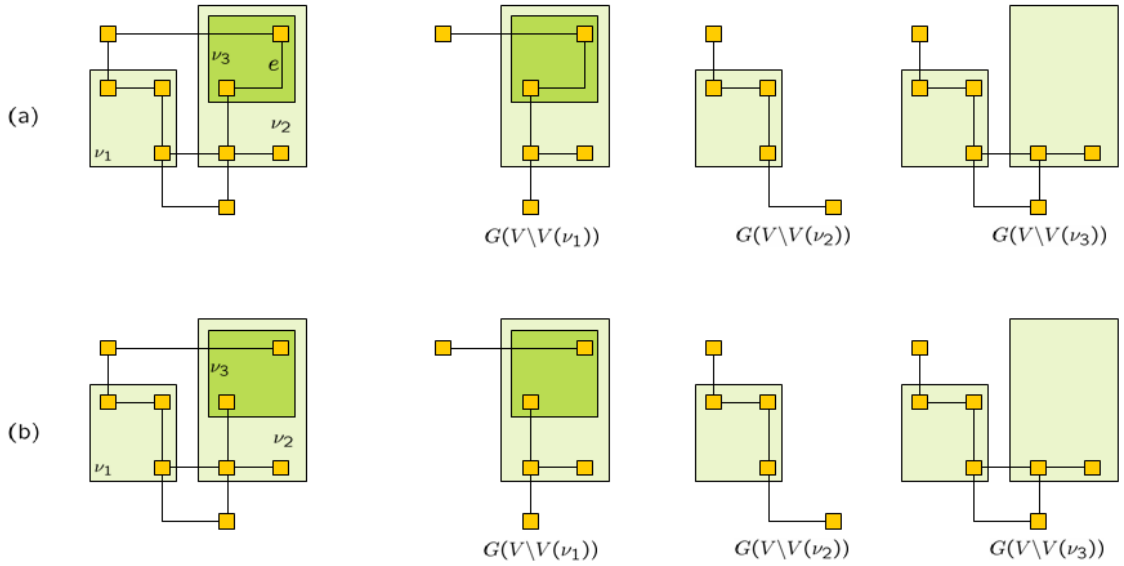


Abbildung 2.8: (a) Der Clustergraph $C = (G, T)$ aus Abbildung 2.4 und die jeweils durch die Entfernung eines Clusters induzierten Komplement-Graphen. C ist vollständig zusammenhängend. (b) $C = (G(V, E\backslash e), T)$ ist nicht vollständig zusammenhängend

Anders ausgedrückt bedeutet Theorem 2.3.2, dass jeder C-planare Clustergraph durch Hinzufügen bestimmter, zusätzlicher Kanten vollständig zusammenhängend gemacht werden kann, ohne die C-Planarität zu verletzen.

Die theoretischen Aussagen in [CW03] bezüglich dieser Graphklasse führen zu einer Idee eines Lösungs-Modells für das Maximum C-planare Subgraphen Problem, das informell bereits in Kapitel 1 vorgestellt wurde. Die Theoreme 2.3.1 und 2.3.2 bilden dabei die Grundlage sowohl in Bezug auf die Korrektheit des Modells, als auch die grundsätzliche Idee des algorithmischen Vorgehens zur Optimierung. Das MCPSP wird im nächsten Kapitel formalisiert.

Kapitel 3

Ein Modell für das MCPSP

In den vorangegangenen Kapiteln wurde das C-Planaritäts-Problem vorgestellt und motiviert. In diesem Kapitel betrachten wir ein allgemeineres Problem, das *Maximum C-planare Subgraphen Problem*. Dieses wird in Abschnitt 3.4 formalisiert und ein Ansatz zur optimalen Lösung entwickelt. Das Lösungs-Modell kann als ILP formuliert werden. Es werden daher zunächst in Abschnitt 3.1 einige Grundlagen zu kombinatorischer Optimierung und zu linearer und ganzzahliger linearer Optimierung gegeben. Zur Untersuchung des Modells für das MCPSP wird ein Branch-and-Cut Algorithmus implementiert. Dabei handelt es sich um ein Verfahren zur optimalen Lösung kombinatorischer Optimierungsprobleme. Die Verfahren Branch-and-Cut und Branch-and-Bound im Allgemeinen werden in den Abschnitten 3.2 und 3.3 erläutert. Die Herleitung des ILPs sowie dessen detaillierte Beschreibung werden in Abschnitt 3.5 gegeben.

3.1 Kombinatorische Optimierung

Viele in der Praxis auftretende algorithmische Probleme können als *kombinatorische Optimierungsprobleme* formuliert werden. Ein kombinatorisches Optimierungsproblem besteht aus einer endlichen Grundmenge E von Elementen, einer Teilmenge $\mathcal{I}$ der Potenzmenge von E, also $\mathcal{I} \subseteq 2^E$ (die Menge der zulässigen Lösungen), und einer Kostenfunktion $c : E \to \mathbb{R}$, die jedem Element aus E einen reellen Wert zuweist. Für jede zulässige Lösung $F \in \mathcal{I}$ sind die Kosten $c(F)$ dieser Lösung definiert als $\sum_{e \in F} c(e)$. Ziel ist es nun; eine zulässige Lösung $\hat{F} \in \mathcal{I}$ zu finden, deren Kosten $c(\hat{F})$ größt- bzw. kleinstmöglich sind.

Die meisten interessanten Probleme, die sich als kombinatorisches Optimierungsproblem formulieren lassen, besitzen einen Lösungsraum der exponentielle Größe in der Grundmenge E hat, zum Beispiel $2^{|E|}$ oder $|E|!$. Eine vollständige Enumeration und Auswertung aller möglichen Lösungen ist demnach meist inpraktikabel bei zunehmender Größe der Grundmenge und damit des Lösungsraums. Das Ziel der kombinatorischen Optimierung ist es, Algorithmen zu entwickeln, die wesentlich schneller als eine vollständige Enumeration sind. In diesem Zusammenhang besitzt die *(ganzzahlige) lineare Programmierung* eine große Bedeutung und bildet die Grundlage moderner Branch-and-Cut Verfahren zur exakten Lösung kombinatorischer Optimierungsprobleme.

3.1.1 Lineare Optimierungsprobleme

Bei einem linearen Optimierungsproblem geht es darum, eine lineare Zielfunktion über einem durch ein System linearer Ungleichungen , den Nebenbedingungen, gegebenen Gültigkeitsbereich zu maximieren bzw. zu minimieren. Lineare Optimierungsprobleme sind in vielen Anwendungen vertreten. Formal ist ein lineares Optimierungsproblem, oder auch *lineares Programm (LP)*, wie folgt definiert:

Definition 3.1.1. ***Lineares Programm.*** *Gegeben seien zwei reell-wertige Vektoren $b \in \mathbb{R}^m$ und $c \in \mathbb{R}^n$ mit $m, n \in \mathbb{N}$ und eine $m \times n$ Matrix $A^{m,n}$ mit Elementen $a_{i,j} \in \mathbb{R}$. Gesucht ist ein Vektor $\hat{x} \in \mathbb{R}^n$, der unter allen Vektoren $x \in \mathbb{R}^n$ den größten/kleinsten Wert $c^T x$ hat, und alle Nebenbedingungen $Ax \leq b$ erfüllt. Dabei bezeichnet man $c^T x$ als die zu maximierende/minimierende* Zielfunktion. *Die durch die Ungleichungen $Ax \leq b$ beschriebenen Nebenbedingungen werden auch* Restriktionen *oder* Constraints *genannt.*

Lineare Programme können in vielfältiger Form auftreten. Anstelle von Ungleichungen können zum Beispiel Gleichungen gegeben sein oder Variablen nur positive oder beliebige reelle Werte annehmen dürfen. Alle Varianten lassen sich jedoch in die durch George B. Dantzig vorgeschlagene und verbreitete Standardform überführen (*s.to* steht dabei für *subject to*):

$$\begin{aligned} & max\ c^T x \\ s.to \quad & Ax \leq b \\ & x \in \mathbb{R} \\ & x \geq 0 \end{aligned}$$

3.1.2 Ganzzahlige und binäre lineare Programme

Bei der Formulierung kombinatorischer Optimierungsprobleme als lineare Programme ist meistens gefordert, dass ein Teil der Variablen nur ganzzahlige Werte annehmen darf. In diesem Fall spricht man von einem *gemischt ganzzahligen linearen Programm (MILP)*. Sind alle Variablen auf ganzzahlige Werte beschränkt, nennt man das lineare Programm ein *ganzzahliges lineares Programm (ILP)*. Der spezielle, aber sehr häufig auftretende Fall, dass die Variablen nur die Werte 0 und 1 annehmen dürfen, wird als *binäres lineares Programm (0/1-LP)* bezeichnet. Bei dem LP unten handelt es sich um ein 0/1-LP in Standardform.

$$\begin{aligned} & max\ c^T x \\ s.to \quad & Ax \leq b \\ & x \in \{0;1\}^n \end{aligned}$$

3.1.3 LP und ILP

Ganzzahlige lineare Programmierung ist im Allgemeinen NP-schwierig. Das heißt, unter der Annahme $NP \neq P$ gibt es keinen Polynomialzeit-Algorithmus, der ILPs optimal löst. Lineare Programmierung hingegen ist in P. LPs polynomieller Größe, das heißt, LPs mit polynomiell vielen Constraints in der Anzahl der Variablen, sind effizient lösbar. Den Beweis dazu lieferte Khachian 1979 konstruktiv durch die Entwicklung der *Ellipsoid-Methode*, bei der es sich um einen Polynomialzeit-Algorithmus zur Lösung LPs polynomieller Größe handelt. Diese Methode stellt zwar ein wichtiges und bahnbrechendes theoretisches Ergebnis dar, ist aber aufgrund der hohen Konstanten meist nicht praktikabel. In der Praxis wird weitgehend die 1947 von George Dantzig erfundene *Simplex-Methode* verwendet. Bei dieser handelt es sich um einen Algorithmus, für den zwar worst-case LPs existieren (zum Beispiel die *Klee-Minty-Cubes* [GHZ98]), auf denen dieser zur Lösung exponentiell viele Iterationen durchführt, der aber auf den meisten in der Praxis auftretenden LPs sehr viel schneller ist.

LP-Relaxierung

Bei den Ganzzahligkeitsbedingungen der Variablen in einer ILP-Formulierung handelt es sich offensichtlich nicht um lineare Constraints. Entfernt man in dem ILP die Ganzzahligkeitsbedingungen, so erhält man ein lineares Programm. Man spricht dabei von der *LP-Relaxierung* des ILPs. Durch die Entfernung der Ganzzahligkeitsbedingungen vergrößert sich im Allgemeinen die Menge der zulässigen Lösungen. Betrachtet man zum Beispiel ein 0/1-LP, bei dem die Variablen also nur die Werte 0 und 1 annehmen dürfen, und entfernt die Ganzzahligkeitsbedingungen, so können die Variablen nun beliebige reelle Werte im Intervall $[0, 1]$ annehmen. Insbesondere gilt deshalb auch, dass sich die Menge der optimalen Lösungen im Allgemeinen ebenfalls ändert. Optimale Lösungen der LP-Relaxierung sind nun in der Regel *fraktional* und nicht zwangsläufig ganzzahlig. Um diesen Sachverhalt zu veranschaulichen, betrachten wir ein Beispiel, das ein ILP und dessen LP-Relaxierung näher beleuchtet. Gegeben sei das folgende, einfache, zwei-dimensionale ILP:

$$\begin{aligned} & max\ y \\ s.to.\quad & -x + y \leq 1 \\ & 3x + 2y \leq 12 \\ & 2x + 3y \leq 12 \\ & x, y \in \mathbb{Z}_0^+ \end{aligned}$$

Abbildung 3.1 zeigt die geometrische Interpretation dieses ILPs und dessen Relaxierung. Die zulässigen (ganzzahligen) Lösungen des ILP sind als helle Punkte gekennzeichnet. Die konvexe Hülle der zulässigen Lösungen, also der kleinste konvexe Raum, der alle zulässigen Lösungen enthält, entspricht der Schnittmenge der gepunkteten Linien und der x- und y-Achsen. Die beiden optimalen ILP-Lösungen $(x = 1, y = 2)$ und $(x = 2, y = 2)$ mit jeweils Zielfunktionswert 2 sind Eckpunkte der konvexen Hülle. Optimale Lösungen eines ILPs oder LPs sind *immer* Eckpunkte der konvexen Hülle des Lösungsraums.

Betrachten wir nun die Relaxierung des ILPs, das heißt wir entfernen die Ganzzahligkeitsbedingungen $x, y \in \mathbb{Z}^+$, bzw. ersetzen sie durch die Bedingungen $x, y \in \mathbb{R}^+$. Dadurch vergrößert sich der Lösungsraum um die schraffierte Fläche. Alle reell-wertigen Punkte, die im Lösungsraum liegen, sind nun zulässige Lösungen der LP-Relaxierung, und nicht nur die Integer-Punkte. Insbesondere hat sich nun durch das Entfernen der Ganzzahligkeitsbedingungen auch die Menge der optimalen Lösungen geändert. Das Beispiel zeigt, dass die fraktionale Lösung LP_{opt} einen höheren Zielfunktionswert erzielt, als die beiden optimalen ILP-Lösungen.

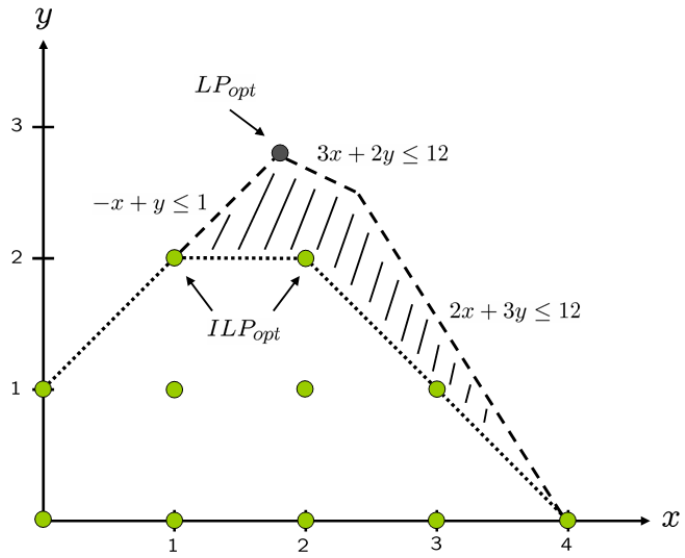


Abbildung 3.1: ILP und LP-Relaxierung

3.2 Branch-and-Bound

Unter einem Branch-and-Bound Verfahren versteht man ein sehr allgemeines, grundlegendes Konzept zur exakten Lösung (NP-schwieriger) kombinatorischer Optimierungsprobleme, durch das versucht wird eine vollständige Enumertation aller zulässigen Lösungen zu vermeiden. Dies geschieht anhand des sukzessiven Aufbaus eines Suchbaums, des Branch-and-Bound Baums, dessen Knoten jeweils eine Teilmenge des Suchraums (bzw. Lösungsraums) repräsentieren, derart, dass der durch einen Nachfolgeknoten definierte Lösungsraum vollständig im durch seinen Vorgängerknoten definierten Lösungsraums enthalten ist. Die Blätter des Branch-and-Bound Baums entsprechen genau einem „Punkt“ des Lösungsraums. Zulässige Lösungen des kombinatorischen Optimierungsproblems entsprechen demnach Blättern des Branch-and-Bound Baums. Je tiefer ein Knoten im Baum liegt, desto kleiner und spezieller ist der durch diesen repräsentierte Suchraum. Die Knoten des Branch-and-Bound Baums nennt man auch *Subprobleme*. Zu Beginn besteht dieser Baum nur aus dem Wurzel-Knoten, der den vollständigen Lösungsraum darstellt. Zu jedem Zeitpunkt wird nun versucht für das aktuell betrachtete Subproblem, eine möglichst gute obere und untere Schranke zu berechnen. Im Fall eines Maximierungsproblems entspricht jede zulässige Lösung für das Problem einer unteren Schranke. Die global besten Schranken werden fortlaufend mit den neu berechneten verglichen und gegebenenfalls verbessert.

Ist die aktuell berechnete obere Schranke eines Subproblems niedriger als die global beste untere Schranke, so kann der Teilbaum des Branch-and-Bound Baums dessen Wurzel das aktuell betrachtete Subproblem ist, *abgeschnitten* werden, da in diesem enthaltene Lösungen nicht besser sein können, als die bereits beste gefundene. Durch dieses *Bounding-Prinzip* erhofft man sich schon frühzeitig unprofitable Teilbäume abschneiden zu können, um somit den Suchraum stetig zu verkleinern. Abbildung 3.2 zeigt einen typischen Verlauf der Schranken in einem Branch-and-Bound Algorithmus. Für den praktischen Einsatz sind reine Branch-and-Bound Verfahren jedoch trotzdem meist nicht praktikabel.

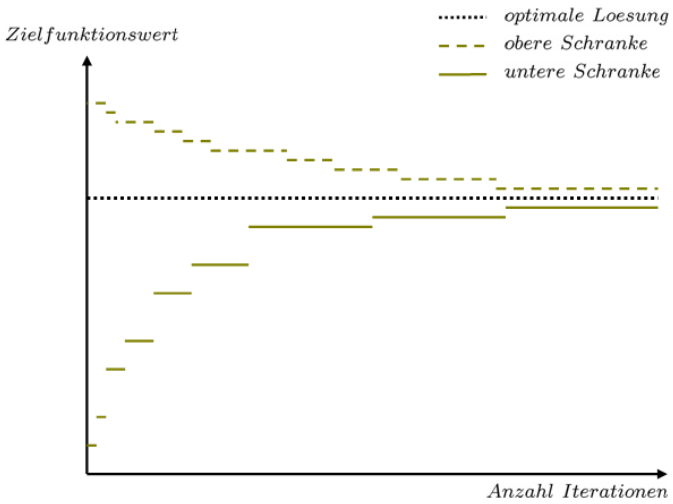


Abbildung 3.2: Typische Entwicklung der Schranken in einem Branch-and-Bound Verfahren, bezogen auf ein Maximierungsproblem

Branch-and-Bound mittels linearer Programmierung

Die Suche nach einer optimalen Lösung eines ILPs kann nun auf folgende Weise mittels Branch-and-Bound erfolgen: jedes Subproblem wird durch eine LP-Relaxierung des ILPs repräsentiert, bei der jeweils eine bestimmte Teilmenge der Variablen auf einen zulässigen ganzzahligen Wert fixiert ist. Dadurch spezifizieren die LP-Relaxierungen jeweils eine bestimmte Teilmenge des Lösungsraums. An einem Subproblem wird nun die korrespondierende LP-Relaxierung optimal gelöst. Ist die optimale Lösung fraktional (was im Allgemeinen der Fall ist), so muss *gebrancht* werden. Dabei werden neue Subprobleme erzeugt, indem eine bestimmte Variable mit fraktionalem LP-Wert auf jeweils einen ihrer zulässigen ganzzahligen Werte fixiert wird. Im Fall eines 0/1-LPs werden beim Branchen also zwei neue Subprobleme erzeugt, bei der die ausgewählte, fraktionale, binäre Variable in dem einen LP auf 0, in dem anderen auf 1 gesetzt wird. Die Werte für diese Variable bleiben in allen weiteren Subproblemen, die aus diesen abgeleitet werden, fix. Dadurch entsteht die Branch-and-Bound typische, sukzessive Spezifizierung des Suchraums und damit der Enumeration der Lösungen. Bezüglich eines ILPs stellt jede optimale Lösung der LP-Relaxierung eine obere Schranke für die Optimallösung des ILPs dar (bezogen auf ein Maximierungsproblem). Hingegen ist natürlich jede zulässige Lösung für das ILP eine untere Schranke für die Optimallösung des ILPs. Dadurch ergeben sich also die oberen und

unteren Schranken, die wie oben beschrieben zum Bounding, also zum Abschneiden unprofitabler Subprobleme, benutzt werden. Im worst-case müssen allerdings bei einem 0/1-LP mit n Variablen 2^n viele LPs erzeugt und gelöst werden, um die optimale ILP-Lösung zu erhalten.

In Anlehnung an die LP-Dualität wird im Folgenden die Schranke, die aus zulässigen Lösungen für das ILP hervorgeht, *primale* Schranke genannt. Die Schranke, die durch optimale Lösungen der LP-Relaxierung hervorgeht, wird *duale* Schranke genannt. Diese Begriffsbildungen ersparen die Unterscheidung zwischen einem Maximierungs- und einem Minimierungsproblem. Das zur Implementierung des Branch-and-Cut Algorithmus verwendete Framework ABACUS verwendet ebenfalls diese Terminologie.

3.3 Branch-and-Cut

Haben die einzelnen LPs polynomielle Größe, so können diese auch in polynomieller Zeit gelöst werden. In der Regel ist es allerdings selbst bei LPs polynomieller Größe oft nicht ratsam, das LP von Anfang an vollständig (also mit allen theoretisch notwendigen Constraints) zu erzeugen. Es hat sich gezeigt, dass häufig schon eine sehr kleine Teilmenge der Constraints ausreicht, um das LP optimal lösen zu können. Das heißt, die restlichen Constraints sind implizit durch die Lösung erfüllt, ohne dass sie explizit zum LP hinzugefügt wurden. Die Idee ist daher, zunächst nur mit einer (evtl. sogar leeren) Teilmenge der Constraints zu starten, und zusätzliche Constraints nur „bei Bedarf“ hinzuzufügen, zu *separieren*. Es werden also jeweils nur solche Constraints zum LP hinzugefügt, die durch die aktuelle LP-Lösung verletzt sind. Auf diesem Prinzip basierend lässt sich nun ein iteratives Verfahren zur sukzessiven Verschärfung und Lösung eines LP definieren, das sogenannte *Schnittebenen-Verfahren*.

3.3.1 Schnittebenen-Verfahren

Gegeben sei ein ILP.

$$\begin{aligned} & max\ c^T x \\ s.to.\quad & Ax \ \leq\ b \\ & \forall\ x \in \mathbb{Z} \end{aligned}$$

Definition 3.3.1. ***Schnittebenen-Verfahren***

1. *Initialisiere die LP-Relaxierung des ILPs mit einer Teilmenge der Restriktionen.*
2. *Löse die LP-Relaxierung optimal (zum Beispiel mittels Simplex-Methode). Sei x^* die optimale Lösung zu diesem LP.*
3. ***Separationsproblem***
   (a) *Entscheide, ob es weggelassene Constraints $a^T x \leq a_0$ gibt, die durch die Lösung x^* verletzt sind, d.h. ob $a^T x^* > a_0$ gilt.*

(b) falls ja, füge den Constraint $a^T x \leq a_0$ zum LP hinzu und gehe zu Schritt (2) des Schnittebenen-Verfahrens.

(c) falls nein, so handelt es sich bei x^* um eine optimale Lösung des LPs, da alle Constraints erfüllt sind (die weggelassenen sind also implizit erfüllt).

Kombiniert man nun Branch-and-Bound mit dem Schnittebenen-Verfahren zur Lösung der einzelnen Subprobleme, so spricht man von einem *Branch-and-Cut Verfahren.*

Beim *Separationsproblem* handelt es sich um ein vollkommen applikationsspezifisches Problem, bei dem entschieden bzw. berechnet werden muss, ob es durch die aktuelle LP-Lösung verletzte Constraints gibt, oder nicht. Ein wesentlicher Faktor für gute Branch-and-Cut Algorithmen sind effiziente Separations-Algorithmen. Es sei noch erwähnt, dass bei der gerade vorgestellten Definition des Schnittebenen-Verfahrens grundsätzlich davon ausgegangen wird, dass ein Constraint, der durch die aktuelle LP-Lösung verletzt ist, auch durch den Separations-Algorithmus gefunden wird. Ansonsten entspräche die am Ende gefundene LP-Lösung nicht zwangsläufig einer optimalen Lösung des betrachteten LPs, da es möglicherweise noch verletzte Constraints gibt. In einem Branch-and-Cut Verfahren ist dies allerdings nicht zwingend erforderlich. Es ist durchaus möglich, verletzte Constraints nur heuristisch zu separieren. Dadurch besteht natürlich die Möglichkeit, dass die LP-Relaxierung nicht optimal gelöst werden kann, in dem Sinne, dass es noch verletzte Constraints gibt, die durch den heuristischen Separations-Algorithmus nicht separiert werden konnten. In diesem Fall muss „vorzeitig“ gebrancht werden. Die Korrektheit des Branch-and-Cut Verfahrens geht dadurch jedoch offensichtlich nicht verloren.

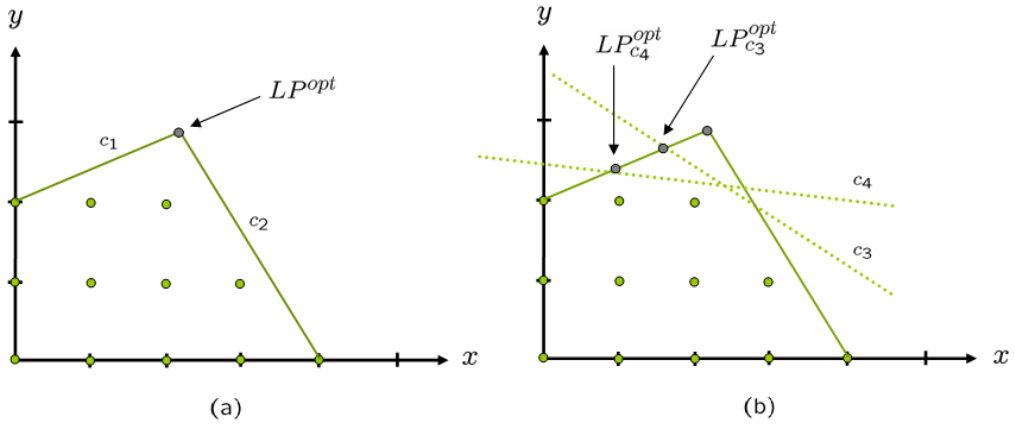


Abbildung 3.3: Beispiel zum Schnittebenen-Verfahren

Betrachten wir als Beispiel Abbildung 3.3. Das zugrunde liegende 2-dimensionale LP habe wie zuvor die simple Zielfunktion $max\ y$ und enthalte die vier Constraints c_1, c_2, c_3 und c_4, sowie Nicht-Negativitätsbedingungen für die beiden Variablen x und y. Das LP soll nun mittels Schnittebenen-Verfahren gelöst werden. Wir starten dazu mit einer Teilmenge der Constraints (c_1 und c_2) und lösen das LP anhand dieser optimal (siehe (a)), wodurch wir die Lösung LP^{opt} erhalten. Als nächstes wird untersucht, ob es weggelassene Constraints gibt, die durch LP^{opt} verletzt sind. c_3 und c_4 sind beide verletzt. Wenn der Separations-Algorithmus nun zuerst c_3 separiert, zum LP hinzufügt und dies erneut optimal löst, so wäre die Lösung $LP^{opt}_{c_3}$ noch durch c_4 verletzt. Separiert der Algoritmus allerdings

zuerst c_4, so wäre in der nächsten Iteration c_3 implizit durch die gefundene Lösung $LP_{c_4}^{opt}$ erfüllt und die Optimierung damit beendet, obwohl nicht explizit alle Constraints zum LP hinzugefügt wurden.

Resümee bezüglich Branch&Cut-Verfahren

Mit Hilfe von Schnittebenen-Verfahren können zum Teil auch exponentiell große LPs in polynomieller Zeit gelöst werden. Insbesondere zeigten Grötschel, Lovacz und Schrijver 1981, dass ein lineares Programm *genau dann* in polynomieller Zeit optimiert werden kann, wenn das zugrunde liegende Separationsproblem in polynomieller Zeit gelöst werden kann. Bei der Lösung kombinatorischer Optimierungsprobleme mittels linearer Programmierung, bezieht sich dies natürlich auf die optimale Lösung eines einzelnen Subproblems (LP-Relaxierung) und nicht auf das zugrunde liegende ILP.

Der Erfolg des Schnittebenen-Verfahrens und Branch-and-Cut spricht für sich. Besonders durch die großen Erfolge in Bezug auf das *Traveling Salesman Problem (TSP)*, für das bereits sehr große Instanzen beweisbar optimal gelöst werden konnten (siehe z.B. [ABCC01], *http://dimacs.rutgers.edu/Challenges/*), was mit einem reinen Branch-and-Bound Algorithmus nicht möglich ist, hat diese Technik in den letzten Jahren zunehmend an Bedeutung und Anerkennung gewonnen. Dies rechtfertigt und untermauert den Einsatz von Branch-and-Cut Techniken als sinnvollen und erfolgversprechenden Lösungs-Ansatz zur optimalen Lösung NP-schwieriger kombinatorischer Optimierungsprobleme.

3.4 Maximale C-planare Subgraphen

Nachdem nun alle zum Verständnis nötigen Kenntnisse und Werkzeuge vorgestellt wurden, betrachten wir das *Maximum C-planare Subgraphen Problem*. Das Problem besteht darin, für einen gegebenen Clustergraphen einen C-planaren Subgraphen zu finden, der unter allen C-planaren Subgraphen maximal viele Kanten enthält.

Es existieren, genau wie im Fall allgemeiner Graphen, Planarisierungsalgorithmen für Clustergraphen [BDM01]. Im Falle klassischer Graphen berechnen Planarisierungs-Verfahren zunächst einen planaren Subgraphen. Von diesem ausgehend werden die entfernten Kanten dann sukzessiv wieder in den Graphen eingefügt (siehe Abschnitt 2.1.2). Das Problem für einen gegebenen Graphen $G = (V, E)$ einen *maximalen* planaren Subgraphen zu berechnen, ist das *Maximum planarere Subgraphen Problem*.

Definition 3.4.1. ***Maximum planarer Subgraph (MPSP).*** *Gegeben sei ein Graph $G = (V, E)$. Gesucht ist ein planarer Subgraph $G' = (V, E')$ mit $E' \subseteq E$ maximaler Kardinalität. Das heißt:*

$$|E'| = max\{|E''| \mid G'' = (V, E'') \text{ planarer Subgraph von } G\}$$

Dann ist $G' = (V, E')$ ein maximum planarer Subgraph *von G.*

MPSP ist NP-schwierig [LG77]. In Bezug auf Clustergraphen lässt sich dieses Problem nun ganz ähnlich definieren. Dabei handelt es sich um das *Maximum C-planare Subgraphen Problem*.

Definition 3.4.2. ***Maximum C-planarer Subgraph (MCPSP).*** *Gegeben sei ein Clustergraph $C = (G = (V, E), T)$. Gesucht ist ein C-planarer Subgraph $C' = (G' = (V, E'), T)$ mit $E' \subseteq E$ maximaler Kardinalität. Das heißt:*

$$|E'| = max\{|E''| \mid C'' = (G'' = (V, E''), T) \text{ C-planarer Subgraph von } C\}$$

Dann ist $C' = (G' = (V, E'), T)$ ein maximum C-planarer Subgraph *von C.*

Das MCPSP ist ebenfalls NP-schwierig. Es lässt sich zeigen, dass MPSP polynomiell auf MCPSP reduzierbar ist.

Theorem 3.4.1. *MCPSP ist NP-vollständig.*

Beweis. Gegeben sei ein Graph $G = (V, E)$. Wir konstruieren in polynomieller Zeit aus G einen Clustergraphen $C = (G, T)$, sodass gilt:

$$C \text{ ist C-planar } \Leftrightarrow G \text{ ist planar}$$

Erzeuge einen Inklusionsbaum T, der nur aus dem root-Cluster μ besteht (also der Cluster, der die gesamte Knotenmenge V enthält) und keinen weiteren Cluster. Dann ist der durch den Cluster μ induzierte Subgraph $G(\mu) = G(V)$ der Graph G selbst.
„⇒“ Ist $C = (G, T)$ C-planar, so ist G per Definition planar.
„⇐“ Ist G planar, so ist $G(\mu)$ trivialerweise auch planar, wegen $G = G(\mu)$. Da μ der einzige Cluster (der root-Cluster) von C ist, ist somit auch C C-planar. □

Ein maximaler C-planarer Subgraph kann nun zum Beispiel als Ausgangspunkt für ein Planarisierungs-Verfahren für Clustergraphen dienen. Insbesondere liefert eine optimale Lösung für MCPSP nun aber auch automatisch die Antwort auf die Frage, ob der gegebene Clustergraph C-planar ist.

Beweis. Enthält der berechnete maximum C-planare Subgraph $C' = (G = (V, E'), T)$ für einen gegebenen Clustergraphen $C = (G = (V, E), T)$ alle Kanten aus E, das heißt, gilt $E = E'$, so handelt es sich bei C' um den ursprünglichen Clustergraphen C, und dieser ist somit C-planar. Enthält C' hingegen nicht alle Kanten, also $E' \subset E$, so ist C auch nicht C-planar. Andernfalls könnten sonst die Kanten $E \backslash E'$ zu C' hinzugefügt werden, ohne die C-Planarität zu verletzen. Dies stünde aber im Widerspruch zur Maximalität von C'. □

3.4.1 Lösungs-Modell für das MCPSP

Die Idee, das MCPSP für einen gegebenen Clustergraphen $C = (G, T)$ optimal zu lösen, ist nun wie folgt:

1. Gegeben sei ein beliebiger Clustergraph $C = (G, T)$
2. Füge eine *kleinstmögliche* Anzahl neuer Kanten zum Graphen hinzu, derart, dass der resultierende Graph vollständig zusammenhängend ist.
3. Entferne gleichzeitig eine *minimale* Anzahl an (Original-)Kanten aus dem Graphen, um ihn planar zu machen.

4. Erhalte dadurch einen maximum C-planaren Subgraphen C' von C. Enthält dieser alle Kanten aus E, so ist der Clustergraph C C-planar.

Die Korrektheit dieses Vorgehens folgt aus den theoretischen Erkenntnissen zu vollständig zusammenhängenden Clustergraphen von Cornelsen und Wagner [CW03], wie sie in Abschnitt 2.3.5 dargestellt wurden. Der auf diese Weise berechnete Clustergraph C' ist vollständig zusammenhängend und planar. Somit ist C' auch C-planar. C ist nun genau dann C-planar, wenn $E' = E$ gilt, der berechnete maximum C-planare Subgraph C' von C also alle Kanten aus E enthält. Diese Schlussfolgerung ist jedoch nur dann gültig, wenn C' aus einer Maximierung der Originalkanten hervorgeht! Dies hat zweierlei Gründe:

1. Fall 1: C ist C-planar. Es ist nicht bekannt *welche* Kanten zu C hinzugefügt werden müssen, um diesen vollständig zusammenhängend zu machen. Grundsätzlich können also durch eine „falsche“ Wahl der zusätzlichen Kanten neue, vorher noch nicht vorhanden gewesene, nicht-planare Strukturen entstehen. Es gilt allerdings, dass es immer eine Möglichkeit gibt, Kanten so hinzuzufügen, dass die Eigenschaft der Planarität nicht verletzt wird. Eine Maximierung der Originalkanten erzwingt also in diesem Fall, dass keine Kanten gelöscht werden und der resultierende C-planare Subgraph C' alle Originalkanten enthält.

2. Fall 2: C ist nicht C-planar. In diesem Fall müssen also Originalkanten gelöscht werden, um einen C-planaren Subgraphen zu erhalten. Die Maximierung dieser erzwingt allerdings, dass nur eine minimal nötige Anzahl von Originalkanten gelöscht wird, um einen C-planaren Subgraphen zu erhalten, sodass der durch die Originalkanten induzierte Clustergraph C' auch tatsächlich ein maximum C-planarer Subgraph von C ist.

Das oben beschriebene Lösungs-Modell für das MCPSP kann nun als 0/1-LP formuliert werden.

3.5 ILP-Formulierung für das MCPSP

Das MCPSP ist offensichtlich ein kombinatorisches Optimierungsproblem. Die Menge der Kanten E des Graphen entspricht der endlichen Grundmenge. Die Menge der zulässigen Lösungen entspricht derjenigen Teilmenge $\mathcal{I}$ der Potenzmenge 2^E der Kanten, die einen C-planaren Subgraphen induzieren. Die Kosten $c(e)$ eines Elements e der Grundmenge (also die Kosten einer Kante) entsprechen dem Wert 1. Die Kosten einer Lösungsmenge $I \in \mathcal{I}$ sind definiert als $\sum_{e \in I} c(e)$ und entsprechen demnach der Anzahl der in $\mathcal{I}$ enthaltenen Kanten. Ziel ist es, eine Kantenteilmenge maximaler Kardinalität zu finden, also eine Menge $\hat{I} \in \mathcal{I}$ mit maximalem Wert $c(\hat{I})$.

Die korrespondierende Formulierung als 0/1-LP ist wie folgt: für jede Kante $e \in E$, *Originalkante*, existiert eine 0/1-Variable x_e, deren Wert wie folgt interpretiert wird: steht die 0/1-Variable einer Kante $e \in E$ auf 1, so gehört die entsprechende Kante zu dem durch die ILP-Lösung induzierten Subgraphen. Ist der Wert der Variablen 0, so gehört die Kante nicht zum Subgraphen. Es ist nun das Ziel, Constraints zum 0/1-LP hinzuzufügen, die sicherstellen, dass der induzierte Graph planar und vollständig zusammenhängend ist. Um

vollständigen Zusammenhang eines nicht vollständig zusammenhängenden Graphen zu erhalten, müssen also neue Kanten, *Zusammenhangskanten*, eingefügt werden. Dementsprechend reicht es nicht aus, nur Variablen für die Kanten des Graphen zu erzeugen, sondern auch für jede potentielle neue Kante. Das heißt, es existiert zusätzlich zu jeder nicht vorhandenen Kante $e \notin E$ eine korrespondierende 0/1-Variable y_e, mit derselben Interpretation.

Als nächstes wird beschrieben, wie genau die Constraints des ILPs aussehen müssen, um diese beiden Eigenschaften zu erhalten.

3.5.1 Planaritäts-Constraints

Die Planarität des durch eine ILP-Lösung induzierten Graphen wird durch die sogenannten Kuratowski-Constraints sichergestellt. Diese können unmittelbar aus dem Satz von Kuratowski abgeleitet werden. Zur Wiederholung: ein Graph ist genau dann planar, wenn er keine K_5 und $K_{3,3}$ Subdivisions enthält. Die Planaritäts-Constraints sollen nun genau dies sicherstellen. Nehmen wir an, der gegebene Graph enthält eine K_5 Subdivision. Diese sei bestimmt durch die Kantenmenge $E_{K_5} = \{e_1, \ldots, e_k\}$. Es reicht aus, *eine* Kante aus der Subdivision zu entfernen, um diese planar zu machen. Es muss also in diesem Fall ein Constraint erzeugt werden, der die Entfernung einer Kante erzwingt. Dazu wird die Summe über denjenigen k Variablen gebildet, die zu den zur Subdivision gehörenden k Kanten korrespondieren. Der Wert dieser Summe (also die Summe der LP-Werte dieser Variablen) darf höchstens $k-1$ betragen, muss also um mindestens den Wert 1 kleiner sein als die Anzahl der Kanten in der Subdivision. Eine (ganzzahlige) LP-Lösung, die diesen Constraint erfüllt, korrespondiert dann zu einem Graphen, aus dem mindestens eine Kante der Subdivision entfernt wurde. Formal haben die Kuratowski-Constraints also folgendes Aussehen:

Sei S_K eine Kuratowski-Subdivision der Kardinalität k. Seien weiterhin $\{e_1, \ldots, e_k\}$ die entsprechenden Kanten der Subdivision, und $\{x_{e_1}, \ldots, x_{e_k}\}$ die korrespondierenden binären Variablen des ILPs (die x_{e_i} beziehen sich hier natürlich sowohl auf zu Originalkanten als auch auf zu Zusammenhangskanten korrespondierende Variablen). Dann ist folgender Kuratowski-Constraint verletzt und erzwingt im ganzzahligen Fall, dass mindestens eine Kante aus S_K entfernt wird:

$$\sum_{1 \leq i \leq k} x_{e_i} \leq k - 1$$

3.5.2 Zusammenhangs-Constraints

Als nächstes betrachten wir die Constraints, die den vollständigen Zusammenhang des Graphen sicherstellen. Diese lassen sich anhand von *Minimum Cuts* auf dem gegebenen Graphen ableiten.

Ein *Schnitt (Cut)* auf einem Graphen $G = (V, E)$ ist eine Partition der Knoten von G in zwei disjunkte Mengen A und $V \backslash A$. Eine solche Knotenpartition definiert außerdem auch eine eindeutige Kantenmenge $E' \subseteq E$, nämlich genau diejenigen Kanten $\{v, w\}$ für die gilt: $v \in A$ und $w \in V \backslash A$. Der Wert, bzw. die Kardinalität eines Cuts entspricht dabei der

Kardinalität eben dieser Kantenmenge. Als *Mincut* bezeichnet man einen Cut minimaler Kardinalität.

Gegeben sei ein Graph $G = (V, E)$. Nehmen wir an, dass G zusammenhängend ist. Jeder Mincut auf G hat dann eine Kardinalität von mindestens 1, da aufgrund des Zusammenhangs jeder Cut mindestens eine Kante enthält. Ist G hingegen nicht zusammenhängend, so hat jeder Mincut den Wert, bzw. Kardinalität 0. Das bedeutet, dass ein Graph genau dann zusammenhängend ist, wenn der Wert eines Mincuts auf diesem Graphen mindestens 1 beträgt. Abbildung 3.4 zeigt zur Veranschaulichung ein Beispiel. (a) zeigt einen zusammenhängenden Graphen. Einer der möglichen Mincuts ist durch die gestrichelte Linie gekennzeichnet. Die entsprechende Knotenpartition definiert die beiden Mengen $A = \{u_1, \ldots, u_5\}$ und $V \backslash A = \{v_1, v_2\}$. Der Mincut enthält eine Kante, nämlich $E_M = \{\{u_3, v_1\}\}$, und hat somit den Wert 1. (b) zeigt den Mincut eines nicht zusammenhängenden Graphen. Die beiden disjunkten Mengen der gekennzeichneten Knotenpartition sind $A = \{u_1, \ldots, u_4\}$ und $V \backslash A = \{v_1, \ldots, v_3\}$. Die korrespondierende Kantenmenge E_M ist leer und der Mincut hat somit den Wert 0.

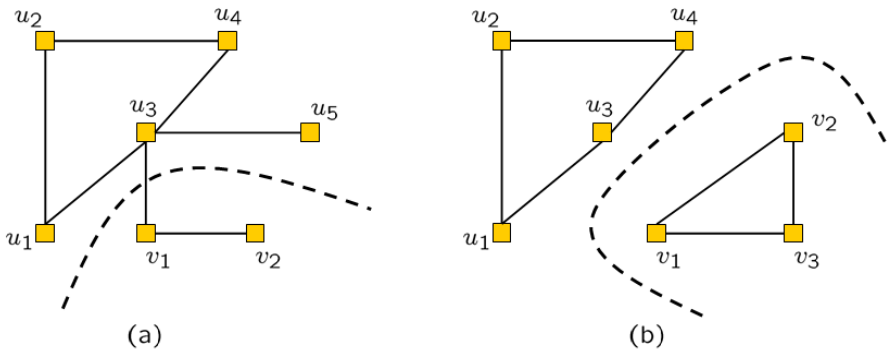


Abbildung 3.4: Mincuts auf einem Graphen

Gegeben sei nun ein Clustergraph $C = (G, T)$. Nehmen wir an, der durch einen Cluster ν induzierte Subgraph $G(\nu)$ ist nicht zusammenhängend. Der Einfachheit halber bestehe $G(\nu)$ aus zwei Zusammenhangskomponenten. Betrachten wir einen Mincut M auf $G(\nu)$ mit Knotenpartition A und $V(\nu)\backslash A$, $|A| = k$ und $|V(\nu)\backslash A| = l$. Der Wert von M beträgt 0. Um Zusammenhang zu erzwingen, muss mindestens eine neue Zusammenhangskante zwischen einem Knoten aus A und einem Knoten aus $V(\nu)\backslash A$ eingefügt werden. Mindestens eine der binären Variablen des ILPs, die zu den potentiellen neuen Zusammenhangskanten zwischen den Knoten aus A und $V(\nu)\backslash A$ korrespondieren, muss also den Wert 1 erhalten. Seien $u_i, 1 \leq i \leq k$ die Knoten aus A und $v_j, 1 \leq j \leq l$ die Knoten aus $V \backslash A$. Folgender *Cut-Constraint* erzwingt dann in einer ganzzahligen Lösung den Zusammenhang von $G(\nu)$:

$$\sum_{e=(u_i,v_j)} x_e \geq 1$$

Grundsätzlich muss die obige Ungleichung natürlich für jeden beliebiegen Cut von $G(\nu)$ erfüllt sein, damit der Zusammenhang von $G(\nu)$ erzwungen wird. Dies ist jedoch offensichtlich der Fall, wenn die Ungleichung für jeden Mincut von $G(\nu)$ gilt.

Analog lassen sich diese Constraints auch für die Komplementgraphen definieren.

3.5.3 Zielfunktion

Damit eine optimale Lösung des ILPs auch zu einem C-planaren Subgraphen maximaler Kardinalität korrespondiert, muss das Optimierungsziel, wie bereits motiviert, die Maximierung der Originalkanten sein. Zum Anderen wird auch noch die Minimierung der neuen Zusammenhangskanten gewichtet mit in die Zielfunktion aufgenommen. Ohne gleichzeitige Minimierung der Zusammenhangskanten würden unter Umständen mehr Kanten hinzugefügt werden, als nötig sind, um den Graphen vollständig zusammenhängend zu machen. Durch jede zusätzliche Kante können wieder nicht-planare Substrukturen entstehen, die erst wieder durch Extraktion entsprechender Kuratowski-Constraints zerstört werden können und müssen. Dies würde zu einem unnötigen blow-up der einzelnen LPs im Branch-and-Cut Algorithmus führen. Auf die Gewichtung der zu Zusammenhangskanten korrespondierenden Variablen in der Zielfunktion durch einen Faktor ϵ wird an späterer Stelle im folgenden Kapitel in Abschnitt 4.5.1 eingegangen.

3.5.4 Formale Definition des ILP

Gegeben sei ein Clustergraph $C = (G, T)$ mit $G = (V, E)$. e_O bezeichne eine Kante aus E (Originalkante) und e_C eine Kante, die nicht in E vorhanden ist (Zusammenhangskante). Für jede Kante e_O wird eine korrespondierende binäre Variable x_{e_O}, und für jede Kante e_C eine binäre Variable y_{e_C} erzeugt. Seien für einen Cluster ν außerdem $G(\nu)$ der durch $V(\nu)$ induzierte Subgraph und $G(V \backslash V(\nu))$ der durch die Entfernung von $G(\nu)$ induzierte Komplementgraph. Weiterhin bezeichne $M_{\text{Mincut}}(G(\nu))$ die Menge aller Mincuts in dem durch die Knotenmenge $V(\nu)$ induzierten Graphen $G(\nu)$ und $E(M)$ die durch einen Mincut M eindeutig induzierte Kantenmenge. Dann lässt sich das MCPSP wie folgt als ILP formulieren:

$$\max \quad \sum_{e_O \in E} x_{e_O} - \epsilon \sum_{e_C \notin E} y_{e_C}$$

s.to.

$$\begin{aligned}
\sum_{e_O \in E : e_O \in E(M)} x_{e_O} \quad + \quad \sum_{e_C \notin E : e_C \in E(M)} y_{e_C} \quad &\geq \quad 1 \qquad \forall\ M \in M_{\text{Mincut}}(G(\nu)) \\
\sum_{e_O \in E : e_O \in E(M)} x_{e_O} \quad + \quad \sum_{e_C \notin E : e_C \in E(M)} y_{e_C} \quad &\geq \quad 1 \qquad \forall\ M \in M_{\text{Mincut}}(G(V \backslash V(\nu))) \\
\sum_{e_O \in E : e_O \in S} x_{e_O} \quad + \quad \sum_{e_C \notin E : e_C \in S} y_{e_C} \quad &\leq \quad |S| - 1 \quad \forall \text{ Kuratowski-Subdivisions } S
\end{aligned}$$

$$x_{e_O}, y_{e_C} \in \{0, 1\} \quad \forall\ e_O \in E,\ e_C \notin E, \quad \forall \text{ Cluster } \nu \in T, \quad \epsilon \in \mathbb{R}$$

Jede optimale, ganzzahlige 0/1-Lösung des ILPs induziert einen maximum C-planaren Subgraphen des gegebenen Clustergraphen. Das ILP stellt also eine vollständige und korrekte Beschreibung des MCPSP dar.

Kapitel 4

Implementierung des Branch-and-Cut Algorithmus

In diesem Kapitel wird die konkrete Implementierung des formulierten ILPs für das MCPSP in einen Branch-and-Cut-Algorithmus im Detail erläutert. In Abschnitt 4.1 werden zunächst einige grundsätzliche Dinge bezüglich der Implementierung erläutert und das verwendete Framework ABACUS vorgestellt. Im darauf folgenden Abschnitt 4.2 wird der durch den implementierten Branch-and-Cut Algorithmus definierte Optimierungsablauf dargestellt. Abschnitt 4.3 erläutert ausführlich die algorithmische Umsetzung der Constraint-Separierung. In Abschnitt 4.4 wird die implementierte und verwendete Heuristik zur Suche nach guten zulässigen Lösungen und somit zur Verbesserung der primalen Schranke vorgestellt. Es folgen einige Details zur Zielfunktion in Abschnitt 4.5. Abschnitt 4.6 beschreibt die verwendeten Branching- und Enumerationsstrategien.

4.1 Allgemeiner Aufbau des Algorithmus

Ein vollständiger Branch-and-Cut Algorithmus besteht im Allgemeinen aus folgenden Bestandteilen:

- Effiziente Separations-Algorithmen zur dynamischen Erzeugung der Constraints.
- Heuristik(en) zur Suche nach guten zulässigen Lösungen und zur Verbesserung der primalen Schranke.
- Enumerations- und Branchingstrategien, die definieren, wie und in welcher Reihenfolge neue Subprobleme erzeugt und offene Subprobleme bearbeitet werden.
- Ein LP-Solver zur Lösung der einzelnen Subprobleme (LP-Relaxierungen).
- Aufbau des Branch-and-Bound Baums sowie die Erzeugung, Speicherung Verwaltung der Subprobleme.
- Geeignete, effiziente Datenstrukturen zur, Speicherung und Verwaltung der Variablen und Constraints.

Die Implementierung eines solchen Algorithmus enthält sehr stark applikationsspezifische Bestandteile, was sich in erster Linie auf die Separationsalgorithmen und Heuristiken bezieht. Das grundsätzliche Prinzip eines solchen Verfahrens ist hingegen unabhängig von der konkreten Applikation. Daher können insbesondere die drei letzten Punkte in der Auflistung oben etwas losgelöster vom konkreten Kontext betrachtet werden, da es sich dabei um allgemeinere Bestandteile handelt. Es existieren mittlerweile speziell zur Implementierung von Branch-and-Cut Algorithmen ausgelegte Frameworks, die einen Großteil der gerade als eher applikationsunabhängig eingestuften Bestandteile implementieren, und eine flexible und weitgehend komfortable Implementierung ermöglichen. ABACUS ist ein solches Framework, das frei zur Nutzung zur Verfügung steht und auch zur Implementierung des Algorithmus für das MCPSP benutzt wurde.

4.1.1 ABACUS

ABACUS (**A** **B**ranch **A**nd **CU**t **S**ystem) ist ein objekt-orientiertes in C++ geschriebenes Framework, das eine flexible und robuste Implementierung von Branch-and-Cut und Branch-and-Price Algorithmen unterstützt, zur Lösung ganzzahliger und gemischtganzzahliger linearer Optimierungsprobleme. Es wurde an der Universität Köln 1995 von Stefan Thienel und Michael Jünger entwickelt [JT97]. Das ABACUS-Framework übernimmt diejenigen Aufgaben im Branch-and-Cut-Algorithmus, die weitgehend applikationsunabhängig sind. Dem Programmierer wird dadurch die Arbeit wesentlich erleichtert, da sich dieser auf die Implementierung der applikationsspezifischen Teile des Algorithmus konzentrieren kann, ohne sich zum Beispiel Gedanken über die Verwaltung des Branch-and-Bound Baums machen zu müssen. ABACUS hält dazu Klassen bereit, von denen einige für den Programmierer weitgehend „unsichtbar“ sind. Andere Klassen müssen abgeleitet und applikationsspezifisch implementiert werden. Es steht Funktionalität bezüglich der Verwaltung des Branch-and-Bound Baums, der Variablen und Constraints, und der Lösung der durch die einzelnen Subprobleme induzierten LPs bereit. Für Letzteres wird der kommerzielle LP-Solver *CPlex* unterstützt und verwendet, der unter anderem eine Implementierung der Simplex-Methode enthält. Der hohe Grad an Flexibilität wird in erster Linie durch die Verwendung rein virtueller und virtueller Funktionen erreicht, die von ABACUS jeweils an bestimmten Stellen im Optimierungsprozess aufgerufen werden, und somit die Schnittstelle zur konkreten Applikation bilden. Einige Funktionen müssen daher applikationsspezifisch implementiert werden. Andere besitzen default-Implementierungen, die jedoch bei Bedarf überschrieben werden können. Wieder andere Methoden beinhalten per default keinerlei Funktionalität, können aber implementiert und somit als Einstiegspunkt genutzt werden, um aktiv in bestimmte Bereiche des Optimierungsprozesses einzugreifen.

Abbildung 4.1 zeigt ein Diagramm der Applikations-Basisklassen von ABACUS sowie die davon abgeleiteten Klassen für das MCPSP (dies sind die Klassen mit dem Präfix „`MaxCPlanar`“). Bei den dargestellten ABACUS-Klassen handelt es sich um diejenigen Klassen, die für den Programmierer am wichtigsten sind. Insbesondere gilt, dass in jedem auf ABACUS basierenden Branch-and-Cut Algorithmus Klassen implementiert werden müssen, die von den `MASTER`, `SUB`, `VARIABLE` und `CONSTRAINT` abgeleitet sind. Es folgt eine kurze Beschreibung dieser Basisklassen.

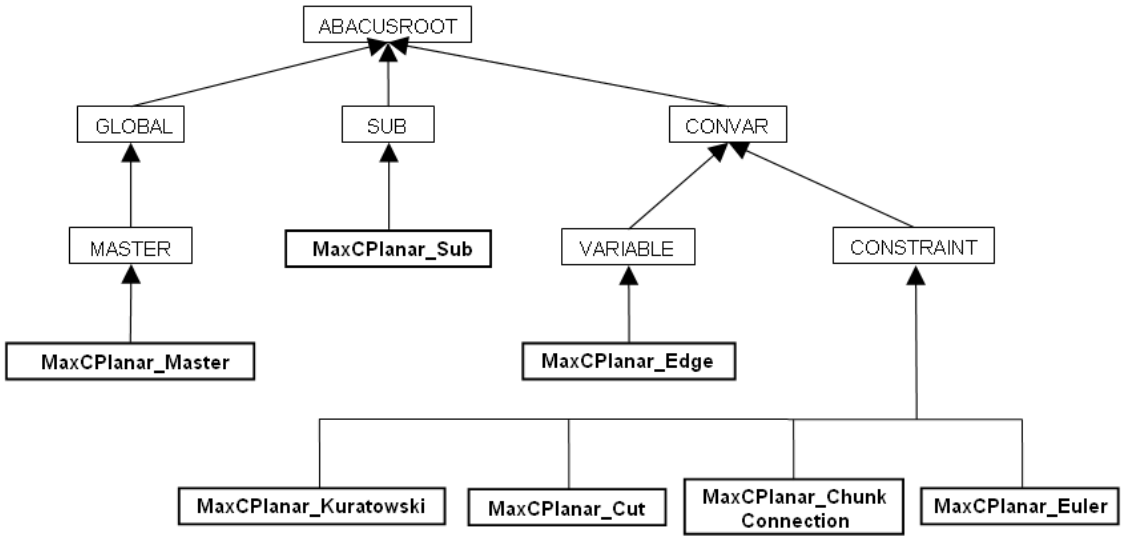


Abbildung 4.1: Basis-Klassen des ABACUS-Framework

Master-Klasse

Die Klasse `MASTER` ist eine der zentralen Klassen des ABACUS-Frameworks. Sie startet, kontrolliert und steuert den Optimierungsprozess und speichert und verwaltet globale Datenstrukturen für die Optimierung. Sie ist abgeleitet von der Klasse `GLOBAL`, in der grundlegende, globale Daten definiert sind, wie zum Beispiel Werte für numerisch bedingte Toleranzen, globale Parameter für den Optimierungsprozess, Ausgabe-Datenströme etc. Nahezu jedes Objekt des Frameworks besitzt einen Zeiger auf ein Objekt der Klasse `GLOBAL` oder eine davon abgeleitete Klasse, wie zum Beispiel die `MASTER`-Klasse.

Subprobleme

Die Klasse `SUB` repräsentiert ein Subproblem, das heißt einen Knoten des Branch-and-Bound Baums. Sie beinhaltet unter anderem Methoden zur Lösung des durch das Subproblem definierten LPs (also eine Schnittstelle zu den Methoden des verwendeten LP-Solvers *CPlex*), sowie virtuelle Funktionen, die zur Implementierung von Separations- oder Spaltengenerierungs-Algorithmen und Heuristiken überschrieben werden können.

Variablen, Constraints

Von `CONSTRAINT` und `VARIABLE` abgeleitete Klassen werden zur Definition und Speicherung der benötigten Variablen und Constraints benutzt. Jede Variable und jeder Constraint ist also jeweils ein eigenes Objekt. Aufgrund der großen Gemeinsamkeiten zwischen den Variablen des primalen LPs und den Constraints des dualen LPs (oder umgekehrt) sind die beiden Basis-Klassen `CONSTRAINT` und `VARIABLE` von einer gemeinsamen Superklasse

`CONVAR` abgeleitet. Variablen und Constraints werden in einer durch ABACUS verwalteten Datenstruktur, den sogenannten *Pools*, gespeichert.

4.2 Grundsätzlicher Optimierungsablauf

In diesem Abschnitt wird der konkrete Ablauf des Branch-and-Cut Algorithmus bzw. der durch diesen implementierte Optimierungsprozess beschrieben.

1. Zu Beginn wird der gegebene Clustergraph $C = (G, T)$ eingelesen und ein Objekt der abgeleiteten Klasse `Master` instanziiert. Von dieser wird die Methode `optimize()` aufgerufen, wodurch der Optimierungsprozess gestartet wird.

2. `optimize()` ruft als erstes die virtuelle Methode `initializeOptimization()` auf. Diese beinhaltet default-mäßig keine Funktionalität. Das Framework sieht vor, dass diese überschrieben und dazu genutzt wird, um zum Beispiel initiale Variablen und Constraints zu erzeugen, globale Werte und Datenstrukturen zu initialisieren und gegebenenfalls eine initiale duale Schranke zu berechnen. Konkret ist diese Methode wie folgt implementiert:

   - Als erstes werden die Variablen erzeugt und zum Standard-Pool hinzugefügt. Für jede vorhandene Kante $e \in E$, sowie für jede nicht vorhandene Kante $e \notin E$ wird jeweils eine Variable erzeugt. Diese werden dabei jeweils durch ein Objekt der von `ABA_VARIABLE` abgeleiteten Klasse `Edge` repräsentiert. Die Klassifikation als Original- oder Zusammenhangskante erfolgt über einen boolschen Wert.
   - Weiterhin werden initiale Constraints erzeugt, wie sie in Abschnitt 4.3.3 beschrieben werden. Diese werden durch die von `CONSTRAINT` abgeleiteten Klassen `ChunkConnection` und `Euler` umgesetzt.
   - Zum Schluss wird noch eine (sehr einfache) Heuristik gestartet, die eine duale Schranke initialisiert. Diese prüft den zugrunde liegenden Graphen lediglich auf Planarität. Im negativen Fall werden mit Hilfe des BoyerMyrvold-Planaritätstests (siehe Abschnitt 4.3.2) eine Reihe Kuratowski-Subdivisions extrahiert und überprüft, ob unter diesen zwei Kanten-disjunkte Subdivisions vorhanden sind. Falls ja, wird die duale Schranke mit dem Wert $|E|-2$ initialisiert, da mindestens zwei Kanten entfernt werden müssen, ansonsten mit $|E|-1$. Ist der Graph planar, wird die duale Schranke einfach auf den Wert $|E|$ gesetzt.

3. `optimize()` ruft nun die Methode `generateSon()` der Klasse `Sub` auf, wodurch das erste Subproblem, und damit die Wurzel des Branch-and-Bound Baums erzeugt wird.

4. Als nächstes wird die durch das erzeugte Subproblem induzierte LP-Relaxierung mittels Schnittebenen-Verfahren optimal gelöst. Dies geschieht auf folgende Weise:

   (a) Im aktuell betrachteten Subproblem wird (von ABACUS gesteuert) die Funktion `solveLP()` aufgerufen, die anhand der aktuell im Pool vorhandenen Constraints und Variablen (die Anzahl der Variablen ist während der gesamten Optmierung konstant) ein entsprechendes LP erzeugt und gelöst. Intern wird dazu das in CPlex implementierte Simplex-Verfahren aufgerufen.

(b) Anschließend wird die dynamische Separation der Constraints gestartet. Es wird dazu die Methode `separate()` aufgerufen, und von dieser aus zunächst der Separations-Algorithmus für die Cut-Constraints. Konnten keine verletzten Cuts separiert werden, wird der Separations-Algorithmus für die Kuratowski-Constraints aufgerufen. Die Suche nach verletzten Kuratowski-Constraints wird also nur dann gestartet, wenn zuvor in dieser Iteration keine Cuts separiert wurden. Der Grund für diese Reihenfolge ist in erster Linie der folgende: jede neu zum Graphen hinzugefügte Kante kann unter Umständen neue nicht-planare Substrukturen im Graphen induzieren. Daher werden als erstes alle verletzten Cut-Constraints separiert und zum LP hinzugefügt, bevor mit der Separation der Planaritäts-Constraints begonnen wird. Die Separations-Algorithmen sind detailliert in Abschnitt 4.3 beschrieben.

(c) Wurden neue Constraints extrahiert, so werden diese zum LP hinzugefügt und dieses erneut gelöst. Die Schritte (a) und (b) werden iterativ solange durchgeführt, bis keine Constraints mehr separiert werden können.

5. Während der iterativen Optimierung des LPs wird stetig von ABACUS überprüft, ob der aktuelle Zielfunktionswert des LPs (also die lokale duale Schranke) niedriger als die globale, primale Schranke ist. Falls ja, wird die Optimierung des aktuellen LPs beendet und der Teilbaum des Branch-and-Bound Baums, dessen Wurzel das aktuelle Subproblem ist, abgeschnitten, da in diesem Zweig keine bessere Lösung mehr gefunden werden kann. Dieses Bounding-Prinzip wurde in Abschnitt 3.2 beschrieben.

6. Nach der Optimierung eines LPs wird geprüft, ob die berechnete LP-Lösung eine zulässige Lösung für das ILP darstellt. Dazu stellt ABACUS die rein virtuelle Methode `feasible()` bereit, die der Applikation entsprechend implementiert werden muss. In unserem Fall stellt die LP-Lösung genau dann eine zulässige Lösung für das ILP dar, wenn gilt:

   - Die Lösung ist ganzzahlig.
   - Der durch die 1-Variablen induzierte Graph ist planar und vollständig zusammenhängend.

7. Die Ganzzahligkeit der Lösung ist trivial nachzuprüfen. Die Planaritäts-Eigenschaft wird durch einen Aufruf des BoyerMyrvold-Planaritätstests ermittelt. Zum Test auf vollständigen Zusammenhang werden zunächst für jeden Cluster ν nacheinander die induzierten Subgraphen $G(\nu)$ erzeugt und auf Zusammenhang getestet. Anschließend werden der Reihe nach die Cluster-induzierten Komplementgraphen $G(V \backslash V(\nu))$ berechnet und jeweils auf Zusammenhang getestet. Der Test erfolgt jeweils mittels eines einfachen DFS. Sind sowohl alle Cluster-induzierten als auch alle Komplementgraphen zusammenhängend, so ist der durch die Lösung induzierte Graph vollständig zusammenhängend. Ist dieser außerdem planar, handelt es sich um eine zulässige Lösung für das ILP. Die Funktion `feasible()` liefert in diesem Fall `true` zurück, ansonsten `false`.
   Es wird im positiven Fall außerdem noch getestet, ob die primale Schranke verbessert werden kann. Dazu wird geprüft, ob der Zielfunktionswert der aktuellen, lokalen Lösung größer ist, als der Zielfunktionswert der bisher global besten, primalen Lösung. Falls ja, wird die globale, primale Schranke entsprechend aktualisiert.

8. Heuristiken zur Suche nach zulässigen Lösungen werden vom ABACUS-Framework her durch die virtuelle Methode `improve()` der Klasse `SUB` aufgerufen. Default-mäßig besitzt diese Methode keinerlei Funktionalität, kann aber in einer abgeleiteten Klasse überschrieben werden, um an dieser Heuristiken zu implementieren. `improve()` wird grundsätzlich nach jeder Iteration des Schnittebenen-Verfahrens aufgerufen. In unserer konkreten Implementierung dieser Methode wird nun zunächst überprüft, ob im vorangegangenen Separationsschritt Constraints separiert wurden. Falls ja, wird die Heuristik nicht aufgerufen. Falls nicht, so wird als nächstes überprüft, ob die LP-Lösung ganzzahlig ist. Falls ja, so wird die Heuristik ebenfalls nicht aufgerufen. Nur wenn die gerade betrachtete LP-Lösung fraktional ist und keine Constraints mehr separiert werden konnten, wird die Heuristik gestartet, also unmittelbar vor einem Branching-Schritt.
   Die Gründe dafür liegen zum Einen darin begründet, dass es sich aufgrund der Art der implementierten Heuristik nicht anbietet, diese für eine ganzzahlige LP-Lösung aufzurufen. Zum Anderen wird das betrachtete LP durch die Separation und Hinzunahme verletzter Constraints zunehmend verschärft, sodass erst „abgewartet“ wird, bis keine Constraints mehr separiert werden konnten. Die implementierte Heuristik ist im Detail in Abschnitt 4.4 beschrieben.
   Findet die Heuristik nun eine zulässige Lösung, deren Zielfunktionswert höher als der Wert der globalen primalen Schranke ist, so wird diese entsprechend verbessert. Enthält die berechnete Lösung alle Originalkanten des Graphen, so wird die Optimierung beendet, da der gegebene Graph somit C-planar ist.

9. Zu jedem Zeitpunkt des Algorithmus besteht der Branch-and-Bound Baum aus einer Menge offener, noch nicht bearbeiteter Subprobleme, und einer Menge bereits bearbeiteter oder abgeschnittener Subprobleme. Es muss also jeweils am Ende der Optimierung des aktuellen Subproblems entschieden werden, welches der noch offenen Subprobleme als nächstes ausgewählt und bearbeitet wird. Dies geschieht anhand sogenannter *Enumerations-Strategien*. Die verwendete Enumerations-Strategie ist in Abschnitt 4.7 beschrieben.

10. Konnte am Ende der Optimierung des aktuellen Subproblems keine zulässige Lösung gefunden werden, so wird ein Branching-Schritt auf dem aktuellen Subproblem S durchgeführt. Beim Branching wird zunächst anhand der verwendeten Branching-Strategie eine der fraktionalen Variablen x_f ausgewählt und anhand dieser zwei neue Subprobleme S_0 und S_1 erzeugt, sodass in dem einen Subproblem die ausgewählte Variable x_f auf den Wert 0 fixiert ist, in dem anderen auf 1. Es gilt also $S_0 = S(x_f = 0)$ und $S_1 = S(x_f = 1)$. Nach welcher Strategie die Branching-Variable ausgewählt wird, ist in Abschnitt 4.6 erläutert.

Terminierung der Optimierung

Der Algorithmus terminiert wenn eine der folgenden Bedingungen erfüllt ist:

- Berechnet die Heuristik einen C-planaren Graphen, der alle Originalkanten enthält, oder findet der LP-Solver eine zulässige, ganzzahlige Lösung, in der alle zu Originalkanten korrespondierenden Variablen den Wert 1 besitzen, so wird die Optimierung an dieser Stelle beendet, da offensichtlich eine optimale Lösung gefunden wurde. Der gegebene Clustergraph ist in diesem Fall C-planar.

- Wird eine ganzzahlige, zulässige Lösung gefunden, deren Zielfunktionswert die globale duale Schranke erreicht oder übersteigt, so handelt es sich um eine Optimallösung. Die Optimierung wird daher beendet.
- Besitzt der Branch-and-Bound Baum keine offenen Subprobleme mehr, und können keine weiteren erzeugt werden, da bereits alle betrachtet bzw. abgeschnitten wurden, so korrespondiert die aktuell global beste primale Lösung zu einer Optimallösung.

Da es zu jedem Clustergraphen einen maximum C-planaren Subgraphen gibt, existiert auch immer eine optimale Lösung.

4.3 Separierung der Constraints

In diesem Abschnitt wird detailliert beschrieben, wie die Algorithmen zur dynamischen Separation der Constraints implementiert wurden.

4.3.1 Cut-Constraints

Die Cut-Constraints werden mit Hilfe eines Mincut-Algorithmus separiert. Dazu wurde ein Algorithmus implementiert, der auf einem Ansatz von Wagner und Stoer [SW97] basiert. Dieser beruht nicht auf dem Max-Flow = Mincut Theorem, sondern ist Graphen- bzw. Mengen-orientiert. Die Laufzeit des Algorithmus ist $O(n\,log(n))$ wobei n die Anzahl der Knoten im Graphen ist.

Wie bereits in Abschnitt 3.5.2 erläutert, ist der Wert eines Mincuts auf einem zusammenhängenden Graphen mindestens 1. Für eine ganzzahlige LP-Lösung kann der vollständige Zusammenhang des durch die Lösung induzierten Graphen dadurch überprüft werden, indem man einen Mincut auf jedem der Cluster-induzierten Graphen und Komplementgraphen berechnet. Da die LP-Lösungen aber im Allgemeinen fraktional sind, induziert die LP-Lösung nicht eindeutig, ob eine Kante zum Graphen gehört oder nicht. Es muss daher eine relaxierte Variante von Zusammenhang betrachtet werden. Die Kanten erhalten nun ein Gewicht gemäß ihres LP-Wertes. Für jeden Cluster ν werden nun die korrespondierenden *gewichteten* Graphen $G_w(\nu)$ und $G_w(V \backslash V(\nu))$ betrachtet. Auf diesen wird jeweils mittels des implementierten Mincut-Algorithmus ein Mincut berechnet. Gibt es einen Graphen, für den der berechnete Mincut einen Wert von echt kleiner als 1,0 hat, so kann ein verletzter Cut-Constraint separiert werden. Dieser hat folgendes Aussehen:

Seien $e_1 \dots e_k$ die Kanten eines Cluster-induzierten Graphen oder Komplementgraphen G und $w_{e_1} \dots w_{e_k}$ die entsprechenden Kantengewichte, die den LP-Werten der korrespondierenden Variablen $x_{e_1} \dots x_{e_k}$ entsprechen. Dies bezieht sich natürlich sowohl auf die Originalkanten, als auch die potentiellen Zusammenhangskanten. Falls der Wert des berechneten Mincuts auf G echt kleiner als 1,0 ist, so ist folgender Cut-Constraint verletzt und kann separiert werden:

$$\sum_{i=1}^{k} w_{e_i} \geq 1,0 \quad \Leftrightarrow \quad \sum_{i=1}^{k} x_{e_i} \geq 1,0$$

Die Berechnung eines Mincuts hat eine worst-case Laufzeit von $O(n\,log(n))$. Für jeden Cluster werden zwei Mincuts berechnet, für den Cluster-induzierten Graphen und den Komplementgraphen. Daher ergibt sich für die Cut-Separierung pro Iteration eine worst-case Laufzeit von $O(c\ (n\,log(n)))$, wobei c die Anzahl der Cluster ist.

4.3.2 Kuratowsi-Constraints

Zur Separation der Kuratowski-Constraints dient ein kürzlich in der OGDF implementierter Planaritätstest für allgemeine Graphen. Dieser ist eine Implementierung des in [BM04] vorgestellten Linearzeit-Algorithmus, der im Rahmen einer vorangegangenen Diplomarbeit [Sch07] implementiert worden ist. Der Algorithmus liefert in seiner Grundform als Rückgabe die Information, ob der übergebene Graph planar ist. Außerdem liefert er im positiven Fall direkt eine planare Einbettung des Graphen. Interessant für die nun betrachtete Problemstellung, die effiziente Separierung von Kuratowski-Constraints, ist nun die Tatsache, dass der Algorithmus im Rahmen der genannten Diplomarbeit dahingehend modifiziert worden ist, dass er im negativen Fall eine Menge von Kuratowski-Subdivisions extrahiert. Dazu werden heuristische Strategien verwendet, die insbesondere darauf ausgelegt sind, möglichst ähnliche Subdivisions zu finden. Die maximale Anzahl an extrahierten Kuratowski-Subdivisions lässt sich anhand eines Parameters einstellen, wobei allerdings eine fest implementierte Grenze definiert ist, da es unter Umständen exponentiell viele Subdivisions geben kann. Diese ist jedoch hoch genug, sodass dies zum Zwecke der Constraint-Separierung vollkommen ausreicht. Diese Einstellungsmöglichkeit erlaubt es, die Performance der Separation anhand verschiedener Einstellungen zu untersuchen (siehe Abschnitt 5.3.1). Der Algorithmus durchläuft den Graphen außerdem anhand eines randomisierten DFS. Welche Kuratowski-Subdivisions letztendlich extrahiert werden ist daher teils vom konkreten Weg des DFS abhängig. Aufgrund der Begrenzung der maximal extrahierten Subdivisions pro Separations-Iteration werden demnach möglicherweise nicht bei jedem Aufruf des Planaritätstests verletzte Subdivisions extrahiert. Ob sich daher bei der Separation ein wiederholtes Aufrufen des Planaritätstests positiv auf die Performance des Branch-and-Cut Algorithmus auswirkt, ist ebenfalls experimentell untersucht worden (siehe Abschnitt 5.3.1).

Jeder durch eine zulässige, ganzzahlige Lösung induzierte Graph enthält aufgrund seiner Planaritätseigenschaft keine Kuratowski-Subdivisions. Demnach ist für eine solche Lösung auch kein Kuratowski-Constraint verletzt. Falls eine ganzzahlige LP-Lösung einen nicht-planaren Graphen induziert, so können für diesen mit Hilfe des implementierten BoyerMyrvold-Planaritätstests verletzte Kuratowski-Constraints separiert werden. Die Lösungen der LP-Relaxierung sind allerdings im Allgemeinen fraktional. Die Separierung verletzter Kuratowski-Constraints für eine fraktionale Lösung ist nun nicht mehr auf diese einfache Weise möglich, denn der durch die fraktionale Lösung induzierte Graph ist nicht eindeutig definiert. Vielmehr gibt es im fraktionalen Fall keinen induzierten Graphen mehr. Dieser muss zunächst „approximiert“ werden. Dies geschieht wie folgt: anhand einer unteren und oberen Schranke $x_{low} \in [0, 1]$ und $x_{high} \in [0, 1]$ wird entschieden, ob eine Kante zum *Support-Graphen* hinzugenommen wird oder nicht. Alle Kanten, deren korrespondierende Variablen einen LP-Wert von höchstens x_{low} haben, werden nicht zum Supportgraphen hinzugefügt. Alle Kanten, deren Variablen einen LP-Wert von mindestens x_{high} haben, werden zum Support-Graphen hinzugefügt. Bei einem LP-Wert zwischen x_{low} und x_{high} wird die entsprechende Kante randomisiert gesetzt, wobei die Wahrscheinlichkeit, ob

die Kante zum Support-Graphen hinzugefügt wird oder nicht, dem LP-Wert entspricht. Mittels dieser Vorgehensweise erhofft man sich einen Graphen zu erzeugen, durch den die aktuelle (fraktionale) LP-Lösung möglichst gut repräsentiert wird.

Es kann nun allerdings vorkommen, dass bestimmte durch die LP-Lösung verletzte Kuratowski-Constraints nicht extrahiert werden können, da die korrespondierenden Subdivisions durch den berechneten Support-Graphen nicht verletzt sind. Da der Support-Graph jedoch zum Teil randomisiert berechnet wird, ist es möglich, dass durch eine andere Auswahl der Kanten ein Graph entsteht, in dem diese Subdivisions enthalten sind, und somit entsprechende Constraints separiert werden können. Ob sich bei der Separation der Kuratowski-Constraints die wiederholte randomisierte Berechnung von Support-Graphen bezüglich der Performance des Algorithmus bemerkbar macht, wurde experimentell untersucht (siehe Abschnitt 5.3.1).

Gute Werte für die Schranken x_{low} und x_{high} lassen sich nur durch experimentelle Analyse der Separierung ermitteln. Setzt man Variablen zu „großzügig“ so ist der ermittelte Support-Graph möglicherweise zu dicht und man findet viele Kuratowski-Subdivisions, die eigentlich gar nicht verletzt sind. Setzt man die Kanten zu spärlich, so werden möglicherweise keine Subdivisions gefunden, obwohl es verletzte gibt. Werte, die in vielen Fällen zu guten Ergebnissen führen, wurden experimentell ermittelt (siehe Abschnitt 5.3.1).

4.3.3 Initiale Constraints

Bei der Initialisierung der Optimierung (in der Methode `initializeOptimization()`) werden noch einige initiale Constraints erzeugt, die zu Beginn der Optimierung zum root-LP hinzugefügt werden und global für jedes Subproblem Gültigkeit besitzen. Dabei handelt es sich zum Einen um gewöhnliche, initial verletzte Cut-Constraints, zum Anderen um Constraints, die sich aus der oberen Schranke für die maximal mögliche Anzahl von Kanten in einem planaren Graph ergeben (siehe Theorem 2.1.3). Diese sind durch die Klassen `ChunkConnection` und `Euler` repräsentiert (siehe Abbildung 4.1).

Zur initialen Erzeugung von Cut-Constraints werden iterativ alle Cluster-induzierten Subgraphen betrachtet. Sei $G(\nu)$ der gerade betrachtete Subgraph. Besteht dieser aus $k \geq 2$ Chunks $\nu^1 \dots \nu^k$, so werden k Constraints der Form

$$\sum_{e=\{v,w\}:v\in V(\nu^i),w\in V(\nu)\setminus V(\nu^i)} x_e \geq 1 \qquad 1 \leq i \leq k$$

erzeugt. Außerdem wird für jeden Cluster ν ein Constraint der Form

$$\sum_{e\in V(\nu)} x_e \leq 3|V(\nu)| - 6$$

erzeugt. Letzterer schränkt also initial die maximale Anzahl an Kanten pro Cluster-induziertem Graphen ein (im ganzzahligen Fall). Im Falle stark nicht-planarer Clustergraphen, bzw. Cluster-induzierter Subgraphen wird dadurch bereits zu Beginn der Optimierung der maximal erreichbare Zielfunktionswert eingeschränkt, falls die Anzahl der Kanten die durch den Constraint definierte Grenze überschreitet. Die Auswirkungen auf die Performance des Algorithmus sind in Abschnitt 5.5 dargestellt.

4.3.4 Constraints in der LP-Relaxierung

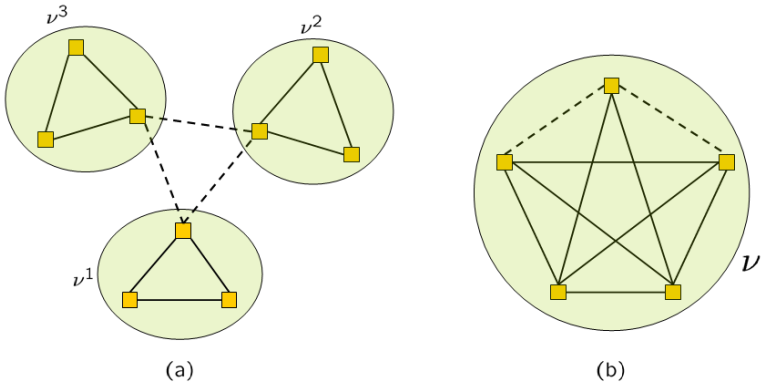


Abbildung 4.2: Erfüllte Constraints im fraktionalen Fall

Zur Veranschaulichung, dass die Cut- und Kuratowski-Constraints in Bezug auf die LP-Relaxierungen nicht ausreichen, um ganzzahlige Lösungen zu erzwingen, sind in Abbildung 4.2 zwei entprechende Situationen dargestellt. Betrachten wir dazu den in (a) dargestellten Cluster-induzierten Graphen. Dieser besteht aus drei Chunks ν^1, ν^2 und ν^3. Um den Cluster zusammenhängend zu machen, müssen mindestens zwei Kanten hinzugefügt werden, die die Chunks verbinden. Die drei entsprechenden Cut-Constraints sind:

$$\sum_{e=\{v,w\}:v\in V(\nu^1),w\in(V(\nu^2)\cup V(\nu^3))} x_e \geq 1$$

$$\sum_{e=\{v,w\}:v\in V(\nu^2),w\in(V(\nu^1)\cup V(\nu^3))} x_e \geq 1$$

$$\sum_{e=\{v,w\}:v\in V(\nu^3),w\in(V(\nu^1)\cup V(\nu^2))} x_e \geq 1$$

Im ganzzahligen Fall müssten mindestens zwei der betroffenen Variablen auf den Wert 1 gesetzt werden, um diese drei Constraints zu erfüllen. Im Kontext der LP-Relaxierung wird nun allerdings eine Situation ähnlich der folgenden auftreten. Der Algorithmus setzt drei der Variablen jeweils auf den Wert 0,5. Diese Situation ist in (a) dargestellt. Die gestrichelten Kanten sollen dabei die drei auf 0,5 gesetzten Variablen/Kanten repräsentieren. Die drei Constraints sind auf diese Weise ebenfalls alle erfüllt. Die Lösung ist allerdings fraktional und erzielt einen besseren Zielfunktionswert, da hierbei bildlich gesprochen nur $\frac{3}{2}$ Kanten hinzugefügt wurden anstatt zwei im ganzzahligen Fall. In einer solchen Situation reichen die das ILP vollständig definierenden Constraints also nicht aus, um eine zulässige Lösung zu erzwingen. Es muss gebrancht werden. Ein ähnliches Bild ergibt sich auch im Falle der Kuratowski-Constraints. (b) zeigt einen Cluster, dessen induzierter Graph dem K_5 entspricht. Aufgrund der verletzten Planaritäts-Eigenschaft, ist folgender Kuratowski-Constraint verletzt:

$$\sum_{e\in V(\nu)} x_e \leq |V(\nu)| - 1 = 9$$

Im ganzzahligen Fall muss also mindestens eine Kante gelöscht werden, bzw. eine der Variablen den Wert 0 erhalten, um diesen Constraint zu erfüllen. In der Relaxierung

könnten aber stattdessen auch zwei der Kanten jeweils den Wert 0,5 erhalten (siehe (b)). Dies verbessert zwar nicht unmittelbar den Zielfunktionswert, möglicherweise kann durch die fraktionalen Werte aber an „anderer Stelle“ etwas eingespart werden.

Insgesamt kommt man um Branching also nicht herum. Durch intensives Studium der Strukturen optimaler Lösungen gelingt es möglicherweise zusätzliche Klassen von Constraints zu beschreiben, die durch zulässige, ganzzahlige Lösungen erfüllt sind, aber solche Situationen, wie die oben dargestellten, im fraktionalen Fall verbieten. Die Suche nach zusätzlichen, die LP-Relaxierung eines ILPs verschärfenden Constraints fällt in das Forschungsgebiet der *polyedrischen Kombinatorik*.

4.4 Primale Heuristik

Ein sehr wichtiger, bzw. eher obligatorischer Bestandteil in einem Branch-and-Bound bzw. Branch-and-Cut Algorithmus sind Heuristiken zur Berechnung möglichst guter zulässiger Lösungen zur stetigen Verbesserung der primalen Schranke. Bei solchen Heuristiken handelt es sich im Allgemeinen um kombinatorische Algorithmen, die ausgehend von der aktuellen, im Allgemeinen fraktionalen LP-Lösung versuchen, eine zulässige Lösung für das zugrunde liegende Problem zu berechnen. Die fraktionalen LP-Werte der Variablen stellen dabei meist ein Indiz dar, wie eine gute Lösung aussehen könnte. Es macht daher Sinn, sich bei der Berechnung an diesen Werten zu orientieren. Es ist auch durchaus denkbar und sinnvoll, mehrere verschiedene Heuristiken zu entwerfen und zu verwenden. Im Rahmen des in dieser Diplomarbeit implementierten Algorithums wurde jedoch zunächst nur eine Heuristik implementiert, die im Folgenden beschrieben wird.

4.4.1 Primale Heuristik für das MCPSP

In der verwendeten Algorithmen-Bibliothek OGDF ist ein Algorithmus zum Testen der Cluster-Planarität eines C-zusammenhängenden Clustergraphen implementiert. Dieser basiert auf dem in [EFC95] vorgestellten Algorithmus. Die Idee der Heuristik besteht darin, ausgehend von der aktuellen (fraktionalen) LP-Lösung zunächst einen C-zusammenhängenden, C-planaren Subgraphen zu berechnen, und diesen dann durch sukzessives Hinzufügen weiterer Kanten derart zu erweitern, dass der resultierende Graph weiterhin C-planar bleibt und möglichst viele Originalkanten enthält. Die Heuristik besteht damit aus zwei Phasen.

Phase 1: Die Idee ist es, zunächst einen Spannbaum T_s auf dem zugrunde liegenden Graphen G zu berechnen, derart, dass für alle Cluster $\nu \in T$ gilt: $T_s(V(\nu))$ ist ein Spannbaum für $G(\nu)$. Gelingt dies, so stellt T_s eine zulässige Eingabe für den implementierten C-Planaritätstests dar. Dies folgt aus der Tatsache, dass T_s für jeden Cluster einen zusammenhängenden Subgraphen darstellt, und T_s somit C-zusammenhängend ist. Außerdem ist T_s auch C-planar, da jeder Baum C-planar ist, egal wie die über diesem definierte Clusterstruktur aussieht (dies ist eine einfache Beobachtung, sodass auf einen Beweis verzichtet wird). Der Spannbaum T_s soll nun möglichst solche Kanten enthalten, deren korrespondierende Variablen ein hohen LP-Wert haben, da ein hoher LP-Wert ein (wenn auch recht naives, aber dennoch begründetes) Indiz dafür darstellt, dass die korrespondierende Kante

auch zu einer guten Lösung gehört. Die Berechnung eines solchen Spannbaums wird auf folgende Weise erreicht:

Der Inklusionsbaum T wir bottom-up durchlaufen. Dadurch gilt die Invariante, dass zu jedem Zeitpunkt alle Subcluster des gerade betrachteten Clusters bereits besucht wurden.

- Ist der aktuell betrachtete Cluster ν ein Blatt-Cluster, enthält also keine weiteren Cluster, so wird auf diesem ein Spannbaum berechnet. Dazu wird zunächst der induzierte Graph $G(\nu)$ betrachtet, aus dem dann alle Kanten entfernt werden. Ausgehend von diesem Graphen werden nun iterativ wieder Kanten hinzugefügt. Dabei wird jedesmal überprüft, ob durch die Hinzunahme der Kante ein Kreis entsteht. Falls ja, wird die Kante wieder verworfen und die nächste getestet. Falls nicht, wird sie beibehalten. Sobald der Graph $|V(\nu)| - 1$ Kanten enthält, handelt es sich um einen Spannbaum von $G(\nu)$ und der nächste Cluster wird betrachtet.

- Ist der aktuell betrachtete Cluster μ ein innerer Knoten des Inkulsionsbaums, beinhaltet also weitere Cluster, so wird folgendermaßen vorgegangen: seien $\nu_1, \ldots, \nu_k,\ k \in \mathbb{N}$ die entsprechenden Kinder-Cluster von μ. Es gilt (aufgrund obiger Invariante), dass für diese bereits Spannbäume berechnet wurden. Seien dies die Graphen $T_s(\nu_1), \ldots, T_s(\nu_k)$. Genau wie im Fall der Blatt-Cluster wird auch hier zunächst mit dem Kanten-leeren induzierten Graphen $G(\mu)$ gestartet. Nun werden alle Kanten e, die in den Spannbäumen $T_s(\nu_i),\ i \in \{1, \ldots, k\}$ der Kinder-Cluster enthalten sind, zu $G(\mu)$ hinzugefügt. Von diesem ausgehend werden nun genau wie im Fall eines Blatt-Clusters iterativ weitere Kanten hinzugefügt, um einen Spannbaum $T_s(\mu)$ für $G(\mu)$ zu erhalten.

Ist auf diese Weise ein Spannbaum $T_s(\mu_r)$ für den root-Cluster μ_r berechnet worden, so stellt dieser auch einen C-zusammenhängenden Spannbaum für den zugrunde liegenden Graphen G dar. Die Berechnung der einzelnen Spannbäume gelingt *immer*, da man es ja im Prinzip jeweils mit vollständigen Graphen zu tun hat (Originalkanten plus Zusammenhangskanten). Es gilt noch zu erläutern, wie genau die Reihenfolge der zu testenden Kanten festgelegt wird.

Betrachten wir also die Spannbaum-Berechnung für einen Cluster ν. Um letztendlich eine möglichst gute zulässige Lösung zu erhalten, ist es natürlich das Bestreben auch möglichst viele Originalkanten zu den Spannbäumen hinzuzufügen. Diese werden daher zunächst bevorzugt getestet.

- Als erstes werden alle Originalkanten, deren korrespondierende Variablen einen LP-Wert von 1,0 haben, randomisiert permutiert und in der resultierenden Reihenfolge getestet.

- Konnte dadurch noch kein Spannbaum erzeugt werden, so werden die übrigen Kanten (sowohl Original- als auch Zusammenhangskanten) nun nach absteigendem LP-Wert getestet. Dies geschieht allerdings auch mit einem gewissen Grad an Randomisierung: die Liste der nach absteigendem LP-Wert sortierten Kanten wird in n_l Listen unterteilt, sodass Liste i , $1 \leq i \leq n_l$, diejenigen Kanten enthält, deren LP-Wert im Interval $[\frac{n_l-(i-1)}{n_l}; \frac{n_l-i}{n_l}]$ liegen. In jeder dieser Listen werden die enthaltenen Kanten dann zufällig permutiert. Anschließend werden diese wieder in der Reihenfolge

$1, \dots n_l$ zu einer Liste verschmolzen und die Kanten in dieser Reihenfolge getestet. Die Anzahl n_l der Listen, und damit implizit die Intervallgröße der LP-Werte einer Liste, ist in der Heuristik frei einstellbar. Durch die freie Wahl der Intervallgröße der Permutations-Listen lässt man sich die Möglichkeit offen, gute Werte experimentell zu ermitteln (siehe Abschnitt 5.3.1).

Phase 2: Nachdem man nun einen C-planaren, C-zusammenhängenden Subgraphen berechnet hat (der Spannbaum T_s), wird dieser nun durch sukzessives Hinzufügen weiterer Kanten vergrößert. Dazu wird der in der OGDF implementierte Planaritätstest für C-zusammenhängende Clustergraphen verwendet. Alle Originalkanten, die nicht zum Spannbaum T_s in Phase 1 hinzugefügt wurden, werden nach absteigendem LP-Wert sortiert. Die Sortierung erfolgt dabei randomisiert, genau wie es bereits bei der Beschreibung von Phase 1 erläutert wurde. In dieser Reihenfolge werden die Kanten nun iterativ zu T_s hinzugefügt. Jedesmal, wenn eine neue Kante eingefügt wird, wird mit Hilfe des C-Planaritätstests überprüft, ob der um diese Kante erweiterte Graph immer noch C-planar ist. Falls ja, wird die Kante beibehalten. Falls nein, wird sie wieder entfernt und verworfen. Am Ende erhält man auf diese Weise einen C-planaren Subgraphen des gegebenen Clustergraphen.

Theorem 4.4.1. *Der durch die Heuristik berechnete Graph ist C-planar.*

Beweis. Der in Phase 1 berechnete Subgraph T_s ist per Konstruktion ein Spannbaum und somit C-planar, da ein Baum unabhängig von der Clusterstruktur immer C-planar ist. T_s ist außerdem per Konstruktion C-zusammenhängend und daher eine zulässige Eingabe für den verwendeten C-Planaritätstest. In Phase 2 zu T_s hinzugefügte Kanten werden nur dann behalten, wenn der Graph anschließend immer noch C-planar ist, und ansonsten wieder aus T_s entfernt. Der resultierende Clustergraph am Ende von Phase 2 ist somit C-planar. □

4.5 Zielfunktion

In diesem Abschnitt soll noch die Zielfunktion diskutiert werden. Diese maximiert die Anzahl der Originalkanten und minimiert die Anzahl der zusätzlichen Zusammenhangs-Kanten, wobei diese durch einen Faktor $\epsilon \in \mathbb{R}$ gewichtet werden. Die Minimierung der Zusammenhangskanten in der Zielfunktion erfüllt den Zweck, dass nicht unnötig viele Kanten hinzugefügt werden, bzw. nicht „wahllos“ entsprechende Variablen einen Wert $> 0,0$ zugewiesen bekommen. Jedes Hinzufügen einer neuen Kante kann dazu führen, dass der durch die LP-Lösung induzierte Graph anschließend nicht mehr planar ist. Dadurch müssten unter Umständen vermeidbare Kuratowski-Constraints separiert werden, um die „unnötig“ hinzugefügten Kanten wieder zu entfernen, was das LP unnötig aufbläht. Durch die Minimierung wird dies zumindest etwas eingeschränkt.

Durch eine geschickte Wahl des Koeffizienten ϵ kann außerdem sichergestellt werden, dass der Wert einer zulässigen Lösung, aufgerundet auf die nächst höhere ganze Zahl, immer der Anzahl der durch die Lösung induzierten Originalkanten entspricht. Dazu muss garantiert werden, dass der Wert der Minimierungssumme in der Zielfunktion $\epsilon \sum_{e_C \notin E} y_{e_C}$ immer innerhalb eines fixen Intervalls liegt. Dadurch lässt sich die primale Schranke im Fall einer

besseren gefundenen zulässigen Lösung um einen zusätzlichen additiven Wert anheben. Dies wird konkret im folgenden Abschnitt erläutert.

4.5.1 Wahl des Gewichtungsfaktors ϵ

Der Koeffizient ϵ ist so gewählt, dass der Wert der Minimierungssumme in der Zielfunktion bezogen auf eine zulässige, ganzzahlige LP-Lösung nicht größer als 0,1 sein kann. Eine optimale Lösung für das ILP beinhaltet niemals mehr als $2|V|$ Zusammenhangskanten bezüglich des gegebenen Clustergraphen $C = (G, T)$. Diese Schranke ist sicherlich recht „großzügig“ aber korrekt: es gibt maximal $|V|$ Cluster und somit höchstens $|V|$ Cluster-induzierte Graphen und $|V|$ Komplementgraphen. Daher müssen auch maximal $2|V|$ Kanten hinzugefügt werden, um vollständigen Zusammenhang zu erhalten. Wählt man ϵ nun als $\frac{0{,}1}{2|V|}$, so liegt der Wert der Minimierungssumme in der Zielfunktion immer zwischen 0,1 und 0,0.

Man betrachte nun eine gefundene zulässige Lösung $\hat{x}$ mit k Originalkanten, die besser ist, als die bis dato beste, globale, primale Lösung. Aufgrund obiger Beobachtung liegt der LP-Wert dieser Lösung im Intervall $[(k-1)+0,9; k]$. Eine bessere Lösung für das MCPSP (nicht zwangsläufig das ILP) als die betrachtete, zeichnet sich nun dadurch aus, dass diese mehr als k Originalkanten beinhaltet. In diesem Fall hätte eine bessere Lösung also analog zum vorangegangenen Argument einen LP-Wert von mindestens $k+0,9$. Jedes LP, dessen lokale, duale Schranke also echt kleiner als $k+0,9$ ist, muss demnach nicht weiter betrachtet werden, da es keine Lösung enthalten kann, die mindestens $k+1$ Originalkanten enthält (die Variablen der Originalkanten sind ja mit Faktor 1 gewichtet). Daher ist es korrekt, die primale Schranke nicht nur auf den reinen LP-Wert von $\hat{x}$ zu verbessern, sondern direkt auf den Wert $k+0,89$ anzuheben, da es eben keine besseren zulässigen Lösungen für das MCPSP geben kann, die einen LP-Wert im Intervall $[k; k+0,89]$ haben.

Der Wert $0,1$ wirkt recht willkürlich gewählt. Es handelt sich dabei lediglich um eine vorsichtige Wahl. Zum Einen möchte man die Miniminierungssumme in der Zielfunktion so klein wie möglich halten, zum Anderen besteht aber die Gefahr, dass sich bei zu kleiner Wahl von ϵ numerische Ungenauigkeiten ergeben. Tests anhand verschiedener Werte wurden im Rahmen dieser Diplomarbeit jedoch nicht durchgeführt.

4.5.2 Perturbation der Zusammenhangskanten

In vielen technischen Bereichen gibt es sogenannte *Perturbations*-Techniken. Die genaue Bedeutung hängt vom Kontext ab. Grundsätzlich versteht man darunter jedoch das bewusste „Stören“ oder „Verwackeln“ bestimmter Vorgänge oder Werte. In Bezug auf den Branch-and-Cut Algorithmus soll unter der Perturbation von Variablen das Modifizieren der Variablen-Koeffizienten in der Zielfunktion durch einen kleinen Term verstanden werden. Bevor darauf näher eingegangen wird, wollen wir uns zunächst zur Motivation eine Situation anschauen, wie sie häufig während der Optimierung der LP-Relaxierungen auftritt.

Ist der durch einen Cluster ν induzierte Subgraph $G(\nu)$ nicht-zusammenhängend, so müssen Kanten zu $G(\nu)$ hinzugefügt werden. Besteht $G(\nu)$ aus vielen Chunks, so gibt es auch viele Möglichkeiten von Kantenkombinationen, diese zu verbinden. Wenn die zu

Zusammenhangskanten korrespondierenden Variablen alle exakt denselben Koeffizienten in der Zielfunktion haben, sind diese für den Algorithmus gleichwertig. Indem man die Koeffizienten dieser Variablen leicht *perturbiert* sind diese nicht mehr exakt gleichwertig, sondern geringfügig vom Algorithmus unterscheidbar.

Die Variablen der Zusammenhangskanten werden nicht willkürlich perturbiert, sondern anhand des *graphentheoretischen Abstandes* der zu der entsprechenden Kante inzidenten Knoten. Der graphentheoretische Abstand entspricht dabei der Anzahl der Kanten eines kürzesten Pfades zwischen den Knoten. Dies ist in Abbildung 4.3 anhand eines Ausschnittes eines einfachen Gittergraphen veranschaulicht, bei dem Knoten, die auf einer gemeinsamen Diagonalen liegen, zu demselben Cluster gehören. Die graphentheoretischen Abstände der Knoten v_1 zu v_2 und v_2 zu v_3 haben jeweils den Wert 2, der Abstand von v_1 zu v_3 beträgt 4 (die Pfade, die diese Abstände definieren, verlaufen entlang der hervorgehobenen Kanten in (a)). Um den Cluster μ zusammenhängend zu machen, müssen zwei Kanten hinzugefügt werden. In diesem Fall wäre die „naheliegendste“ Möglichkeit die in (b) dargestellte. Die Lösung in (c) wirkt intuitiv „ungünstiger“. Der Ansatz ist nun, die Koeffizienten der Zusammenhangskanten mit geringerem graphentheoretischen Abstand etwas günstiger zu machen, und somit eine Art Güte-Bewertung der Zusammenhangskanten zu erhalten. Das präsentierte Beispiel ist natürlich sehr speziell konstruiert. Es handelt sich bei dieser Art der Koeffizienten-Vergabe lediglich um einen möglichen Ansatz. Wie sich dies auf die Perfomance des Algorithmus auswirkt, ist in Abschnitt 5.7 dargestellt. Es sind durchaus andere Ansätze denkbar. Im Rahmen dieser Diplomarbeit wurde jedoch nur die Auswirkung der Perturbation anhand des graphentheoretischen Abstandes untersucht.

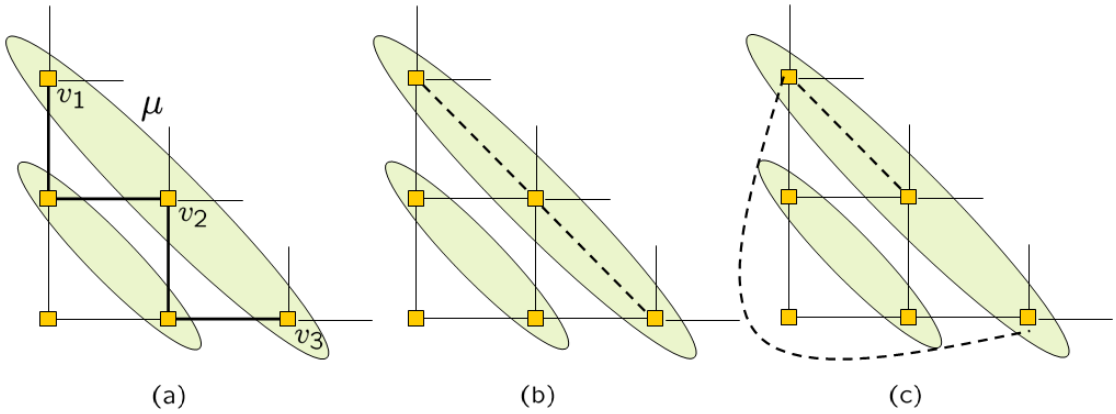


Abbildung 4.3: Graphentheoretischer Abstand

Die Werte zur Perturbation der Koeffizienten werden nun konkret wie folgt gewählt:

Für jede Zusammenhangskante wird durch einfache Breitensuche der graphentheoretische Abstand der inzidenten Knoten berechnet, und diese nach absteigendem Wert sortiert. Sei n_c die Anzahl der Zusammenhangskanten. Sei außerdem i die Position einer Kante

innerhalb der sortierten Reihenfolge. Dann ergibt sich der Koeffizient ϵ_i aus:

$$\begin{aligned} \delta_i &= \frac{(0,1\epsilon)}{n_c} * i \qquad 0 \leq i \leq n_c - 1 \\ \epsilon_i &= \epsilon - \delta_i \end{aligned}$$

Die jeweils von ϵ abgezogenen Werte sind also linear skaliert im Intervall $[0; 0,1\epsilon]$ und liegen vom Wert her alle unterhalb von ϵ. Dies ist sehr wichtig, damit die Minimierungssumme in der Zielfunktion des ILPs nach wie vor maximal den Gesamtwert $0,1$ annimmt. Wäre diese Eigenschaft verletzt, bzw. könnte dies nicht garantiert werden, so könnte das in Abschnitt 4.5.1 geschilderte Vorgehen zur Anhebung der primalen Schranke nicht mehr in dieser Form angewendet werden.

4.6 Branching

ABACUS stellt einige verbreitete Branching-Strategien bereit, die direkt benutzt werden können, ohne eigene Branching-Regeln implementieren zu müssen. Darunter fällt unter anderem die Strategie `CloseHalfExpensive`. Bei dieser wird als erstes diejenige bzw. werden diejenigen fraktionalen Variablen ermittelt, deren aktueller LP-Wert am dichtesten am Wert $0,5$ liegt. Von diesen Variablen wird dann diejenige als Braching-Variable ausgewählt, die den höchsten (kleinsten) Koeffizienten in der Zielfunktion hat, bezogen auf ein Maximierungsproblem (Minimierungsproblem). Die Idee hinter dieser Auswahlstrategie ist wie folgt: eine Variable, von der man noch keinerlei Hinweise bezüglich deren Werte-Tendenz hat, als Branching-Variable auszuwählen, ist meist vielversprechender. Der Grund, von solchen Variablen dann diejenige zu wählen, die den größten (kleinsten) Zielfunktions-Koeffizienten hat, ist naheliegend. Je stärker sich Änderungen des Wertes einer Variablen auf den Wert der Zielfunktion auswirken, desto stärker fällt dies ins Gewicht. Letztendlich hängt eine geeignete Branching-Strategie jedoch auch noch von der konkreten Applikation ab. `CloseHalfExpensive` stellt dennoch im Allgemeinen eine gute Strategie dar.

Der Branch-and-Cut Algorithmus für das MCPSP verwendet ebenfalls diese Strategie, allerdings etwas modifiziert. Dazu wurden die Memberfunktionen `selectBranchingVariableCandidates()` und `selectBranchingVariable()` überschrieben. Erstere wird von ABACUS benutzt um „Kandidaten“ für die Branching-Variable zu ermitteln, Zweitere um aus diesen dann konkret eine auszuwählen. `selectBranchingVariable()` wird von ABACUS an entsprechender Stelle aufgerufen und innerhalb dessen die Methode `selectBranchingVariableCandidates()`. Die vorgenommenen Änderungen sind dahingehend, dass bei der Auswahl der Branching-Variable eine Unterscheidung zwischen Original- und Zusammenhangskanten gemacht wird. Die Überlegung ist, dass es möglicherweise erfolgversprechender ist, vorzugsweise zunächst auf Originalkanten zu branchen, da diese einen hohen Zielfunktions-Koeffizienten haben $(1,0)$ und daher stärker ins Gewicht fallen. Zusammenhangskanten sind außerdem nur Mittel zum Zweck, um den Graphen vollständig zusammenhängend zu machen. Die konkrete Auswahl erfolgt auf folgende Weise:

1. Zunächst wird eine Branching-Variable mittels der in ABACUS standard-mäßig implementierten Methoden anhand der Strategie `CloseHalfExpensive` ermittelt.

Dabei handelt es sich um eine fraktionale Variable, deren aktueller LP-Wert um höchstens $0,15$ von $0,5$ abweicht, und unter diesen den höchsten Zielfunktionswert hat. Sei dies die Variable $\hat{x}$.

2. Handelt es sich bei $\hat{x}$ um eine zu einer Originalkante korrespondierenden Variable, so wird diese auch als Branching-Variable übernommen. Falls nicht, wird das Intervall um den Wert $0,5$ herum um einen Wert x_b vergrößert.

3. Es wird nun überprüft, ob es zu Originalkanten korrespondierende fraktionale Variablen gibt, deren LP-Wert im Intervall $[0,5 - x_b; 0,5 + x_b]$ liegt.

4. Falls ja, wird eine derjenigen Variablen als Branching-Variable ausgewählt, deren LP-Wert am dichtesten an $0,5$ liegt.

5. Falls nicht, werden diese Kandidaten verworfen und die ursprünglich bestimmte Variable $\hat{x}$ als Branching-Variable genommen. In diesem Fall handelt es sich dann um eine zu einer Zusammenhangskante korrespondierenden Variable.

Der Wert x_b lässt sich frei einstellen, sodass experimentelle Untersuchungen anhand verschiedener Werte möglich sind (siehe Abschnitt 5.3.1).

4.7 Enumeration

Die Auswahl des nächsten zu bearbeitenden, offenen Subproblems im aktuellen Branch-and-Bound Baum erfolgt mittels einer sogenannten *Enumerations-Strategie bzw. -Regel*. Vier Standard-Enumerations-Regeln sind in ABACUS implementiert:

- `BFS`. Der Branch-and-Bound Baum wird mittels Breitensuche durchlaufen. Offene Subprobleme werden in dieser Reihenfolge bearbeitet.
- `DFS` Analog wird der Branch-and-Bound Baum mittels Tiefensuche durchlaufen.
- `DiveAndBest`. Sucht zunächst in die Tiefe, bis eine zulässige Lösung gefunden wurde, und sucht dann mittels `BestFirstSearch` weiter.
- `BestFirstSearch`. Es wird als nächstes immer dasjenige offene Subproblem betrachtet, dessen LP-Relaxierung den besten Zielfunktionswert erzielt hat.

In dem implementierten Branch-and-Cut Algorithmus wird die Enumerations-Strategie `BestFirstSearch` verwendet.

Kapitel 5

Experimentelle Analyse

In diesem Kapitel werden die durchgeführten Experimente beschrieben und deren Ergebnisse analysiert. In Abschnitt 5.1 wird zunächst beschrieben und motiviert, woraufhin der Algorithmus untersucht werden soll. Zur Durchführung einer experimentellen Analyse eines Algorithmus benötigt man ein Benchmark-Set von Eingabeinstanzen. Die zu diesem Zweck konstruierten Benchmark-Instanzen werden in Abschnitt 5.2 vorgestellt. Im darauf folgenden Abschnitt 5.3 wird als erstes auf die einstellbaren Parameter des Algorithmus eingegangen und einige Testreihen durchgeführt, die Aufschluss über „gute“ und „schlechte“ Einstellungen geben sollen. Es folgen verschiedene Experimente anhand konkreter Fragestellungen und spezieller Klassen von Clustergraphen. Diese werden in den Abschnitten 5.4 bis 5.8 beschrieben. Eine Einschätzung bezüglich der Praxistauglichkeit wird in Abschnitt 5.9 gegeben. Der letzte Abschnitt 5.10 diskutiert mögliche weitere interessante Experimente, die in folgenden Arbeiten durchgeführt werden könnten.

5.1 Ziele der Analyse

Eines der Ziele der experimentellen Analyse ist es, den Algorithmus (und damit auch das entworfene Modell) auf Praxistauglichkeit zu überprüfen. Da es sich bei dem MCPSP um ein algorithmisch schwieriges Problem handelt, ist davon auszugehen, dass die Performance des Algorithmus ab einer gewissen Eingabegröße und in Bezug auf „schwierigen“ Instanzen inpraktibal wird oder diese nicht mehr gelöst werden können. In diesem Zusammenhang wird auch untersucht, ob sich bestimmte strukturelle Eigenschaften von Clustergraphen herausstellen lassen, die für den Algorithmus besonders schwierig sind. Dazu wird der Algorithmus anhand verschiedener Graphklassen untersucht, die zu diesem Zweck konstruiert wurden. Hierbei wird insbesondere auch daraufhin abgezielt zu ergründen, warum die Optimierung auf bestimmten Clustergraphen schwierig ist, was also die Schwierigkeit ausmacht. Im vorangegangenen Kapitel wurde bereits an mehreren Stellen angemerkt, dass der Algorithmus parametrisiert ist, sodass verschiedene Einstellungen, die jeweils bestimmte Bereiche der Optimierung betreffen, getestet werden können.

Die Experimente wurden alle auf einem 2.4GHz Opteron Rechner mit 4 CPU-Kernen und 32 GByte RAM durchgeführt.

5.2 Benchmark-Set

Im Folgenden werden die Graphklassen vorgestellt, die das zugrunde liegende Benchmark-Set bilden. Es erfolgt zunächst eine kurze Übersicht über die einzelnen Graphklassen, die dann anschließend detaillierter beschrieben werden.

5.2.1 Graphklassen

- **Rome-Graphen**
  - Die Graphen aus der Rome-Library [BGLT97] ohne Clusterstruktur (bzw. nur mit root-Cluster). Diese sind entsprechend ihrer Größe in zwei Gruppen eingeteilt: Rome-Graphen bis maximal 25 Knoten (`RootClusterRome`) und Rome-Graphen bis zu 50 Knoten (`RootClusterRomeLarge`).
  - Rome-Graphen bis Größe 25, bei denen die Clusterstruktur randomisiert mittels BFS erzeugt sind (`MaxDepth`).
  - Rome-Graphen bis Größe 25 mit vollständig randomisierter Clusterstruktur (`RandomDepth`).
- **Kreise in Clusterkreisen.** Clustergraphen wie in [CBPP04] beschrieben (`ClusterCycles`).
- **Gittergraphen** mit verschiedenen Clusterstrukturen (`Grids`).
- **Wheel-Graphen.** Spezielle Klasse zum Testen des Algorithmus bei zunehmender Anzahl von Zusammenhangskomponenten in einem Cluster (`Wheel`).
- **Vollständige Graphen.** Spezielle Klasse zum Testen des Algorithmus bei steigender Nicht-Planarität (`KGraphs`).

Rome-Graphen

Die vier Benchmark-Klassen, die anhand der Rome-Graphen konstruiert wurden, dienen dazu, einen möglichst allgemeinen Eindruck bezüglich der Praxistauglichkeit des Algorithmus zu erhalten, da die Graphen aus der Rome-Library aus praktischen Anwendungen stammen. Die Rome-Library hat sich bezüglich experimenteller Untersuchungen von Graphen-Algorithmen als eine Art *Quasi-Standard* entwickelt. Für unsere Zwecke werden allerdings Clustergraphen benötigt. Durch die randomisierte Erzeugung von Clusterstrukturen über diesen Graphen sind die so gebildeten Benchmark-Sets zumindest bedingt geeignet, um Aussagen bezüglich der Praxistauglichkeit des Algorithmus treffen zu können. Die Graphen der beiden Klassen (`RootClusterRome`) und (`RootClusterRomeLarge`) besitzen wie erwähnt keine Clusterstruktur, sodass der Algorithmus bezüglich dieser Graphen im Prinzip das MPSP löst.

Durch die randomisierten Clusterstrukturen über den Rome-Graphen in den Klassen `MaxDepth` und `RandomDepth` besitzen die erzeugten Clustergraphen unterschiedliche Eigenschaften. Bei der Klasse `RandomDepth` sind die Cluster vollkommen randomisiert erzeugt, in der Klasse `MaxDetph` wurde bei der Erzeugung der Clusterstrukturen jeweils eine Maximaltiefe vorgegeben, die allerdings nicht zwangsläufig auch erreicht werden muss. Die

höchste auftretende Clustertiefe in diesen beiden Klassen ist 4, die maximale Anzahl an Clustern 6.

Gittergraphen

Die Benchmark-Klasse `Grids` umfasst eine Reihe von Graphen unterschiedlicher Größe, die eine einfache „Gitterstruktur“ besitzen. Auf diesen Graphen sind verschiedene Clusterstrukturen definiert, sodass sich die resultierenden Grid-Clustergraphen in fünf strukturell verschiedene Klassen einteilen lassen. Die einzelnen Unterklassen, die sich durch die speziellen Clusterstrukturen ergeben, sind nachfolgend aufgelistet. Es wird jeweils noch kurz auf deren Eigenschaften eingegangen. Abbildung 5.1 zeigt graphische Beispiele zu den jeweiligen Klassen. Die erzeugten Graphen der Klassen `BFS`, `Columns` und `ColumnGaps` liegen in einer Größenordnung von 4×4 bis 10×10. Die `Narrow`-Gittergraphen reichen von 2×6 bis 4×10.

- **BFS.** Die Gittergraphen werden mittels Breitensuche durchlaufen. Für die Knoten jeweils eines Levels der Breitensuche wird ein Cluster erzeugt. Dadurch ergibt sich eine „diagonale“ Clusterstruktur, sodass die jeweils Cluster-induzierten Subgraphen keine Kanten enthalten (siehe Abbildung 5.1 (a)). Bei den so erzeugten Graphen handelt es sich also um nicht-C-zusammenhängende, deren Komplementgraphen ebenfalls nicht zusammenhängend sind.

- **Column.** Die Cluster werden „spaltenweise“ erzeugt, sodass sich dadurch eine Clusterstruktur wie in Abbildung 5.1 (b) ergibt. Die Cluster-induzierten Graphen dieser Graphklasse sind somit alle zusammenhängend, allerdings besteht kein vollständiger Zusammenhang.

- **ColumnGaps.** Diese Klasse entspricht den `Column`-Grids. Allerdings werden hierbei randomisiert „Lücken“ in die einzelnen Cluster eingefügt. Damit ist gemeint, dass manche Knoten einer Cluster-Spalte in den root-Cluster verlegt werden. Dadurch entsteht ein gewisser Grad an Nicht-C-Zusammenhang innerhalb der Cluster. Abbildung 5.1 (c) veranschaulicht dies. Dabei ist zu beachten, dass es sich bei den Clustern jeweils derselben Spalte des Graphen auch nach wie vor um **einen** Cluster handelt, auch wenn dies aus der Zeichnung (zum Zwecke des Erhalts der Gitterstruktur des zugrunde liegenden Graphen) nicht hervorgeht. Die Lückengrößen reichen von 1 bis 3.

- **Narrow.** Narrow-Gittergraphen haben im Prinzip die gleiche Struktur wie die `Column`- und `ColumnGaps`-Graphen. Auch hier sind die Cluster spaltenweise erzeugt. Der Unterschied besteht im Verhältnis von Spaltenanzahl zu Zeilenanzahl. Hierbei ist die Anzahl der Spalten im Vergleich zu einem Graphen gleicher Größe der Klasse `Columns` geringer, die Anzahl der Knoten pro Spalte dafür größer, sodass sich eine „spitzere“ Form ergibt. Abbildung 5.1 (d) zeigt dies. Auch bei dieser Klasse sind auf einem Teil der Graphen randomisiert Lücken in die Cluster eingefügt worden.

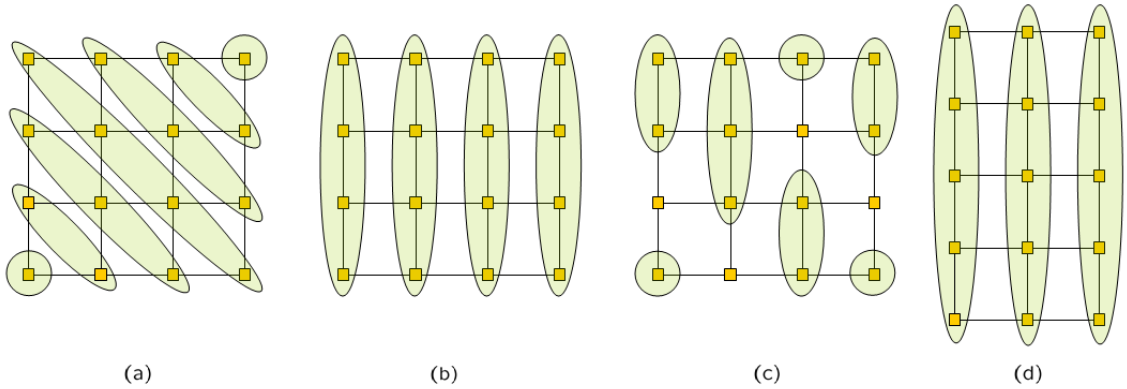


Abbildung 5.1: Benchmark-Klasse Grids

Kreise in Clusterkreisen

In [CBPP04] stellen Cortese et. al. eine spezielle Klasse von Clustergraphen vor. (siehe Abschnitt 2.3.3). Graphen, die in diese Klasse fallen, können je nach Clusterstruktur hochgradig nicht-C-zusammenhängend als auch C-zusammenhängend sein. Zur Untersuchung, wie sich der Branch-and-Cut Algorithmus in Bezug auf diese Graphklasse verhält, wurden diese mit in das Benchmark-Set aufgenommen. In Abbildung 5.2 ist repräsentativ ein Graph dieser Klasse dargestellt. (a) zeigt, dass der zugrunde liegende Graph ein Kreis ist. Die für das Benchmark-Set erzeugten Clustergraphen variieren in der Anzahl der Knoten (also in der Größe des Kreises) und der Anzahl der Cluster, wobei die Cluster randomisiert gebildet wurden. Dabei musste darauf geachtet werden, dass die Cluster ebenfalls einen Kreis bilden, wenn man die jeweils Cluster-induzierten Graphen zu einem Knoten kollabiert (siehe b). Die Clustertiefe ist durchgängig 1 (die theoretischen Aussagen in [CBPP04] beziehen sich allerdings auch auf Clustergraphen mit tieferer Clusterstruktur).

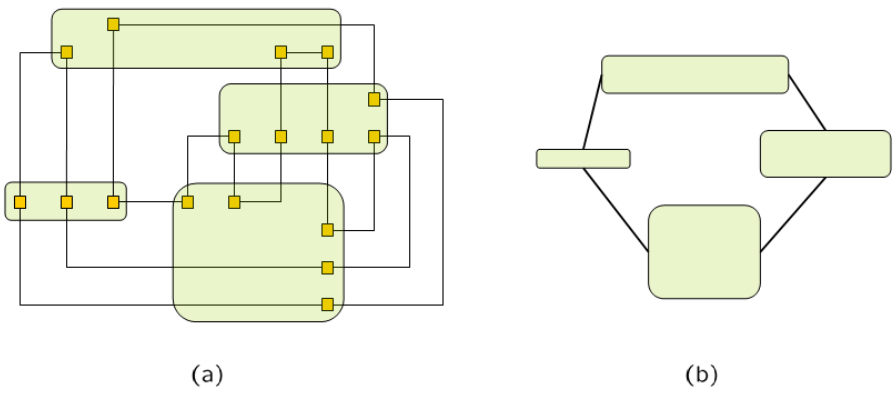


Abbildung 5.2: Benchmark-Klasse ClusterCycles

5.3 Parameter-Test

Der implementierte Branch-and-Cut Algorithmus ist an einigen Stellen parametrisiert, sodass dieser anhand verschiedener Einstellungen, die jeweils bestimmte Bereiche der Optimierung beeinflussen, untersucht werden kann. Dies betrifft unter anderem Einstellungen zum Variieren der Randomisierungsstrategie der primalen Heuristik, der Art und Weise wie Kuratowski-Constraints separiert werden und die Auswahl einer Branching-Variable. Welche Parameter-Einstellungen möglicherweise besser sind als andere, lässt sich zum Teil begründet vermuten. Dies muss jedoch anhand einer experimentellen Untersuchung verifiziert werden. Die in diesem Abschnitt vorgestellten Experimente dienen genau diesem Zweck. Aufgrund der hohen Anzahl möglicher Einstellungs-Kombinationen wird sich bei den einzelnen Experimenten jeweils auf bestimmte Bereiche der Optimierung beschränkt. Die Suche nach guten Parameter-Einstellungen kann daher nur bedingt durchgeführt werden, da diese separat und unter Konstanthaltung der jeweils übrigen Parameter getestet werden. Es sollte jedoch gelingen können, besonders ungünstige Einstellungen zu identifizieren, sodass zumindest davon ausgegangen werden kann, in Bezug auf die Peformance-Tests keine allgemein schlechten Einstellungen benutzt zu haben.

Damit die Testreihen vernünftig planbar und von ihrer zeitlichen Dauer zumindest annähernd abschätzbar sind, wurde für die durchgeführten Experimente ein zeitliches Abbruch-Kriterium von 25 Minuten vorgesehen. Bei Instanzen, deren Optimallösung nach 25 Minuten noch nicht durch den Algorithums gefunden werden konnte, wird die Optimierung dann beendet. Zeitliche Abbruch-Kriterien sind natürlich schwer zu bestimmen und können durchaus „unfair“ sein. Eine Instanz, auf der die Optimierung nach 25 Minuten abgebrochen wird, könnte wenige Minuten später gelöst worden sein. Um jedoch insbesondere die Tests, bei denen eine große Fülle von Graphen betrachtet wird (wie zum Beispiel bei den in diesem Abschnitt betrachteten Parameter-Tests) durchführbar zu gestalten, ist ein Abbruch-Kriterium nötig. Ziel der Experimente ist es außerdem nicht, jede gerechnete Instanz optimal zu lösen, sondern den implementierten Algorithmus für das Modell in Bezug auf sein Verhalten zu analysieren und Informationen über Rechenzeit-Verläufe, schwierige Instanzen und die Praxistauglichkeit zu erhalten. Bei machen Experimenten wurde dieses Abbruch-Kriterium allerdings auch nicht verwendet. An entsprechender Stelle wird dies jedoch explizit erwähnt.

5.3.1 Einstellbare Parameter

Tabelle 5.3.1 listet die „von außen“ einstellbaren Parameter des Branch-and-Cut Algorithmus auf und gibt eine kurze Beschreibung, wie diese jeweils den Optimierungsprozess beeinflussen.

Die allgemeinen Tests der Parameter-Einstellungen wurden anhand der Benchmarkklasse `MaxDepth` durchgeführt. Diese beinhaltet insgesamt 6117 Clustergraphen. Durch die randomisierten Clusterstrukturen besitzen die erzeugten Clustergraphen außerdem unterschiedliche Eigenschaften, wobei die Anzahl an planaren Graphen jedoch dominiert.

Nummer	Wertebereich	Beschreibung
1	$\mathbb{N}^+$	Die maximale Anzahl der wiederholten Heuristik-Aufrufe pro Iteration.
2	$[1, \ldots, 10]$	Die Anzahl und damit die Intervallgrößen der erzeugten Permutationslisten in der Heuristik, innerhalb derer die nach ihrem aktuellen LP-Wert zugeordneten Kanten permutiert werden (siehe Abschnitt 4.4).
3	$\mathbb{N}^+$	Die maximale Anzahl an Aufrufen des Planaritätstests zur Extraktion verletzter Kuratowski-Subdivisions (Abschnitt 4.3.2).
4	$\mathbb{N}^+ \cup -1$	Die maximale Anzahl an Kuratowski-Subdivisions, die in einer Separations-Iteration extrahiert werden. -1 bedeutet, es werden so viele „wie möglich“ extrahiert.
5	$\mathbb{N}^+$	Begrenzt die maximale Anzahl an wiederholten Supportgraph-Berechnungen pro Separations-Iteration, anhand derer nach verletzten Kuratowski-Subdivisions gesucht wird (siehe Abschnitt 4.3.2).
6	$[0,0; 1,0]$	Die obere Schranke, bis zu welchem LP-Wert Kanten noch deterministisch zum Supportgraphen bei der Kuratowski-Extraktion hinzugefügt werden.
7	$[0,0; 1,0]$	Die untere Schranke, bis zu welchem LP-Wert Kanten deterministisch *nicht* zum Supportgraphen hinzugefügt werden.
8	$\{true, false\}$	Ein Flag, das indiziert, ob die Variablen für die Zusammenhangskanten perturbiert werden sollen, oder nicht (siehe Abschnitt 4.5.2).
9	$[0,0; 0,5]$	Der Wert, der das Intervall um $0,5$ herum definiert, innerhalb dessen bevorzugt Originalkanten als Branching-Variable genommen werden (siehe Abschnitt 4.6).

Tabelle 5.1: Einstellbare Parameter des Branch-and-Cut Algorithmus

Kuratowski-Constraints Separierung

Der Algorithmus bietet Freiraum in Bezug auf die Art und Weise, wie die Extraktion der Kuratowski-Constraints ablaufen soll. Anhand der Parameter 3-7 kann diese manipuliert werden. Aufgrund der Vielzahl an Einstellungsmöglichkeiten wurden zwei separate Testreihen durchgeführt. Im ersten Experiment beschränken wir uns auf die Parameter 3-5, also die maximale Anzahl von Planaritätstest-Aufrufen, die maximale Anzahl berechneter Supportgraphen und die maximale Anzahl extrahierter Kuratowski-Constraints pro Separations-Iteration. Alle anderen Parameter sind konstant. Insgesamt wurden 12 Läufe durchgeführt, die sich aus unterschiedlichen Kombinationen der drei variablen Parameter ergeben. Die Tabelle unten listet diese kompakt auf.

Parameter-Nr.	1	2	3	4	5	6	7	8	9
Wertebereich	2	5	$\{1;3\}$	$\{5;10;20\}$	$\{1;3\}$	0,7	0,3	$false$	0,3

Abbildung 5.3 zeigt die Ergebnisse der Läufe. Die getesteten Graphen wurden dazu jeweils nach ihren Eigenschaften eingeteilt (c-zusammenhängend, vollständig zusammenhängend, planar, C-planar und nicht-planar). In jedem der vier Diagramme sind die durchschnittlichen Laufzeiten (y-Achse) jeder Eigenschaftsklasse (x-Achse) aufgetragen. Man beachte dabei, dass die Klassen nicht disjunkt sind. C-planare Clustergraphen sind automatisch auch planar. Die Graphen können außerdem jeweils C-zusammenhängend und vollständig zusammenhängend sein. Jedes Diagramm stellt jeweils drei Läufe gegenüber, bei denen die maximale Anzahl der pro Iteration separierten Kuratowski-Subdivisions variieren, wobei die anderen beiden Parameter jeweils konstant sind. *c-con* bezeichnet den C-Zusammenhang, *cc-con* den vollständigen Zusammenhang. Die Tripel (x, x, x) = (Planaritätstest-Aufrufe,Subdivisions,Supportgraphen) in der Legende kennzeichnen die jeweils zugrunde liegenden Parameter-Einstellungen.

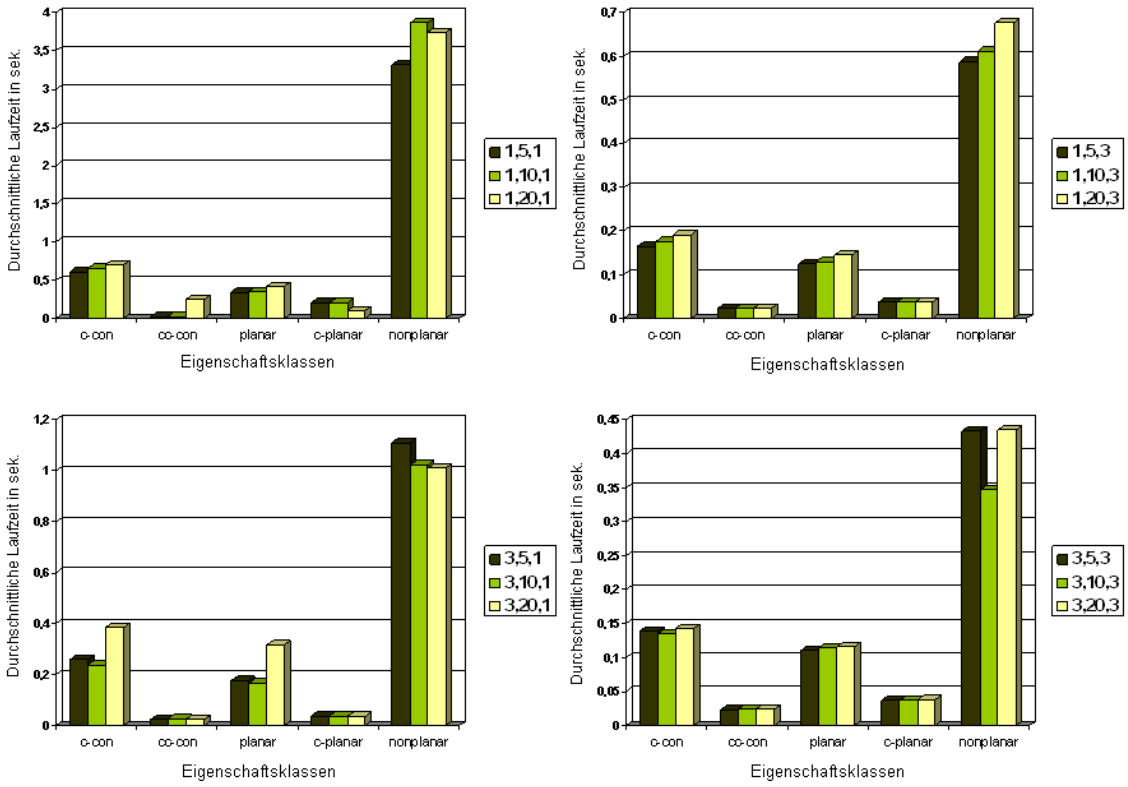


Abbildung 5.3: Unterschiedliche Einstellungen bezüglich der Kuratowski-Separierung

Aus den Diagrammen ist zu erkennen, dass eine mehrmalige (im durchgeführten Experiment maximal dreimalige) Wiederholung der Supportgraph-Berechnung im Durchschnitt zu kürzeren Laufzeiten führt. Vergleicht man zum Beispiel die Durchschnittslaufzeiten der beiden oberen Diagramme, so ergibt sich für die Klasse der nicht-planaren Clustergraphen eine fast um den Faktor 6 kürzere Durchschnittslaufzeit. Dies muss jedoch mit Vorsicht betrachtet werden. Die meisten Graphen dieser Benchmark-Klasse werden vom Branch-and-Cut Algorithmus unabhängig von den gesetzten Parametern innerhalb einer zehntel Sekunde gelöst (siehe auch Abschnitt 5.9). Die Unterschiede in den durchschnittlichen Laufzeiten werden demnach in erster Linie durch, im Verhältnis zur Gesamtmenge, weni-

ge Clustergraphen verursacht, auf denen der Algorithmus länger rechnet. Tabelle 5.2 stellt die tatsächlichen Laufzeiten in Sekunden für einige der „schwierigeren“ Clustergraphen gegenüber.

Einstellung	(1/10/1)	(1/10/3)	(3/10/1)	(3/10/3)
	104	28,84	58,25	1,8
	14,93	14,21	14,5	12,61
	15,62	9,85	10,72	2,12
	20,74	2,65	0,12	0,13
	182,52	33,81	45,51	16,42
	102,72	0,13	91,66	0,14
	1215,77	77,95	133,88	33,57
	61,5	16,06	69,69	11,12
	158,61	111,12	138,08	69,47
	184,77	55,24	33,9	5,4
	42,45	17,82	12,25	6,17
	63,78	11,55	34,01	11,33
	14,98	16,12	16,13	13,66
	885,03	0,03	0,09	0,05
	258,05	208,57	239,13	207,41

Tabelle 5.2: Absolute Laufzeiten schwieriger Clustergraphen der Benchmark-Klasse Max-Depth in Bezug auf unterschiedliche Einstellungen zur Kuratowski-Separation

Das Ergebnis entspricht in etwa dem, was in Abschnitt 4.3.2 bereits vermutet und motiviert wurde. Die beiden Parameter-Einstellungen (1/10/3) und (3/10/3) erzielen insgesamt die „besten“ Ergebnisse. Durch die wiederholte Berechnung der Supportgraphen konnten offensichtlich verletzte Constraints gefunden werden, die bei nur einmaliger Berechnung eines Supportgraphen nicht gefunden wurden. Dies resultiert in besseren LP-Relaxierungen. Die Einstellung (3/10/3) ist im Durchschnitt noch etwas besser als (1/10/3). Wiederholtes Aufrufen des BoyerMyrvold-Planaritätstest zur Extraktion von Kuratowski-Subdivisions wirkt sich also ebenfalls positiv auf die Performance aus. Durch die randomisierten DFS-Durchläufe bei jedem Aufruf des Planaritätstest variiert jeweils die Mengen der extrahierten Subdivisions, sodass durch mehrmaliges Durchlaufen des Graphen offensichtlich verletzte Subdivisions gefunden werden können, die bei nur einmaligem Durchlaufen nicht gefunden werden.

In Bezug auf die Anzahl der maximal pro Separations-Iteration extrahierten Subdivisions scheint 5 oder 10 ein guter Wert zu sein. Aus Abbildung 5.3 geht hervor, dass die Läufe, in denen bis zu 20 Subdivisions extrahiert wurden, im Schnitt eine gering höhere Laufzeit haben. Weitere Kuratowski-Subdivisions scheinen also bei den betrachteten Graphgrößen keinen ersichtlichen Vorteil mehr zu bringen. Bei größeren Graphen könnte dies jedoch durchaus anders sein.

Aufgrund dieser Beobachtungen ist es noch interessant sich anzuschauen, ob eine weitere Erhöhung der wiederholten Supportgraph-Berechnung auf den betrachteten Graphen zu noch höheren Rechenzeitverkürzungen führt. Wir betrachten dazu einen Lauf mit den Parameter-Einstellungen (3/10/6). Die Tabelle unten stellt die Durchschnittslaufzeiten in

Sekunden der Läufe (3/10/3) und (3/10/6) gegenüber.

Eigenschaft	c-con	cc-con	planar	non-planar	c-planar
Einstellung (3/10/3)	0,135	0,024	0,115	0,347	0,038
Einstellung (3/10/6)	0,152	0,023	0,132	0,365	0,038

Es zeigt sich, dass eine weitere Erhöhung der wiederholten Supportgraph-Berechnung bezüglich der betrachteten Graphen keinen ersichtlichen Vorteil mehr bringt. Es sind keine großen Unterschiede zu erkennen. Bei der Separierung der Kuratowski-Constraints werden mit der Einstellung (3/10/6) häufiger Supportgraphen berechnet. Diese scheinen jedoch keine neuen Gewinn bringenden Constraints zu liefern. Es entsteht daher zum Teil sogar ein leichter Rechenzeit-Overhead gegenüber der Einstellung (3/10/3). Die Parameter-Einstellung „maximal drei BoyerMyrvold-Planaritätstest Aufrufe, maximal 10 extrahierte Kuratowski-Subdivisions, maximal drei Supportgraph-Berechnungen“ pro Separations-Iteration wird daher für die weiteren Experimente verwendet (falls nicht explizit anders angegeben).

Supportgraphen-Erzeugung

Im Folgenden werden Läufe untersucht, bei denen die Parameter 6 und 7 variiert wurden. Diese definieren die Grenzwerte, anhand derer entschieden wird, ob eine Kante bei der Berechnung eines Supportgraphen deterministisch oder randomisiert gesetzt wird. Die Zugehörigkeit einer Kante zum Supportgraphen, wird dabei gemäß des aktuellen LP-Wertes der zur Kante korrespondierenden Variablen bestimmt (siehe Abschnitt 4.3.2). Die getesteten Einstellungen hierbei sind: $(0,8/0,2)$, $(0,8/0,4)$, $(0,8/0,6)$, $(0,7/0,3)$, $(0,7/0,5)$, $(0,6/0,2)$, $(0,6/0,4)$, sowie $(0,8/0,8)$, $(0,7/0,7)$, $(0,6/0,6)$, $(0,5/0,5)$. Bei den letzten vier Einstellungen werden Kanten also nur deterministisch gesetzt. Deshalb wurde für diese vier Tests die Anzahl der Supportgraph-Berechnungen auf 1 gesetzt. Mehrfaches Berechnen ist in diesen Fällen unnötiger Overhead, da sich die jeweiligen Supportgraphen nicht unterscheiden.

Wie in Abschnitt 4.3.2 schon motiviert, liegt die Vermutung nahe, dass grundsätzlich eine randomisierte Erzeugung der Supportgraphen von Vorteil ist. Ist der Wert einer Variablen nahe an $1,0$ $(0,0)$, so stellt dies ein begründetes Indiz dar, dass die entsprechende Kante auch (nicht) zu einer guten zulässigen Lösung gehört. Diejenigen Variablen jedoch, die anhand ihres LP-Wertes keinen Aufschluss bzw. kein Indiz über ihren möglichen Wert in einer guten zulässigen Lösung geben, sollten daher randomisiert gesetzt werden. Die naheliegende Vermutung ist also, dass die deterministischen Varianten $(0,8/0,8)$, $(0,7/0,7)$, $(0,6/0,6)$ und $(0,5/0,5)$ am schlechtesten abschneiden. In Bezug auf die randomisierten Varianten lässt sich nur schwer vorhersagen, welche Grenzwerte im Schnitt zu „guten“ Supportgraphen führen.

Abbildung 5.4 (a) stellt die durchschnittlichen Laufzeiten der randomisierten Varianten gegenüber. Es ist ersichtlich, dass besonders die Varianten, bei denen die untere Schranke relativ hoch liegt, wie bei $(0,7/0,5)$ und $(0,8/0,6)$, offensichtlich keine gute Wahl darstellen. Die Schranken $(0,8/0,2)$ und $(0,7/0,3)$ erzielen im Schnitt die besten Resultate. Die Ergebnisse der deterministischen Varianten sind in (b) dargestellt. Wie erwartet liegen die erzielten Durchschnittslaufzeiten hierbei höher, als bei den randomisierten Varianten. Bei

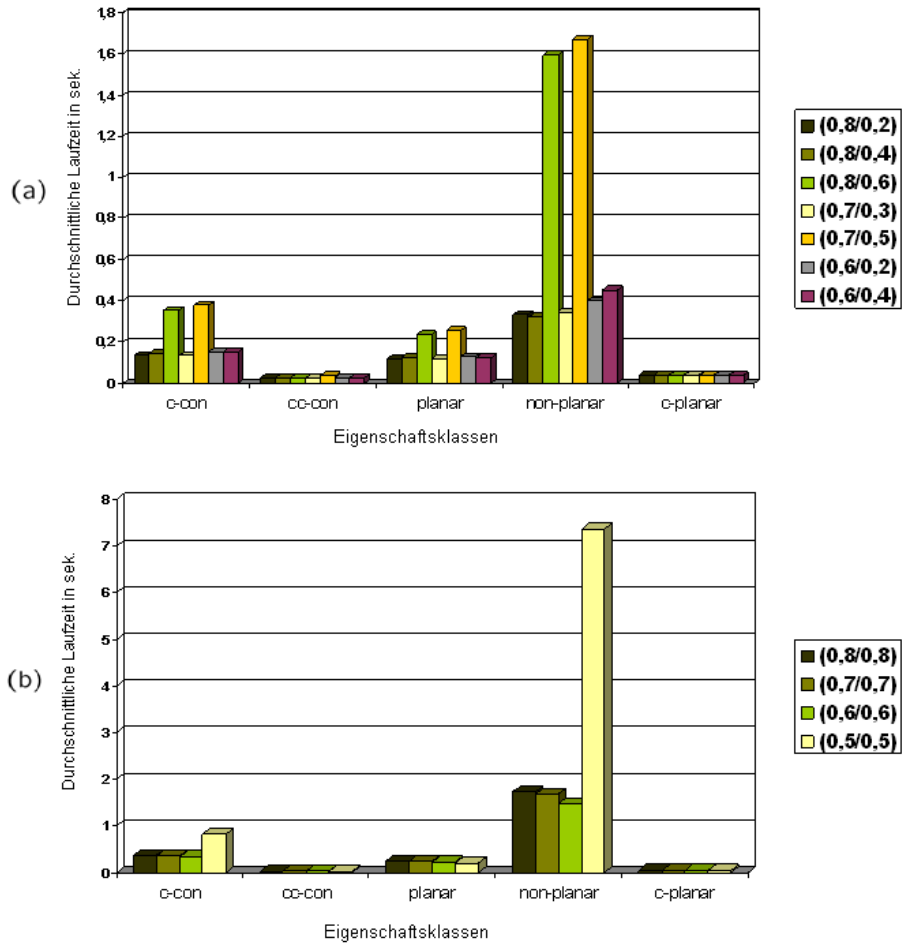


Abbildung 5.4: Durchschnittslaufzeiten bezüglich unterschiedlicher Grenzwerte beim deterministischen (Nicht-)Hinzufügen von Kanten bei der Supportgraph-Berechnung

der sehr naiven Einstellung $(0,5/0,5)$ konnte einer der nicht-planaren Graphen nicht innerhalb des zeitlichen Limits von 25 Minuten optimiert werden, was sich auch deutlich auf die Durchschnittslaufzeit niederschlägt. Die bereits beim Test der Kuratowski-Separation verwendeten Grenzwerte $0,7$ und $0,3$ werden weiterhin verwendet.

Unterschiedliche Einstellungen bezüglich der Heuristik

Die Heuristik kann im Algorithmus anhand der Parameter 1 und 2 beeinflusst werden. Diese definieren zum Einen, wie häufig die Heuristik maximal nacheinander aufgerufen wird, zum Anderen, wie groß die Intervalle der Permutations-Listen bei der Berechnung der Kanten-Reihenfolge sind (siehe Abschnitt 4.4). Die durchgeführten Läufe zielen darauf ab zu untersuchen, ob sich ein wiederholtes Aufrufen der Heuristik bezahlt macht, und welcher „Grad“ an Randomisierung zu guten Ergebnissen führt. Je weniger Permutations-Listen erzeugt werden, desto größer sind die entsprechenden Intervalle. Da die Kanten innerhalb einer Liste zufällig permutiert werden, und diese dann in der resultierenden Reihenfolge

getestet werden, ergibt sich durch wenige Listen eine stärkere Randomisierung.

Es wurden insgesamt neun Läufe durchgeführt, bei denen die Heuristik jeweils maximal ein-, zwei- oder dreimal aufgerufen wurde, und die Anzahl der Permutations-Listen dabei jeweils auf einen der Werte 3, 5 und 7 fixiert ist. Es ist zu vermuten, dass in manchen Fällen durch mehrmaliges Aufrufen der Heuristik Lösungen gefunden werden, die bei nur einmaligem Aufrufen aufgrund einer „ungünstigen“ Testreihenfolge nicht gefunden werden konnten.

Anhand der Graphen der Klasse `MaxDepth` sind keine nennenswerten Unterschiede bei der Wahl verschiedener Werte für die Anzahl der Permutations-Listen ersichtlich. Abbildung 5.5 stellt daher repräsentativ die Laufzeiten für diejenigen Graphen, auf der die Heuristik länger als 10 Sekunden zur Optimierung brauchte, bei der Verwendung von fünf Permutations-Listen in der Heuristik dar.

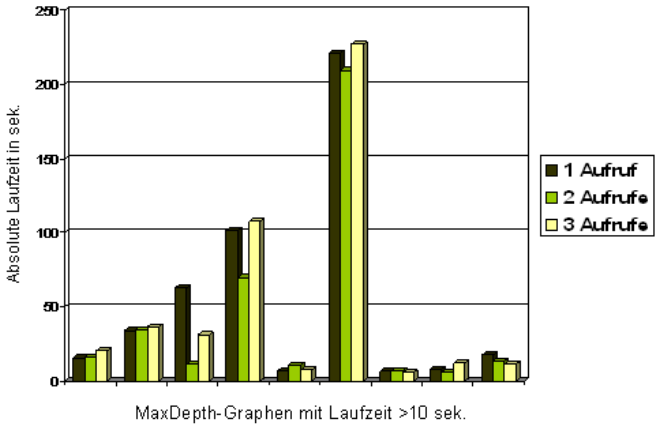


Abbildung 5.5: Heuristik-Test anhand der MaxDepth-Graphen

Ein absolut eindeutiger Trend lässt sich bezüglich der Anzahl wiederholter Heuristik-Aufrufe nicht feststellen. Mehrmaliges Aufrufen der Heuristik scheint jedoch in manchen Fällen durchaus gewinnbringend zu sein. Die Laufzeiten zeigen auch, dass bei den betrachteten Graphen erstaunlicherweise dreimaliges Aufrufen der Heuristik keinen erkenntlichen Vorteil gegenüber dem zweimaligen Aufrufen erzielt. Teilweise sind die Laufzeiten aufgrund des entstehenden Rechenzeit-Overheads sogar leicht höher.

Die erzielten Laufzeiten der Graphen der Benchmark-Klasse `Narrow` spiegeln ein ähnliches Bild wieder. Eine eindeutige Aussage, welche Einstellungen die besten Ergebnisse erzielen, ist nicht möglich. Die Laufzeitunterschiede bei unterschiedlicher Wahl der Anzahl der Permutations-Listen fällt hierbei jedoch etwas stärker aus. Abbildung 5.6 stellt diese bezüglich der „schwierigeren“ Graphen dieser Klasse dar.

Um genauere und aussagekräftigere Ergebnisse bezüglich der Heuristik-Einstellungen zu erzielen, sind offensichtlich intensivere Tests erforderlich, anhand zu diesem Zweck besser geeigneterer Benchmark-Sets. Im Rahmen dieser Diplomarbeit belassen wir es mit dem Test verschiedener Einstellungen für die Heuristik bei den gerade betrachteten, und verwenden die Einstellung „zwei Heuristik-Aufrufe“, da diese in Bezug auf die betrachteten Graphen der Klasse `MaxDepth` im Durchschnitt geringfügig die kürzesten Laufzeiten nach sich zieht (siehe Abbildung 5.5), und verwenden außerdem sieben Permutations-Listen.

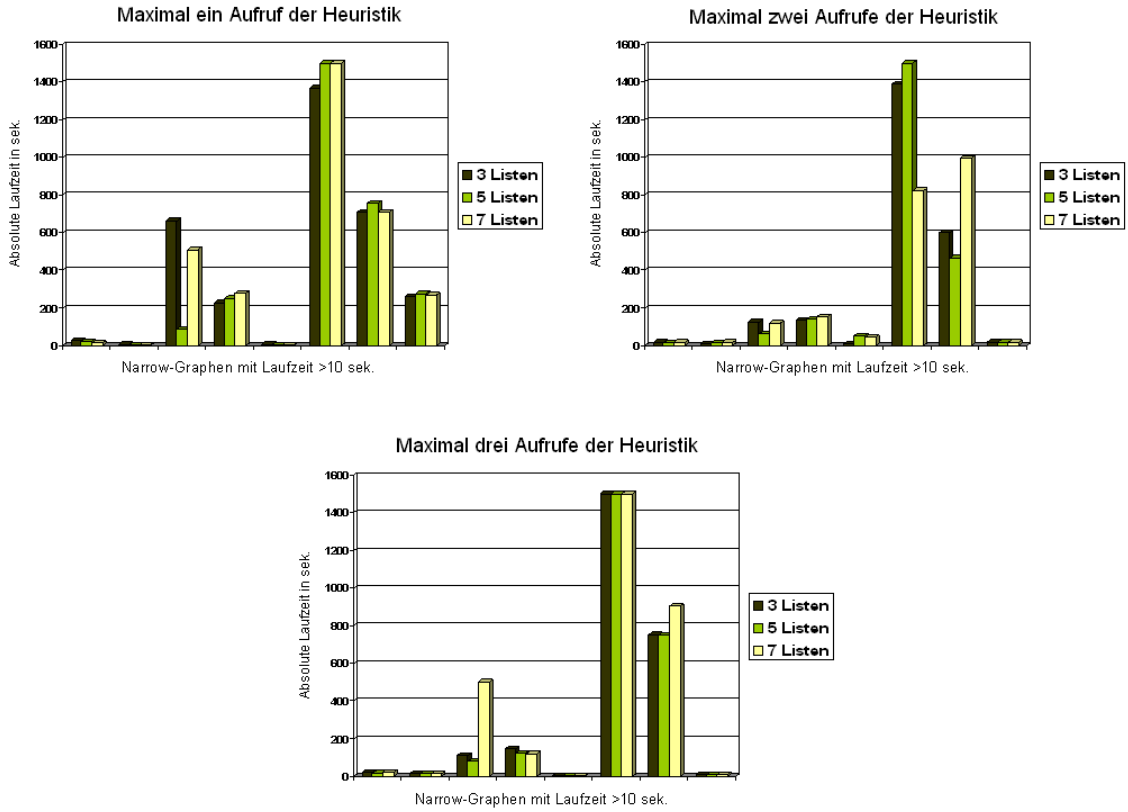


Abbildung 5.6: Heuristik-Test anhand der Narrow-Graphen

Testen der Branching-Strategie

Anhand des Parameters 9 lässt sich die verwendete Branching-Strategie beeinflussen. Es wurden Läufe durchgeführt, bei denen dieser Parameter jeweils auf einen der Werte $\{0,1; 0,2; 0,3; 0,4\}$ gesetzt wurde. Wie der Parameter die Branching-Strategie modifiziert, ist in Abschnitt 4.6 beschrieben. Grob gesagt gilt, je höher der Wert dieses Parameters ist, desto bevorzugter werden fraktionale Originalkanten gegenüber fraktionalen Zusammenhangskanten (bzw. deren Variablen) als Branching-Variable ausgewählt. Die zu Originalkanten korrespondierenden Variablen sind die „Kostenträger“ der Zielfunktion. Sie bestimmen im Wesentlichen den Zielfunktionswert, wohingegen die Zusammenhangskanten nur Mittel zum Zweck sind, und nur durch einen kleinen Wert ϵ gewichtet in die Zielfunktion eingehen. Da die betrachteten Graphen der Klasse `MaxDepth` im Durchschnitt sehr dünn sind (also relativ wenige Kanten enthalten), ist die Anzahl der Originalkanten im Verhältnis zur Anzahl der Zusammenhangskanten recht klein. Aus diesen Gründen liegt die Vermutung nahe, dass es vorteilhaft ist, zuerst vorzugsweise auf den Originalkanten zu branchen. Ob sich dies bestätigt zeigen die Ergebnisse der vier Läufe in Abbildung 5.7.

Für die meisten der betrachteten Graphen macht die zunehmende Bevorzugung der Originalkanten kaum einen Unterschied. Bezogen auf die nicht-planaren Graphen, zeigt sich jedoch die Tendenz, dass ein bevorzugtes Branching auf den Originalkanten von Vorteil ist.

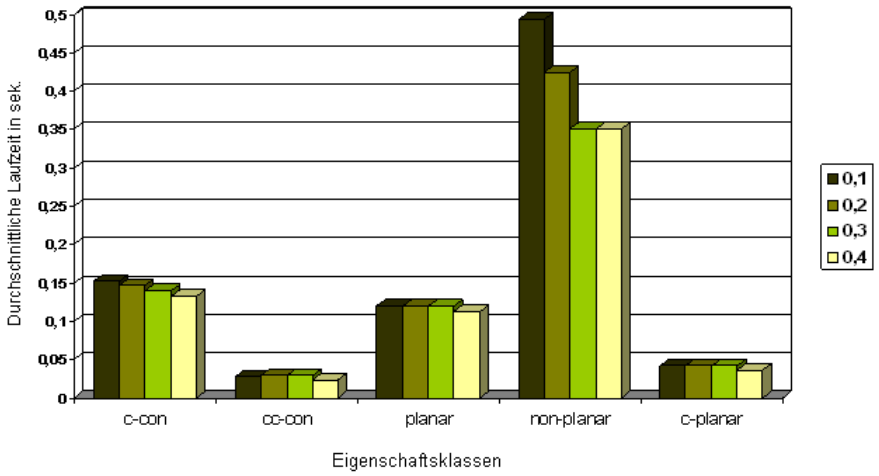


Abbildung 5.7: Test der Branching-Strategie

Die meisten der betrachteten nicht-planaren Graphen sind jedoch nur sehr „schwach nicht-planar“. Es müssen daher meist nur sehr wenige Kanten entfernt werden, um diese planar bzw. C-planar zu machen. Um zu untersuchen, wie sich diese Tendenz bei stark nicht-planaren Graphen entwickelt, ist das betrachtete Benchmark-Set nicht geeignet. Da diese Branching-Strategie, zumindest auf den betrachteten Graphen, allerdings auch keinen negativen Einfluss zu haben scheint, wird der Wert $0,4$ bei allen weiteren Tests beibehalten.

5.4 Untersuchung von Nicht-C-Zusammenhang

Dieser Abschnitt berschreibt eine Reihe von Experimenten, um nicht-C-zusammenhängende Clustergraphen näher zu untersuchen. Bis auf einige spezielle Klassen von Clustergraphen, wie zum Beispiel die in [CBPP04] beschriebenen, scheint der Nicht-C-Zusammenhang eines Graphen in Bezug auf die Entwicklung eines effizienten C-Planaritätstest eine besondere Schwierigkeit darzustellen. Aufgrund dessen wird diese Eigenschaft anhand des Branch-and-Cut Algorithmus intensiv untersucht. Interessante Fragestellungen diesbezüglich sind zum Beispiel:

- Wie entwickelt sich das Laufzeitverhalten des Branch-and-Cut Algorithmus bei zunehmendem Grad an Nicht-C-Zusammenhang der Clustergraphen?
- Sind bestimmte nicht-C-zusammenhängende Graphen einfacher zu lösen gegenüber anderen nicht-C-zusammenhängenden Clustergraphen, die dieselben Eigenschaften besitzen, sich aber bezüglich der Clusterstruktur oder der Struktur des zugrunde liegenden Graphen unterscheiden?
- In welchem anzahlmäßigen Verhältnis stehen die separierten Cut- und Kuratowski-Constraints zueinander?

5.4.1 Zunehmende Anzahl von Chunks in einem Cluster

Wir wollen das Verhalten des Algorithmus bei zunehmender Anzahl von Zusammenhangskomponenten in einem nicht-zusammenhängenden Cluster untersuchen. Grundsätzlich steigt die Anzahl potentieller Möglichkeiten, die einzelnen Chunks des Clusters durch zusätzliche Kanten zu verbinden, erheblich mit zunehmender Chunk- und Knotenanzahl des Clusters. Der genaue Optimierungsablauf wird dabei ebenfalls studiert, um möglichst ergründen zu können, worin genau die „Schwierigkeit“ liegt?

Dazu betrachten wir eine speziell zu diesem Zweck konstruierte Klasse von „Wheel“-Graphen, wie sie in Abbildung 5.8 (a) dargestellt sind. Die Graphen sind der Größe nach skaliert, sodass die Anzahl der Zusammenhangskomponenten des einzigen Clusters immer um 1 zunimmt. Die erzeugten Wheel-Clustergraphen besitzen außerdem alle dasselbe Verhältnis zwischen der Anzahl der Knoten plus Kanten des Graphen zu der Anzahl an neuen Zusammenhangskanten, die hinzugefügt werden müssen, um diesen C-zusammenhängend zu machen. Es handelt sich extra um eine sehr einfache Graphen- und Clusterstruktur, damit sich die Optimierung möglichst ausschließlich auf die Herstellung des C-Zusammenhangs „beschränkt“. Vereinfachend ist außerdem die Tatsache, dass die einzelnen Chunks des einzigen Clusters eines Graphen dieser Graphklasse jeweils nur aus einem einzigen Knoten bestehen.

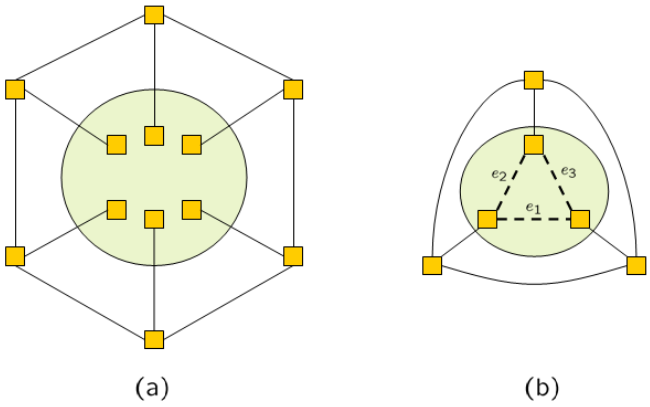


Abbildung 5.8: Wheel-Graphen zum Testen des Algorithmus bei zunehmendem Nicht-C-Zusammenhang. (a) zeigt die allgemeine Struktur dieser Graphen. (b) zeigt den Graphen W_2

Bei einer Graphgröße von bis zu 16 Knoten und 16 Kanten findet der Algorithmus eine optimale Lösung innerhalb einer zehntel Sekunde. Diese Graphgröße impliziert, dass sieben Kanten hinzugefügt werden müssen. Ab dann steigt die Optimierungszeit rapide an. Der Graph W_{11} mit einer Chunk-Anzahl von 11 und damit 10 hinzuzufügender Kanten, konnte nicht mehr innerhalb von zwei Stunden gelöst werden.

Wir betrachten nun das Verhalten der Optimierung etwas genauer. Die betrachteten Clustergraphen sind alle C-planar. Es müssen also keine Originalkanten gelöscht werden. Der Algorithmus beschäftigt sich demnach vollständig mit der Herstellung des C-Zusammenhangs.

- **Wie verhält sich der Algorithmus in Bezug auf die Herstellung des C-Zusammenhangs?** Mit zunehmender Anzahl von Chunks steigt die Anzahl der Auswahlmöglichkeiten von Zusammenhangskanten, um die Chunks zu verbinden. Damit steigt allerdings auch die Anzahl unzulässiger Kantenkombinationen, also solche, durch deren Hinzufügen der resultierende Graph nicht-planar wird. Anders ausgedrückt, steigt die rein rechnerische Anzahl verschiedener Kombinationsmöglichkeiten von Zusammenhangskanten wesentlich stärker an als die Anzahl an zulässigen Kombinationen. Der Algorithmus hat jedoch keinerlei Informationen darüber, welche Kantenauswahl gut (resultierender Graph ist immer noch planar) und welche schlecht (resultierender Graph ist nicht mehr planar) ist. Schlecht gewählte Kanten zur Herstellung des Zusammenhangs resultieren in verletzten Kuratowski-Subdivisions. Diese werden also beim Schnittebenenverfahren extrahiert und zur entsprechenden LP-Relaxierung hinzugefügt. Der Algorithmus „versucht“ also die falsch gesetzten Kanten wieder rückgängig zu machen und andere zu wählen. Aufgrund der Tatsache, dass sowohl die Cut- als auch die Kuratowski-Constraints in Bezug auf die Relaxierung nur sehr schwach sind, stellt sich schnell die bereits in Abschnitt 4.3.4 geschilderte Situation ein, dass zwar keine Constraints mehr separiert werden können, die Lösung aber fraktional ist.

- **Warum findet die Heuristik nicht die optimale Lösung?** Das Vorgehen der Heuristik ist, dass zunächst ein Spannbaum T_s auf dem zugrunde liegenden Graphen berechnet wird, derart, dass jeder Cluster-induzierte Subgraph (bezüglich der Kanten aus T_s) C-zusammenhängend ist (siehe Abschnitt 4.4.1). Im Fall der betrachteten Wheel-Graphen besitzt der einzige Cluster-induzierte Graph (abgesehen vom root-Cluster) keine Originalkanten. Der Spannbaum muss und wird also vollständig aus Zusammenhangskanten aufgebaut. Dies geschieht anhand der aktuellen LP-Werte der Kanten. Die Heuristik tut dies zwar zu einem gewissen Grad randomisiert, jedoch spiegelt der berechnete Spannbaum im Wesentlichen die aktuelle Lösung wieder. Es werden also insbesondere meist auch solche Kanten zum Spannbaum hinzugefügt, die einen von $0,0$ verschiedenen Wert haben. War die „Kantenauswahl“ des Algorithmus schlecht (nicht-planare Strukturen), so erzeugt die Heuristik im Allgemeinen dennoch einen Spannbaum, der im Wesentlichen aus eben diesen Kanten besteht. Beim iterativen Hinzufügen weiterer Kanten in Phase 2 der Heuristik, kommt diese daher irgendwann an eine Stelle, an der sich die ungünstige Wahl der zur Spannbaum-Berechnung verwendeten Zusammenhangskanten dahingehend auswirkt, dass nicht alle Originalkanten hinzugefügt werden können, da dies sonst in einem nicht-C-planaren Graphen resultieren würde. Dadurch findet die Heuristik nicht (oder zumindest erst sehr spät) die optimale Lösung, in der alle Originalkanten enthalten sind.
  Es zeigt sich also, dass die implementierte Heuristik in Bezug auf stark nicht-C-zusammenhängende Clustergraphen schlecht performt.

Die Situation, dass die aktuelle LP-Lösung fraktional ist und keine Constraints mehr separiert werden können, stellt sich in Bezug auf die betrachteten Wheel-Graphen schon sehr früh ein. Bereits beim zweiten Wheel-Graphen W_2, dessen Cluster aus drei Chunks besteht (Abbildung 5.8 (b)), entsteht folgende Situation: insgesamt müssen zwei Kanten zum Graphen hinzugefügt werden. Es gibt genau 3 Cut-Constraints für diesen Fall: $e_1 + e_2 \geq 1$, $e_1 + e_3 \geq 1$ und $e_2 + e_3 \geq 1$ die schon zu Beginn der Optimierung im root-LP vorhanden

sind (siehe Abschnitt 4.3.3). Bereits hier berechnet der Algorithmus schon keine ganzzahlige Lösung mehr, denn anstatt zwei der Variablen jeweils auf $1, 0$ zu setzen, setzt er alle drei Kanten jeweils auf den Wert $0, 5$. Dadurch sind die drei Constraints ebenfalls erfüllt und die Zielfunktion wird dabei nur mit $1, 5\epsilon$ anstatt 2ϵ „belastet“. An dieser Stelle muss also schon gebrancht werden, was auch passiert. Die Heuristik findet jedoch sofort eine optimale Lösung, da, egal wie die Zusammenhangskanten zur Herstellung des C-Zusammenhangs bei der Spannbaum-Berechnung gewählt werden, aufgrund der kleinen Graphgröße keine nicht-planaren Strukturen entstehen können und somit in Phase 2 der Heuristik auch alle Originalkanten hinzugefügt werden können, ohne die C-Planarität zu verletzen. Je mehr Chunks ein Cluster jedoch enthält, desto mehr potentielle Zusammenhangskanten gibt es, wodurch der Algorithmus zunehmend mehr Möglichkeiten hat, die Kanten fraktional zu setzen, sodass keine Constraints verletzt sind.

Graphen	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8	W_9	W_{10}
Anzahl Knoten	4	6	8	10	12	14	16	18	20	22
Anzahl Chunks	2	3	4	5	6	7	8	9	10	11
Gelöste LPs	1	1	8	9	51	5	200	449	5272	47192
B&B-Tiefe	1	3	4	3	8	2	12	16	24	36
Anzahl K-Cons	0	0	0	0	4	0	3	17	685	22069
Anzahl C-Cons	1	3	7	10	27	10	103	239	3191	29866

Tabelle 5.3: Eckdaten der Optimierung bezüglich der Wheel-Graphen

Interessant ist es noch, sich zum Beispiel die Anzahl der erzeugten Constraints anzusehen. Tabelle 5.3 stellt diese zusammen mit einigen anderen Eckdaten gegenüber. *K-Cons* bezeichnet dabei die Anzahl der Kuratowski-Constraints, und *C-Cons* die Anzahl der Cut-Constraints. Bemerkenswert ist hierbei vor allem, dass die Anzahl an Kuratowski-Constraints bei steigender Graphgröße erheblich zunimmt. Beim letzten der gelösten Wheel-Graphen sind es bereits über 20000. Die Graphen sind C-planar, es müssen keine Originalkanten entfernt werden. Die Kuratowski-Constraints werden also bildlich gesprochen nur deshalb separiert, um die ungünstig gesetzten Zusammenhangskanten (durch deren Hinzufügen nicht-planare Strukturen enstanden sind) wieder herauszunehmen und statt dessen andere zu wählen. Der Mangel an Informationen oder Indizien für die Wahl „günstiger bzw. guter“ Zusammenhangskanten, sowie die Schwäche der Cut- und Kuratowski-Constraints in Bezug auf die LP-Relaxierung wird hier deutlich. Der Algorithmus ist zu großen Teilen allein damit beschäftigt, gemachte „Fehler“ wieder rückgängig zu machen. Ein Versuch, dem ein wenig entgegenzuwirken, wurde bereits in Abschnitt 4.5.2 motiviert und wird in Abschnitt 5.7 experimentell untersucht.

5.5 Untersuchung der Euler-Constraints

Ähnlich der Tests zur Untersuchung des Algorithmus auf stark nicht-C-zusammenhängenden Graphen im vorangegangenen Abschnitt, wollen wir auch das Verhalten des Algorithmus bei stark nicht-planaren Graphen untersuchen. Dazu betrachten wir zunächst wieder eine Reihe von konstruierten Graphen, die mit zunehmender Größe nicht-planar, aber alle vollständig zusammenhängend sind. Dadurch kann die Herstellung der Planarität

isoliert betrachtet werden, da sich der Algorithmus nicht zusätzlich mit der Herstellung von Zusammenhang beschäftigen muss. Wir betrachten zu diesem Zweck die vollständigen Graphen K_5 - K_{15}, bei denen die Clusterstruktur nur aus dem root-Cluster besteht. Bis hin zum K_{11} werden die Graphen noch innerhalb einer Sekunde gelöst. Ab dann steigt die benötigte Rechenzeit jedoch an. Der K_{15} benötigte ca. $3\frac{1}{2}$ Minuten zur Optimierung. Tabelle 5.4 stellt einige weitere Eckdaten zusammen.

Graphen	K_5	K_6	K_7	K_8	K_9	K_{10}	K_{11}	K_{12}	K_{13}	K_{14}	K_{15}
Gelöste LPs	2	3	2	8	31	43	195	895	2447	6453	49931
B&B-Tiefe	1	1	1	1	1	1	1	2	1	3	3
K-Cons	0	2	2	22	162	273	1768	8793	24229	64083	496148
Entf. Kanten	1	3	6	10	15	21	28	36	45	55	66

Tabelle 5.4: Kuratowski-Graphen mit Euler-Constraints

Die Tatsache, dass die Graphen bis zum K_{11} hin noch innerhalb einer Sekunde gelöst werden können, ist überraschend. Es handelt sich schließlich um vollständige Graphen, aus denen bei zunehmender Größe auch zunehmend mehr Kanten entfernt werden müssen. Die Information, dass Kanten entfernt werden müssen, bzw. einige zu Originalkanten korrespondierende Variablen den Wert $0,0$ erhalten müssen, bekommt der Algorithmus nur dadurch, dass entsprechend Kuratowski-Constraints separiert werden. Dass dies so schnell geschieht, ist erstaunlich. Die Ursache dafür sind jedoch die initial zum root-LP hinzugefügten *Euler-Constraints* (siehe Abschnitt 4.3.3). Diese schränken den maximal erreichbaren Zielfunktionswert von anfang an erheblich ein, da sie die maximale Anzahl an Originalkanten (im ganzzahligen Fall) auf $3|V| - 6$ beschränken. Für den K_6 zum Beispiel, der 15 Kanten besitzt, bedeutet dies konkret, dass der Zielfunktionswert des ersten LPs nur den Wert $6 * 3 - 6 = 12$ annehmen kann, anstatt 15. Wenn diese Constraints weggelassen werden, ändert sich die Performance des Algorithmus drastisch. Bereits der K_8 kann dann schon nicht mehr innerhalb von 25 Minuten gelöst werden. Bis zu diesem Zeitpunkt wurden bereits $3 * 10^6$ Kuratowski-Constraints separiert.

Es zeigt sich also, dass das initiale Hinzufügen dieser Constraints die Optimierung bei **sehr** dichten Graphen stark beschleunigen kann. Zu beachten ist natürlich, dass es sich bei den betrachteten vollständigen Graphen um absolute Extrem-Beispiele handelt. Die Euler-Constraints machen sich bei dünneren Graphen nicht bemerkbar. Bereits das Splitten weniger Kanten innerhalb sehr dichter, nicht-planarer Substrukturen bewirkt, dass die Euler-Grenze schon nicht mehr erreicht wird, und die Constraints damit hinfällig werden. Außerdem gilt, dass die Constraints nur bezüglich der jeweils Cluster-induzierten Graphen erzeugt werden, „Cluster-übergreifende“ nicht-planare Strukturen also in der Regel davon unberücksichtigt bleiben. Die Hinzunahme der Euler-Constraints beeinflusst die Optimierung jedoch auch nicht negativ, sodass diese durchaus standardmäßig hinzugenommen werden sollten, da sie unter Umständen bei dichten, stark nicht-planaren Graphen einen positiven Effekt haben können.

5.6 Gittergraphen und ClusterCycles

In diesem Abschnitt untersuchen wir den Algorithmus anhand der Benchmark-Klassen `Grids` und `ClusterCycles`. Unter diesen befinden sich auch eine Reihe teils stark nicht-C-zusammenhängender Clustergraphen, sodass auch ein Vergleich mit den bereits betrachteten Wheel-Graphen möglich ist. Diese beiden Klassen zeichnen sich außerdem dadurch aus, dass deren zugrunde liegende Graphen immer dieselbe Struktur haben, sich also jeweils nur durch die Größe und die definierte Clusterstruktur unterscheiden.

Als erstes betrachten wir die Gittergraphen untereinander. Die erzeugten Unterklassen unterscheiden sich in der Art, wie die Cluster auf den Gittergraphen gebildet wurden. Die zugrunde liegenden Graphen implizieren jeweils eine „Gitterstruktur“ (siehe Abbildung 5.1 und Abschnitt 5.2).

Grid-Klassen Column, ColumnGaps, Narrow und BFS

Die Graphen der Klasse `Columns` sind alle C-zusammenhängend und C-planar. Einzig der Zusammenhang der Komplementgraphen muss hergestellt werden. Aufgrund der C-Planarität dieser Graphen und insbesondere der Tatsache, das kein C-Zusammenhang hergestellt werden muss, lässt vermuten, dass diese verhältnismäßig schnell zu lösen sind. Die Graphen der Klasse `ColumnGaps` unterscheiden sich von diesen dahingehend, dass die Cluster „Lücken“ enthalten, das heißt, aus jedem Cluster wird eine gewisse Anzahl an Knoten in den root-Cluster verschoben (siehe Abschnitt 5.2.1 und Abbildung 5.1). Dadurch sind die Cluster nicht mehr zusammenhängend. Wir betrachten Lücken bis zu einer Größe von 2. Aufgrund der Größenordnung der betrachteten Graphen machen größere Lücken keinen Sinn (die Cluster des 4×4 Gittergraphen bestünden bei einer Lückengröße von 3 nur aus einem einzigen Knoten). Durch das Verschieben von Knoten in den root-Cluster ändern sich die Eigenschaften der Graphen noch weiter. Zum Einen sind nun zwar die Komplementgraphen alle zusammenhängend, zum Anderen geht ab einer gewissen Lückengröße allerdings die C-Planarität verloren. Die Cluster können nun nicht mehr wie in Abbildung 5.1 dargestellt „sequentiell“ angeordnet werden, ohne die Planarität zu verletzen.

Die Graphen der Klasse `Columns` konnten allesamt wie vermutet sehr schnell gelöst werden. Selbst der größte dieser Clustergraphen (10×10) mit 100 Knoten und 180 Kanten wurde vom Algorithmus in knapp zwei Sekunden gelöst. Die Tatsache, dass die Komplementgraphen nicht zusammenhängend sind, scheint also hier kein großes Problem darzustellen.

Die Graphen aus der Klasse `ColumnGaps`, bei denen die Lückengröße 1 beträgt (also jeweils 2 Chunks pro Cluster), sind nach wie vor alle C-planar und konnten ebenfalls in derselben zeitlichen Größenordnung gelöst werden. Anders dagegen die Graphen, bei denen die Lücken eine Größe von 2 haben. Die Graphen sind nun alle durchweg nicht mehr C-planar, es müssen also nun auch Originalkanten entfernt werden. Bis zum 6×6 Graphen konnten diese noch in relativ kurzer Zeit gelöst werden. Ab dem 6×7 Graphen (also ein Knoten pro Cluster mehr) explodiert die Laufzeit jedoch. Der 7×7 Graph konnte nicht mehr innerhalb von 2 Stunden gelöst werden. Tabelle 5.5 stellt die Eckdaten zusammen, die in diesem Zusammenhang interessant sind.

Graphen	G_{4x4}	G_{4x5}	G_{4x6}	G_{5x5}	G_{5x6}	G_{6x6}	G_{6x7}
Gelöste LPs	2	26	72	6	214	119	17020
B&B-Tiefe	1	3	6	1	22	12	34
K-Cons	6	118	481	41	1218	657	124543
C-Cons	3	10	10	6	15	15	778
Entf. Kanten	1	1	1	1	2	1	2
Hinzugefügte Kanten	3	4	7	5	6	7	12
Laufzeit in sek.	0,02	0,25	1,19	0,10	6,75	5,14	913,15

Tabelle 5.5: Ergebnisse: ColumnGaps-Graphen mit Lücken der Größe 2

Beachtlich ist hierbei unter anderem das Verhältnis von Kuratowski-Constraints zu Cut-Constraints. Obwohl nur verhältnismäßig wenige Kanten gelöscht werden müssen, ist die Anzahl separierter Kuratowski-Constraints sehr hoch. Einer der Gründe ist bereits anhand der Wheel-Graphen erkenntlich geworden (Abschnitt 5.4.1): „ungünstiges“ Hinzufügen von Kanten um den Graphen C-zusammenhängend zu machen, kann wieder darin resultieren, dass neue nicht-planare Substrukturen entstehen, die erst wieder durch Separation verletzter Kuratowsi-Constraints entfernt werden müssen.

Die `Narrow`-Gittergraphen besitzen im Gegensatz zu den beiden vorangegangenen Klassen eine „schmalere“ Gitterstruktur. Durch Einfügen von entsprechend großen Lücken werden die Graphen genau wie im Falle der `ColumnGaps`-Graphen nicht-C-planar. Vergleicht man diese nun mit den „eher quadratischen“ `ColumnGaps`-Graphen so zeigt sich, dass bei gleicher Lückengröße der Grad der Nicht-C-Planarität bei den `Narrow`-Graphen geringer ausfällt. Der Grund dafür ist die geringere Spalten- und damit Clusteranzahl. Bei einem Gittergraphen der Größe 6×6 werden bei einer Lückengröße von 2 zum Beispiel 12 Knoten in den root-Cluster verschoben, bei einem Gittergraphen der Größe 8×4 nur 8. Die Performance des Algorithmus bricht bei den diesen Clustergraphen daher noch nicht so früh ein, wie bei den `ColumnGaps`-Graphen. Fügt man jedoch Lücken der Größe 3 in die Cluster der `Narrow`-Graphen ein, so stellt sich auch hier schon früher der Performance-Einbruch des Algorithmus ein. Tabelle 5.6 stellt die Laufzeiten in Sekunden gegenüber. Ein „–“ in einer Spalte bedeutet, dass es in der entsprechenden Graphklasse keinen Graphen mit exakt dieser Größe gibt.

Graphgröße	24	25	27	28	30	32	36	40	42
Column Gaps 2	0,43	0,2	–	–	13,47	–	42,45	–	959,76
Narrow Gaps 2	0,96	–	0,48	5,14	2,24	6,15	22,71	46,95	–
Narrow Gaps 3	58,01	–	462,46	54,85	811,28	217,9	1318.22	>1500	–

Tabelle 5.6: Laufzeiten der ColumnGaps- und Narrow-Graphen

Die Gittergraphen der Klasse `BFS`, bei denen die Clusterstruktur mittels Breitensuche erstellt wurde und daher die Cluster „diagonal“ angeordnet sind (siehe Abbildung

5.1), sind durchweg C-planar, aber weder C-zusammenhängend noch Komplementgraph-zusammenhängend. Wohingegen der 5×6 BFS-Gittergraph noch in ca. 3 Minuten gelöst wurde, konnte der 6×6 Graph schon nicht mehr innerhalb einer Stunde gelöst werden. Die Schwierigkeit besteht hierbei offensichtlich in der „gleichzeitigen“ Herstellung des Zusammenhangs sowohl der Cluster-induzierten Graphen als auch der Komplementgraphen. Insbesondere steigt hierbei durch zunehmende Graphgröße auch der Grad an Nicht-C-Zusammenhang. Durch die dichte Gitterstruktur und die spezielle Anordnung der Cluster gibt es außerdem auch nur sehr wenige „zulässige“ Möglichkeiten für den Algorithmus, den Graphen C-zusammenhängend zu machen, da sehr viele Kombinationen von Zusammenhangskanten zu nicht-planaren Strukturen führen würden.

Die erzielten Laufzeiten der Graphen der Klasse `Grids` sind in Abschnitt 5.9 in Abbildung 5.11 zusammengefasst dargestellt.

Cycles in Cluster-Cycles

Die Performance des Algorithmus auf den in [CBPP04] beschriebenen ClusterCycle-Graphen verhält sich ähnlich, wie es bisher auch schon bei den vorangegangen Graph-Klassen beobachtet wurde. Die konstruierten Graphen dieser Klasse (siehe Abschnitt 5.2.1) sind aufgrund der Kreisstruktur alle Komplementgraph-zusammenhängend und teils C-zusammenhängend, teils hochgradig nicht-C-zusammenhängend, je nachdem wie die Clusterstruktur auf dem Graphen definiert ist (siehe Abbildung 5.2). Es wird auch hierbei wieder deutlich, dass die reine Größe der Graphen (bezogen auf die Anzahl der Knoten) keinerlei Aufschluss über die Schwierigkeit des Graphen gibt, sondern diese auch zu großen Teilen, wie bereits an vorangegangenen Experimenten gesehen, von der konkreten Clusterstruktur abhängt. So auch bei den Graphen dieser Klasse. Der zugrunde liegende Graph ist jeweils ein simpler Kreis. Die Performance des Algorithmus auf diesen Graphen reicht jedoch von wenigen Hundertstel Sekunden bis hin zu mehreren Stunden. Die Laufzeiten sind hierbei in erster Linie durch die Herstellung des C-Zusammenhangs dominiert. Die Verteilung der Laufzeiten bezüglich der Graphen dieser Klasse sind in Abschnitt 5.9 in Abbildung 5.11 dargestellt.

5.7 Auswirkungen der Kanten-Perturbation

In Abschnitt 4.5.2 wurde ein möglicher Ansatz vorgestellt, durch leichtes *perturbieren* der Zielfunktions-Koeffizienten der Variablen, diese für den Algorithmus unterscheidbar zu machen. Die Idee ist es, die Zielfunktions-Koeffizienten der zu Zusammenhangskanten korrespondierenden Variablen umso stärker zu perturbieren, je kleiner der graphentheoretische Abstand der zu der entsprechenden Kante inzidenten Knoten ist. Das heißt, je kleiner der Abstand, desto kleiner der Koeffizient der Variablen in der Zielfunktion, wodurch diese Variable geringfügig „billiger“ wird. Anhand der Benchmark-Klassen `Grids`, `Wheel` und `ClusterCycles`, die jeweils viele und teils hochgradig nicht-C-zusammenhängende Graphen beinhalten, wurden daher erneut Testläufe durchgeführt, um die Auswirkungen dieser Art der Koeffizienten-Perturbation zu untersuchen.

Das Ergebnis ist erstaunlich. Bei sehr vielen der schwierigeren Graphen dieser Klassen führt die Perturbation der Zusammenhangskanten zu teils erheblichen Laufzeitverbesse-

rungen. Alle Graphen der speziellen Klasse `Wheel` konnten innerhalb einer Sekunde gelöst werden, wohingegen die 4 größten Graphen dieser Klasse zuvor nicht innerhalb des Zeitlimits von 25 Minuten gelöst werden konnten. Die `Wheel`-Graphen sind allerdings sehr speziell konstruiert und erfüllen aufgrund ihrer Struktur exakt, dass Knoten mit kleinem graphentheoretischem Abstand auch potentiell besser sind. Bei den Graphen der anderen Klassen sind die Laufzeitverbesserungen jedoch in vielen Fällen ähnlich extrem. Die Ergebnisse sind in Abbildung 5.9 dargestellt. Bei den Gittergraphen der Klasse `BFS` sind die Laufzeiten stark von der Größe der Gittergraphen abhängig, da bei diesen die Clusterstruktur deterministisch erzeugt wurde und daher der Grad des Nicht-C-Zusammenhangs bei diesen Graphen stetig mit der Größe der Graphen zunimmt. Bei den Graphen der anderen Klassen ergibt sich ein „unregelmäßigeres“ Bild, da die Schwierigkeit der Instanzen durch die randomisierte Clusterstruktur nur gering von der reinen Größe der Graphen abhängt. Außerdem wurden in Bezug auf die Graphen der Klassen `Narrow`, `ColumnGaps` und `ClusterCycles` nur die Laufzeiten derjenigen Instanzen abgebildet, auf denen der Algorithmus länger als 10 Sekunden zur Lösung benötigte.

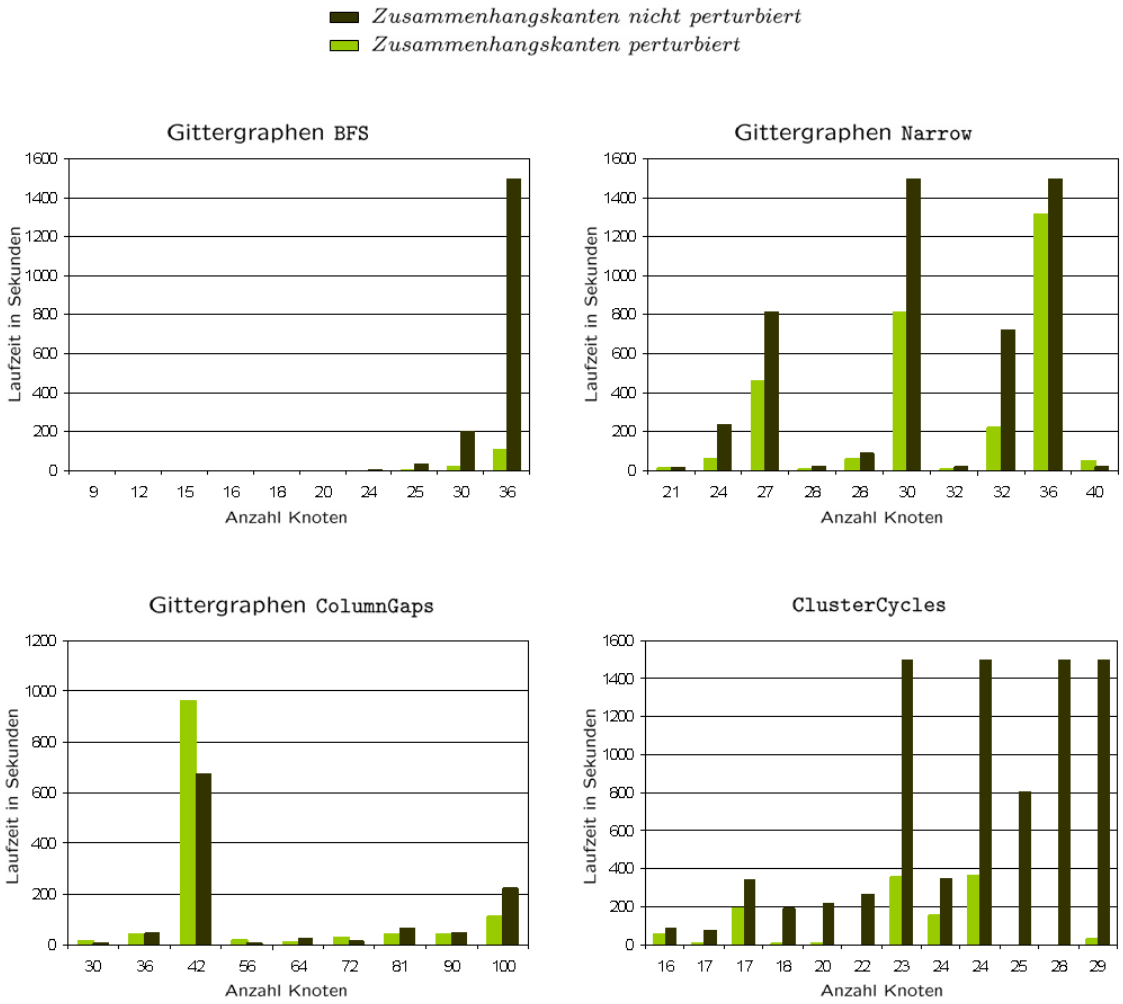


Abbildung 5.9: Laufzeiten bei Perturbation der Zusammenhangskanten

Bemerkenswert ist vor allem, dass ein paar der Graphen der Klasse `ClusterCycles` bei Benutzung der Variablen-Perturbation sogar innerhalb weniger Sekunden gelöst werden konnten, aber ohne Perturbation nicht innerhalb von 25 Minuten gelöst wurden. Da die `Wheel`-Graphen in Bezug auf den gewählten Ansatz im Prinzip „ideale“ Graphen darstellen, lässt sich anhand dieser auch am leichtesten nachvollziehen, was passiert: zur Erfüllung der separierten Cut-Constraints müssen einige Zusammenhangskanten einen Wert $> 0,0$ erhalten. Dazu werden nun vom Algorithmus solche Variablen ausgewählt die am günstigsten sind und diesen Zweck erfüllen. Am günstigsten sind nun genau diejenigen Variablen (Zusammenhangskanten), deren Abstand im Graphen am kleinsten ist. Bei den `Wheel`-Graphen sind dies diejenigen, die im Cluster direkt „nebeneinander liegen“. Die Heuristik testet nun in Phase 1 bei der Spannbaum-Berechnung im Wesentlichen zuerst die Variablen mit dem höchsten LP-Wert. Die Heuristik findet daher meist schon früh eine gute Auswahl von Zusammenhangskanten, die zusammen mit den Originalkanten keine nicht-planaren Strukturen implizieren, und damit eine optimale Lösung.

Das Vorgehen, die Variablen-Koeffizienten anhand des graphentheoretischen Abstandes der korrespondierenden Knoten zu modifizieren, scheint also - zumindest bezogen auf die betrachteten Clustergraphen - ein sehr erfolgversprechender Vorstoß zu sein, potentiell „gute und schlechte“ Zusammenhangskanten zu unterscheiden. Die beobachteten Erfolge müssen jedoch nicht zwangsläufig aus der Verwendung des graphentheoretischen Abstandes zur Wahl der Koeffizienten herrühren. Unter Umständen ist bereits die bloße Tatsache, dass die Variablen geringfügig unterscheidbar sind, ein ausschlaggebender Faktor dafür, dass der Algorithmus nun besser performt. Um dies näher zu untersuchen, sollten in zukünftigen Arbeiten weitere Ansätze zur Wahl der Perturbationswerte für die Koeffizienten verwendet werden. Eine ausgefeiltere, genauere Klassifizierung bzw. „Einschätzung“ der potentiellen Güte einer Zusammenhangskante ist jedoch nach wie vor eine wichtige Problemstellung. In diesem Bereich steckt sicherlich noch großes Potential zur Verbesserung der Performance des Algorithmus.

5.8 Tiefe der Clusterstruktur

Es gilt, dass bei vollständig zusammenhängenden Clustergraphen C-Planarität der Planarität des zugrunde liegenden Graphen entspricht. Daher stellen grundsätzlich weder die bloße Anzahl der im Clustergraphen vorhandenen Cluster, noch die Tiefe der Inklusionsstruktur ein geeignetes Maß für die potentielle Schwierigkeit eines Clustergraphen bezüglich des MCPSP dar. Impliziert die definierte Clusterstruktur vollständigen Zusammenhang, so hat diese keinerlei Auswirkungen auf die Performance des Algorithmus in dem Sinne, dass sich das Verhalten des Algorithmus im Vergleich zu demselben Graphen ohne Clusterstuktur nicht unterscheidet (es würde in beiden Fällen das MPSP gelöst). Erst wenn die Clusterstruktur nicht-zusammenhängende Subgraphen bzw. Komplementgraphen induziert, ändert sich auch die Performance des Algorithmus. Sinnvolle und allgemeine Aussagen, die einen unmittelbaren Zusammenhang zwischen der Schwierigkeit des gegebenen Clustergraphen bezüglich des Modells für das MCPSP und der Tiefe der über diesem definierten Inklusionsstruktur herstellen, sind daher kaum möglich. Grundsätzlich lässt sich jedoch sagen, dass bei zufällig erzeugter Clusterstruktur jeder zusätzliche Cluster potentiell ein neues „Risiko“ darstellt, den Grad an nicht vollständigem Zusammenhang zu erhöhen, und dadurch auch die potentielle Schwierigkeit, das MCPSP bezüglich dieses

Clustergraphen zu lösen.

5.9 Resümee bezüglich der Praxistauglichkeit

Handfeste Aussagen in Bezug auf die Praxistauglichkeit des Algorithmus sind schwierig zu treffen, da zu diesem Zweck ein geeignetes Benchmark-Set von Realworld-Clustergraphen erforderlich ist. Die erzeugten Clustergraphen in den Klassen `RandomDepth`, `MaxDepth`, `RootClusterRome` und `RootClusterRomeLarge` besitzen als zugrunde liegende Graphen jedoch Graphen aus der Rome-Library, bei denen es sich um (klassische) Graphen aus praktischen Anwendungen handelt. Anhand dieser lässt sich zumindest ein erster Eindruck verschaffen, ob sich der Algorithmus potentiell für den praktischen Einsatz eignet. Abbildung 5.10 stellt die Laufzeiten des Algorithmus bezüglich dieser Graphklassen zusammen. Dabei sind die Laufzeiten zur besseren Übersicht in Klassen eingeteilt.

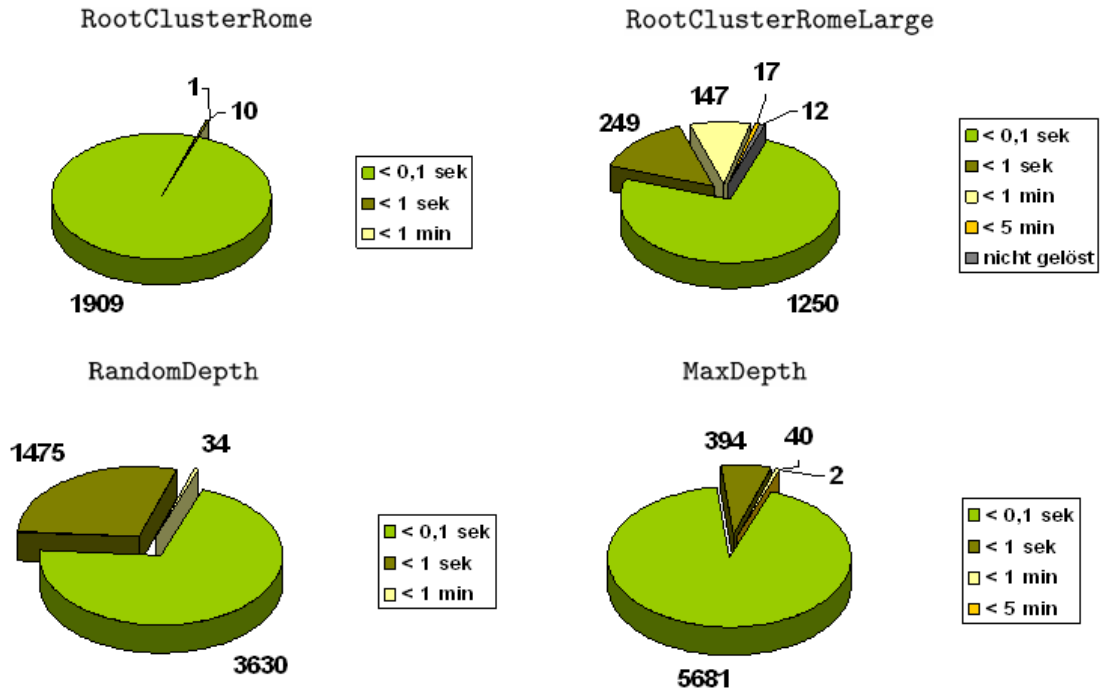


Abbildung 5.10: Laufzeiten der Rome-Graphen Benchmark-Sets

Die Graphen der Klasse `RootClusterRomeLarge` haben alle eine Größe von bis zu maximal 50 Knoten und 75 Kanten. Die anderen drei Klassen bestehen aus Graphen mit einer maximalen Größe von 25 Knoten und 42 Kanten. Der mit Abstand größte Teil der Graphen dieser Klassen kann innerhalb einer Sekunde gelöst werden. Bei den Graphen der Klasse `RootClusterRome` und `RootClusterRomeLarge` handelt es sich allerings um die reinen Rome-Graphen, bei denen die Clusterstruktur nur aus dem root-Cluster besteht, sodass hier also im Prinzip das MPSP gelöst wird. Diese haben somit bezüglich der Performance auf Clustergraphen nur bedingt Aussagekraft. Bei nicht-planaren Graphen dieser Klasse (also ohne definierte Clusterstruktur) stößt der Algorithmus bei Graphgrößen zwischen ca.

40 und 50 Knoten zum Teil schon an seine Grenzen. In Relation zur Menge der gelösten Graphen handelt es sich dabei allerdings noch um verhältnismäßig wenige.

Clusterstrukturen, die einen hohen Grad an Nicht-Zusammenhang sowohl bezüglich der Cluster-induzierten Graphen als auch der Komplementgraphen induzieren, stellen potentiell eine höhere Schwierigkeit dar. Wir haben gesehen, dass dies insbesondere an dem Mangel an Information liegt, welche Zusammenhangskanten „besser“ sind als andere. Ein moderater Grad an Nicht-Cluster-Zusammenhang stellt jedoch häufig kein allzu großes Problem dar. Abbildung 5.11 zeigt für die Graphen der Klassen `Grids` und `ClusterCycles`, wie die Laufzeiten des Algorithmus auf diesen anteilmäßig verteilt sind. Das Diagramm für die `Grid`-Graphen beinhaltet alle Unterklassen dieser Klasse. Die betrachteten Graphgrößen reichen von 3×3 bis 7×7 bezüglich der „quadratischeren“ Gittergraphen der Klassen `Column`, `ColumnGaps` und `BFS`, und von 5×2 bis 10×4 in Bezug auf die „spitzeren“ Gittergraphen der Klasse `Narrow`. Ab einer Graphgröße von mehr als 50 Knoten, also mindestens 7×8 Gittergraphen konnten die Graphen des Klassen `BFS` und `ColumnGaps` bei einer Lückengröße von 2 nicht mehr innerhalb des zeitlichen Limits von 25 Minuten gelöst werden. Aufgrund der auf diese Weise veränderten Clusterstruktur sind diese Gittergraphen größtenteils nicht mehr C-zusammenhängend und nicht-C-planar. Wie aus Abbildung 5.11 hervorgeht konnten dennoch noch einige der Graphen innerhalb kurzer Zeit gelöst werden.

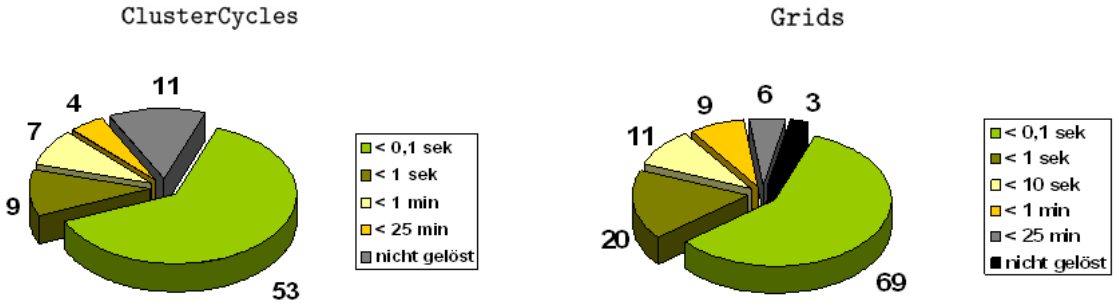


Abbildung 5.11: Laufzeiten der ClusterCycle und Grid Benchmark-Sets

Die betrachteten Größen der `ClusterCycle`-Graphen, also die Größe des zugrunde liegenden Kreises, reichen von 10 bis 30 Knoten, die Anzahl der gebildeten Cluster von 3 bis 5. Die Anzahl an nicht gelösten Instanzen ist hier im relativen Vergleich zu den anderen Graphklassen recht hoch. Dennoch konnten auch knapp 75% der Graphen dieser Klasse in weniger als einer Sekunde gelöst werden.

Insgesamt lässt sich festhalten, dass die Schwierigkeit eines konkreten Clustergraphen in Bezug auf den Branch-and-Cut Algorithmus und damit das entworfene Lösungs-Modell für das MCPSP natürlich zum Einen vom Grad der Nicht-Planarität des zugrunde liegenden Graphen, aber insbesondere auch maßgeblich von seiner konkreten Clusterstruktur und die dadurch induzierten Substrukturen abhängt. Die reine Größe des zugrunde liegenden Graphen gibt nur wenig Aufschluss über die potentielle Schwierigkeit, ebenso die Anzahl der Cluster und die maximale Tiefe der Inklusionsstrukturen. Ein Großteil der betrachteten Clustergraphen mittlerer Größe (bis ca. 50 Knoten) konnte schnell gelöst werden, so dass es durchaus denkbar ist, den Algorithmus auch praktisch einzusetzen. Auffallend bei allen

durchgeführten Experimenten ist außerdem, dass es unter den beobachteten Laufzeiten hauptsächlich Extrema gibt. In Bezug auf die betrachteten Benchmark-Sets konnte ein bestimmter Clustergraph meistens entweder sehr schnell optimiert werden, oder nur sehr langsam bzw. „gar nicht“. Im Mittelfeld befinden sich nur recht wenige Clustergraphen.

Bei dem implementierten Branch-and-Cut Algorithmus handelt es sich um einen ersten Schritt. Es bietet sich noch viel Platz für Ansätze und Techniken, diesen weiter zu verbessern, vor allem im Hinblick auf Bereiche, die im Rahmen der durchgeführten Untersuchungen als besonders schwierig herausgestellt wurden. Einige mögliche Ansätze sind in Kapitel 6 angerissen.

5.10 Weiterführende Experimente und Fragestellungen

Die im Rahmen dieser Arbeit durchgeführten experimentellen Untersuchungen geben einen ersten Eindruck auf die Performance des Algorithmus und die Güte des Modells für das MCPSP. Anhand einiger speziell konstruierter Clustergraphen konnte gezeigt werden, welche Strukturen besonders schwierig sind und warum der Algorithmus auf diesen schlecht performt. Zur weiteren Untersuchung des Algorithmus und um tiefere Einblicke in die Schwierigkeit des MCPSP zu erhalten, sind in zukünftigen Arbeiten unter anderem folgende Experimente und Fragestellungen denkbar:

- Intensivere Untersuchung nicht-planarer und damit nicht-C-planarer Clustergraphen. Die betrachteten Benchmark-Sets beinhalten hauptsächlich planare oder recht dünne Graphen, sodass der Grad an Nicht-Planarität bei diesen sehr gering ist.

- Untersuchungen des Algorithmus anhand von Clustergraphen, die nicht C-planar sind, aber deren zugrunde liegender Graph planar ist. Zum Teil wurde dies schon im Rahmen dieser Arbeit betrachtet, zum Beispiel anhand der `Grid`-Clustergraphen, deren zugrunde liegender Graph grundsätzlich planar ist, die Clustergraphen aber je nach Clusterstruktur nicht C-planar sind. Um diesbezüglich möglichst allgemeine Aussagen formulieren zu können, sind intensivere Tests erforderlich anhand ausgeklügelterer Benchmark-Sets.

- Um konkretere Aussagen über die Praxistauglichkeit des Algorithmus treffen zu können, ist ein breites Spektrum an Realworld-Clustergraphen als Benchmark-Set wünschenswert.

- Weitere Experimente bezüglich der Schwierigkeit des Nicht-C-Zusammenhangs. Wie ist zum Beispiel die Performance des Algorithmus, wenn der Grad des Nicht-C-Zusammenhangs eines Clustergraphen zwar hoch, aber über viele Cluster verteilt ist.

- Intensivere Tests der Heuristik. Die Ergebnisse aus Abschnitt 5.3.1 haben keinen großen Aufschluss darüber gegeben, welche Einstellungen im Durchschnitt zu den besten Ergebnissen führen. Denkbar ist zum Beispiel, die verwendeten Einstellungen von Iteration zu Iteration zu verändern, oder diese bei mehrmaligem Aufrufen der Heuristik zu variieren.

Ein Ausblick auf weitere mögliche Verbesserungen und Adaptionen des Algorithmus und in diesem Zusammenhang geläufige Techniken werden im nächsten Kapitel diskutiert.

Kapitel 6

Zusammenfassung und Ausblick

In dieser Diplomarbeit wurde das NP-schwierige *Maximum C-planare Subgraphen Problem (MCPSP)* vorgestellt und motiviert. Basierend auf den theoretischen Erkenntnissen zu vollständig zusammenhängenden Clustergraphen [CW03] konnte eine ILP-Formulierung für das MCPSP entworfen werden, für das ein Branch-and-Cut Algorithmus implementiert wurde anhand dessen das MCPSP optimal gelöst werden kann. Durch experimentelle Untersuchung des Algorithmus konnte eine erste Einschätzung gegeben werden, ob dieser und damit das entwickelte Lösungs-Modell zum praktischen Einsatz geeignet ist. Die anhand der Rome-Graphen erzeugten Clustergraphen kleiner bis moderater Größe (ca. 30-50 Knoten) werden im Allgemeinen, bis auf wenige schwierigere Graphen, sehr schnell gelöst. Es konnten zum Teil anhand weiterer speziell konstruierter Benchmark-Instanzen für bestimmte Strukturen in Clustergraphen gezeigt werden, dass diese für die Optimierung schwierig sind, was insbesondere den Nicht-Zusammenhang von Cluster-induzierten Graphen betrifft.

Der im Rahmen dieser Diplomarbeit implementierte Branch-and-Cut Algorithmus ist ein erster Ansatz und bietet noch viele mögliche Einstiegspunkte für weitere Verbesserungen und Ergänzungen. Dies beinhaltet unter anderem folgende Punkte:

- Geschicktes Preprocessing zur Variablenreduktion
- Spalten-Generierung. Dabei handelt es sich um ein in Branch-and-Cut Verfahren weit verbreitetes Konzept, dass relativ analog zur Separation von Constraints betrachtet werden kann.
- Weitere Heuristiken. Das Verwenden mehrerer Heuristiken zur Suche nach zulässigen Lösungen bietet viel Potential.
- Formulierung zusätzlicher, die LP-Relaxierungen verschärfender Constraints. Dies ist jedoch keine triviale Aufgabe. Vielmehr existiert in diesem Zusammenhang ein eigenes Forschungsgebiet, die *Polyedrische Kombinatorik*.

6.1 Preprocessing

Mit *Preprocessing* bezeichnet man im Allgemeinen applikationsspezifische Methoden oder Algorithmen die anfangs auf der Eingabe für den eigentlichen Lösungsalgorithmus angewendet werden. Ziel solcher Preprocessing-Methoden ist es, die Eingabegröße zu verkleinern, indem sie die Eingabe auf „problem-unabhängige“ Teile überprüfen, und diese gegebenenfalls abschneiden. Problem-unabhängig ist dabei so zu verstehen, dass möglicherweise bestimmte Teile der Eingabe-Instanz zur Problem-Lösung nicht betrachtet werden müssen, da diese keinen Einfluss auf die Menge der optimalen Lösungen haben. Zur exakten Lösung NP-schwieriger Optimierungsprobleme sind geschickte Preprocessing-Algorithmen von großer Bedeutung. Das Abschneiden nicht-relevanter Teile der Eingabe-Instanz führt zu einer Reduzierung der Komplexität der Instanz, bezogen auf die zugrunde liegende Problemstellung. Es handelt sich also dabei um ein Vorgehen, die Eingabe möglichst zu „vereinfachen“.

Betrachten wir den Branch-and-Cut Algorithmus für das MCPSP. Eine durch ein Preprocessing bedingte Verkleinerung der Eingabegröße, also des gegebenen Clustergraphen, führt zu einer Reduktion der benötigten Variablen. Dadurch werden die für jedes Subproblem erzeugten LPs kleiner. Variablen-Reduktion durch vorheriges, geschicktes Preprocessing ist daher durchaus erfolgversprechend. In Bezug auf das MCPSP stellt sich dies jedoch als äußerst schwierig heraus. Mögliche Fragestellungen sind hierbei:

- Kann man effizient bestimmte Substrukturen im Graphen ausfindig machen, die sich abschneiden lassen, ohne die Menge der optimalen Lösungen zu verändern? „Abschneiden“ bezieht sich in diesem Zusammenhang zum Beispiel auf das Löschen oder Verschmelzen von Knoten, sodass die Graphgröße und damit die Anzahl der Variablen im LP abnimmt. Die definierte Clusterstruktur über dem zugrunde liegenden Graphen macht es sehr schwierig und kaum vorhersehbar, ob es bestimmte Teilgraphen gibt, die keinen Einfluss auf die Menge der optimalen Lösungen haben.

- Kann von bestimmten Kanten des Graphen beweisbar gesagt werden, dass diese niemals in einer optimalen Lösung gelöscht werden, oder von bestimmten potentiellen Zusammenhangskanten, dass diese nicht unbedingt zur Herstellung des vollständigen Zusammenhangs benötigt werden? Auch diese Problemstellung stellt eine Herausforderung dar.

Eine erste (naheliegende) Idee bestimmte Zusammenhangskanten auszuschließen, die im Rahmen dieser Arbeit aufkam, musste schon schnell wieder verworfen werden, da sie sich als falsch erwies. Sie soll dennoch kurz erwähnt werden:

Man betrachte einen zusammenhängenden Cluster ν eines Clustergraphen $C = (G, T)$. Um Planarität zu erhalten müssen gegebenenfalls einige Kanten aus $G(\nu)$ entfernt werden, jedoch niemals so viele, dass $G(\nu)$ anschließend nicht mehr zusammenhängend ist. Denn dies hätte zur Folge, dass der Zusammenhang durch Einfügen einer neuen Zusammenhangskante wiederhergestellt werden muss, was den Zielfunktionswert verringern würde. Daher fällt die „Auswahl“ der zu löschenden Kanten niemals auf solche, durch die der Zusammenhang des Clusters zerstört würde. Die Idee ist somit, dass man für jeden **zusammenhängenden** Cluster ν initial alle zwischen den Knoten aus $V(\nu)$ verlaufenden potentiellen Zusammenhangskanten auf $0,0$ setzen kann. Leider stellte sich dies schnell als Irrtum heraus. Die

Annahme, dass niemals soviele Kanten entfernt werden, so dass ein Cluster-induzierter Subgraph anschließend nicht mehr zusammenhängend ist, ist falsch! Es lassen sich Clustergraphen konstruieren, bei denen die Entfernung einer bestimmten Originalkante in einem Cluster-induzierten Graphen dazu führt, dass dieser zwar anschließend nicht mehr zusammenhängend ist, die Entfernung der Kante jedoch gleichzeitig mehrere nicht-planare Substrukturen zerstört, so dass sich die Entfernung dieser Kante dennoch „lohnt“. Das Problem liegt unter anderem darin, dass die Menge möglicher planarer Einbettungen eines Cluster-induzierten Graphen $G(\nu)$ im Kontext des MCPSP eingeschränkt ist. Besitzt $G(\nu)$ zum Beipiel viele extroverte Kanten (Kanten, die aus $G(\nu)$ „hinauslaufen“), so induziert eine bestimmte planare Einbettung $\mathcal{E}$ von $G(\nu)$ möglicherweise eine Kantenkreuzung zwischen einer der extroverten Kanten $e = \{v, w\},\ v \in V(\nu), w \in V \backslash V(\nu)$ und einer Kante aus $G(\nu)$. Dies kann zum Beispiel passieren, wenn aufgrund der Einbettung $\mathcal{E}$ der Knoten w der extroverten Kante e nicht inzident zum externen Face bezüglich (E) ist, sondern „innerhalb„ von $G(\nu)$ liegt. Solche Einbettungen lassen sich nicht ausschließen, wenn man die Cluster-induzierten Graphen isoliert betrachtet.

Bezogen auf die erste Fragestellung lässt sich folgendes überlegen: Man betrachte einen Cluster ν eines Clustergraphen $C = (G, T)$. Besitzt dieser nur **eine** extroverte Kante, also eine Kante $e = \{v, w\}$ mit $v \in V(\nu),\ w \in V \backslash V(\nu)$, und besitzt der durch ν induzierte Subgraph $G(\nu)$ eine planare Einbettung, in der der Knoten v inzident zum externen Face der Einbettung von $G(\nu)$ liegt, so gilt, dass jeder maximum C-planare Subgraph C' von C alle Kanten aus $G(\nu)$ enthält. Dies ist leicht einzusehen: in jeder konsistenten, C-planaren Zeichnung eines maximum C-planaren Subgraphen von C liegen die Knoten $V(\nu)$ alle vollständig im Inneren der geschlossenen Region, die den Cluster ν repräsentiert. Die Kanten von $G(\nu)$ verursachen daher keine Kante-Region-Kreuzung. Da $G(\nu)$ nur eine extroverte Kante e besitzt, und für diesen außerdem eine planare Einbettung existiert, bei der e inzident zum externen Face von $G(\nu)$ ist, verursachen die Kanten von $G(\nu)$ auch keine Kantenkreuzung mit der Kante e. Demnach enthält jeder maximum C-planare Subgraph von C auch alle Kanten aus $G(\nu)$.
Der Cluster ν und damit der Graph $G(\nu)$ hat also keinen Einfluss auf potentielle Nicht-C-Planarität des Clustergraphen C. $G(\nu)$ kann in diesem Fall also zu einem Knoten kollabiert werden, ohne dadurch die Menge der optimalen Lösungen bezüglich des MCPSP zu verändern. Dennoch handelt es sich hierbei um kein Preprocessing-Verfahren, dass (falls überhaupt anwendbar) die Komplexität des gegebenen Clustergraphen nennenswert veringern kann. In diesem Bereich liegt noch großes Potential zur Verbesserung des Algorithmus.

6.2 Klassifizierung der Zusammenhangskanten

Die vorangegangenen Experimente haben gezeigt, dass insbesondere die Herstellung des Cluster-Zusammenhangs bei stark nicht-zusammenhängenden Clustern für den Algorithmus sehr schwierig ist. Es wurde gezeigt, dass dies im Wesentlichen an der Tatsache liegt, dass der Algorithums keine Informationen darüber hat, welche Zusammenhangskanten potentiell gut und welche ungünstig sind (da durch deren Hinzunahme nicht-planare Substrukturen entstehen). Gerade an dieser Stelle bietet sich also auch noch ein Ansatzpunkt, den Algorithmus zu verbessern, bzw. diesem die Kantenauswahl zu „erleichtern“, indem man versucht, diese zu klassifizieren. Ein erster Ansatz war die Perturbation der

Zielfunktions-Koeffizienten der Zusammenhangskanten (siehe Abschnitt 4.5.2) anhand des graphentheoretischen Abstandes der zu einer Zusammenhangskante inzidenten Knoten. Eine geschicktere Klassifikationen der Zusammenhangskanten, die die potentielle Güte der einzelnen Kanten realistischer wiederspiegelt, kann die Performance des Algorithmus weiter verbessern.

6.3 Spalten-Generierung

Die Spalten-Generierung ist vom Konzept her ähnlich der Separierung von Constraints. Hierbei wird mit einer Teilmenge der Variablen gestartet und das dadurch reduzierte LP zunächst optimal gelöst. Mit Hilfe eines sogenannten *Pricing*-Algorithmus wird dann überprüft, ob es weggelassene Variablen gibt, durch deren Hinzunahme der Zielfunktionswert weiter verbessert werden kann. Falls ja werden diese zum LP hinzugefügt und dieses erneut gelöst. Dieses iterative Vorgehen ist also sehr ähnlich dem Separieren von Constraints. Tatsächlich lässt sich im Kontext der LP-Dualität die dynamische Generierung von Variablen im primalen LP auch als das dynamische Separieren und Hinzufügen von Constraints im dualen LP betrachten, sodass diese Techniken zumindest technisch gesehen zu großen Teilen identisch sind. Der Entwurf von effizienten Pricing-Algorithmen zur dynamischen Erzeugung von Variablen ist genau wie der Entwurf effizienter Separierungs-Algorithmen vollständig applikationsspezifisch. Branch-and-Bound Algorithmen, die Spalten-Generierungs Techniken beinhalten, nennt man auch *Branch-and-Price-* bzw. *Branch-and-Cut-and-Price*Algorithmen.

6.4 Heuristik

Die im Rahmen dieser Arbeit implementierte primale Heuristik zeigt sehr große Schwächen in Bezug auf stark nicht-C-zusammenhängende Clustergraphen. Ein möglicher Ansatz ist es, das bereits umgesetzte Konzept, Zusammenhangskanten nach ihrem graphentheoretischen Abstand zu klassifizieren, in die bereits implementierte Heuristik zu integrieren (oder besser eine zweite zu implementieren), sodass die Zusammenhangskanten in der Reihenfolge ihres graphentheoretischen Abstandes getestet werden, bzw. dies mit in die Generierung der Test-Reihenfolge einfließt.

Grundsätzlich sind mehrere Heuristiken in einem Branch-and-Bound Algorithmus sinnvoll, sodass also auch in den Entwurf weiterer Heuristiken entsprechender Aufwand gesteckt werden sollte.

Literaturverzeichnis

[ABCC01] D. Applegate, R. E. Bixby, V. Chvátal, and W. Cook. Tsp cuts which do not conform to the template paradigm. In M. Jünger and D. Naddef, editors, *Computational Combinatorial Optimization*, volume 2241 of *Lecture Notes in Computer Science*, pages 261–304, 2001.

[AP61] L. Auslander and S. V. Parter. On embedding graphs in the plane. In *Journal of Applied Mathematics and Mechanics*, volume 10, pages 517–523, 1961.

[BDM01] G. Di Battista, W. Didimo, and A. Marcandalli. Planarization of clustered graphs. In *Proc. of The 9th International Symposium on Graph Drawing (GD 2001)*, pages 60–74, 2001.

[Ber85] C. Berge. *Graphs and Hypergraphs*. Elsevier Science Ltd, 1985.

[BGLT97] G. Di Battista, A. Garg, G. Liotta, and R. Tamassia. An experimental comparison of four graph drawing algorithms. In *International Journal of Computational Geometry and Applications*, volume 7, pages 303–325, 1997.

[BL76] K. Booth and G. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *Journal of Computer and System Science*, 13:335–379, 1976.

[BM99] F. Bertault and M. Miller. An algorithm for drawing compound graphs. In *Proc. of The 7th International Symposium on Graph Drawing (GD 1999)*, volume 1731, pages 197–204, 1999.

[BM04] J. M. Boyer and W. Myrvold. On the cutting edge: Simplified O(n) planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004.

[CBPP04] P. F. Cortese, G. Di Battista, M. Patrignani, and M. Pizzonia. Clustering cycles into cycles of clusters (extended abstract). In *Proc. of The 12th International Symposium on Graph Drawing (GD 2004)*, pages 100–110, 2004.

[CW03] S. Cornelsen and D. Wagner. Completely connected clustered graphs. In *Proceedings of the 29th International Workshop on Graph Theoretic Concepts in Computer Science (WG 2003)*, volume 2880, pages 168–179, 2003.

[Dah98] E. Dahlhaus. A linear time algorithm to recognize clustered planar graphs and its parallelization. In *LATIN: Latin American Symposium on Theoretical Informatics*, volume 1380, pages 239–248, 1998.

[EFC95] Peter Eades, Qing-Wen Feng, and Robert F. Cohen. Clustered graphs and C-planarity. Technical report, March 22 1995.

[For02] M. Forster. Applying crossing reduction strategies to layered compound graphs. In Michael T. Goodrich and Stephen G. Kobourov, editors, *Proc. of The 10th International Symposium on Graph Drawing (GD 2002)*, volume 2528 of *Lecture Notes in Computer Science*, pages 276–284, 2002.

[GHZ98] B. Gartner, M. Henk, and G. M. Ziegler. Randomized simplex algorithms on klee-minty cubes. *Combinatorica*, 18(3):349–372, 1998.

[GJ83] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4(3):312–316, 1983.

[GJK+01] C. Gutwenger, M. Jünger, G. W. Klau, S. Leipert, P. Mutzel, and R. Weiskircher. AGD: A library of algorithms for graph drawing. In *Proc. of The 9th International Symposium on Graph Drawing (GD 2001)*, pages 473–474, 2001.

[GJL+02] C. Gutwenger, M. Jünger, S. Leipert, P. Mutzel, M. Percan, and R. Weiskircher. Advances in C-planarity testing of clustered graphs. In *Proc. of The 10th International Symposium on Graph Drawing (GD 2002)*, pages 220–235, 2002.

[GLS05] M. T. Goodrich, G. S. Lueker, and J. Z. Sun. C-planarity of extrovert clustered graphs. In *Proc. of The 13th International Symposium on Graph Drawing (GD 2005)*, volume 3843, pages 211–222, Limerick, Ireland, September 2005.

[GT94] A. Garg and R. Tamassia. On the computational complexity of upward and rectilinear planarity testing. In *SIAM Journal on Computing*, volume 31, pages 601–625, 1994.

[Har95] D. Harel. On visual formalisms. In *Diagrammatic Reasoning*, pages 235–271. The MIT Press, Cambridge, Massachusetts, 1995.

[HMM00] I. Herman, G. Melançon, and M. Scott Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.

[JT97] M. Jünger and S. Thienel. The design of the branch-and-cut system abacus. Technical report, Institut für Informatik, Universität zu Köln, 1997.

[LE96] W. Lai and P. Eades. A graph model which supports flexible layout functions. Technical Report 96–15, Callaghan 2308, Australia, 1996.

[LG77] P. Liu and R. Geldmacher. On the deletion of non-planar edges of a graph. In *Proceedings of the 10th Southeastern Conference on Combinatorics, Graph Theory, and Computing*, pages 727–738. Boca Raton, 1977.

[MW98] P. Mutzel and R. Weiskircher. Two-layer planarization in graph drawing. In *Proc. of the 9th International Symposium on Algorithms and Computation (ISAAC-98)*, volume 1533 of *Lecture Notes in Computer Science*, pages 69–78, 1998.

[PMCC01] H. C. Purchase, M. McGill, L. Colpoys, and D. Carrington. Graph drawing aesthetics and the comprehension of uml class diagrams: an empirical study. In *Proceedings of the 2001 Asia-Pacific symposium on Information visualisation*, volume 9, pages 129–137, 2001.

[Pur97] H. C. Purchase. Which aesthetic has the greatest effect on human understanding? In *Proc. of the 5th International Symposium on Graph Drawing (GD 1997)*, pages 248–261, 1997.

[Pur02] H. C. Purchase. Metrics for graph drawing aesthetics. *Journal of Visual Languages and Computing*, 13(5):501–516, 2002.

[Sch07] J. Schmidt. *Effiziente Extraktion von Kuratowski-Teilgraphen.* Diplomarbeit, Universität Dortmund, Lehrstuhl 11 für Algorithm Engineering, März 2007.

[SW97] M. Stoer and F. Wagner. A simple min-cut algorithm. *Journal of the ACM*, 44(4):585–591, 1997.

[WPCM02] C. Ware, H. C. Purchase, L. Colpoys, and M. McGill. Cognitive measurements of graph aesthetics. *Information Visualization*, 1(2):103–110, 2002.