Bachelorarbeit

Clique als Optimierungsproblem

MaxCLIQUE, MCP, MCC und Weitere

Rico Schrage

01. März 2017

Am Institut für Theoretische Informatik
Leibniz Universität Hannover

Erstprüfer	Prof. Dr. H. Vollmer
Zweitprüfer	Dr. A. Meier
Betreuer	Dr. A. Meier

Erklärung

Hiermit versichere ich, die abgegebene Bachelorarbeit selbstständig verfasst und keine anderen als die angegeben Hilfsmittel und Quellen verwendet zu haben. Weiter wurden alle direkten oder indirekten Zitate als solche kenntlich gemacht.

Weder wurde die Arbeit in dieser oder ähnlicher Form einer Prüfungsbehörde vorgelegt noch wurde sie anderweitig veröffentlicht.

Rico Schrage Hannover, der 01.03.2017

Inhaltsverzeichnis

1 | Einleitung

Das Berechnen von *Cliquen*, ob als Partitionierung oder einzeln, ist ein wichtiges Thema in der Informatik. Es wird ein Beispiel dafür betrachtet: Es gebe eine Gruppe von Personen, dargestellt durch jeweils einen Knoten, diese sind immer dann verbunden, wenn die Personen miteinander befreundet sind. Dadurch entstehe ein Graph. Die Analyse solcher Graphen ist Teil der Social-Network Analyse. Ein wichtiger Bereich dieser Analyse ist die Berechnung von maximal großen Personengruppen [RGG15], die alle (oder fast alle) miteinander befreundet sind. Solche Gruppen werden in der Graphentheorie *Cliquen* (Quasi-Cliquen) genannt. Weitere typische Bereiche in denen maximale *Cliquen*, oder generell die Berechnung von *Cliquen* eine Rolle spielen, sind die Bioinformatik, Visual-Analytics oder Netzwerkanalyse [Bom+99].

Bei der Berechnung solcher Cliquen gibt es allerdings, mit Blick auf die Resultate der Komplexitätstheorie, ein Problem. Denn das Entscheidungsproblem Clique oder auch verwandte Entscheidungsprobleme wie Clique-Cover sind NP-vollständig [Kar72]. Anschaulich bedeutet dieses (falls $\mathcal{P} \neq \mathcal{NP}$), dass das Problem zu den schwersten nicht effizient lösbaren Problemen gehört. Somit existiert kein optimaler polynomieller Algorithmus um eine maximale Clique zu berechnen. Nun haben aber beispielsweise Graphen aus der Social-Network Analyse recht viele Knoten, somit müssen Wege gefunden werden, trotz der Schwierigkeit des Problems, schnell Lösungen zu finden.

In solchen Fällen werden oft Optimierungsprobleme als Ausweg verwendet. Diese suchen nicht zwingend die beste Lösung für ein Problem, sondern vielmehr eine möglichst gute, in der Regel maximale oder minimale Lösung. Es soll erreicht werden, das Problem in polynomieller Laufzeit mit minimalem Qualitätsverlust zu lösen. Dafür können sogenannte Approximationsalgorithmen entwickelt werden, die sich durch eine Performanzrate auszeichnen. Diese Rate ist der Faktor, um den die Lösung vom Algorithmus schlimmstenfalls schlechter ist als die optimale Lösung. Neben den Approximationsalgorithmen gibt es noch die exakten Algorithmen, und die heuristischen Algorithmen. Abgrenzend zu Approximationsalgorithmen haben heuristische keine beweisbaren Garantien, deren Effektivität wird allein durch empirische Tests bestimmt. Exakte Algorithmen liefern, dem Namen entsprechend, optimale Lösungen.

In der folgenden Arbeit werden diese Algorithmengruppen für mehrere Clique-Optimierungsprobleme betrachtet, erläutert und verglichen. Hierbei wird Frage

untersucht, welche verschiedenen algorithmischen Möglichkeiten es gibt und wie effektiv diese sind. Zusätzlichen werden Beispiele solcher vorgestellt und implementiert. Die besonderen Teile dieser Implementierungen sollen zusätzlich erläutert werden. Weiter wird untersucht, wie sich die verschiedenen Clique-Optimierungsprobleme zueinander verhalten. Hierfür werden Reduktionen zwischen den unterschiedlichen Problemen gezeigt.

Neben dem Optimierungsproblem zu Clique werden in dieser Arbeit noch weitere Probleme, in denen es um das Finden von Cliquen geht, betrachtet. Neben den zwei weiteren oft behandelten Clique-Problemen, wovon eines zu einem Karp Entscheidungsproblem [Kar72] gehört, werden schließlich noch weniger intensiv erforschte Probleme behandelt und in die „Welt der Clique-Probleme“ eingeordnet.

1.1. Aufbau

Die Arbeit gliedert sich in sechs Kapitel (exklusive Einleitung und Resümee). Bevor die Optimierungsprobleme behandelt werden, werden in Kapitel 2 einige grundlegende Begriffe definiert. Im folgenden Kapitel 3, wird auf das Maximum Clique Problem eingegangen. Hierbei werden die Möglichkeiten der Approximation untersucht und ein Einblick in Approximationsalgorithmen, heuristische und exakte Algorithmen gegeben. Am Ende vom Kapitel werden Implementierungen der behandelten Algorithmen erläutert. Das Kapitel 4 ist analog aufgebaut und handelt vom Minimum Clique Partition Problem. Das dritte Clique-Problem Minimum Clique Cover wird in Kapitel 5 behandelt. Dort wird neben Approximierbarkeit und Algorithmen, auch eine Reduktion auf (und von) Minimum Clique Partition erläutert. Im vorletzten Kapitel 6 werden noch kurz drei weitere Clique-Probleme steckbriefartig eingeführt, um schließlich im letzten Kapitel 7 die Relationen zwischen allen sechs behandelten Problemen zu betrachten. Außerdem ist im Anhang eine Dokumentation zum sogenannten *Clique-Framework* zu finden, welches für die Implementierung der Algorithmen entwickelt und eingesetzt wurde.

2 | Grundlagen

Zunächst folgen einige grundlegende Definitionen, die zum Verständnis der Clique-Probleme benötigt werden.

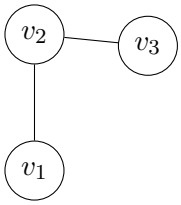
2.1. Graphen

Es wird begonnen, einen Graphen zu definieren und speziellere Notationen einzuführen. Die Kenntnis von grundlegenden Begriffen aus der Graphentheorie wird vorausgesetzt.

Definition 1 (Graph)**.** *Ein* Graph *sei definiert als 2-Tupel bestehend aus der Menge der Knoten V und einer Kantenmenge $E \subseteq \{ U \subseteq V : |U| = 2 \}$.*

$$G = (V, E)$$

Beispiel 1 (Graph)**.** *Gegeben sei der Graph $G = (\{ v_1, v_2, v_3 \}, E)$ wobei die Kantenmenge $E = \{ \{ v_1, v_2 \}, \{ v_2, v_3 \} \}$ ist. Dieser Graph lässt sich wie folgt darstellen:*



Definition 2 (Induzierter Teilgraph)**.** *Es sei $G = \{ V, E \}$ und $V' \subset V$. Dann ist der* induzierte Teilgraph *von V', ein Graph*

$$G' = \Big\{ V', \{ \{ x, y \} : x, y \in V' \wedge \{ x, y \} \in E \} \Big\}.$$

Es handelt sich um den Graphen, der alle Knoten aus V' und alle zwischen diesen Knoten vorkommenden Kanten enthält. Als Abkürzung kann $IT_G(V')$ (oder falls sich G aus dem Kontext ergibt schlicht $IT(V')$) für den induzierten Teilgraphen *der Knotenmenge V' auf dem Graphen G verwendet werden.*

Es folgen noch einige wichtige Definitionen aus der Graphentheorie, welche in der Arbeit Verwendung finden.

Definition 3 (Clique)**.** *Eine* Clique *in einem Graphen $G = (V, E)$ ist eine Teilmenge $V' \subseteq V$ für dessen Elemente gilt, dass jeder Knoten mit allen anderen Knoten in dieser Menge verbunden sein muss. Die höchstmögliche Anzahl an Knoten in einer* Clique *eines Graphen G wird mit $\omega(G)$ bezeichnet.*

Definition 4 (Clique Partition)**.** *Eine* Clique Partition *ist die Aufteilung eines Graphen in $n \in \mathbb{N}$ paarweise disjunkte Mengen $V_0 \uplus V_1 \uplus \cdots \uplus V_n = V$, die jeweils eine Clique bilden (vgl. [CK97]).*

Definition 5 (Clique Überdeckung)**.** *Eine* Clique Überdeckung *ist die Aufteilung eines Graphen in $n \in \mathbb{N}$ Mengen $V_1 \cup V_2 \cup \cdots \cup V_n = V$, die jeweils eine Clique bilden, wobei für alle unterschiedlichen Knoten $u, v \in V$ gilt, dass, falls die beiden Knoten durch eine Kante verbunden sind, sie sich beide in mindestens einer der Mengen V_i $(0 < i \leq n)$ befinden (vgl. [CK97]).*

Zur Clique und Clique Partition existieren zwei stark verwandte Begriffe, die im Folgenden eingeführt werden. Des Weiteren werden Matchings definiert.

Definition 6 (Stabile Menge)**.** *Eine* stabile Menge *(engl.* Independent Set*) in einem Graphen $G = (V, E)$ ist eine Teilmenge $V' \subseteq V$ für dessen Elemente gilt, dass keiner der Knoten mit einem der anderen, in der Menge befindlichen, Knoten eine Verbindung haben darf. Eine* stabile Menge *V' ist also auf dem komplementären Graphen $\overline{G}$ eine Clique.*

Definition 7 (Graph-Coloring)**.** *Es sei $G = (V, E)$ ein Graph. Dann wird eine Abbildung $c\colon V \to F$, mit $F = \{F_1, \ldots, F_k\}$ wobei $k \in \mathbb{N}$, als korrektes* Graph-Coloring *bezeichnet, falls gilt, dass jeder Knotens $v \in V$ eine andere Farbe hat als seine Nachbarn. Dieses entspricht der Aufteilung des Graphen in paarweise disjunkte Mengen, die jeweils stabil sind.*

Definition 8 (Matching)**.** *Es sei ein Graph $G = (V, E)$, dann ist ein* Matching *$M \subseteq E$ eine Menge von Kanten, sodass kein Knoten in V als Endpunkt in mehreren Kanten in M vorkommt [HK73].*

Definition 9 (Maximum-Matching)**.** *Das Matching M nennt man* Maximum-Matching*, falls $|M|$ größtmöglich ist [HK73].*

Definition 10 (Maximum/Minimum-Weighted-Matching)**.** *Es gebe zum Graph G eine Gewichtsfunktion $g\colon E \to \mathbb{N}$. Ein Matching M wird* Maximum/Minimum-Weighted-Matching *genannt, falls sie ein Maximum-Matching ist und $\Sigma_{x \in M} c(x)$ maximal (minimal) ist.*

2.1.1. Unit Disk Graphen

An einigen Stellen werden eingeschränkte Graphen verwendet, um bessere Laufzeiten und Performanzraten zu erreichen. Hierfür sind insbesondere die Unit Disk Graphen interessant.

Definition 11 (Unit Disk Graph)**.** *Es existiert ein* Unit Disk Graph (UDG) *zu einem Graphen G, falls geometrische Informationen (x- und y-Positionen der Knoten) für den Graphen existieren, die die folgende Bedingung erfüllen:*

> *Es gebe Kreise mit jeweils dem Radius ε, deren Mittelpunkte gleich der Positionen der Knoten seien. Damit G ein* UDG *ist, darf jeder Knoten lediglich mit den Knoten verbunden sein, für welche die beiden zugehörigen Kreise sich schneiden (vgl. [CCJ90, Kapitel 1]).*

Ein Modell für einen solchen Graph kann folgendermaßen formal beschrieben werden:

$$M_{UDG} = (V, \varepsilon) \text{ mit } V = \{ (x, y) : x, y \in \mathbb{R} \} \text{ und } \varepsilon \in \mathbb{R}$$

Beispiel 2 (Unit Disk Graph)**.** *Ein Modell für den Graphen aus Beispiel 1 kann wie folgt aussehen:* $M_{UDG} = \{ \{ (0, 0), (0, 2), (2, 2) \}, 1 \}$

Abbildung 2.1 zeigt sehr anschaulich, dass es sich bei dem gezeigten Graphen um einen UDG *handeln muss. Es ist ersichtlich, dass ausschließlich zwischen den Knoten deren Kreise sich schneiden/berühren (zwischen v_2 und v_1, v_2 und v_3) eine Kante verläuft, folglich ist die obige Bedingung erfüllt.*

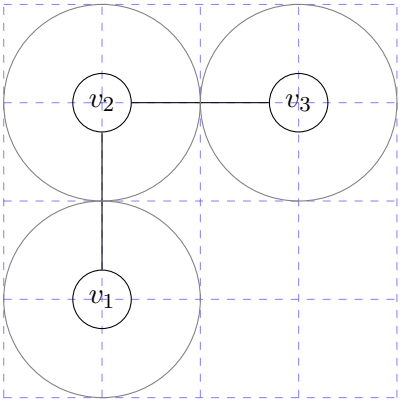


Abbildung 2.1.: Graph aus Beispiel 1 als UDG

2.1.2. Intervall Graphen

Definition 12 (t-Intervall Graph)**.** *Ein t-Intervall Graph bezeichnet einen Schnittgraphen, dessen zugrundeliegendes Modell aus jeweils genau t Intervallen (auf den reellen Zahlen) pro Knoten besteht (vgl. [Rid]).*

Es gibt also genau dort eine Kante zwischen zwei Knoten, wenn einer (oder mehrere) deren Intervalle sich überschneiden. Es sei $G = (V, E)$ ein Graph, dann ist in Modell für einen solchen Graphen gegeben als Menge von Intervallfunktionen, die jeweils jedem Knoten ein Intervall zuweist:

$$I_1, \ldots, I_t \text{ mit } I_i \colon V \to [a_i, b_i].$$

Beispiel 3 (1-Intervall Graph)**.** *Es gelte $I_1(v_1) = [0, 1]$, $I_1(v_2) = [0.5, 2]$, $I_1(v_2) = [1.5, 3]$, durch dieses Modell wird folgender Graph induziert:*

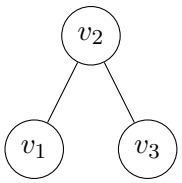


Abbildung 2.2.: Aus dem Modell resultierender Graph

Da sich sowohl die Intervalle von v_2 und v_1 als auch von v_3 und v_2 überschneiden, verläuft zwischen den jeweiligen Knoten eine Kante.

2.1.3. Weitere Graphenklassen

Definition 13 ((Co-)Bipartiter Graph)**.** *Ein Graph G heißt* bipartit*, falls er mit zwei stabilen Mengen überdeckbar ist. Entsprechend heißt ein Graph G* co-bipartit*, falls er mit zwei Cliquen überdeckbar ist, also eine Clique-Überdeckung mit genau zwei Mengen existiert (vgl. [Rid]).*

In der Arbeit werden an bestimmten Stellen noch weitere Graphenklassen im Hinblick auf die Approximierbarkeit der jeweiligen Probleme erwähnt. Diese werden in der Arbeit nicht weiter inhaltlich behandelt, somit sei für deren Definition auf [Rid] verwiesen.

2.2. Optimierungsprobleme

Im Folgenden werden die Optimierungsprobleme und weitere zugehörige Begriffe formalisiert. Hierbei werden die Definitionen aus dem Buch „Komplexität von Algorithmen“ von Meier und Vollmer [MV15] verwendet. Auch die weiteren, hier nicht definierten (z.B. $\mathcal{P}$) Grundbegriffe der Komplexitätstheorie können diesem Buch entnommen werden.

Definition 14 (Optimierungsproblem)**.** *Ein* Optimierungsproblem *sei ein 4-Tupel, wie folgt aufgebaut:*

$$(I,\, s,\, m,\, t).$$

Die Menge I enthält alle Instanzen, welche Strukturen jedweder Art sein können, sie stellen die Eingabe da. Die Abbildung s bildet eine Instanz $x \in I$ auf eine Menge gültiger Lösungen ab. Hierbei sei $S = \bigcup_{x \in I} s(x)$ die Menge aller Lösungen. Das Maß m ist eine Abbildung $m\colon I \times S \to \mathbb{N}$, sie soll eine Lösung qualitativ bewerten. Das Ziel t ist ein Element aus der Menge $\{\, max,\, min \,\}$ und definiert, ob ein maximal großes oder minimal kleines Maß erreicht werden soll.

Definition 15 (Approximationsalgorithmus)**.** *Sei $p = (I,\, s,\, m,\, t)$ ein Optimierungsproblem, dann ist ein zugehöriger* Approximationsalgorithmus *A_p eine Abbildung $A\colon I \to S$. Der Algorithmus bildet eine Instanz $x \in X$ auf eine zugehörige Lösung $y \in S$ ab.*

Definition 16 (Performanzrate)**.** *Sei $p = (I,\, s,\, m,\, t)$ ein Optimierungsproblem, dann ist die* Performanzrate *R_p wie folgt definiert:*

$$R_p(x,\, y) = \begin{cases} \frac{m(x,y)}{m(x,y^*)}, & \textit{falls } t = min. \\ \frac{m(x,y^*)}{m(x,y)}, & \textit{falls } t = max. \end{cases}$$

Hierbei sei $x \in I$ und $y^* \in s(x)$, wobei für y^* gilt, dass dessen Maß max- bzw. minimal ist.

2.2.1. Minimum Graph Coloring

Das Minimum Graph Coloring (MGC) Problem kann wie folgt definiert werden (vgl. [CK97]):

Instanz Ein Graph $G = (V,\, E)$.

Lösung Ein Graph-Coloring von G. Dieses wird durch die paarweise disjunkten stabilen Mengen $V_1, \ldots, V_k$ dargestellt.

Maß Der Parameter k.

Ziel Die Minimierung von k.

2.2.2. Maximum Independent Set

Das Maximum Independent Set (MIS) Problem kann wie folgt definiert werden (vgl. [CK97]):

Instanz Ein Graph $G = (V,\, E)$.

Lösung Eine stabile Menge $V' \subseteq V$ aus G.

Maß Die Anzahl der Knoten $|V'|$ in der gefundenen stabilen Menge.

Ziel Die Maximierung vom Maß.

2.2.3. Die c-Reduktion

Definition 17 (c-Reduktion [Sim90])**.** *Eine* c-Reduktion *(engl.* Continuus Reduction*) ist eine Beziehung zweier Optimierungsprobleme $P1$ und $P2$. Sie ist gegeben durch ein 2-Tupel (f, g). Bei einer c-Reduktion von $P1$ auf $P2$ müssen folgende Bedingungen erfüllt sein:*

1. *Für alle Instanzen x von $P1$ gilt $f(x)$ ist eine Instanz zu $P2$.*
2. *Für alle $f(x)$, wobei x eine Instanz von $P1$ ist, und alle Approximationsalgorithmen zu $P2$ gilt, dass $g(x, A(f(x)))$ eine gültige Lösung zu der Instanz x sein muss. Hierbei muss es einen Faktor r geben, sodass folgendes gilt:*

$$R_{P1}\Big(x, g\big(x, A(f(x))\big)\Big) \leq r \cdot R_{P2}\Big(f(x), A\big(f(x)\big)\Big).$$

Notation 1. *Falls eine c-Reduktion von $P1$ auf $P2$ mit der Faktor r existiert, kann dieses folgendermaßen ausgedrückt werden: $P1 \leq_c^r P2$. Die Schreibweise $P1 \leq_c^{1+\varepsilon} P2$ bedeutet, dass es für jedes $\varepsilon > 0$ eine c-Reduktion von $P1$ auf $P2$ mit $r = 1 + \varepsilon$ gibt.*

3 | Maximum Clique

Als Erstes wird das bekannteste Clique-Optimierungsproblem Maximum Clique (MaxClique) behandelt. Zunächst folgt eine Definition des Problems (vgl. [CK97, Kapitel MaxClique]).

Instanz Ein Graph $G = (V, E)$.

Lösung Eine Clique aus G, also eine Untermenge $V' \subseteq V$ in der jeder Knoten mit jedem anderen enthaltenen Knoten (außer sich selbst) verbunden ist.

Maß Die Anzahl der Knoten $|V'|$ in der gefundenen Clique.

Ziel Die Maximierung vom Maß.

Dieses Problem ist, wie alle hier behandelten Probleme, in die Gruppe der kombinatorischen Optimierungsprobleme einzuordnen. Bezüglich der Beziehungen zu anderen Optimierungsproblemen sind vor allem andere Clique-Optimierungsprobleme und Probleme, die auf dem Suchen von stabilen Mengen basieren, von Interesse. Am wichtigsten ist hierbei die Äquivalenz von MaxClique zu MIS auf dem komplementären Graphen. Weitere Relationen sind in Kapitel 8 zu finden.

Bevor verschiedene effiziente und exakte Lösungsmöglichkeiten des Problems diskutiert werden, folgt ein Überblick über die Approximierbarkeit des Problems.

3.1. Approximierbarkeit

Als Erstes wird das Problem MaxClique in die Hierarchie der Optimierungsprobleme eingeordnet. Die erste Erkenntnis ist, dass es in der Optimierungsklasse $\mathcal{NPO}$ liegt.

Beweis. Zuerst ist zu zeigen, dass $\{ (x,\ y) : y \in s(x) \} \in \mathcal{P}$, das bedeutet jede Lösung des Problems muss polynomiell überprüfbar sein. Die Überprüfung der Richtigkeit jeder Lösung kann in $\mathcal{O}(n^2)$ stattfinden: Es wird für jeden Knoten in der Lösungsmenge geprüft, ob dieser mit allen anderen Knoten in der Menge über eine Kante verbunden ist.

Weiter muss es ein Polynom p geben, sodass für alle Instanzen und deren Lösungen gilt, dass deren Länge durch p beschränkt ist. Da jede Lösungsmenge eine Untermenge

der Knoten ist und die Codierung pro Zeichen $\log |V|$ Platz benötigt, ist jede Lösung durch $\mathcal{O}(|V| \cdot \log |V|)$ beschränkt.

Die Maßfunktion muss polynomiell berechenbar sein. Hier ist das Maß lediglich die Kardinalität der Knotenmenge und somit natürlich auch in polynomieller Zeit berechenbar (Zählen der Knoten liegt in $\mathcal{O}(n)$). □

Das Problem liegt nicht in $\mathcal{APX}$, da kein c-Approximationsalgorithmus existiert (ausgehend von $\mathcal{P} \neq \mathcal{NP}$), denn Håstad hat gezeigt [Hås96], dass MaxClique nicht innerhalb von $|V|^{1/2-\varepsilon}$ für jedes $\varepsilon > 0$ approximierbar ist.

Die aktuell beste bekannte Performanzrate ist $\mathcal{O}(n(\log\log n)^2/(\log n)^3)$. Sie wurde 2004 von U. Feige mittels eines Algorithmus entdeckt [Fei04]. Auf diesen wird später im Abschnitt 3.2.1 genauer eingegangen.

Die Tatsache, dass kein c-Approximationsalgorithmus existiert, ist eher unbefriedigend, daher werden häufig eingeschränkte Graphen zur Lösung des Problems verwendet. Im Folgenden ist eine Übersicht über die erreichbaren Performanzraten und deren zugehörige Laufzeitklasse, auf ausgewählten eingeschränkten Graphen zu finden:

Tabelle 3.1.: Laufzeit & Performanz auf eing. Graphen

Graphenklasse	Performanzrate	Laufzeit	Referenz
unit disk	1	polynomiell	[CCJ90]
overlap			[Rid]
circle			[Rid]
(co-)bipartite			[Rid]
t-interval	t	polynomiell	[FGO15]
t-track			
t-circular interval	$2t$	linear	
t-circular track			

Es ist zu sehen, dass MaxClique in relativ vielen bekannten Graphen (unit disk, overlap, circle, (co-)bipartite) exakt polynomiell lösbar ist, somit liegt auch das zugehörige Entscheidungproblem auf diesen Graphen in $\mathcal{P}$. Des Weiteren gibt es eine Reihe von Graphen (t-interval, t-track, t-circular interval, t-circular track) auf denen das Problem einen c-Approxmiationsalgorithmus besitzt und somit in $\mathcal{APX}$ liegt. Zu t-Intervall Graphen wird im Abschnitt 3.2.2 ein Algorithmus vorgestellt.

3.2. Approximationsalgorithmen

Nachdem die Approximierbarkeit von MaxClique betrachtet wurde, werden hier die Möglichkeiten zur Approximation untersucht. Der Vorteil von Approximationsalgorithmen ist, dass sie, im Vergleich zu heuristischen Algorithmen, eine bewiesene obere Schranke aufweisen.

Der bedeutendste Approximationsalgorithmus, da er die beste obere Schranke aufweist, ist der Algorithmus von Feige [Fei04], welcher wiederum aus mehreren verschiedenen Algorithmen besteht. Weiter zu erwähnen ist der Algorithmus von Boppana und Halldórsson [BH92], welcher als gedanklicher Vorgänger zum Algorithmus von Feige betrachtet werden kann. Im Folgenden wird ein Teil von Feiges Algorithmus beschrieben und der Gedankengang, der schließlich zur besten bekannten oberen Schranke führt, skizziert. Weiter gibt es einige Approximationsalgorithmen auf eingeschränkten Graphen (siehe Tabelle 3.1).

3.2.1. Algorithmus von Feige

Der Algorithmus wurde 2004 von U. Feige publiziert [Fei04] und basiert auf dem Entfernen von Teilgraphen. Er dient eher dazu, die obere Schranke von MaxClique zu erläutern, als tatsächlich Verwendung zu finden.

Definition 18 (Schwacher Teilgraph)**.** *Im Folgenden wird ein Teilgraph S* schwach *genannt, falls er keine Clique mit der Mindestgröße $|S|/2k$ enthält.*

Notation 2 ($N(S_{ij})$)**.** *Die Funktion $N(S_{ij})$ liefert eine Knotenmenge $\subseteq V'' \setminus S_{ij}$, bei der jedes Element zu allen Knoten in S_{ij} eine Verbindung haben muss.*

Algorithmus 1: Feiges Algorithmus

```
1:  procedure FEIGE(G, t, k)                                   ▷ G = (V, E)
2:      G' := G                                                ▷ G' = (V', E')
3:      G'' := G'                                              ▷ G'' = (V'', E'')
4:      C := ∅
5:      OuterWhile:                                            ▷ Schleifenmarkierung
6:      while true do
7:          if |V''| < 6kt then
8:              return C
9:          p = |V''|/2kt
10:         P_{1...p} := Disjunkte Partitionen von V'' mit jeweils der Länge 2kt
11:         for P_i in P do
12:             for all subsets S_ij in P_i with |S_ij| = t do
13:                 if IT(S_ij) is a clique ∧ |N(S_ij)| ≥ |V''|/2k − t then
14:                     C := C ∪ S_ij
15:                     G'' := IT(N(S_ij))
16:                     continue OuterWhile                    ▷ Beende aktuelle Iteration
17:         G', G'' := IT(V' \ V'')
```

Der Algorithmus basiert grundlegend auf das Finden und Entfernen von schwachen Teilgraphen. Im Zuge der Überprüfung auf schwache Teilgraphen wird dann eine möglichst große Clique gebildet. Es wird vom Algorithmus ein Graph mit einer Clique der Größe n/k, t, und k als Eingabe erwartet. In der Regel wird $t = \Theta(\log n / \log\log n)$ gesetzt, um eine polynomielle Laufzeit zu garantieren (siehe Abschnitt „Laufzeit“) [Fei04, Kapitel 2].

Der Algorithmus wird in Phasen und Iterationen eingeteilt, hierbei gibt es mindestens eine Phase, welche mehrere Iterationen enthalten kann. Es folgt eine Beschreibung dieser beiden Teile.

PHASE: Eine Phase führt maximal $|V'|/t$ Iterationen aus. Innerhalb einer Phase wird auf G', einem Teilgraphen von G, operiert (in der ersten Phase gilt $G' = G$). Sie kann aus zwei Gründen beendet werden, zum einen wenn eine Clique mit der gewünschten Größe $t \log_{3k}(|V'|/6kt)$[Fei04] gefunden wurde, oder falls die aktuelle Iteration mit G'' einen schwachen Teilgraphen gefunden hat. Im letzteren Fall folgt die nächste Phase.

ITERATION: Eine Iteration operiert auf G'', einem Teilgraphen von G' (in der ersten Iteration einer Phase gilt $G'' = G'$). Sie fängt damit an, zu prüfen, ob die aktuelle Knotenmenge V'' kleiner als $6kt$ ist. Falls dem so ist, kann die Ergebnisclique C ausgegeben werden und der Algorithmus terminiert. Falls das nicht der Fall ist, wird V'' in $p = |V''|/2kt$ disjunkte Partitionen gespalten. Jede dieser Partitionen muss eine Länge von $2kt$ haben. Es wird im originalen Algorithmus davon ausgegangen, dass $2kt$ $|V''|$ ohne Rest teilt, in der Praxis muss diese Annahme aufgegeben werden (siehe Abschnitt 3.5). Danach wird jede Partition P_i $(0 < i \leq p)$ weiter aufgeteilt in alle möglichen Teilmengen S_{ij} der Länge t $(0 < j \leq \binom{p}{t})$. Diese werden dann einzeln untersucht. Falls der induzierte Teilgraph von S_{ij} eine Clique ist und falls es in V'' mindestens $|V''|/2k - t$ Knoten gibt, die mit allen Knoten in S_{ij} verbunden sind, kann die Ergebnisclique C um S_{ij} erweitert werden. Der induzierte Teilgraph von $N(S_{ij})$ bildet dann den neuen Graphen G”, die aktuelle Iteration wird beendet und eine neue beginnt. Hierbei ist anzumerken, dass es genau eine Phase gibt, in der dies geschieht.

In dem Fall, dass kein S_{ij} die genannte Bedingung (Zeile 13 im Pseudocode) erfüllt, wird V'' als schwach markiert und die aktuelle Phase beendet. Die neue Phase wird dann mit dem induzierten Teilgraphen von $V' \setminus V''$ als G' begonnen und G'' wird wieder auf G' gesetzt.

Um den Algorithmus besser zu verstehen, werden die Bedingungen in Zeile 7 und 13 (Teil nach „$\wedge$“) untersucht, angefangen mit Zeile 7:

$$|V''| < 6kt$$

Durch die Bedingung wird garantiert, dass die gefundene Clique mindestens $t \cdot \log_{3k}(|V'|/6kt)$ groß ist. Die Größe setzt sich wie folgt zusammen: Jede Iteration fügt t Knoten zu C hinzu und es werden mindestens $\log_{3k}(|V'|/6kt)$ Iterationen benötigt, um

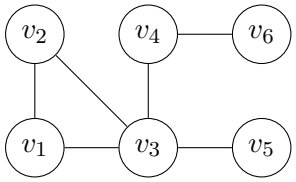
$|V''|$ auf $6kt$ zu reduzieren. Die Anzahl der Iterationen ergibt sich durch die minimale Größe von $|N(S_{ij})|$ nach jeder Iteration. Diese ergibt sich in Zeile 13:

$$|N(S_{ij})| \geq \frac{|V''|}{2k} - t$$

Wenn nun $t < {}^{|V''|}/_{6k}$, dann folgt, dass $|V''|$ nach mindestens $\log_{3k}({}^{|V'|}/_{6kt})$ Iterationen die Größe $6kt$ annimmt [Fei04, Lemma 2.2, 2.4].

Beispiel

$$G = (\{ v_1, v_2, v_3, v_4, v_5, v_6 \}, E)$$
$$\text{mit } E = \left\{ \begin{matrix} \{ v_1, v_2 \}, \{ v_2, v_3 \}, \{ v_1, v_3 \}, \\ \{ v_4, v_6 \}, \{ v_5, v_3 \}, \{ v_4, v_3 \} \end{matrix} \right\}$$



Um den Algorithmus zu starten wird G verwendet, außerdem müssen noch k und t bestimmt werden: Für k wird 2 gewählt, da ${}^{6(\text{Größe vom Graphen})}/_{3(\text{Größe der max. Clique}))} = 2$, t wird auf 0.1 gesetzt (es muss gelten $t \ll {}^{n}/_{k}$).

Zunächst beginnt die erste Phase: G' hat anfangs den Wert G, C ist leer und G'' ist initial auf G' gesetzt. Es beginnt die erste Iteration der ersten Phase.

Der Wert von $|V''|$ beträgt 6 und ist somit kleiner als $6 \cdot 2 \cdot 0.1 = 1.2$, dementsprechend kann noch nicht terminiert werden. V'' wird in $P_{1\ldots6}$ aufgeteilt. Hierbei muss die Größe der Partitionen aufgerundet werden.

$$P_1 = \{ v_1 \}, \; P_2 = \{ v_2 \}, \; P_3 = \{ v_3 \}$$
$$P_4 = \{ v_4 \}, \; P_5 = \{ v_5 \}, \; P_6 = \{ v_6 \}$$

Für jeden dieser Partition werden alle Teilmengen betrachtet, die eine Größe von t (hier 0.2, aufgerundet 1) haben. Da die Partitionen den Teilmengen bereits entsprechen, können diese betrachtet werden. Es müssen alle Knoten einzeln auf die Bedingung in Zeile 13 geprüft werden. $P_1 = \{ V_1 \}$ erfüllt diese, daher kann auf die Überprüfung der anderen Mengen verzichtet werden. Nun wird v_1 zu C hinzugefügt, G'' auf $IT(\{ v_2, v_3 \})$ gesetzt und die nächste Iteration beginnt.

Anfangs wird erneut die Abbruchbedingung geprüft: Da $6kt < 2$, wird V'' erneut partitioniert, und zwar in folgende Mengen:

$$P_1 = \{ v_2 \}, \; P_2 = \{ v_3 \}$$

Auch hier entsprechen die Teilmengen S_{ij} den Partitionen. Nun wird für v_2 die Bedingung in Zeile 13 geprüft, welche zu `true` ausgewertet wird. Daher muss C um v_2 erweitert werden, G'' wird auf $IT(\{ v_3 \})$ gesetzt und die nächste Iteration beginnt.

Hier gilt nun $6kt > 1$, daher terminiert der Algorithmus schon in der ersten Phase und gibt folgendes Ergebnis aus:

$$C = \{ v_1, v_2 \}$$

Dieses ist um den Faktor 1.5 schlechter als das optimale Ergebnis. Für noch niedrigere t kann sogar eine optimale Lösung erreicht werden. Der Algorithmus funktioniert mit der von Feige [Fei04] vorgeschlagenen Wahl von $t = {}^{\log n}/_{\log\log n}$ für kleine n nicht gut, denn bis einschließlich $n = 25$ terminiert der Algorithmus sofort in der ersten Iteration und liefert somit eine leere Menge als Lösung.

Laufzeit & Performanz

Für obigen Algorithmus gibt Feige [Fei04] an, dass immer eine Clique der Größe $\Omega({}^{\log n}/_{\log\log n})$ gefunden wird. Falls es eine Clique der Größe $\mathcal{O}({}^{n}/_{\log n})$ gibt, dann beträgt die Performanzrate sogar $\mathcal{O}({}^{n(\log\log n)^2}/_{(\log n)^3})$. Um nun für alle Fälle diese Performanzrate zu erreichen, trifft Feige folgende Fallunterscheidung für Graphen mit einer Clique der Größe ${}^{n}/_{k}$:

$$\textsc{MaxClique}(G, k) = \begin{cases} \mathit{HB}(G, k), & \text{falls } k \leq \frac{\log n}{2\log\log n} \\ \mathit{MFeige}(G, k), & \text{falls } \frac{\log n}{2\log\log n} < k < \log n \\ \mathit{Feige}(G, k), & \text{falls } \log n \leq k \leq (\log n)^3 \\ \mathit{OneVertex}(G, k), & \text{falls } k > (\log n)^3 \end{cases} .$$

$\mathit{HB}(G, k)$ sei hierbei der Algorithmus von Halldorsson und Boppana, den sie 1992 beschrieben haben [BH92]. $\mathit{OneVertex}(G, k)$ gebe einen zufälligen Knoten aus. $\mathit{MFeige}(G, k)$ ist eine leicht veränderte Version des obenstehenden Algorithmus von Feige, beschrieben in [Fei04, Kapitel 3].

Als Laufzeit wird von Feige eine Komplexität von $\mathcal{O}(\binom{2kt}{t} n^c)$ mit einer Konstanten c angegeben. Der Binomialkoeffizient resultiert aus der Zerlegung von V'' in Partitionen und weiter in die Teilmengen S_{ij}. Nun müssen, um eine polynomielle Laufzeit zu garantieren, k und t eingeschränkt werden. Es sei $k \leq (\log n)^b$ für eine Konstante $b > 0$, womit, wenn $t = {}^{\log n}/_{\log\log n}$, eine polynomielle Laufzeit für diese k sichergestellt ist.

Schlussbemerkung

Bei dem betrachteten Algorithmus ergeben sich einige praktische Probleme, denn obwohl der Algorithmus eine relativ hervorragende Performanzrate (zumindest auf einem Intervall von k) liefert, so ist er in der Praxis schwierig nutzbar, vor allem für kleine n, da er in dem Bereich, bei Feige's vorgeschlagenen Funktion für t, sehr schwache Ergebnisse liefert. Bei obigem Beispiel findet der Algorithmus für Feige's Wahl

von t beispielsweise gar keine Clique, da er sofort bei der ersten Abbruchbedingung terminiert. Folglich sollte in diesem Bereich t anders gewählt werden. Nebenbei sei gesagt, dass auch die Voraussetzung, das k zu kennen, suboptimal ist, da dieser Wert in der Regel unbekannt ist und somit schlimmstenfalls alle Werte von k ($0 < k \leq |V|$) getestet werden müssen. Allerdings ist der Algorithmus anderen polynomiellen Algorithmen, was die garantierte Qualität der Lösung angeht, überlegen. Dennoch wird hier deutlich, warum Maximum Clique als hart approximierbar gilt.

Es gibt zwei Lösungen, um bessere Ergebnisse in polynomieller Zeit zu erzielen. Eine Möglichkeit besteht darin, heuristische Algorithmen zu nutzen. Bei der anderen wird die Instanzmenge von MaxClique eingeschränkt, also ein eingeschränkter Graph verwendet.

3.2.2. Algorithmus von Francis, Gonçalves und Ochem

Nachdem ein Approximationsalgorithmus gezeigt wurde, der eine insgesamt schwache Performanzrate liefert, wird nun ein c-Approximationsalgorithmus erläutert, der MaxClique auf k-Intervallgraphen löst (k sei eine natürliche Zahl). Es handelt sich um den Algorithmus von Francis, Gonçalves und Ochem [FGO15].

Das Prinzip des Algorithmus ist simpel: Im ersten Schritt wird aus dem Graphen (inkl. Modell) ein gerichteter Graph mit gefärbten Kanten erstellt, um dann im zweiten Schritt aus jeweils immer einem Knoten und deren, auf einem Farbpaar, erreichbaren Nachbarn die maximale Clique des induzierten Teilgraphen zu berechnen.

1. Schritt: Es wird über alle Kanten des Graphen iteriert und jeder Kante wird eine Richtung zugewiesen. Die Abbildungen $I_1, \ldots, I_t$ beschreiben die Intervalle eines jeden Knoten und dienen als Modell für den Graphen $G = (V, E)$. Nun wird für jede Kante $\{ u, v \} \in E$ eine Richtung bestimmt. Falls es ein i und j gibt ($0 < i, j \leq t$), sodass $u_i \in I_j(v)$, verläuft die neue Kante von u nach v und wird mit der Farbe i gefärbt. Falls dieses nicht der Fall ist, verläuft die Kante genau anders herum und der Kante wird die Farbe j zugewiesen. Die entstehende Kantenmenge heißt K. Die Funktion, die jede Kante auf die Farbe abbildet, heißt C.

2. Schritt: Es werden alle Knoten, die in der Kantenmenge K vorkommen, zusammen mit jeder beliebigen Farbkombination aus zwei verschiedenen Farben i und j betrachtet, und eine neue Knotenmenge aus dem betrachteten Knoten u und deren Nachbarn ${N_i}^+(u)$ und ${N_j}^+(u)$ gebildet. Aus der resultierenden Knotenmenge wird der induzierte Teilgraph gebildet und dessen maximale Clique berechnet. Hierbei sei bemerkt, dass es sich bei $IT(s)$ um einen Graphen handelt, der von genau zwei Cliquen überdeckbar ist, weshalb die Berechnung der maximalen Clique in polynomieller Zeit möglich ist [Rid]. Die berechnete Clique wird gespeichert, falls noch keine größere Clique gefunden wurde. Sobald dieses für alle Knoten gemacht wurde, terminiert der Algorithmus.

Algorithmus 2: Algorithmus von Francis, Gonçalves und Ochem

```
1:  procedure FGO(G, (I_1, ..., I_t))                          ▷ G = (V, F)
2:      K := ∅
3:      C: G.V → { 1, ..., t }
4:      for all edges e in G.E do
5:          u ∈ e
6:          v ∈ e \ { u }
7:          Es gilt I_i(u) = [u_i, u'_i]
8:          if Es existiert ein i, j sodass u_i ∈ I_j(v) then
9:              oe := { u → v }
10:             C(oe) = i
11:         else
12:             oe := { v → u }
13:             C(oe) = j
14:         K := K ∪ oe
15:     max := ∅
16:     for all vertices u in K do
17:         for all i = 1, ..., t do
18:             for all j = i, ..., t do
19:                 if i == j then
20:                     continue
21:                 s := { u } ∪ N_i^+(u) ∪ N_j^+(u)
22:                 c :=maxClique(IT(s))
23:                 if |c| > |max| then
24:                     max := c
25:     return max
```

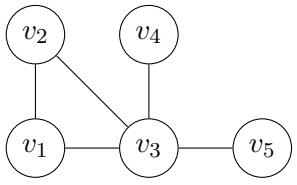
$N_i{}^+(u)$: Diese Funktion liefert eine Menge aus Knoten, zu denen von u aus eine mit Farbe i gefärbte Kante führt.

Beispiel

Als Beispiel zum Algorithmus dienen folgender Graph und Modell als Eingabe:

$$G = (\{ v_1, v_2, v_3, v_4, v_5 \}, E)$$

$$\text{mit } E = \left\{ \begin{matrix} \{ v_1, v_2 \}, \{ v_2, v_3 \}, \{ v_1, v_3 \}, \\ \{ v_5, v_3 \}, \{ v_4, v_3 \} \end{matrix} \right\}$$



Das Modell sei ein 3-Intervall Graph:

$I_1 = \{ v_1 \to [0, 1], v_2 \to [5, 6], v_3 \to [2, 3], v_4 \to [7, 8], v_5 \to [3, 4] \}$
$I_2 = \{ v_1 \to [1.5, 2], v_2 \to [11, 12], v_3 \to [4, 4.5], v_4 \to [50, 60], v_5 \to [42, 43] \}$
$I_3 = \{ v_1 \to [10, 11], v_2 \to [3, 3.5], v_3 \to [15, 16], v_4 \to [16, 17], v_5 \to [20, 21] \}$

Im ersten Schritt wird aus dem Beispielgraph G ein gerichteter Graph mit gefärbten Kanten, wie in den Zeilen 4-14 beschrieben, erstellt. Die erste Kante ist $\{ v_1, v_2 \}$ (die obige Variable u sei immer der erste Knoten der Kante), hierbei wird geprüft, welche Intervalle dieser Knoten sich schneiden. Es schneiden sich [11, 12] von v_2 aus dem Intervall I_2 und [10, 11] von v_1 aus I_3. Somit muss die Kante von v_2 nach v_1 verlaufen, da 11 (untere Grenze von I_2) in [10, 11] liegt. Die Farbe der Kante ist somit 2. Nachdem dies für alle Kanten durchgeführt wurde, entsteht letztlich folgender Graph:

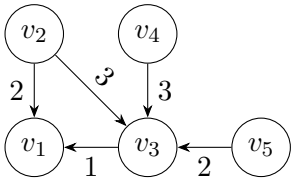


Abbildung 3.1.: Graph K mit Farben

Nun beginnt der zweite Schritt (Zeilen 17-31). Es wird jeder Knoten bei allen Farbpaaren (i, j) mit $i \neq j$, wobei die Reihenfolge (von i und j) irrelevant ist, betrachtet (Es gibt drei Paare: (1, 2), (1, 3) und (2, 3)). Zuerst wird v_1 betrachtet, da der Knoten keine ausgehenden Kanten hat, besteht die zu bildende Knotenliste (vgl. Zeile 24) bei jedem Farbpaar nur aus dem Knoten selbst und hat somit eine Clique der Größe 1 auf dem induzierten Graphen. Bei Betrachtung von v_2 resultieren folgende induzierte Graphen:

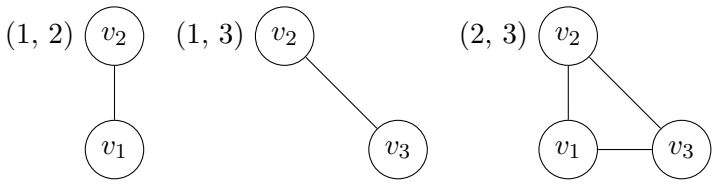


Abbildung 3.2.: Induzierte Graphen bei Betrachtung von v_2

Es ist ersichtlich, dass hier die maximale Cliquegröße 3 beträgt. Diese kann einfach für jeden Knoten wiederholt werden: v_3 liefert $\{ v_3 \}$ und $\{ v_3, v_1 \}$. Offenbar hat keiner der daraus induzierten Teilgraphen eine größere Clique als 3. Der Knoten

v_4 liefert $\{ v_4 \}$ und $\{ v_4, v_3 \}$, deren induzierte Teilgraphen ebenfalls keine größeren Cliquen hervorbringen. Zum Schluss bleibt v_5 zu betrachten, welcher sich selbst und $\{ v_3, v_5 \}$ als Knotenmenge liefert und somit kann hier auch keine größere Clique gefunden werden. Die Lösung ist somit $\{ v_2, v_1, v_3 \}$ und dementsprechend hier sogar optimal.

Laufzeit & Performanz

Die Laufzeit ist polynomiell beschränkt. Der erste Schritt ist in quadratischer Zeit ausführbar, da ein Intervall-Graph im ungünstigsten Fall $n(n-1)/2$ Kanten hat. Die Schritte, die von Zeile 5-14 ausgeführt werden, sind in konstanter Zeit durchführbar.

Der zweite Schritt ist in kubischer Laufzeit möglich. Es finden n Durchläufe statt und die Anzahl der inneren Schleifen ist nur abhängig von t und somit nicht relevant. In einem Durchlauf wird die maximale Clique eines co-biparten Graphen berechnet, was in quadratischer Zeit durchführbar ist [Rid, Class: co-bipartite]. Die restlichen Operationen in einem Durchlauf sind maximal linear beschränkt.

Es folgt eine Gesamtlaufzeit von $\mathcal{O}(n^3)$.

Der Algorithmus ist ein c-Approxminationsalgorithmus und hat, falls die Eingabe ein t-Intervall-Graph ist, eine Performanzrate von t. Der Knoten, der die beste gefundene Clique liefert, hat mindestens $1/2(\omega(G) - 1)$ ausgehende Kanten, da dieser mindestens genauso viele ausgehende wie eingehende Kanten haben muss [FGO15]. Nun werden hiervon $2/t$ betrachtet („2“ wegen dem Farbpaar und „$1/t$“ da es insgesamt t Farben gibt), womit bei der größten in Schritt 2 gefundenen Clique eine Mindestgröße von $1 + 1/t(\omega(G) - 1) > 1/t(\omega(G))$ [FGO15] folgt. Folglich handelt es sich um einen t-Approximationsalgorithmus.

Schlussbemerkung

Der Algorithmus ist ein gutes Beispiel dafür, dass eine maximale Clique auch mittels Approximationsalgorithmus, unter gewissen Einschränkung, in polynomieller Laufzeit gefunden werden kann. Als allgemeine Lösung für MaxClique eignet sich der Algorithmus aber nicht, da nicht immer ein Modell für einen t-Intervall-Graphen gegeben bzw. leicht berechenbar ist. Denn leider ist das Problem ein t-Intervall-Modell für beliebige Graphen zu finden – vor allem falls das t minimal sein soll – NP-vollständig [WS84], weshalb obiger Algorithmus nicht in allen Fällen effizient einsetzbar ist.

3.3. Exakte Algorithmen

Nachdem die Möglichkeiten der Approximation untersucht wurde, wird jetzt die nächste Algorithmengruppe betrachtet: die exakten Algorithmen. Im Folgenden wird eine kurze Übersicht über wichtige exakte Algorithmen gegeben, danach wird der repräsentative Algorithmus MCQ [TS03] vorgestellt.

Wenn es um exakte Algorithmen zu MaxClique geht, dann muss auch gleichzeitig über das Branch-and-Bound (B&B) Prinzip geschrieben werden. Viele dieser exakten Algorithmen arbeiten nach diesem, es besagt hier, dass der Suchraum, also der Graph, aufzuteilen ist (Branching) und bestimmte Zweige abzuschneiden sind (Bounding). Nun wird also die Effizienz eines solchen exakten Algorithmus vor allem dadurch bestimmt, wie der Suchraum geteilt wird und durch welche Überlegungen entschieden wird, welche Teile nicht weiter betrachtet werden müssen. Alle im Folgenden angeschnittenen Algorithmen arbeiten nach diesem Prinzip.

Am Anfang der Geschichte von modernen exakten Algorithmen steht der Algorithmus CP. Er wurde 1990 von Carraghan und Pardalos [CP90] beschrieben. Dieser liefert das Grundgerüst von B&B. Hier wurden Zweige mittels der simplen Bedingung, dass die aktuelle Clique und die weiter zu betrachtenen Knoten nicht größer als die beste schon gefundene Clique sein darf, abgeschnitten.

Im selben Jahr wurde ebenfalls der erste B&B Algorithmus mit Teilgraph-Färbungen BT von Babel und Tinhofer [BT90] vorgestellt. In diesem wird das Bounding gegenüber CP optimiert in dem durch ein Coloring auf betrachteten Teilgraphen eine obere und untere Schranke für die Cliquengröße errechnet wird.

Im Jahre 2003 wurde dieses Prinzip durch Tomita's und Seki's MCQ [TS03] erweitert. Dieser wurde in den folgenden Jahren (2007, 2010) mehrfach durch Tomita [WH15] zum MCR und dann MCS Algorithmus durch bessere Vorsortierungen verbessert.

Ebenfalls von Interesse ist der Algorithmus MaxCLQ von Li und Quan, der einen ganz neuen Ansatz einbringt. Hier wird ein MaxSat Algorithmus benutzt, um die obere Grenze der Cliquengröße weiter zu verbessern, genaueres ist [LQ10] zu entnehmen.

3.3.1. MCQ

Im Folgenden wird der Algorithmus MCQ vorgestellt, der von Tomita und Seki im Jahre 2003 veröffentlicht wurde ([TS03]).

Der Algorithmus nutzt zwei verschiedene Mengen: die Kandidaten R und die aktuell gefundene Clique Q. Des Weiteren benötigt er einen Graphen G als Eingabe. Schließlich wird ein Zuordnung N der Länge $|V|$ benötigt, welche jedem Knoten eine Farbklasse zuweist.

Nun werden die drei verschiedenen Teile vom Algorithmus erläutert. Es sei angemerkt, dass im Folgenden und im Pseudocode davon ausgegangen wird, dass Mengen geordnet sein können.

Initialisierung: Zum Starten von MCQ wird die definierte Methode MCQ aufgerufen. Danach wird zuerst V nach Knotengrad absteigend sortiert. Als nächstes wird N initialisiert mithilfe eines Greedy-Coloring Algorithmus, woraufhin V nach den Farbklassen sortiert wird (für genaueres siehe Abschnitt „Bounding“). Schließlich wird die Methode Expand mit der geordneten Kandidatenliste V und der Fäbung N aufgerufen, dieses führt zum nächsten Abschnitt: dem Branching.

Branching: Für das Branching wird die Kandidatenliste rückwärts durchlaufen und nach dem Testen der Bounding Bedingung (siehe Abschnitt C) wird der aktuelle Kandidat in die Clique Q eingefügt. Nun wird eine eingeschränkte Kandidatenliste erstellt, denn es können nun nur noch Kandidaten in Frage kommen, die mit dem aktuellen Kandidaten verbunden sind. Falls noch Kandidaten existieren, also die neue Kandidatenliste R_p größer ist als 0, wird für die neue Kandidatenliste eine neue Färbung N' erstellt, R_p dementsprechend sortiert und durch den rekursiven Aufruf von Expand(R_p, N') wird das Branching mit den neuen Parametern ausgeführt. In dem Fall, dass es keine Kandidaten mehr gibt, wird geprüft, ob die Lösung die aktuell beste ist und sie wird in Q_{max} gespeichert. Am Ende der Schleife wird nun der aktuelle Kandidat p aus Q und R entfernt, um den nächsten Kandidaten ausprobieren zu können. Es ist erkennbar, dass der Algorithmus einem Suchalgorithmus für Bäume ähnelt, er basiert auf das Erstellen von aussichtsreichen Verzweigungen und dessen Durchsuchung nach einer optimalen Lösung.

Bounding: Während das Branching im Grunde in vielen B&B Algorithmen so ähnlich durchgeführt wird, stellt dieser Teil die Besonderheit des Algorithmus dar. Das Bounding wird über eine nicht optimale Lösung zum Coloring-Problem durchgeführt. Die Methode Number-Sort(R, N) berechnet diese Lösung mit einem Greedy-Algorithmus. Daraufhin sortiert die Methode die geordnete Menge R aufsteigend nach den Farbklassen, falls zwei Knoten die gleiche Klasse haben, entscheidet der ursprüngliche Index über die Sortierung, dies bedeutet, dass die Vorsortierung in diesem Fall entscheidend ist. Es folgt eigentlichen Bedingung:

$$|Q| + N[p] \leq |Q_{max}|$$

Falls diese Bedingung erfüllt wird, kann der Branch geschlossen werden. Es wird also die Suche für den aktuellen Kandidaten abgebrochen und die folgenden Kandidaten werden nicht weiter betrachtet. Wegen des rückwärtigen Durchlaufs (und wegen des Entfernens von p am Ende der Schleife) durch die Kandidatenliste, der bewirkt, dass der aktuelle Kandidat eines Branches immer die höchste Farbklasse aller Kandidaten in R hat, ist dies legitim. Da die maximale Cliquengröße in R durch die Anzahl der Farbklassen (folgt aus der Definition eines Graph-Colorings) beschränkt sein muss ($\omega(R) \leq |N|$), kann ein Branch nicht aussichtsreich sein, wenn er nicht die Möglichkeit bietet, die aktuelle Clique um genügend Knoten zu erweitern, um schließlich eine maximale Clique zu bilden (für eine genauere Betrachtung siehe [TS03]).

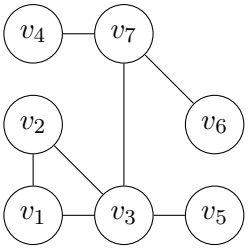
Algorithmus 3: MCQ

```
1:  global Q := ∅                                          ▷ Aktuelle Clique
2:  global Q_max := ∅                                      ▷ Größte gefundene Clique
3:  global E_g := ∅
4:
5:  procedure MCQ(V, E)                                    ▷ G := (V, F)
6:      N := ∅                                             ▷ Farbklassen der Knoten
7:      E_g := E
8:      Sortiere Knotenmenge V absteigend nach dem Knotengrad
9:      NUMBER-SORT(V, N)
10:     EXPAND(V, N)
11:     return Q_max
12:
13: procedure EXPAND(R, N)
14:     for p in R ordered-reverse do                      ▷ Menge wird rückwärts durchlaufen
15:         if |Q| + N[p] ≤ |Q_max| then                   ▷ Zentrale Heuristik
16:             return
17:         Q := Q ∪ { p }
18:         R_p := R ∩ { x : x ∈ V, { x, p } ∈ E }
19:         if |R_p| > 0 then
20:             N' := ∅
21:             NUMBER-SORT(R_p, N')
22:             EXPAND(R_p, N')
23:         else if |Q| > |Q_max| then
24:             Q_max := copy(Q)                           ▷ Q muss hier kopiert werden
25:         R, Q \= { p }
26:
27: procedure NUMBER-SORT(R, N)
28:     max := 1
29:     C_1, ··· , C_|V| := ∅
30:     for p in R ordered do                              ▷ Greedy-Coloring
31:         k := 1
32:         while C_k ∩ { x : x ∈ V, { x, p } ∈ E } ≠ ∅ do
33:             k++
34:         if k > max then
35:             max := k
36:         N[p] := k
37:         C_k := C_k ∪ {p}
38:         R := R \ {p}
39:     i := 1
40:     for k to max do                                    ▷ Neusortierung
41:         for j to |C_k| do
42:             R[i] := C_k[j]
43:             i++
```

Beispiel

Es wird der folgende Graph als Eingabe verwendet.

$$G = (\{ v_1, v_2, v_3, v_4, v_5, v_6, v_7 \}, E)$$
$$E = \left\{ \begin{array}{c} \{ v_1, v_2 \}, \{ v_2, v_3 \}, \{ v_1, v_3 \}, \{ v_4, v_7 \}, \\ \{ v_6, v_7 \}, \{ v_5, v_3 \}, \{ v_7, v_3 \} \end{array} \right\}$$



Zuerst ist die Initialisierung durchzuführen. Hierbei müssen die Knoten nach dem Grad sortiert werden, es entsteht die folgende Menge: $\{ v_3, v_7, v_1, v_2, v_4, v_5, v_6 \}$. Daraufhin wird N initialisiert und V dadurch neu sortiert, es entstehen folgende Mengen:

$$N_i = \{ v_1 \to 2, v_2 \to 3, v_3 \to 1, v_4 \to 1, v_5 \to 2, v_6 \to 1, v_7 \to 2 \}$$
$$V_i = \{ v_3, v_4, v_6, v_7, v_1, v_5, v_2 \}$$

Jetzt wird EXPAND mit V und N aufgerufen. Der folgende Verlauf wird vereinfacht schematisch in Abbildung 3.3 dargestellt.

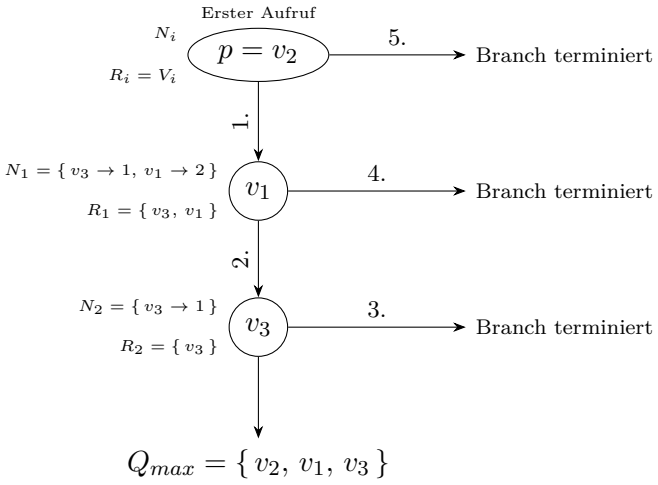


Abbildung 3.3.: Verlauf nach dem Aufruf von EXPAND

Wegen der rückwärtigen Iteration ist der erste betrachtete Kandidat v_2. Dieser erfüllt die Bounding-Bedingung (Zeile 15), folglich werden R und Q aktualisiert, N neu berechnet und EXPAND mit den neuen Mengen R_1 und N_1 aufgerufen. Nun wird

wieder der letzte Kandidat v_1 betrachtet, auch dieser Kandidat erfüllt die Bounding-Bedingung. Danach wird auch hier erneut R, Q und N wie im obigen Teil beschrieben, aktualisiert und EXPAND wird mit den Mengen R_2 und N_2 aufgerufen. Der nächste betrachtete Kandidat ist v_3, welcher wieder die Bounding-Bedingung erfüllt. Auf ein erneutes Aufrufen von EXPAND kann verzichtet werden, da R nach erneuter Aktualisierung leer ist, daher wird $Q = \{ v_2, v_1, v_3 \}$ an die Stelle Q_{max} kopiert, womit der Pfad fast beendet ist.

Letztlich werden noch die zusätzlichen Aufrufe von EXPAND sukzessive beendet (es wird bei Tiefe 1 noch eine Iteration angefangen und durch die Bounding-Bedingung direkt wieder beendet) und das Programm fährt mit der Iteration im ersten EXPAND Aufruf fort. Zuletzt wird der nächte Knoten in der Liste R_i untersucht, und zwar der Knoten v_5. Dieser erfüllt allerdings nicht die Bounding-Bedingung. Es wird also der letzte Aufruf von EXPAND abgebrochen und Q_{max} zurückgegeben, womit der Algorithmus erfolgreich durchlaufen wurde. Die Lösung lautet:

$$Q_{max} = \{ v_2, v_1, v_3 \}$$

Schlussbemerkung

MCQ liegt in $\mathcal{O}(2^n)$, denn es werden im ungünstigsten Fall alle Untermengen von V überprüft. Die Komplexität einer einzelnen Überprüfung ist polynomiell, also für die Gesamtkomplexität nicht relevant. Es gibt bei $|V| = n$ genau 2^n Untermengen, woraus schließlich eine Gesamtkomplexität von $\mathcal{O}(2^n)$ resultiert.

Im Geschwindigkeitsvergleich zu vielen anderen Algorithmen, wie dem simplen MC, liegt MCQ klar vorne, allerdings gilt er schon als veraltet und wurde durch Algorithmen wie MCS ersetzt. Er zeigt allerdings die Funktionisweise des Graph-Coloring Ansatzes, welcher auch heute noch als *state-of-the-art* gilt, und eignet sich dementsprechend als Einstieg in moderne, exakte Algorithmen für MAXCLIQUE [Pro12].

3.4. Heuristische Algorithmen

Nachdem mehrere approximative und exakte Algorithmen untersucht wurden, werden jetzt die heuristischen Algorithmen betrachtet. In der Literatur ist ein heuristischer Algorithmus im Vergleich zum Approximationsalgorithmus ein Algorithmus ohne beweisbare obere Schranke. Die Güte wird hierbei durch empirische Versuche festgestellt. Da MAXCLIQUE allerdings nur schwierig approximierbar ist, werden in der Literatur häufiger heuristische Algorithmen und seltener Approximationsalgorithmen betrachtet [WH15].

3.4.1. Heuristiken

Es gibt für MaxClique eine Vielzahl an Heuristiken und Metaheuristiken. Einige der am häufigsten verwendeten werden im Folgenden erläutert.

Greedy: Wie für die meisten Probleme, gibt es auch für MaxClique, Algorithmen die den Greedy-Ansatz verfolgen. In den meisten Fällen bedeutete dies, dass eine Clique sukzessive nach bestimmten Kriterien aufgebaut wird und dabei immer nur die nächstbeste Verbesserung gesucht wird. Algorithmen mit dieser Heuristik terminieren, falls kein weiterer Knoten mehr hinzugefügt werden kann.

Local-Search: Bei dieser Metaheuristik wird eine initiale Clique erstellt und durch eine sogenannte Nachbarschaftsfunktion, welche verwandte Lösungen enthält, iterativ weitmöglichst verbessert. Weiter wird die Heuristik in der Literatur oft in zwei unterschiedliche Kategorien aufgeteilt. Zum einen die *legal strategy* und zum anderen die *k-fixed penalty strategy*. Der Unterschied ist hierbei der Suchraum, bei ersterem dürfen alle gültigen Cliquen betrachtet werden, bei zweiterem nur Cliquen der Größe k, wobei k sich gegebenenfalls stetig erhöht. Aktuell ist es wohl die effektivste Heuristik für MaxClique [WH15].

Evolutionär: Die Funktionsweise dieser Metaheuristik ist an der Evolution von Lebewesen angelehnt. Es wird eine Lösung zufällig erzeugt und bewertet, danach werden bis zu einem bestimmten Punkt verschiedene Teile (sog. Individuen) gewählt und kombiniert. Es folgt eine Veränderung von dem Ergebnis (Mutation). Das Ergebnis dieser Veränderung wird wieder evaluiert, um daraufhin vielversprechende Ergebnisse auszuwählen und weiter zu verwenden. Dieser Prozess wiederholt sich, bis eine bestimmte Bedingung erfüllt wurde [Xue01]. Es handelt sich um eine sogenannte Metaheuristik, eine abstrakte Lösungsmöglichkeit für nahezu beliebige Probleme. Das Grundgerüst eines evolutionären Algorithmus kann beispielsweise wie folgt aussehen. Die Evaluation sei in der Selektion inbegriffen.

Algorithmus 4: Mögliches Skelett eines evolutionären Algorithmus

```
1: procedure MACOL(Instanz, m)
2:     P := populationInit(Instanz)
3:     repeat
4:         I := selektion(Instanz, P, m)
5:         NI :=rekombination(P, I)
6:         mutation(I, NI)
7:         P := selektion(P, NI)
8:     until some condition
9:     return beste Lösung in P
```

Die vorherrschende Meinung zu der Metaheuristik ist, dass sie für MaxClique gegenüber klassischer Heuristiken wie Local-Search im Nachteil ist. Somit wird sie vor allem in Kombination mit anderen Heuristiken genutzt [WH15, Kapitel 5].

3.4.2. Übersicht

Nun werden Algorithmen für die in der vorherigen Sektion beschriebenen Heuristiken erläutert. Diese wurden [WH15] und [Xue01] entnommen.

Greedy: Der Algorithmus DAGS [GLC04] verfolgt einen fortschrittlichen Greedy Ansatz, indem er nicht immer nur neue Knoten hinzufügt, sondern teilweise auch welche austauscht, wenn daraus eine vermutlich besser erweiterbare Clique resultiert.

Local-Search: Als typische Vertreter der Heuristik gelten CLS [PMB11] und das später betrachtete DLS [PH06].

Evolutionär: Algorithmen, die nur die evolutionäre Metaheuristik verwenden, sind auf Grund der bereits angesprochenen Ineffektivität selten aufzufinden. Allerdings gibt es einige Hybridalgorithmen, die den Ansatz mit der *local-search* Heuristik kombinieren [WH15].

Im nächsten Abschnitt wird ein repräsentativer Vertreter der Local-Search Heuristik, der Algorithmus DLS [PH06], welcher trotz des Alters immer noch einer der besten Algorithmen seiner Art darstellt [WH15], erläutert.

3.4.3. DLS-MC

In diesem Abschnitt wird der Algorithmus DLS-MC, beschrieben von Hoos und Pullan [PH06], erläutert. Die Abkürzung DLS steht für *dynamic local search*. Hierbei wird eine konstruktive und zufallsbasierte Suche kombiniert. Mit dieser Suche ist das Suchen von exakten Lösungen auf Teilgraphen und dessen Erweiterung gemeint. Um einen geeigneten Knoten zur Erweiterung zu finden, haben die Knoten des gegebenen Graphen dynamische Penalty-Werte, welche die Wahl des nächsten zu betrachtenden Knoten bestimmen.

Es folgen einige, in dieser Sektion wichtige, Notationen und der Pseudocode, hierbei sei zu beachten, dass es sich bei der Prozedur DLS-MC um den Algorithmus handelt, während der Rest Hilfsprozeduren sind.

Notation 3 ($N_I(C)$)**.** *Die Funktion $N_I(C)$ enthält alle Knoten, die mit allen in C vorhanden Knoten eine Verbindung haben.*

Notation 4 ($N_L(C)$)**.** *Die Funktion $N_L(C)$ enthält alle Knoten, die mit allen außer einem in C vorhanden Knoten eine Verbindung haben.*

Algorithmus 5: DLS-MC

```
1:  global numSteps := 0
2:
3:  procedure DLS-MC(G, tcs, pd, maxSteps)
4:      Füge zufälliges Element aus V zu C hinzu.
5:      P := Initialisiere den Penalty-Wert aller Knoten mit 0.
6:      while numSteps < maxSteps do
7:          v = expand(G, C)
8:          if |C| == tcs then
9:              return C
10:         C' := C
11:         v := plateauSearch(G, C, C')
12:         while N_I(C) ≠ ∅ do
13:             v = expand(G, C)
14:             if |C| == tcs then
15:                 return C
16:             v := plateauSearch(G, C, C')
17:         Erhöhe den Penalty-Wert für alle Knoten in C um 1.
18:         Verringere den Penalty-Wert für alle Knoten alle pd Zyklen um 1.
19:         if pd > 1 then
20:             C := { v }
21:         else
22:             v := Zufälliges Element aus V
23:             C := C ∪ { v }
24:             Entferne alle Knoten von C, die keine Verbindung zu v haben.
25:     return error
26:
27: procedure EXPAND(G, C)
28:     while N_I(C) ≠ ∅ do
29:         v := selectMinPenalty(N_I(C))
30:         C := C ∪ { v }
31:         numSteps + +
32:     return v
33:
34: procedure PLATEAUSEARCH(G, C, C')
35:     while N_I(C) == ∅ ∧ N_L(C) ≠ ∅ ∧ C ∩ C' ≠ ∅ do
36:         v := selectMinPenalty(N_L(C))
37:         C := C ∪ { v }
38:         Entferne den Knoten aus C, der nicht mit v verbunden ist.
39:         numSteps + +
40:     return v
```

Der Algorithmus erwartet als Eingabe einen Graphen G, eine Anforderung an die Größe der resultierenden Clique *tcs*, einen Wert der bestimmt wie oft der Penalty-Wert aller Knoten verringert wird *pd* und eine Begrenzung der Schritte *maxSteps*. Zurückgegeben wird eine Clique mit der Größe *pd* oder *error*. Es gibt eine globale Variable *numSteps*. Sie enthält die Anzahl der bereits erfolgten Schritte.

Am Anfang des Algorithmus wird ein zufälliges Element aus V ($G = (V, E)$) zu der aktuellen Clique C hinzugefügt und die Penalty-Werte werden initialisiert.

Es wird solange iteriert, bis entweder die gegebene maximale Anzahl an Schritten *maxSteps* erreicht oder eine Clique mit mindestens der Größe *tcs* gefunden wurde. Am Anfang einer Iteration steht das Aufrufen der Subroutine EXPAND. Diese Routine fügt zur Clique C solange Knoten aus $N_I(C)$, mit minimalen Penalty-Wert, hinzu, bis $|N_I(C)|$ Null ergibt. Danach folgt die Prüfung, ob C bereits die nötige Größe erreicht hat. Als nächstes wird die Subroutine PLATEAUSEARCH aufgerufen. Diese Funktion fügt Knoten mit minimalem Penalty-Wert aus $N_L(C)$ zur Clique hinzu und entfernt dabei den Knoten aus C, mit dem der jeweils neue Knoten nicht verbunden ist. Dieses wird solange wiederholt, bis entweder $N_L(C)$ keine Elemente enthält, $N_I(C)$ wieder Elemente enthält oder der Schnitt der aktuellen und alten Clique leer ist.

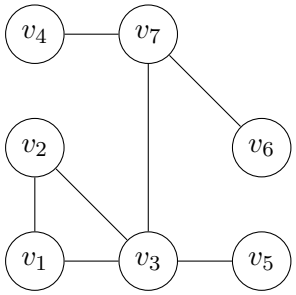
Der nächste Schritt ist, EXPAND aufzurufen, die Überprüfung, ob *tcs* erreicht ist, zu wiederholen und schließlich PLATEAUSEARCH erneut auszuführen. Dieser Schritt wird solange wiederholt, bis es nach PLATEAUSEARCH keine Knoten mehr gibt, die zu allen Knoten in C eine Verbindung haben. Es sei noch bemerkt, dass bei der Wahl eines Elements mit niedrigstem Penalty-Wert in den vorangegangenen Schritten niemals ein Knoten doppelt gewählt wird. Außerdem wird nie ein Knoten mit einem Penalty-Wert 10 oder höher genutzt. Als letzter Schritt werden die Penalty-Werte dem Pseudocode entsprechend angepasst und es werden zwei Fälle unterschieden. Falls *pd* größer als 1 ist, wird C geleert und der letzte zu C hinzugefügte Knoten wird eingefügt. Ist dem nicht so, wird ein zufälliges Element aus V zu C hinzugefügt und alle Knoten entfernt, die zu diesem Knoten keine Verbindung haben.

Beispiel

In diesem Beispiel wird der untere Graph als Eingabe dienen. Die Variablen *tcs*, *maxSteps* und *pd* werden auf 3, 10 und 2 gesetzt.

$$G = (\{ v_1, v_2, v_3, v_4, v_5, v_6, v_7 \}, E)$$

$$E = \left\{ \begin{array}{c} \{ v_1, v_2 \}, \{ v_2, v_3 \}, \{ v_1, v_3 \}, \{ v_4, v_7 \}, \\ \{ v_6, v_7 \}, \{ v_5, v_3 \}, \{ v_7, v_3 \} \end{array} \right\}$$



Als Erstes wird ein zufälliges Element zu C hinzugefügt, dieses sei hier v_7. Die Abbildung P wird für alle Knoten auf 0 gesetzt. Als nächstes wird EXPAND(G, C) aus Zeile 7 aufgerufen. Die Funktion sucht nach dem Knoten mit dem niedrigsten Penalty-Wert aus $N_I(C)$, hier sind alle Werte gleich, also wird wieder ein zufälliger Knoten gewählt, hier v_2. Da es keine Knoten mehr gibt, die zu v_7 und v_2 eine Verbindung haben, ist EXPAND hier fertig. Die aktuelle Clique C lautet nun:

$$C = \{ v_2, v_7 \}$$

Die Größe ist kleiner als 3 und somit fährt der Algorithmus fort. Als Nächstes wird PLATEAUSEARCH(G, C, C') (Zeile 11) aufgerufen, wobei C' eine Kopie von C ist. Da immer noch alle Penalty-Werte gleich sind, wird wieder ein zufälliger Knoten aus $N_L(C)$ $(= V \setminus C)$ gewählt, hier sei v_3 dieser Knoten. Da v_3 keine Verbindung zu v_7 hat, wird v_7 entfernt. Somit ist $N_I(C)$ nicht mehr leer, womit die Suche endet. Folgende Knoten sind nun in C enthalten:

$$C = \{ v_2, v_3 \}$$

Es wird jetzt wieder EXPAND aufgerufen (Zeile 13). Es existiert noch ein Knoten der zu v_2 und v_3 eine Verbindung hat, nämlich v_1. Damit ist die Erweiterung bereits beendet und es wird geprüft, ob *tcs* bereits erreicht ist. Dieses ist nun der Fall und somit wird die folgende Clique zurückgegeben:

$$C = \{ v_2, v_3, v_1 \}$$

Performance

Da die Komplexität, und vor allem die Performanzrate, wegen des Zufallselementes der Suche bei diesem Algorithmus keine große Aussagekraft haben, wird in diesem Unterkapitel eine durch die Autoren Pullan und Hoos [PH06, Kapitel 3] durchgeführte empirische Analyse betrachtet.

Es wurden 80 verschiedene Graphen getestet. Der Parameter *tcs* wurde auf die tatsächlich größte bekannte Clique der jeweiligen Graphen gesetzt. Für *pd* wurden verschiedene Werte probiert und nur das beste Ergebnis berücksichtigt. Die *maxSteps* wurden auf 100 000 000 gesetzt, um die Wahrscheinlichkeit zu erhöhen, dass der Algorithmus tatsächlich mit einem Ergebnis terminiert. Im Ergebnis wurden von den 80 Testfällen 77-Mal eine Lösung der Größe *tcs* gefunden. Bei den meisten Tests (67) wurden die Lösungen innerhalb einer Sekunde gefunden. Des Weiteren wurde gezeigt, dass DLS-MC in vielen Fällen eine bessere Ausführungszeit als die damaligen *state-of-the-art* Algorithmen DAGS und GRASP liefert. Es sollte angemerkt werden, dass DLS-MC zwar auch schneller als viele neuere Algorithmen ist (in der Breite gesehen), aber gegenüber dem modernen Algorithmus CLS häufig unterlegen ist [PMB11, Kapitel 3] (es ist zu beachten, dass hier PLS und CLS verglichen werden, da aber PLS sehr ähnliche Werte aufweist wie DLS [Pul06], muss CLS auf dem dort betrachteten Benchmark eine bessere Performance liefern als DLS).

Schlussbemerkung

Auch wenn die mangelnde Sicherheit einer Performanzrate auf den ersten Blick verunsichert, so ist DLS-MC doch immer noch einer der praktisch effizientesten heuristischen Algorithmen der heutigen Zeit. Somit bleibt die Erkenntnis, dass die Performanzrate in der Praxis, zumindest beim Approximieren von MaxClique, nicht die allerhöchste Relevanz haben muss. Problematisch an dem Algorithmus ist die Varianz, im Hinblick auf die Laufzeit, beim Berechnen einer Clique auf dem jeweils gleichen Graphen, wobei diese durch Parallelisierung etwas abgeschwächt werden kann [PH06, Kapitel 4].

3.5. Implementierungen

Nun da viele verschiedene Algorithmen erläutert wurden, sollen sie implementiert werden. Es soll hier insbesondere um die Wahl der Datenstrukturen und die Umsetzung von vereinfachenden Notationen gehen.

Alle Algorithmen werden in Lua, mit einer Auswahl an als Modul bereitgestellter Datenstrukturen implementiert. Lua läuft hierbei in einem Java-Interpreter und die über `require` eingebundenen Module sind in Java implementiert. Die Ein-, Ausgabe und Ausführung wird über ein Java-Programm verwaltet. Genauere Informationen zu dem verwendeten Java-Programm sind im Anhang zu finden. Es folgt eine kurze Übersicht der verwendeten Datenstrukturen.

Tabelle 3.2.: Datenstrukturen [Vig16][Ora]

	Laufzeit			
Datenstruktur	Einfügen	Entfernen	Lesen	Kurzerläuterung
ArrayList	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	Typische Liste, die ein C-Style Array als Speicher nutzt.
ArrayGraph	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	Knoten werden in einer ArrayList gespeichert, Kanten in einer 2D ArrayList.
Int2ObjectHashMap	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	Auf Integer als Schlüssel spezialiserte HashMap
Int2IntHashMap	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	Auf Integer als Schlüssel und Wert spezialiserte HashMap
Object2IntHashMap	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	Auf Integer als Wert spezialiserte HashMap

Es sollte schließlich noch erwähnt werden, dass obige Datenstrukturen amortisierte Laufzeiten von $\mathcal{O}(1)$ auf fast allen Operationen haben (außer bei dem Entfernen auf ArrayList/ArrayGraph). Außer den aufgezählten Datenstrukturen, wurden noch weitere, auf primitive Typen spezialisierte, ArrayLists verwendet (auf Typ x spezialisierte Datenstrukturen speichern Daten dieses Typs effizienter) [Vig16][Ora].

MCQ

Die Mengen Q, Q_{max}, R und R_p werden als ArrayLists's implementiert, da alle Operationen ähnlich wichtig sind, ist die ArrayList die beste Wahl, da sie wenig Overhead mitbringt und sehr gute amortisierte Laufzeiten hat.

Die Abbildung der Farbklassen wird mithilfe einer Int2IntHashMap implementiert. Dank der Spezialisierung der Map ist der Objekt-Overhead minimal. Des Weiteren liefern auch HashMaps sehr gute amortisierte Laufzeiten auf allen Operationen.

Der Graph wird als ArrayGraph implementiert. Hier war vor allem der schnelle Lesezugriff entscheidend, da der Rest der Operationen nicht genutzt wird. Außerdem ist der Platzverbrauch, durch das interne Nutzen von spezialisierten Listen, minimal.

Die Sortierung in Zeile 8 vom gegebenen Pseudocode ist implementiert durch einen iterativen Mergesort Algorithmus mit einer Laufzeit von $\mathcal{O}(n \log n)$. Es handelt sich hierbei um die Methode `sort(...)` aus der Klasse `java.util.Collections` der Java Standard Library [Ora].

Feige's Algorithmus

G', G'' sind in der Implementierung ArrayLists (aus dem gleichen Grund wie bei MCQ), die zusammen mit den Kanten in G, welcher als ArrayGraph implementiert ist, einen Graphen bilden. Durch diese Methode kann die Berechnung der induzierten Teilgraphen komplett weggelassen werden.

Die Teilmengen der Größe t sind als 2D ArrayList implementiert, da hier die Leseoperation überwiegt. Die Berechnung der Teilmengen erfolgt über einen einfachen rekursiven Algorithmus, welcher die Fälle „enthalten“ und „nicht enthalten“ für jeden Knoten rekursiv durchprobiert.

Weiter ist die Wahl von k und t von Interesse. Da k in der Regel unbekannt ist, werden alle $k's$ ausprobiert und die beste korrekte Lösung wird als endgültige Lösung verwendet. Feige schlägt $\log n / \log \log n$ für t vor, was aber für kleine n oft keine Ergebnisse liefert, da die Bedingung $t \ll n/k$ verletzt werden würde. Somit wird, um diese Bedingungen garantiert zu erfüllen, t halbiert bis t hinreichend viel größer ist als n (k ist unbekannt), konkret wurde der Faktor 50 gewählt.

DLS

Die Variable P wird als ArrayList implementiert und ständig sortiert gehalten (gleiches Sortierverfahren wie bei MCQ), dadurch wird das Suchen nach dem minimalen Penalty-Wert, welches dominiert, enorm beschleunigt. C ist wie in den vorangegangenen Implementierungen eine Knoten-Liste. G wird analog zu Feige's Algorithmus implementiert.

Als Zufallsfunktion wird die Lua-Funktion `math.random()` verwendet, welche gleichverteilt einen Wert zwischen 0 und 1 zurückgibt.

Algorithmus von Francis, Gonçalves und Ochem

Ein Intervall wird durch ein spezielles Objekt dargestellt. Die Intervall-Listen I_i sind ArrayLists von Intervall-Objekten. Der Graph ist erneut ein ArrayGraph. Alle Teilgraphen (s, c) sind lediglich Knotenlisten, die als ArrayLists implementiert sind. Auch hier kann daher unter Zunahme des Graphen G auf die Berechnung induzierter Teilgraphen verzichtet werden. Der gerichtete Graph wird als Int2ObjectHashMap implementiert. Für die Zuordnung der Klassen zu den Kanten wird eine Int2IntHashMap verwendet. Es werden in beiden Fällen HashMaps benutzt, da sie laufzeittechnisch hier ohne Konkurrenz sind, da keine weiteren vereinfachenden Faktoren vorliegen. Die Spezialisierungen der HashMaps gewähren des Weiteren eine bessere Speicherausnutzung.

Weiter wird die 8te Zeile betrachtet. Hier wird doppelt verschachtelt über alle Intervalle iteriert und der Schnitt der Intervalle wird berechnet. Je nach Fall wird ein entsprechender Eintrag im gerichteten Graphen (es gilt beim Eintrag $Key \rightarrow Value$, dass es eine Kante von *Key* nach *Value* gibt) und in die Klassen-Map geschrieben.

Erwähnenswert ist weiter die Implementierung der Zeile 25. Hier soll die maximale Clique eines co-bipartiten Graphen in polynomieller Zeit berechnet werden. In der Implementierung wurde dieses über einen relativ langen Weg gelöst. Die Beschreibung aller Implementierungdetails würde zu weit führen, daher folgt eine kurze Skizze vom Lösungsweg.

1. Der Graph wird komplementiert.
2. Auf dem komplementierten Graph wird der Hopcroft-Karp Algorithmus [HK73] ausgeführt, woraus ein maximales Matching folgt.
3. Durch das Theorem von König [Bon97] kann dieses Matching in eine minimale Knotenüberdeckung konvertiert werden.
4. Das Komplement einer Minimalen Knotenüberdeckung ist eine maximal große stabile Menge.
5. Da der Graph anfangs komplementiert wurde, ist das Komplement der minimalen Knotenüberdeckung die maximale Clique des Ursprungsgraphen.

4 | Minimum Clique Partition

Das nächste zu behandelnde Clique-Optimierungsproblem ist Minimum Clique Partition (MCP). Es folgt eine Definition (angelehnt an [CK97]) des Problems.

Instanz Ein Graph $G = (V, E)$.

Lösung Eine Menge an paarweise disjunkten Teilmengen V_i mit $0 \leq i < n$ von V, wobei jeder der Teilmengen auf dem induzierten Teilgraphen eine Clique darstellen muss. Jeder Knoten muss in genau einer dieser Teilmengen enthalten sein.

Maß Anzahl der Teilmengen n.

Ziel Die Minimierung von n.

Das Problem ist auf dem komplementären Graphen äquivalent zu MGC. Daher sind alle Ergebnisse zu diesem Problem auch für MCP gültig. Im Folgenden wird dementsprechend an vielen Stellen auf Publikationen verwiesen, die das genannte Ergebnis ursprünglich für das Coloring Problem, bewiesen bzw. entwickelt haben.

4.1. Approximierbarkeit

Das Problem ist zuerst in die Hierarchie der Optimierungsprobleme einzuordnen. Minimum Clique Partition liegt in $\mathcal{NPO}$.

Beweis. Definitionsgemäß muss zuerst gezeigt werden, dass jede Lösung polynomiell überprüfbar ist. Dies ist hier der Fall, indem zuerst geprüft wird, ob die Partitionierung gültig ist, also jeder Knoten in genau einer Teilmenge liegt. Dafür erfolgt eine Iteration für jeden Knoten über alle Teilmengen, dieses ist in $\mathcal{O}(n^2)$ durchführbar. Als nächstes folgt die Prüfung für jede Teilmenge, ob der induzierte Graph eine Clique darstellt, wofür einfach für jeden Knoten betrachtet wird, ob dieser zu jedem der anderen Knoten eine Verbindung hat. Somit ergibt sich eine Laufzeit der Prüfung von $\mathcal{O}(n^2)$.

Die Länge der Lösung muss polynomiell beschränkt sein. Da es sich um eine Partition handelt, ist die Länge durch $\mathcal{O}(|V| \cdot \log n)$ beschränkt.

Die Maßfunktion muss polynomiell berechenbar sein. Dieses ist trivialerweise der Fall, da es sich hier um das Zählen von Mengen handelt. □

Goldreich, Bellare und Sudan [BGS95] haben gezeigt, dass eine Approximierung nicht innerhalb von $|V|^{1/7-\varepsilon}$ für alle $\varepsilon > 0$ möglich ist. Hierdurch kann eine Mitgliedschaft in $\mathcal{APX}$ ausgeschlossen werden.

Die obere Schranke wurde von Halldórsson 1993 [Hal93] bewiesen und liegt bei $\mathcal{O}(|V|((\log\log|V|)^2/(\log|V|)^3))$.

Des Weiteren ist auch hier, womöglich wegen der schwachen Garantien auf allgemeinen Graphen, ein bemerkenswert oft behandeltes Thema die Approximierbarkeit auf eingeschränkten Graphen. Folgend einige Graphenklassen und deren derzeit bekannten erreichbaren Performanzraten.

Tabelle 4.1.: Laufzeit & Performanz auf eing. Graphen

Graphenklasse	Performanzrate	Laufzeit	Referenz
unit disk	$1+\varepsilon$	$n^{\mathcal{O}(1/\varepsilon^2)}$	[DP11]
penny	$3/2$	polynomiell	[Cer+11]
max-3-degree	$5/4$		[Cer+08]
circle	$\log Opt$	$\mathcal{O}(n^3 \log\log n)$	[KS06]
subtree filament	$\mathcal{O}(log\|V\|)$	polynomiell	

4.2. Approximationsalgorithmen

Bei Approximationsalgorithmen zu Minimum Clique Partition gilt Ähnliches wie bei MaxClique, auf allgemeinen Graphen existieren nur wenige Algorithmen, wovon der bedeutendste von Halldórsson [Hal93] entwickelt wurde, dieser hat die beste bekannte obere Schranke, als Performanzrate, für das Problem. Dieser Sachverhalt lässt sich dadurch erklären, dass MCP schwierig zu approximieren ist, woraus resultiert, dass Algorithmen mit beweisbaren Schranken oft vergleichsweise schwache Ergebnisse liefern.

Im Gegensatz zu allgemeinen Graphen liegt das Problem auf eingeschränkten Graphen sehr häufig in $\mathcal{APX}$. Daraus resultiert die Existenz von vielen Approximationsalgorithmen auf eingeschränkten Graphen (siehe Tabelle 4.1). Insbesondere die UDGs werden in diesem Zusammenhang oft behandelt, deshalb wird im folgendem Kapitel ein $\mathcal{PTAS}$, veröffentlicht von Pirwani und Salavatipour [PS12], für das Finden einer Partition auf UDGs vorgestellt.

4.2.1. Algorithmus von Pirwani und Salavatipour

Algorithmus 6: Algorithmus von Pirwani und Salavatipour

```
1:  procedure MinCP(G, M_UDG, ε)                                  ▷ G = (V, E)
2:      k := ⌈16/ε⌉, C_min := ∅
3:      for ⌈log n⌉ times do
4:          G_{0,0} := Gitter mit Regionen der Größe k × k.
5:          Wähle zufälliges (a, b) ∈ [0, k) × [0, k).
6:          Verschiebe G_{0,0} um a Einheiten nach rechts und um b nach oben. Das
                Resultat wird G_{a,b} genannt.
7:          for all regions in G_{a,b} do
8:              Berechne C_i für die Punkte in region.
9:          C_{a,b} := Vereinigung von allen Mengen C_i
10:         if |C_{a,b}| < |C_min| then
11:             C_min := C_{a,b}
12:     return C_min
```

Hier wird der Algorithmus von Pirwani und Salavatipour [PS12] für den Fall, dass die geometrischen Informationen des Graphen vollständig bekannt sind, beschrieben. Als Eingabe wird ein Graph G, ein UDG-Modell M_{UDG} und eine Variable $\varepsilon > 0$ erwartet. Das ε im Modell vom UDG wird konstant auf 1 gesetzt. Bei der Ausgabe sind zwei Fälle zu beachten. Falls G zusammen mit dem Modell kein UDG darstellt, wird eine Lösung ohne Garantien ausgegeben. Ist der Graph zusammen mit dem Modell ein gültiger UDG, wird eine Lösung ausgegeben, die mit hoher Wahrscheinlichkeit maximal um den Faktor $1 + \varepsilon$ ($\varepsilon > 0$) schlechter ist als die optimale Lösung.

Als Erstes berechnet der Algorithmus $k = \lceil 16/\varepsilon \rceil$. Danach wird ein Gitter „erdacht“, welches über $k \times k$ große Quadrate verfügt. Das gedachte Gitter ist unendlich groß und die genaue Lage ist, wegen der folgenden Verschiebung, irrelevant. Nun wird das Gitter um jeweils einen zufälligen Wert $\in [0, k)$ nach rechts und nach oben geschoben. Als Nächstes werden aus den Punktemengen aller nicht leeren Regionen jeweils eine optimale Lösung für MCP berechnet, dazu später mehr. Falls ein Punkt genau zwischen zwei Regionen liegt, darf dieser Punkt nur für eine der beiden Regionen betrachtet werden. Nachdem dieses für alle Regionen durchgeführt wurde, werden alle diese Ergebnisse vereinigt und ergeben für den gesamten Graphen eine korrekte Lösung. Dieses alles wird $\log n$-mal durchgeführt und die beste der errechneten Lösungen wird als Ergebnis von der Prozedur zurückgegeben.

Nun zur Berechnung der optimalen Lösung in einer Region der Größe k (Zeile 8). Es gibt eine Menge an Trennlinien T die paarweise separierende Linien enthalten, welche jeweils zwei verschiedene Cliquen voneinander trennen. Ein Satz von Pirwani und Salavatipour [PS12] besagt, dass es eine optimale Lösung gibt, für die eine solche Menge T existiert. Um diese zu finden, muss jede Kombination an Linien und

daraus resultierende Partitionierungen geprüft werden, für eine effizientere Art siehe Abschnitt 4.5. Die polynomielle Laufzeit des Vorgehens ist wie folgt begründbar: Es gibt in einem $k \times k$ Quadrat maximal $\mathcal{O}(k^2)$ Cliquen, da der durch k begrenzte Raum in $1/2 \times 1/2$ große Teile gespaltet werden kann [PS12]. Wegen der UDG Eigenschaft kann in jedem dieser Teile nur eine Clique sein. Es werden ungünstigstenfalls für jeden Knoten die Separierung zu einer Clique gesucht, folglich läuft die Teilprozedur in $\mathcal{O}(n^{k^2})$ [PS12].

Beispiel

Folgender Graph wird als Beispielgraph benutzt. Der Übersicht halber wird die formale Definitionen der Mengen weggelassen.

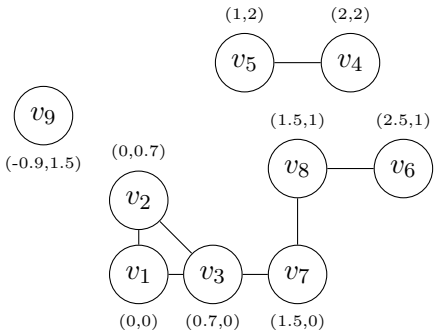


Abbildung 4.1.: Beispiel UDG-Graph

Am Anfang ist der Wert von ε zu wählen, es wird hier auf 8 gesetzt, um die Durchführung anschaulich zeigen zu können. Es folgt eine Regionsgröße von 2. Nun muss ein Gitter mit solchen Regionensgrößen über den Graphen gelegt und um beispielsweise (1, 0) verschoben werden.

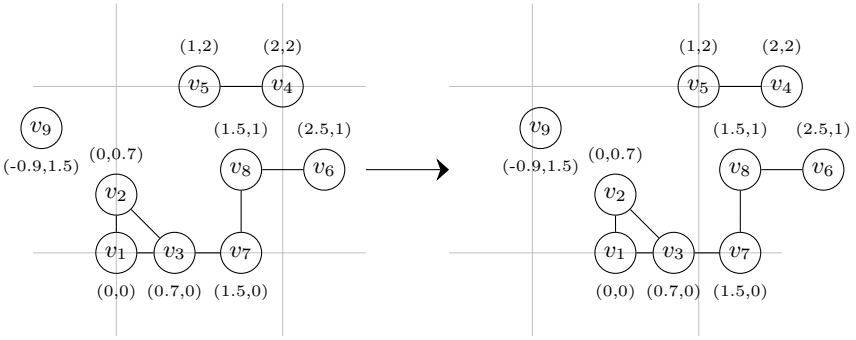


Abbildung 4.2.: links) UDG mit Gitter; rechts) UDG mit verschobenem Gitter

Die obere linke Region wird als R_{00} und die untere rechte als R_{22} bezeichnet. Da R_{00} leer ist, ist R_{10} als Erstes zu betrachten. In dieser Region liegt v_5, somit ist die optimale Partition trivial und lautet $V_0 = \{ v_5 \}$. Für R_{20} ist die Lösung ebenfalls trivial und lautet $V_1 = \{ v_4 \}$. Da R_{01} leer ist, wird nun R_{11} betrachtet. Es werden nun separierende Linien gesucht, die in eine optimale Lösung resultieren.

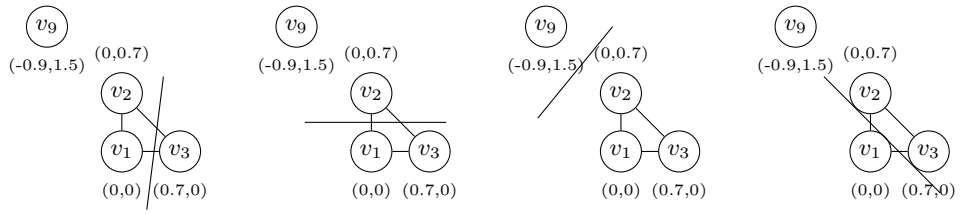


Abbildung 4.3.: Alle Möglichkeiten der ersten Trennlinie

Es ist erkennbar, dass es nach Platzierung der ersten Trennlinie nur eine optimale Lösung gibt, es ist die Linie zwischen v_9 und v_2. Da nun alle Teile des Graphen eine einzelne Clique bilden, ist die Region abgearbeitet und die Lösung ist somit $V_2 = \{ v_9 \}$, $V_3 = \{ v_1, v_2, v_3 \}$. Bei der letzten relevanten, alle weiteren sind leer, Region R_{21} ist analog vorzugehen, daraus resultiert beispielsweise $V_4 = \{ v_8, v_7 \}$, $V_5 = \{ v_6 \}$. Damit ist ein Durchlauf fertig (die weiteren $\log 9$ Durchläufe werden an dieser Stelle weggelassen) mit dem Ergebnis:

$$\begin{array}{lll} V_0 = \{ v_5 \} & V_2 = \{ v_9 \} & V_4 = \{ v_8, v_7 \} \\ V_1 = \{ v_4 \} & V_3 = \{ v_1, v_2, v_3 \} & V_5 = \{ v_6 \} \end{array}$$

Das Maß der Lösung ist 6 und da die optimale Lösung ein Maß von 5 hat, ist diese um $6/5 < 1 + \varepsilon$ schlechter.

Laufzeit & Performanz

Es wird $\log n$ Mal der Gesamtalgorithmus durchgeführt, es existieren ungünstigstenfalls $\mathcal{O}(n)$ viele nichtleere Quadrate, und da das Finden der optimalen Lösungen, wie vorher schon erläutert, in $\mathcal{O}(n^{k^2})$ liegt, folgt eine Gesamtlaufzeit von $\mathcal{O}(\log n \cdot n^{k^2+1})$.

Die Performanzanalyse von Pirwani und Salavatipour basiert auf Wahrscheinlichkeiten, konkret der Markow-Ungleichung, diese liefert für eine Zufallsgröße eine obere Schranke, für das Überschreiten eines gewissen Wertes. Auf die Zufallsvariable, wie viele Kanten der Lösungsmenge von Raster geschnitten werden, bezogen, entsteht folgende Ungleichung:

$$P\big[\text{Eine Kante in } C \text{ wird von } G_{a,b} \text{ geschnitten}\big] \leq \frac{2}{k}.$$

Das Symbol δ stelle im Folgenden die Anzahl der Partitionen bei einer optimalen Lösung dar. Nun, da für jede Clique die Wahrscheinlichkeit $< 2/k$ ist, geschnitten zu werden, beträgt die Erwartung der Anzahl an geschnittenen Cliquen $2/k \cdot \delta$. Somit beträgt die Wahrscheinlichkeit, dass nur $4/k \cdot \delta$ Cliquen geschnitten werden $1/2$. In diesem Fall gibt es also ungünstigstenfalls $4 \cdot 4/k \cdot \delta$ zu viele Cliquen in der Lösung, da jede geschnittene Clique in bis zu 4 Teile geteilt wird. Da dies alles allerdings nur zu einer Wahrscheinlichtkeit von 50% zutrifft, wird der Algorithmus $\log n$-mal wiederholt, um die Wahrscheinlichkeit zu maximieren. Da es sich um einen Überschuss handelt, hat die Partitionierung im Erfolgsfall eine Größe von maximal $\delta + 16/k \cdot \delta$ Cliquen [PS12]. Es ist erkennbar, dass es sich nicht um ein richtiges $\mathcal{PTAS}$ handelt.

Schlussbemerkung

Der vorgestellte Algorithmus ist eine simplere, aber auch eingeschränktere, Version des hauptsächlich von Pirawani und Salavatipour [PS12] veröffentlichten Algorithmus. Allerdings ist dieser deutlich schlechter implementierbar. Daher fiel die Wahl auf diese Variante.

Weiter sei auf eine Nachfolgearbeit von Dimitrescu und Pach [DP11] verwiesen, welche die Laufzeit des Hauptalgorithmus von Pirwani und Salavatipour weiter verbessern.

4.3. Exakte Algorithmen

Da Minimum Clique Partition auf dem komplementären Graphen äquivalent zu Minimum Graph Coloring ist, wird eine exakte Lösung zu MCP vor allem mithilfe von Algorithmen für MinColoring berechnet. Hier werden die bedeutendsten Vertreter kurz angeschnitten.

Am Anfang steht der Algorithmus von E.L. Lawer aus dem Jahre 1976 [Law76] mit einer Laufzeit von $\mathcal{O}(1 + \sqrt[3]{3}^n)$. Sein Algorithmus nutzt das Prinzip der dynamischen Programmierung, konkret iteriert er über alle Teilmengen, betrachtet dessen maximale stabile Mengen und berechnet durch die vorher ermittelte Farbzahl eine neue.

25 Jahre später wurde dieser Algorithmus durch D. Eppenstein [Epp03] weiter verbessert auf eine Laufzeit von $\mathcal{O}((4/3 + 3^{4/3}/4)^n)$. Er erreichte dies durch die Reduzierung der betrachteten stabilen Mengen und der Nutzung eines speziellen Algorithmus, um 3-färbbare Teilgraphen zu behandeln.

Der früheste der heutigen *state-of-the-art* Algorithmen wurde 2006 von Mèndez-Dìaz und Zabala veröffentlicht [MZ06]. Es handelt sich um einen *branch-and-cut* (Kombination von *branch-and-bound* und Verkleinern der Lösungsmenge durch Hinzufügen einer neuen Bedingung, hierbei wird das Problem als ganzzahliges lineares

Optimierungsproblem umgeschrieben) Algorithmus. Weiter interessant ist der *branch-and-price* (wieder eine Variation von *branch-and-bound*, ähnlich zu *branch-and-cut*) Algorithmus von Malaguti, Monaci und Toth [MMT11] aus dem Jahre 2010, ein auf DSATUR (Kombination aus Greedy-Heuristik und Vorsortierung) basierender Algorithmus von Segundo [Seg12] und ein auf Constraints basierender Algorithmus von Zhou, Li, Haund und Xu [Zho+14].

4.4. Heuristische Algorithmen

Nun werden einige heuristische Algorithmen betrachtet, wobei auch hier angemerkt werden muss, dass es sich im Folgenden um Algorithmen handelt, die für Minimum Graph Coloring entwickelt wurden. Die häufigsten Heuristiken decken sich weitestgehend mit denen für MaxClique, daher werden diese und einige Unterformen hier lediglich kurz angerissen. Weiter werden einige der bedeutendsten Vertreter der verschiedenen Heuristiken beschrieben.

Greedy: Der *Greedy*-Ansatz wurde vor allem am Anfang der heuristischen Algorithmen für Minimum Clique Partition angewendet. Hier war ein bedeutender Vertreter die iterative Greedy-Methode von Culberson und Luo [CL93]. Eine weiterer und wohl der bekannteste Vertreter vom Greedy-Ansatz war der DSATUR Algorithmus, welcher von Brelaz [Bré79] eingeführt wurde. Dessen Neuerung war eine Vorsortierung der Knoten. Darauf basierend wurden auch viele andere Vorsortierungen/Reihenfolgen getestet [OS06].

Local-Search: Bei der *lokalen Suche* werden in der Lösungsmenge Änderungen aufgrund von lokalen Begebenheiten vorgenommen bis eine gewisse Lösungsgüte erreicht ist. Eine wichtige Weiterentwicklung ist hier die sogenannte Tabu-Suche, entwickelt von Glover im Jahre 1986 [GM86]. Hierbei wird im Kern eine Tabu-Liste eingeführt, in der Elemente eingefügt werden, um sie bis zu einem gewissen Punkt für den Algorithmus als nicht verwendbar zu markieren. Die erste Implementierung dieser Metaheuristik kam von Hertz und Werra [HW87]. Als eine weitere Unterart kann *iterated-local-search* angesehen werden, dieses versucht, die Schwäche vo dem *local-search* Ansatz, nämlich, dass es oft nur lokale Minima (oder in anderen Problemen Maxima) findet, auszugleichen, indem es nacheinander, mithilfe des bisherigen Verlaufes, lokale Suchen mit günstig gewählten Startpunkten startet.

Evolutionär: Die *evolutionäre* Metaheuristik basiert auf der Evolutionstheorie. Zur Erinnerung, es werden Generationen von Lösungen gebildet, gewählt, mutiert und wieder von vorne begonnen. Es ist einer der am häufigsten verwendeten Metaheuristiken in aktuellen Algorithmen, allerdings sind sie auch bei MCP alleine nicht konkurrenzfähig, daher werden sie nahezu ausschließlich in hybriden Algorithmen eingesetzt.

HYBRID: Die *hybriden* Lösungsverfahren sind eine Kombination von mehreren Heuristiken und Metaheuristiken. Viele heutige *state-of-the-art* Algorithmen verfahren nach multiplen Heuristiken, und genau solche werden im Folgenden betrachtet. Eine häufig betrachtete Kombination sind die sogenannten memetische Algorithmen. Diese basieren auf der evolutionären Metaheuristik, welcher eine lokale Suche als Mutationsschritt benutzt. Ein populärer Vertreter ist hierbei MACOL von Lü und Hao [LH10]. Ein weiterer Hybrid wurde von Wu und Hao entwickelt: Der EXTRACOL Algorithmus [WH12]. Dieser basiert auf einer Kombination von Tabu-Suche und dem Nutzen von stabilen Mengen. Ein sehr interessanter und höchst konkurrenzfähiger Ansatz kam 2016 wieder auf: Die Kombination eines memetischen Alogrithmus mit einem lernenden Automaten, genannt AMACOL, veöffentlicht von Mirsaleh und Mebodi [MM16].

Wegen der aktuellen hohen Popularität memetischer Algorithmen im Bereich von MINIMUM CLIQUE PARTITION (bzw. MGC) wird nun im Folgenden ein solcher vorgestellt.

4.4.1. MACOL

In diesem Kapitel wird der Algorithmus MACOL (*memetic algorithm for the coloring problem*) [LH10] behandelt. Er wurde für MINIMUM GRAPH COLORING entwickelt, daher wird im Folgendem eine Variante behandelt, die auf dem invertierten Graphen arbeitet.

Als Eingabe wird ein beliebiger Graph $G = (V, E)$, die Größe der Population p, die Selektionsgröße m ($2 \leq m \leq p$), die Anzahl der Partitionen k, die Tabu-Iterationszahl t und die maximale Anzahl an Iterationen L benötigt. Die Ausgabe ist eine Menge entsprechend der Definition einer Lösung für das MINCLIQUEPART Problem.

Algorithmus 7: MACOL

```
1:  procedure MACOL(G, p, m, k, t, L)
2:      P := populationInit(G, k, p)                    ▷ P = { P_1, ..., P_p }
3:      for i = 1, p do
4:          P_i := tabuSearch(G, P_i, t)
5:      P' := min_{0...p}(f(P_i))
6:      repeat
7:          I := selection(G, P, m)
8:          P_0 := crossover(G, I)
9:          P_0 := tabuSearch(G, P_0, t)
10:         if f(P_0) < f(P') then          ▷ f(x) = Anzahl der Konflikte in x
11:             P' := P_0
12:         P := updatePopulation(G, P_0, P)
13:         L--
14:     until korrekte Partition wurde gefunden OR L == 0
15:     return P'
```

Die im Pseudocode auftauchende Funktion f ist definiert als die Anzahl an Knotenpaaren, die in der gleichen Partition liegen, aber nicht durch eine Kante verbunden sind. Weiter ist der evolutionäre Aufbau vom Algorithmus erkennbar, mit Selektion (SELECTION), Kombination (CROSSOVER) und Mutation (TABUSEARCH). Alle einzeln erwähnten Prozeduren werden im Folgenden erläutert.

POPULATIONINIT: Bei der Initialisierung werden p möglichst korrekte und divergente k-Clique-Partitionierungen mithilfe der DANGER Heuristik [GPR93] erstellt. Die Heuristik verwendet spezifische Formeln, abhängig von dem Knoten, dessen Nachbarn und ihren Farben. Es gibt hierbei zwei verschiedene Formeln: Eine für die Selektion und eine für die Zuordnung zu einer Clique. Hier werden – im Gegensatz zum Original – die Ergebnisse dieser Funktion (nach Normalisierung) als Wahrscheinlichkeitswerte aufgefasst und dadurch ein Zufallselement erzeugt.

SELECTION: Wählt zufällig m Individuen, hier Clique-Partitionierungen, aus.

CROSSOVER: Hierbei handelt es sich um den Kern des Algorithmus und um den Teil, der ihn von anderen memetischen Algorithmen abhebt. Es soll aus den im vorherigen Schritt gewählten Individuen eine neue, möglichst gute Clique-Partition $P_0 = \{ V_{01}, \ldots, V_{0k} \}$ erstellt werden. Es werden nun also k Knotenmengen $V_{0i} (0 \leq i \leq k)$ erstellt, die jeweils möglichst korrekte Cliquen sein sollen. Hierfür wird sukzessive jede Menge V_{0i} auf die größte Knotenmenge, aus denen in Zeile 7 selektierten Partitionierungen, gesetzt. Nachdem ein neues V_{0i} bestimmt wurde, müssen alle Elemente in V_{0i} aus den restlichen selektierten Mengen entfernt werden. Hierbei ist zu beachten, dass jedes Mal, wenn eine größte Knotenmenge auf diese Weise gewählt wurde, die dazugehörige Partition für die nächsten $\lceil {}^{p-1}/_{2} \rceil$ neuen Knotenmengen von P_0 nicht mehr berücksichtigt wird. Nun ist es möglich, dass einige Knoten am Ende ohne Zuweisung verbleiben. Diese werden schlicht zufällig in eine der Cliquen eingefügt.

TABUSEARCH: Die Funktion ist eine typische Anwendung der Tabu Metaheuristik. Es werden alle Möglichkeiten betrachtet, einen Knoten, mit Konflikt innerhalb der Partition in eine andere Partition einzufügen. Es wird letztlich der Wechsel durchgeführt, der die geringsten Kosten aufweist, also mit insgesamt am wenigsten Konflikten verbunden ist. Weiter wird der ausgeführte Wechsel der Klasse in einer Tabu-Liste vermerkt und für eine variable Anzahl an Iterationen $l = f(P0) + random(1, 10)$ tabuisiert, es ist also nicht möglich, den Wechsel vor Ablauf der Iterationen rückgängig zu machen. Dies wird solange wiederholt, bis sich das Ergebnis t-Mal nicht verbessert hat.

UPDATEPOPULATION: Im letzen Schritt einer Iteration wird geprüft, ob das neu errechnete Element P_0 in die Population P aufgenommen werden soll, und falls ja, welche Clique-Partition die Population verlassen muss. Hierfür werden für jede Partitionierung alle Goodness-Werte [LH10, Def. 1-3] berechnet. Ein solcher Score hängt von der Population und der Distanz zu einer anderen Partitionierung ab. Für P_i $(0 \leq i < p)$ und die Population P ist dieser wie folgt definiert: $h_{i,P} = f(P_i) + e^{0.08n/D_{i,P}}$.

Der Wert $D_{i,P}$ ist die niedrigste Distanz zu einem der anderen Instanzen in P, wobei die Distanz die geringste Anzahl an Knotenwechseln, die benötigt werden, um eine Partitionierung in eine andere umzuformen, darstellt. Nun wird die Färbung mit der höchsten Goodness-Score durch P_0 ersetzt. Falls P_0 die höchste Score hat, wird mit einer Wahrscheinlichkeit von 20% die Partitionierung, welche den zweithöchsten Score aufweist, ersetzt.

Zusammengefasst lässt sich der Ablauf wie folgt beschreiben: Es wird eine möglichst gute Anfangs-Population erstellt. Diese wird dann, bis eine korrekte Lösung gefunden wurde, verbessert. Hierfür wird aus zufällig gewählten Individuen ein, hoffentlich, besseres Individuum geschaffen. Weiter wird versucht dessen Konflikte durch die Tabu-Suche zu verringern. Im Erfolgsfall ersetzt die neue Partition eine weniger gute alte. Dieser Prozess wiederholt sich, bis eine korrekte Lösung gefunden wird.

Beispiel

Es folgt ein Beispiel zur Funktionsweise. Die Eingabeparameter k, p, t, m und L seien 4, 3, 1, 2 und 10. Folgender Graph G soll hierbei als Eingabe dienen.

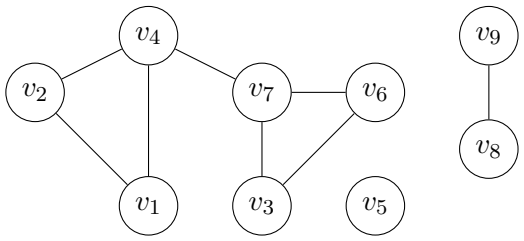


Abbildung 4.4.: Beispielgraph G

Als Erstes müssen zwei verschiedene Clique-Partitionierungen erstellt werden. Um nur das Wesentliche zu betrachten, werden diese vorgegeben.

$$S_1 = \{\{v_2, v_1\}, \{v_7, v_6, v_4\}, \{v_3, v_5\}, \{v_9, v_8\}\}$$
$$S_2 = \{\{v_2, v_7, v_4\}, \{v_1, v_6\}, \{v_3\}, \{v_5, v_8, v_9\}\}$$

Nun wird eine Iteration der Tabu-Suche für jeden dieser Partitionierungen durchgeführt. In P_1 haben v_4, v_6, v_3, v_5 Konflikte. Der beste mögliche Wechsel ist es, v_4 in die erste Partition zu verschieben, es folgt eine Reduzierung der Konfliktzahl um 1. In der zweiten Menge P_2 weisen folgende Knoten einen Konflikt auf: v_2, v_7, v_1, v_6, v_5, v_8, v_9. Ein möglicher bester Wechsel reduziert die Konfliktzahl um 1 und bewegt v_5 in die dritte Partition.

$$P_1 = \{\{v_2, v_1, v_4\}, \{v_7, v_6\}, \{v_3, v_5\}, \{v_9, v_8\}\}$$
$$P_2 = \{\{v_2, v_7, v_4\}, \{v_1, v_6\}, \{v_3, v_5\}, \{v_8, v_9\}\}$$

Die Beste dieser beiden Lösungen ist P_1 mit einer Konfliktzahl von 1, während die von P_2 bei 3 liegt. Es beginnt die erste Iteration. Da hier $p = m$ gilt, wird immer die gesamte Population selektiert. Es folgt die Kombination (CROSSOVER). P_{ij} bezeichne die j-te Partition der i-ten Partitionierung.

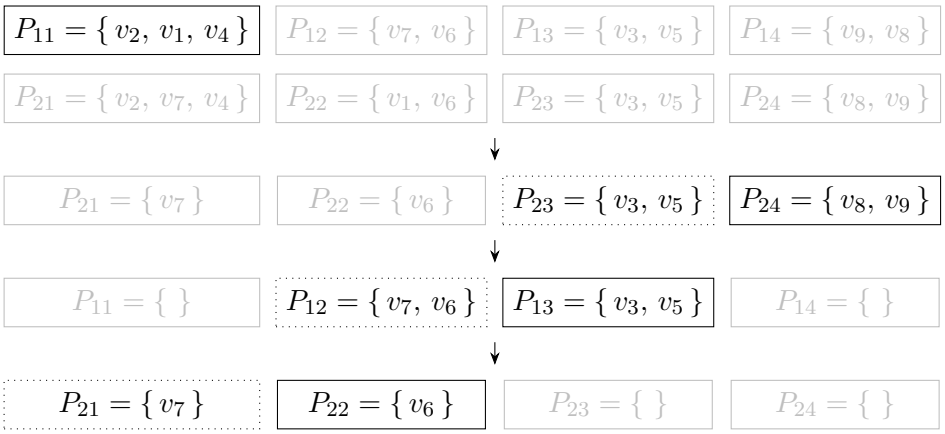


Abbildung 4.5.: Crossover-Schritt (gepunktet = Alternativwahl)

Als erste Partition der neuen Lösung wird P_{11} gewählt. Hiernach werden v_2, v_1, v_4 aus allen weiteren Partitionen entfernt. Da P_1 für die nächste Iteration nicht zur Verfügung steht, wird P_{24} als weitere Partition gewählt und deren Elemente aus allen Mengen entfernt. Dieses wird nach obigem Schaubild weitergeführt bis am Ende der Knoten v_7 ohne Partition in P_0 bleibt. Dieser kann beispielsweise in P_{02} eingefügt werden. Es resultiert folgende Menge:

$$P_0 = \{ \{ v_2, v_1, v_4 \}, \{ v_6, v_7 \}, \{ v_3, v_5 \}, \{ v_9, v_8 \} \}.$$

Für die anschließende Tabu-Suche gibt es nur eine beste Möglichkeit: das Verschieben von v_3 in die zweite Partition. Wegen der geringen Konfliktzahl wurde mit P_0 eine neue beste Partition gefunden. Hieraus folgt ebenfalls, dass P_2 in UPDATEPOPULATION gegen P_0 ausgetauscht wird. Da P_0 mit einer Konfliktzahl von 0 bereits eine optimale Lösung darstellt, terminiert der Algorithmus und liefert:

$$P' = \{ \{ v_2, v_1, v_4 \}, \{ v_3, v_6, v_7 \}, \{ v_5 \}, \{ v_9, v_8 \} \}$$

Laufzeit & Performance

POPULATIONINIT: Die Methode muss eine Laufzeit von $\mathcal{O}(p \cdot n^2)$ haben, da p Partitionierungen erstellt werden müssen, und für alle, wegen der Partitionseigenschaft, n Knoten und deren Umfeld, im ungünstigsten Fall also n^2 Knoten, betrachtet werden müssen.

TabuSearch: Die Komplexität der Tabu-Suche wird von der Prüfung aller Verschiebungen bestimmt und ist durch $\mathcal{O}(k \cdot f(S_x) \cdot n^2)$ (wobei S_x die zu betrachtende Partitionierung sei) beschränkt, da $k \cdot f(S_x)$ Möglichkeiten zum Verschieben existieren und die Berechnung der Konflikte in $\mathcal{O}(n^2)$ liegt.

Selektion: Diese Methode kann in $\mathcal{O}(p)$ ausgeführt werden, da p Partitionierungen zu berücksichtigen sind.

Crossover: Die Laufzeit liegt in $\mathcal{O}(m(k^2 + n))$. Bestimmt wird diese durch die Berechnung der größten Menge, was in $\mathcal{O}(m \cdot k^2)$ geschehen kann. Des Weiteren ist das Entfernen der Knoten, nachdem eine neue Partition zu S_0 hinzugefügt wird, durch $\mathcal{O}(m \cdot n)$ beschränkt.

updatePopulation: Die Laufzeit liegt bei $\mathcal{O}(p \cdot k^3)$, da das Berechnen der Goodness-Scores für jede der Partitionierungen eine Zeitkomplexität von $\mathcal{O}(k^3)$ erfordert [LH10, Sektion 2.6].

Der Gesamtalgorithmus läuft in $\mathcal{O}(n^2)$, weiter sollten die meistens sehr hohe Konstante L und die recht einflussreichen Konstanten k und p zusätzlich beachtet werden.

Mit Blick auf die empirische Studie der Autoren [LH10, Kapitel 3], zeigt sich, dass der Algorithmus im Vergleich zu anderen *state-of-the-art* Algorithmen sehr gute Ergebnisse im Hinblick auf die erreichte Lösungsgüte, errechnet in einer fest vorgegeben Zeit, erzielt. Auf 84% der betrachteten Graphen (DIMACS Benchmark Graphen) wurde von MACOL die kleinste bekannte Färbung gefunden. Wobei hierbei erwähnt werden muss, dass ein Teil dieser Ergebnisse von einigen aktuelleren Algorithmen wie [MM16] weiter verbessert wurden.

4.5. Implementierungen

Auch die in diesem Kapitel erläuterten Algorithmen sollen implementiert werden. Hierfür gilt das gleiche wie in Kapitel 3.5. Es wurden auch die selben Datenstrukturen verwendet, welche in der Tabelle 3.2 aufgelistet wurden. Dementsprechend liefert die Sektion vor allem Hinweise auf die Nutzung von Datenstrukturen und die Implementierung von stark abstrahierten Passagen.

Algorithmus von Pirwani und Salavatipour

Im Folgenden sollen drei Aspekte der Implementierung erläutert werden: die Datenstrukturen, die Implementierung vom Gitter und die Berechnung der optimalen Clique-Partitionierung in einer Region.

Datenstrukturen Alle verwendeten Mengen werden als ArrayLists dargestellt. Bei ganzen Zahlen als Inhalt wurden spezialisierte ArrayLists verwendet, welche unnötige Allokationen von Objekten vermeiden. Weiter ist der Grund hierfür, dass die Leseoperation dominiert, während Entfernen kaum notwendig ist. Der Graph wurde aus den gleichen Gründen als ArrayGraph implementiert. Das Model ist als Int2ObjectHashMap umgesetzt, da dadurch ein schnelles Lesen der zu den Knoten zugehörigen Punkte ermöglicht wird.

Gitter Um das Gitter aufzuteilen, werden durch die Berechnung der niedrigsten und höchsten x- und y-Werte die Anzahl der nötigen Quadrate errechnet (folgend als min_x, max_x, min_y, max_y bezeichnet). Es wird, falls k die Länge/Breite ist, die folgende Anzahl an relevanten Quadraten berechnet:

$$n_q = \left(max \begin{pmatrix} max_x - min_x, \\ max_y - min_y \end{pmatrix} \frac{1}{k} + 1 \right)^2$$

Nun wird in einer doppelt verschachtelten Schleife über die Quadrate iteriert, wobei jede genau $\sqrt{n_q}$-mal durchläuft. Die beiden Laufparameter identifizieren den Ort von dem aktuellen Quadrat. Der genaue Ort wird mithilfe der minimalen x- und y-Werte, der Laufparameter, den Zufallswerten a und b und der Größe k berechnet. Folgende Formeln können dafür verwendet werden:

$$realX = min_x + k(x-1) + a$$
$$realY = min_y + k(y-1) + b$$

Somit stellen obige Koordinaten zusammen mit der Länge/Breite k ein Rechteck dar. In diesem Rechteck werden alle enthaltenen Knoten gesucht und die optimale Clique-Partitionierung innerhalb der Region berechnet.

Optimale Clique-Partitionierung Die Berechnung der optimalen Clique-Partitionierung innerhalb einer Region wird mithilfe eines B&B-Algorithmus erledigt. Der Aufbau ist analog zum Branching-Teil des MCQ-Algorithmus. Hierbei wird die Bedingung, dass keine konvexe Hülle mit einer der anderen Hüllen kollidiert, geprüft, um eine Lösung als Kandidaten zu identifizieren. Beim Bounding wird geprüft, ob die Größe der aktuellen Lösung k^2 überschreitet.

MACOL

Nun wird ein ausgewählter Teil der Implementierung vom MACOL-Algorithmus erläutert. Hierbei werden die verwendeten Datenstrukturen und die Berechnung der Goodness-Score behandelt.

Datenstrukturen Die Mengen werden, je nach Inhalt, als ArrayList oder (auf primitive Typen) spezialisierte ArrayList umgesetzt. Von der schnellen Lese-Performance wird in allen Fällen am meisten profitiert. Des Weiteren wird an einer Stelle eine Object2IntHashMap verwendet, um eine schnelle Abbildung von Partitionierungen auf Konflikt-Werte zu ermöglichen und diese zwischenzuspeichern. Es wurde hierfür eine HashMap gewählt, da diese sowohl schnelles Einfügen als auch Auslesen garantiert. Der Graph wird analog zum Algorithmus von Pirwani und Salavatipour als ArrayGraph implementiert.

Goodness-Score Die Distanz zwischen zwei Partitionierungen zu berechnen, ist nichttrivial. Es gebe zwei Clique-Partitionierungen P_1 und P_2, k sei die Anzahl an Partitionen. Die Partitionen werden mit P_{1i} bzw. P_{2j} (mit $0 \leq j, i < k$) bezeichnet. Um nun die Distanz, wie in Sektion 4.4.1 beschrieben, zu berechnen, wird ein Graph mit $2k$ Knoten erstellt. Jeder dieser Knoten stellt eine Partition von P_1 bzw. P_2 dar. Jeder Knoten, der für eine Partition in P_1 steht, ist mit jedem Knoten, der für eine Partition in P_2 steht, verbunden. Nun wird jede Kante gewichtet. Für zwei Knoten u_1 und v_1, wobei der erste Knoten zu einer Partition P_{1i} gehört und der zweite zu P_{2j}, ist das Kantengewicht $w_{ij} = |P_{1i}| - \mathit{shared}_{ij}$. Der Wert shared_{ij} bezeichne hierbei die Anzahl der gemeinsamen Knoten. Der resultierende Graph ist offenbar ein bipartiter Graph (ein Graph, der mit zwei stabilen Mengen überdeckbar ist). Die Anzahl an Umformungen, die minimal nötig sind, um $P1$ in $P2$ (oder andersherum) zu überführen, kann mithilfe der ungarischen Methode [Kuh55] berechnet werden. Es wird hierbei ein gewichtetes Minimum-Weighted-Matching M erstellt und die Summe der Kantengewichte von den enthaltenen Kanten berechnet. Dieses ist die gewünschte Distanz (die Methode ist angelehnt an [LH10, Sektion 2.6]).

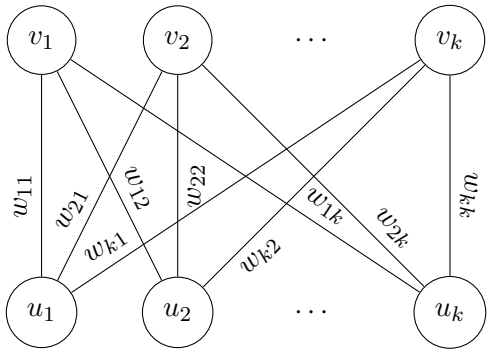


Abbildung 4.6.: Schaubild zum bipartiten Graphen

5 | Minimum Clique Cover

Das letzte der hier behandelten drei Hauptprobleme ist Minimum Clique Cover (MCC). Begonnen wird erneut mit der Definition (angelehnt an [CK97]) des Problems.

Instanz Ein Graph $G = (V, E)$.

Lösung Eine Menge an Teilmengen V_i mit $0 \leq i < n$ von V, wobei jeder der Teilmengen auf dem induzierten Teilgraph eine Clique darstellen muss. Des Weiteren muss jede Kante in G in einem der induzierten Teilgraphen enthalten sein.

Maß Der Parameter n.

Ziel Die Minimierung von n.

5.1. Verhältnis zu MCP

Da hier die Approxmierbarkeit von Minimum Clique Cover im Hinblick auf Minimum Clique Partition bestimmt wird (siehe nächste Sektion dazu), werden die Reduktionen von letzterem auf ersteres und vice versa kurz erläutert.

5.1.1. Reduktion auf MCP

Kou, Stockmeyer und Wong [KSW78] haben gezeigt, dass es eine c-Reduktion von Minimum Clique Cover auf Minimum Clique Partition mit einem Reduktionsfaktor von 1 gibt. Sie schlagen dafür folgenden Algorithmus vor.

Algorithmus 8: Graph: MCC zu MCP

```
1: procedure graphMccToMcp(G)                          ▷ G := (V, E)
2:     G' := (V', E')
3:     V' := { v'_1, ..., v'_m } wobei E := { e_1, ..., e_m }
4:     I_ij := Knoten, die benachbart zu den Kanten e_j oder e_i auf liegen (1 ≤ i <
           j ≤ m).
5:     E' := enthält eine Kante zwischen zwei Knoten n'_i und n'_j, falls I_ij eine Clique
           auf G enthält
6:     return G'
```

Um mithilfe der Reduktion einen beliebigen MCP-Algorithmus zum Berechnen einer Lösung von MCC zu nutzen, wird der obige Algorithmus auf einem Graph G ausgeführt und mittels des MCP-Algorithmus eine Lösung errechnet. Um eine gültige Lösung für MCC zu bekommen, muss jeder Knoten durch die, mit der dazugehörigen Kante verbundenen, Knoten ersetzt werden. Dazu folgt ein kurzes Beispiel.

Beispiel 4. *Gegeben sei folgender Graph (links):*

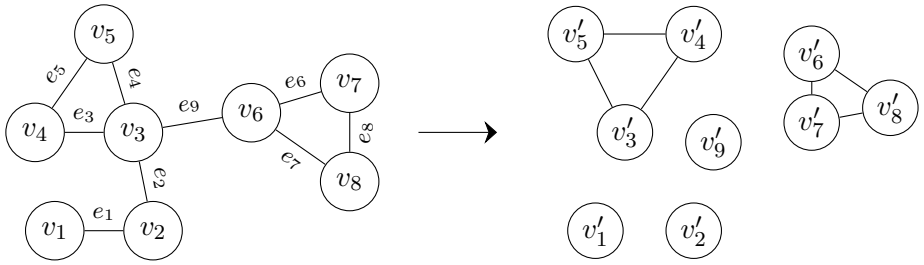


Abbildung 5.1.: links) Ursprungsgraph G; rechts) G'

Nun ist für jede Kante ein Knoten zu erstellen. Es müssen jeweils zwei verbunden werden, falls die zugehörigen Kanten mit ihren verbundenen Knoten eine Clique bilden. Nachdem dieses auf alle neuen Knoten angewendet wurde, resultiert der rechte Graph in Abbildung 5.1. Als nächster Schritt wird ein beispielsweise optimaler MCP*-Algorithmus auf den rechten Graphen angewendet und es folgt die Lösung:*

$$V_0 = \{ v'_1 \} \qquad V_1 = \{ v'_9 \}$$
$$V_3 = \{ v'_5, v'_8, v'_7 \} \qquad V_4 = \{ v'_3, v'_4, v'_5 \} \qquad V_2 = \{ v'_2 \}.$$

Im letzten Schritt werden die zu den neuen Knoten gehörigen, ursprünglichen Kanten betrachtet und ersetzt die Knoten in $V_0, \ldots, V_4$ durch jene Knoten, die mit diesen ursprünglichen Kanten auf dem Ursprungsgraph verbunden sind. Beispielsweise gehört e_1 zu v'_1 und da e_1 mit v_1 und v_2 auf G verbunden ist, muss v'_1 durch diese ersetzt werden. Nach der Durchführung für alle neuen Knoten resultiert folgende Lösung:

$$V_0 = \{ v_1, v_2 \} \qquad V_1 = \{ v_3, v_6 \}$$
$$V_3 = \{ v_5, v_8, v_7 \} \qquad V_4 = \{ v_3, v_4, v_5 \} \qquad V_2 = \{ v_2, v_3 \}$$

Es ist leicht erkennbar, dass diese Lösung sowohl richtig als auch optimal ist.

5.1.2. Reduktion von MCP

Die gleichen Autoren [KSW78] haben auch gezeigt, dass es eine c-Reduktion von Minimum Clique Partition auf Minimum Clique Cover, mit $r = 1 + \varepsilon$ wobei $\varepsilon > 0$, gibt.

Folgender Algorithmus wird von ihnen benutzt, um im ersten Schritt den Graphen passend umzuschreiben.

Algorithmus 9: Graph: MCP zu MCC

```
1: procedure GRAPHMCCTOMCP(G)                          ▷ G := (V, E)
2:     G' := (V', E')
3:     t := |E| + 1
4:     V' := V ∪ { u_1, ..., u_t }                     ▷ U := { u_1, ..., u_t }
5:     E' := E
6:     Füge zu E' von jedem Knoten u_i eine Kante zu allen Knoten hinzu.
7:     return G'
```

Um letztlich eine korrekte Lösung für MCP zu bekommen, muss noch die Lösung von MCC auf G' in eine korrekte Clique-Partitionierung umgeschrieben werden. Dafür wird von den Autoren Algorithmus 10 vorgeschlagen. Die Eingabe hier ist die Lösung von MCC auf G' P und die Variable t aus dem vorherigen Algorithmus.

Algorithmus 10: Lösung: MCP zu MCC

```
1: procedure SOLUTIONMCCTOMCP(P, t)                    ▷ P := { P_1, ..., P_k }
2:     P(i) := alle Cliquen in P, die u_i enthalten, ohne u_i (1 ≤ i ≤ t)
3:     min_i := Das i mit dem niedrigsten Wert |P(i)|.
4:     return P(min_i), nach dem Entfernen doppelter Knoten.
```

Beispiel 5. *Gegeben sei folgender Graph (links), der Parameter t beträgt offenbar* 5.

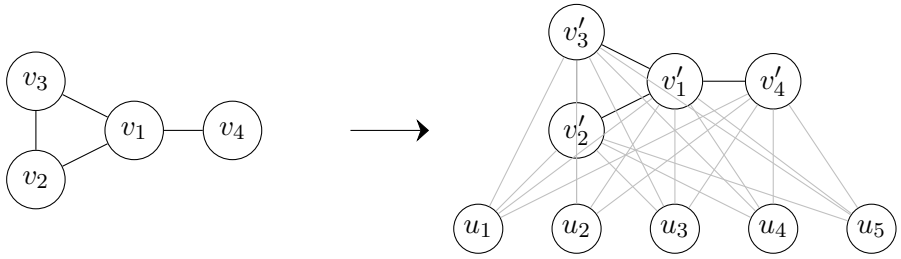


Abbildung 5.2.: links) Ursprungsgraph G; rechts) G'

Es werden jetzt t Knoten erstellt, dem Graphen hinzugefügt und zu allen weiteren Knoten (außer u_i) eine Kante erstellt. Als Folge entsteht der Graph auf der rechten Seite. Nach Anwendung eines beispielsweise optimalen Algorithmus für MCC *auf G' resultiert folgende Lösung:*

$P_0 = \{ v'_1, v'_2, v'_3, u_1 \}$
$P_1 = \{ v'_1, v'_2, v'_3, u_4 \}$
$P_2 = \{ v'_4, u_3 \}$
$P_3 = \{ v'_1, v'_2, v'_3, u_2 \}$
$P_4 = \{ v'_1, v'_4, u_1 \}$
$P_5 = \{ v'_4, u_4 \}$
$P_6 = \{ v'_1, v'_2, v'_3, u_3 \}$
$P_7 = \{ v'_4, u_2 \}$.

Bei der Betrachtung von allen P_i' fällt auf, dass P_2' die wenigsten Knoten insgesamt enthält und auch die einzig gültige Partitionierung ist. Somit kann in diesem Fall das Entfernen doppelter Knoten gespart werden und es ergibt sich folgende Lösung:

$$P_3 = \{ v_1, v_2, v_3 \} \qquad P_7 = \{ v_4 \}.$$

5.2. Approximierbarkeit

Wie bisher alle Clique-Probleme, liegt auch Minimum Clique Cover in $\mathcal{NPO}$.

Beweis. Zuerst muss gezeigt werden, dass jede Lösung polynomiell überprüfbar ist. Hier ist dieses machbar, indem für jede Kante geprüft wird, ob die beiden dazu inzidenten Knoten in einer der Mengen enthalten sind. Als Nächstes muss geprüft werden, ob jede Menge eine Clique darstellt. Beides zusammen ist in $\mathcal{O}(n^3)$ $(n = |V|)$ machbar.

Die Länge der Lösung muss polynomiell beschränkt sein. Jede Menge kann höchstens n Knoten enthalten, und es kann maximal n Mengen geben, wegen der Codierung ist die Länge durch $\mathcal{O}(n^3)$ beschränkt.

Die Maßfunktion muss polynomiell berechenbar sein. Dieses ist der Fall, da es sich hier um das Zählen von ungünstigstenfalls $\mathcal{O}(n)$ Mengen handelt. □

Untere Schranke Eine Möglichkeit, die untere Schranke zu bestimmen, ist, sich der approximativen Äquivalenz [Sim90] zu MinCliquePart zu bedienen. Da sogar MCC $\leq_c^1$ MCP gilt, folgt laut Kann [CK97] für eine beliebige Abbildung $f\colon \mathbb{R} \to \mathbb{R}$:

$$\text{MCP ist nicht approx. in } f(|V|) \Rightarrow \text{MCC ist nicht approx. in } f(|V|)^2.$$

Außerdem wurde 1994 von Lund und Yannakakis [LY94] bewiesen, dass Minimum Clique Cover nicht innerhalb von $|V|^\varepsilon$, mit einem $\varepsilon > 0$, approximiert werden kann. Es folgt somit auch MCC $\notin \mathcal{APX}$.

Obere Schranke Auch hier ist die Schranke mit MinCliquePart verknüpfbar. Laut Halldórsson [CK97] ist MinCliqueCover in $\mathcal{O}(f(|E|))$ approximierbar, wenn MinCliquePart in $f(|V|)$ approximierbar ist.

Beweis. Wie in Sektion 5.1 ausgeführt, gibt es eine C-Reduktion mit $r = 1$ von MCC auf MCP. Wie ebenfalls in dieser Sektion erläutert, werden bei dieser Reduktion alle Kanten in Knoten umgeschrieben, somit lässt sich MCC durch jeden Approximationsalgorithmus für MCP lösen. Nun hängt die Funktion f im Fall von

MCC allerdings linear von der Anzahl der Kanten ab und da im ungünstigsten Fall $|E| = {}^{n(n+1)}/_{2}$ gilt, liegt $f(|V|)$ in jedem Fall in $\mathcal{O}(f(|E|))$. □

Weiter wurde die konkrete obere Schranke $\mathcal{O}(|V|^{2(\log\log|V|)^2/(\log|V|)^3})$ in [Aus+99] bewiesen.

Eingeschränkte Graphen Auch dieses Problem wird, wegen der schweren Approximierbarkeit auf allgemeinen Graphen, sehr oft auf speziellen Graphenklassen untersucht. Eine Auswahl an Ergebnissen sind in der Tabelle 5.1 eingetragen.

Tabelle 5.1.: Laufzeit & Performanz auf eing. Graphen

Graphenklasse	Performanzrate	Laufzeit	Referenz
chordal	1	$\mathcal{P}$	[Gav72]
1-interval circular-arc	1	$\mathcal{P}$	[GLL82]

5.3. Algorithmen

Algorithmen, die speziell für Minimum Clique Cover entwickelt wurden gibt es verhältnismäßig wenige, denn das Problem lässt sich ohne großen Mehraufwand durch die c-Reduktion auf MCP lösen. Dennoch gab es einige sehr praxisnahen Veröffentlichungen, die sich mit der direkten Lösung von MCC beschäftigen.

Exakte Algorithmen Wie für viele Clique-Optimierungsprobleme, so sind auch beim exakten Lösen von MCC *branch-and-bound*, oder Varianten solcher, Algorithmen vorherrschend. Ein aktueller Algorithmus dieser Gruppe wurde von Gramm, Guo, Hüffner und Niedermeier [Gra+06] entwickelt. Dieser nutzt die Methode der Datenreduzierung (engl. *data reduction*). Hierbei wird das zu MCC gehörige Entscheidungsproblem betrachtet und eine Instanz dessen nach bestimmten Regeln in eine einfacher zu lösende Instanz konvertiert, von der bewiesen wurde, dass sie die Lösbarkeit der ursprünglichen Instanz impliziert. Nun macht sich der Algorithmus dieses Prinzip zunutze, um besonders aussichtsreiche Zweige im gedachten Suchbaum zu besuchen.

Approximationsalgorithmen Für MCC gibt es keine erwähnenswerten Approximationsalgorithmen auf generellen Graphen. Von Interesse sind höchstens die Approximationsalgorithmen für MCP.

Heuristische Algorithmen Viele *state-of-the-art* Algorithmen basieren auf der Heuristik von Kellerman [Kel]. Diese (Meta-)Heuristik nummeriert alle Knoten und betrachtet jeden Knoten und die Nachbarn, die eine geringere Nummer zugewiesen bekommen haben, und versucht, den Knoten in eine Clique einzuordnen, die eine Untermenge der betrachteten Nachbarn darstellt (genaueres kann bei [KSW78] nachgelesen werden). Ein sehr aktueller Vertreter dieser Heuristik ist ein Algorithmus von Gramm, Gui, Hüffner und Niedermeier [Gra+06]. Dieser baut auf dem Algorithmus von [KSW78] auf und verbessert seine Laufzeit signifikant.

6 | Weitere Probleme

Außer den drei ausführlich behandelten Clique-Problemen existieren noch weitere Probleme. Eine Auswahl von drei weiteren Optimierungsproblemen werden im Folgenden steckbriefartig erläutert.

6.1. Maximum Quasi-Clique

Bei dem Maximum Quasi-Clique (MQC) Problem wird eine möglichst große γ-Quasi-Clique gesucht, wobei diese eine Kantendichte von γ auf dem induzierten Teilgraphen haben muss. Die formale Definition vom Optimierungsproblem folgt nun, diese wurde in ähnlicher (äquivalenter) Form bereits beispielsweise von [Pat+13] aufgestellt.

Instanz Ein Graph G und eine fixe Konstante $\gamma \in [0, 1]$.

Lösung Eine γ-Quasi-Clique, diese wird durch einen Knotenmenge V' dargestellt. Die Knotenmenge V' ist genau dann eine γ-Quasi-Clique wenn sie auf dem induzierten Teilgraphen eine Kantendichte $\geq \gamma$ hat. Dieses bedeutet, dass $|V'|$ im Verhältnis zu ${}^{n(n-1)}/_{2} \geq \gamma$ sein muss.

Maß Die Anzahl der Elemente in V'.

Ziel Die Maximierung von $|V'|$.

Approximierbarkeit Das Problem ist für $\gamma = 1$ trivialerweise äquivalent zu Maximum Clique, folglich gilt die untere Schranke von MaxClique (siehe Abschnitt 3.1) auch für den allgemeinen Fall von Maximum Quasi-Clique. Somit folgt auch, dass das Problem nicht in $\mathcal{APX}$ liegen kann. Die Mitgliedschaft in $\mathcal{NPO}$ hingegen ist offensichtlich und kann analog zu MaxClique gezeigt werden.

Algorithmen Approximationsalgorithmen sind mir keine bekannt. Gängige Heuristiken für das Problem entsprechen ungefähr den typischen Maximum Clique Heuristiken. Einer der *state-of-the-art* Algorithmen [BHB07] ist sogar eine Abwandlung vom bereits gezeigten DLS-MC Algorithmus. Des Weiteren lassen sich auch exakte Algorithmen von MaxClique für MQC anpassen.

6.2. Minimum Biclique Cover

Das Minimum Biclique Cover (MBC) Problem ist eine Variation von Minimum Clique Cover, wobei statt einer Clique-Überdeckung eine Biclique-Überdeckung gesucht wird. Es gibt noch weitere solcher Variationen für Maximum Clique und Minimum Clique Partition.

Instanz Ein Graph $G = (V, E)$.

Lösung Eine Menge an Teilmengen V_i mit $0 \leq i < n$ von V, wobei jeder der Teilmengen auf dem induzierten Teilgraph eine Biclique darstellen muss. Eine Biclique ist hierbei die Vereinigung von zwei Knotenmengen X und Y, bei denen jeder Knoten aus X mit jedem Knoten aus Y verbunden ist. Innerhalb der Mengen dürfen keine Kanten existieren. Des Weiteren muss jede Kante in E in einem der induzierten Teilgraphen enthalten sein.

Maß Der Parameter n.

Ziel Die Minimierung von n.

Approximierbarkeit Das Problem gehört zur Klasse $\mathcal{NPO}$, der Beweis funktioniert analog zu Minimum Clique Cover (es wird geprüft, ob die Lösungsmengen V_i Bicliquen statt Cliquen sind, dies ist auch in polynomieller Zeit durchführbar). Eine untere Schranke wurde von Chalermsook, Heydrich, Holm und Karrenbauer [Cha+14] bewiesen und zeigt, dass das Problem nicht polynomiell in $n^{1-\varepsilon}$ mit $\varepsilon > 0$ approximierbar ist, somit folgt MBC $\notin \mathcal{APX}$. Von den gleichen Autoren wurde eine obere Schranke gezeigt, wobei bewiesen wurde, dass Approximationsraten der Form $n/(\log n)^{\gamma}$ mit $\gamma > 0$ erreicht werden können.

Algorithmen Ein Vorteil von dem Problem ist, dass es, wie im Kapitel 7 ersichtlich, eine c-Reduktion von/auf Minimum Clique Cover mit $r = 1$ existiert. Folglich ist das Problem auch über die Algorithmen zum MCC lösbar.

6.3. Maximum Edge Clique Partitioning

Das letzte hier behandelte Clique-Problem ist das Maximum Edge Clique Problem (MaxECP). Eine typische Definition lautet wie folgt.

Instanz Ein Graph $G = (V, E)$.

Lösung Sei eine Clique-Partition $V_1, \ldots, V_m$, wobei $G_i = (V_i, E_i)$ $(0 < i \leq m)$ den induzierten Teilgraph von V_i darstellt. Alle G_i müssen jeweils eine Clique sein.

Maß Die Anzahl der Kanten aller Partitionen: $\sum_{i=1}^{m} |E_i|$.

Ziel Die Maximierung des Maßes.

Von diesem Problem existiert eine weitere Variation in der das Ziel die Minimierung des Maßes ist.

Approximierbarkeit Die Autoren von [Des+07] haben bewiesen, dass das Problem nicht innerhalb von $n^{1-o(1)}$ approximiert werden kann (es sei denn $\mathcal{NP} \subseteq \text{ZPTIME}(2^{(\log n)^{\mathcal{O}(1)}})$). Als beste obere Schranke gilt ein Satz von Punnen und Zhang [PZ12]: Wenn sich Maximum Clique in δ approximieren lässt, dann kann das Problem MaxECP in $2(p\delta-1)/p-1$ für alle $p \geq 2$ approximiert werden (je größer p desto langsamer der Algorithmus).

Algorithmen Die im vorherigen Paragraphen zitierten Artikel beschreiben jeweils Approximationsalgorithmen mit schwachen Performanzraten. Darüber hinaus wird von [Des+07] ein 2-Approximationsalgorithmus erläutert, welcher davon ausgeht, dass sich Maximum Clique in polynomieller Zeit exakt lösen lässt. Dieses ist auf einigen Graphenklassen (siehe Kapitel 3) der Fall. Ansonsten ist auch die Nutzung von heuristischen Algorithmen denkbar, welche, wie in Kapitel 3 erläutert, auch auf großen Graphen schnell optimale Lösungen finden können. Somit existiert ein polynomieller 2-Approximationsalgorithmus, wenn sich auch Maximum Clique heuristisch/exakt in polynomieller Zeit optimal lösen lässt.

7 | Relationen

Nachdem insgesamt sechs verschiedene Clique-Probleme und deren Algorithmen untersucht wurden, bleibt jetzt noch übrig, Relationen zwischen den Problemen herzustellen. Dieses wird häufig durch Reduktionen erledigt. Nun gibt es bei herkömmlichen Reduktionen das Problem, dass diese keine Möglichkeit liefern, das Problem auf dem darauf zu reduzierenden Problem zu lösen und eine möglichst gleichwertige Lösung auf dem Ursprungsproblem zu erhalten, also zum Beispiel P1 durch das Lösen von P2 zu lösen und eine Lösung für P1 zu erhalten, die höchstens um r schlechter ist (als die Lösung auf P2). Dieses Szenario ist in der Praxis aber sehr relevant, da sich viele Probleme sehr ähneln und das Finden von neuen Algorithmen zeitaufwendig ist. Die Lösung sind approximationserhaltende Reduktionen, welche genau das Szenario abdecken. Zu diesem Zweck wurde im Kapitel 2 die c-Reduktion eingeführt. In diesem Kapitel werden hauptsächlich einige wichtige Reduktionen zwischen den Clique-Problemen behandelt, wobei auch die zwei eng verwandten Probleme Maximum Independent Set und Minimum Graph Coloring miteinbezogen werden.

7.1. Die „großen“ Drei

Als Erstes werden Reduktionen zwischen den „großen“ drei Clique Probleme Maximum Clique, Minimum Clique Cover und Minimum Clique Partition angeschaut. Die Zusammenhänge zwischen diesen Problemen wurde u.a. von Simon erforscht [Sim90] und lassen sich wie folgt in einer Grafik zusammenfassen.

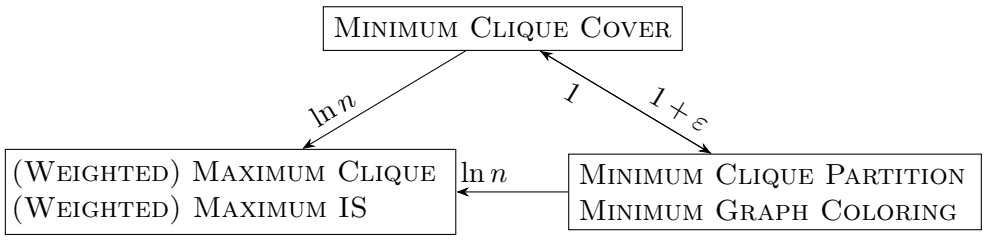


Abbildung 7.1.: Approximationserhaltende Reduktionen

Die Grafik ist so zu lesen, dass immer wenn es einen Pfeil von Problem 1 auf Problem 2 mit der Beschriftung x gibt, eine Reduktion der Form Problem 1 $\leq_c^x$ Problem 2 existiert. Befinden sich Problem 1 und Problem 2 in dem gleichen Rechteck so bedeutet es, dass Problem 1 $\leq_c^1$ Problem 2 und Problem 2 $\leq_c^1$ Problem 1 gilt.

Die Reduktionen von/auf Maximum Clique auf/von Maximum Independent Set ist trivial, es ist offenbar nur das Invertieren des Graphen notwendig.

Die Reduktionen zwischen Minimum Clique Cover und Minimum Clique Partition wurden bereits im Kapitel 5.1 ausführlich behandelt.

Es bleibt die Reduktion von Minimum Clique Partition und Minimum Clique Cover auf Maximum Clique. Die Grundidee hierbei ist, dass ein Algorithmus für MaxClique (bzw. einer Variation von dem Problem siehe hierfür [Sim90, Kapitel 6]) nach dem Greedy-Verfahren angewendet wird, die Lösung aus dem Graphen entfernt und wieder angewendet wird, bis eine gültige Lösung gefunden wurde.

7.2. Die „kleinen“ Drei

Nun kann obige Grafik erweitert werden, hierfür sollen die drei „kleinen“ Clique-Probleme aus Kapitel 6 eingearbeitet werden.

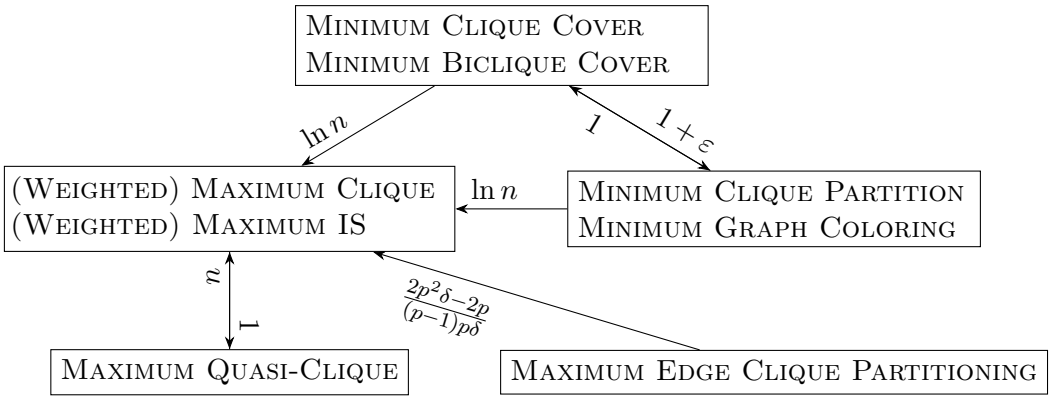


Abbildung 7.2.: Endgültige Übersicht über die Reduktionen

Das Minimum Biclique Cover Problem wurde von Simon [Sim90] untersucht, wobei er eine Reduktion auf/von MCP mit $r = 1$ bewiesen hat.

Die Reduktion Maximum Clique $\leq_c^1$ Maximum Quasi-Clique ist trivial, beim Abbilden auf eine Quasi-Clique Instanz wird $\gamma = 1$ gewählt, dadurch folgt bereits die Existenz der c-Reduktion.

Schließlich kann noch der Algorithmus von Punnen und Zhang [PZ12], welcher eine Performanz von $2(p\delta-1)/p-1$, mit δ als Performanzrate vom MaxClique Algorithmus und $p \geq 2$, verwendet werden. Dieser Sachverhalt ist in eine c-Reduktion umformulierbar. Um den Parameter r angeben zu können, muss die Verschlechterung der Performanz durch die Lösung von MaxECP mittels MaxClique Algorithmus ausgedrückt werden. Durch simple Umformungen ergibt sich der Term $2p^2\delta-2p/(p-1)p\delta$.

Beweis. Eine c-Reduktion ist formal definiert als 2-Tupel der Funktionen (f, g). Wir geben diese Funktionen nun an. Es gilt $f(x) = x$ und $g(x, A(x)) = A'(x, A)$, wobei A' der Algorithmus von Punnen und Zhang sei, A sei ein Algorithmus für MaxClique. Damit ist die c-Reduktion angegeben. Es wird nun gezeigt, dass die beiden geforderten Bedingungen erfüllt werden:

1. Bei beiden Problemen ist die Instanz ein Graph G und da $f(x) = x$ muss $f(x)$ immer dann eine Instanz für das MaxClique sein, falls x eine Instanz für MaxECP ist.
2. Da g als Algorithmus für MaxECP definiert ist, gibt es für alle x auch eine gültige Lösung zurück.

Zuletzt soll der Faktor r bestimmt werden. Die Performanz von A' ist in Abhängigkeit von der Performanz von dem MaxClique Algorithmus A gegeben, somit soll nach der Definition gelten:

$$\frac{2(p\delta-1)}{p-1} \leq r \cdot \delta \text{ (mit } p \geq 2) \tag{7.1}$$

Es ist klar, dass r der Faktor ist, der durch das Ausklammern von δ (linke Seite) entsteht. Dabei ist wie folgt vorzugehen:

$$\begin{aligned}
\frac{2(p\delta-1)}{p-1} &\Leftrightarrow \frac{2p}{p-1}\left(\delta - \frac{1}{p}\right) \\
&\Leftrightarrow \frac{2p}{p-1}\delta - \frac{2p}{(p-1)p} \\
&\Leftrightarrow \delta\left(\frac{2p}{p-1} - \frac{2p}{(p-1)p\delta}\right) \\
&\Leftrightarrow \delta\left(\frac{2p^2\delta-2p}{(p-1)p\delta}\right).
\end{aligned}$$

Somit erfüllt $r = 2p^2\delta-2p/(p-1)p\delta$ die Ungleichung (7.1). Es resultiert die Existenz der beschriebenen c-Reduktion. □

8 | Resümee

Bei der Gesamtbetrachtung der Clique-Optimierungsprobleme fällt immer wieder die Tatsache auf, dass ausnahmslos jedes dieser Probleme hart polynomiell approximierbar ist, es wird in Teilen der Literatur sogar von unapproximierbar gesprochen. Diese Richtung ist auf allgemeinen Graphen eine Sackgasse. Bei der Betrachtung von eingeschränkten Graphen sieht dieses schon wieder ganz anders aus. In der Arbeit wurden einige Möglichkeiten aufgezeigt, Clique-Probleme auf solchen Graphen effizient, oft sogar optimal in polynomieller Zeit, zu lösen.

Des Weiteren zeigen die vielen Studien der Autoren von heuristischen Algorithmen für Clique-Optimierungsprobleme, dass diese Gruppe geeignet ist, um solche Probleme auf allgemeinen Graphen zu lösen. Es war vor allem zu beobachten, dass bei den meisten Clique-Problemen *local-search*, teilweise auch als Hybrid mit der evolutionären Metaheuristik, sehr effektiv ist.

Auch die Bedeutung von approximationserhaltenden Reduktionen ist mehrfach deutlich geworden. Zwei der am stärksten verbreiteten Clique-Probleme Minimum Clique Partition und Minimum Clique Cover sind vor allem über solche Reduktionen lösbar und auch die weniger stark erforschten Probleme aus Kapitel 6 (außer Maximum Quasi-Clique) sind aktuell am besten über geeignete Reduktionen zu lösen. Gerade bei einer großen Anzahl an ähnlichen Problemen ist es nur logisch, aufwändige Algorithmenentwicklung zu sparen und – falls effiziente existieren – stattdessen einmalig eine Reduktion zu finden.

8.1. Ausblick

Neben den in der Arbeit behandelten Aspekten, gibt es in der Literatur für einige Clique-Probleme noch andere Formulierungsarten und damit verbundene Lösungsansätze (beispielsweise ganzzahlige lineare Optimierung) [Bom+99]. Des Weiteren existieren noch eine Menge weiterer Clique-Probleme, von denen viele noch sehr spärlich untersucht oder erst in naher Vergangenheit bekannt wurden. In die erstere Kategorie fallen beispielsweise Maximum Labelled-Clique [DCC11] und Minimum Dominating Clique [KL06]. Eines der wohl neusten Clique-Probleme ist Maximum Fuzzy Clique [BB09].

A | Das Clique-Framework

Das *Clique-Framework* ist ein in Java geschriebenes Konsolenprogramm und wurde entwickelt, um die Implementierung von Algorithmen zu den Clique-Optimierungsproblemen zu vereinfachen. Es stellt die Möglichkeit bereit, die Implementierung in der Skriptsprache Lua zu schreiben. Des Weiteren liefert es nützliche Utility-Funktionen, Datenstrukturen und es kümmert sich komplett um das Einlesen von Eingabeparametern, wie Graphen und Modellen (beispielsweise für UDG- oder Intervall-Graphen). In diesem Kapitel wird das Programm kurz technisch beschrieben und eine Anleitung für die Verwendung geliefert.

A.1. Struktur

Das Programm läuft auf der Konsole und ist modular aufgebaut. Jede Funktion wird durch einen Modus angesprochen. Es ist ein leichtes, neue Modi hinzuzufügen. Jeder Modus wird durch einen speziellen Controller implementiert. Der unvollständige Aufbau lässt sich im folgenden Klassendiagramm darstellen:

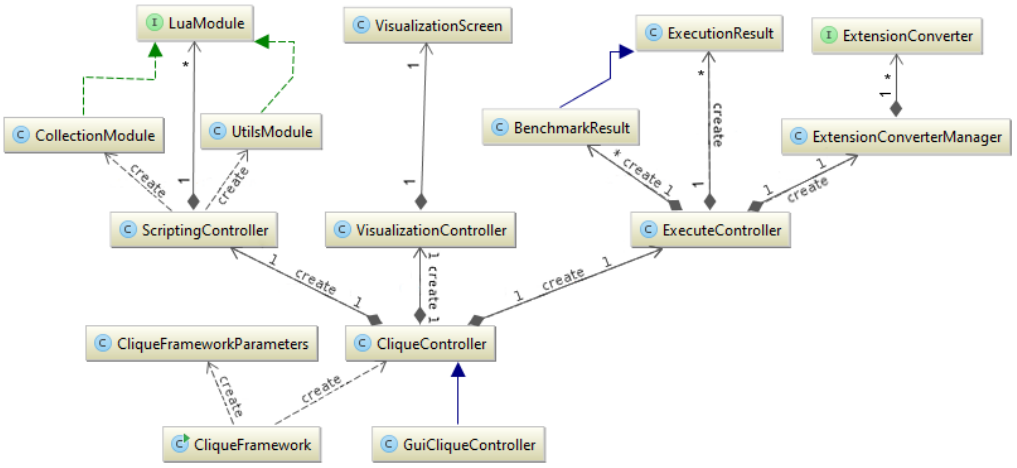


Abbildung A.1.: Das Clique-Framework als Klassendiagramm

Wie ersichtlich ist, arbeiten alle Controller vollkommen separat voneinander, weiter haben sie alle eine eigene innere Struktur. In der Klasse `CliqueFrameworkParameters` werden die Konsolenbefehle definiert. In der Zukunft kann das Framework um eine GUI erweitert werden, indem vom Fassaden-Controller geerbt wird.

A.1.1. Bibliotheken

Das Framework nutzt eine kleine Anzahl von Java-Bibliotheken, im Folgenden werden sie kurz aufgelistet und beschrieben.

LuaJ: LuaJ [Ros15] ermöglicht das Ausführen von Lua-Skripten. Ein Vorteil von dieser Art Skripte auszuführen, ist, dass Java-Klassen und Methoden den Skripten zur Verfügung gestellt werden können.

geom2D: Die Bibliothek [dle13] implementiert eine Vielzahl von geometrischer Klassen und Algorithmen. Dieses ermöglicht dem Framework das Erstellen von konvexen Hüllen und das Prüfen von Polygon-Kollisionen. Es kann noch eine Vielzahl weiterer Aufgaben erfüllen.

fastutil: Diese Abhängigkeit [Vig16] stellt eine große Anzahl an Datenstrukturen bereit. Vom Framework werden vor allem auf primitive Typen spezialisierte Listen und Maps benutzt.

JCommander: Von dieser Bibliothek [Beu16] wird das Parsen der Kommandozeilenargumente übernommen.

reflections: Diese Bibliothek [ron15] bietet erweiterte Funktionen, um Java-Reflections besser nutzen zu können. Es wird alleine für interne Zwecke benutzt.

stern-library: Durch diese Bibliothek [Ste14] werden einige nützliche Algorithmen bereitgestellt. Vom Framework wird der Kuhn-Munkres-Algorithmus [Kuh55] benutzt, um gewichtete Minimum-Matchings zu berechnen.

A.1.2. Lua-API

Um das Schreiben von Lua-Code effizienter und einfacher zu gestalten, bietet das Framework eine Reihe von APIs an, welche mithilfe von Lua-Modulen in Skripten benutzt werden können. Ein Modul kann über die globale Methode `require(modulname)` eingebunden werden. Es gibt zwei verschiedene Lua-Module: *utils* und *collections*.

Utils

sortList: Akzeptiert als Eingabe ein Instanz von `java.util.List<T>` der Java-Standard-Bibliothek und eine anonyme Lua-Funktion, welche sich exakt wie ein `java.util.Comparator<T>` auf dem entsprechenden Typen der gegebenen Liste verhalten muss. Die Methode sortiert die Liste dem Comparator entsprechend mithilfe eines iterativen Mergesort-Algorithmus [Ora].

sortGraphVertices: Funktioniert analog zu *sortList*. Der einzige Unterschied ist, dass ein `org.rschrage.cf.util.graph` statt einer Liste als Eingabe verwendet werden muss.

checkPointHullIntersection: Benötigt als Eingabe zwei Listen von Knoten und ein Model, welches den Knoten einen Punkt (`tmath.geom2d.Point2D`) zuweist. Berechnet die konvexe Hülle der Punktmengen mithilfe vom Graham-Scan Algorithmus [dle13] und prüft, ob die entstehenden Polygone sich schneiden. Falls dieses der Fall ist, wird `true` zurückgegeben.

minimumWeightedMatching: Die API akzeptiert eine Matrix, als Liste von IntArrayLists implementiert, als Eingabe. Diese stellt die Kosten der Kanten zwischen den – gedanklich – zwei Teilen des Graphen. Daraus berechnet die Methode ein Minimum-Matching.

Collections

Die *collection*-APIs benötigen keine Eingabeparameter. Jede einzelne erstellt eine Datenstruktur und gibt sie als Rückgabewert zurück. In der folgenden Tabelle werden die Klassen der jeweils erstellen Objekte, inklusive Package, aufgelistet.

Tabelle A.1.: APIs: Datenstrukturen

API	Datenstruktur
`createArray()`	`java.util.ArrayList<T>`
`createIntArray()`	`it...ints.IntArrayList`
`createIntArrayMap()`	`it...ints.Int2IntArrayMap`
`createIntHashMap()`	`it...ints.Int2IntOpenHashMap`
`createIntObjectHashMap()`	`it...ints.Int2ObjectOpenHashMap`
`createIntIntervalArrayMap()`	`it...ints.Int2ObjectArrayMap<Interval>`
`createIntObjectArrayMap()`	`it...ints.Int2ObjectArrayMap<T>`
`createIntQueue()`	`it...ints.IntArrayFIFOQueue`
`createObjectIntHashMap()`	`it...objects.Object2IntOpenHashMap`
`createTupleArray()`	`org.rschrage.cf.util.TupleArray`
`createArrayGraph()`	`org.rschrage.cf.util.graph.ArrayGraph`

Hinweis: Die Packages `it.unimi.dsi.fastutil.ints/objects` wurden in der obigen Tabelle zur besseren Darstellung mit `it...ints/objects` abgekürzt.

A.2. Anwendung

Um ein für das Framework geschriebenes Lua-Skript ausführen zu können, muss das Skript dem Programm mithilfe der Option `--file` übergeben werden, dieses kann beispielsweise folgendermaßen aussehen:

```
>>./clique-framework --file deinSkript.lua
```

Um der Hauptmethode vom Skript die benötigten Eingabeparameter übergeben zu können, wird `--input` verwendet. Alle Parameter müssen einzeln auf diese Weise übergeben werden, hierbei ist es wichtig, dass es sich um Dateien handeln muss, wobei zu beachten ist, dass die Dateiendung den Inhalt definiert.

```
>>./clique-framework --file skript.lua --input graph.graph
```

Zum Schluss folgt eine Tabelle, welche alle Optionen, die akzeptiert werden, und deren Bedeutung zusammenfasst. Des Weiteren wird in einer anderen Tabelle alle als Eingabe unterstützten Dateiformate aufgelistet.

Tabelle A.2.: Clique-Framework: Konsolenoptionen

Option	Bedeutung	Standard-Wert	Benötigt?
`--gui (-g)`	Zeigt eine GUI für das Framework. Hat aktuell keine Funktion.	`false`	✗
`--file (-f)`	Übergibt einen Pfad zu einem Lua-Skript.	-	✓
`--mode (-m)`	Definiert die Routine, es gibt folgenden Möglichkeiten: `exe, bench, vis`.	`exe`	✗
`--help (-h)`	Ruft eine Beschreibung der Optionen auf.	`false`	✗
`--input (-i)`	Übergibt einen Pfad zu einer Eingabe-Datei.	-	✗

Tabelle A.3.: Clique-Framework: Dateiendungen

Dateiendung	Inhalt
`*.graph`	Die Datei muss einen Graphen nach dem DIMACS-Format [Cai] enthalten. Wird zu einem ArrayGraphen umgewandelt.
`*.udg`	Diese sollte ein Modell für ein UDG enthalten. In jeder Zeile muss ein Knoten zu einem Punkt nach folgendem Format zugewiesen werden: `vertex->(x,y)`. Es wird daraus eine HashMap erstellt, die den Knoten auf den jeweiligen Punkt abbildet.
`*.interval`	Es enthält ein Modell für einen Intervall-Graphen. Jede Zeile enthält eine Intervalfunktion, wie folgt aufgebaut: `vertex->[a b], vertex2[c d]`. Aus der Datei wird eine Menge von HashMaps erstellt, die jeweils eine Intervallfunktion darstellen.
`*.int`	Sie sollte eine ganze Zahl enthalten.

Abkürzungsverzeichnis

MaxClique	Maximum Clique
MCP	Minimum Clique Partition
MCC	Minimum Clique Cover
MaxECP	Maximum Edge Clique Problem
MinECP	Minimum Edge Clique Problem
MQC	Maximum Quasi-Clique
MIS	Maximum Independent Set
MBC	Minimum Biclique Cover
MGC	Minimum Graph Coloring
UDG	Unit Disk Graph
IT	Induzierter Teilgraph
B&B	Branch-and-Bound

Algorithmenverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[Aus+99] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela und M. Protasi. *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer Science & Business Media, 1999.

[BB09] Malay Bhattacharyya und Sanghamitra Bandyopadhyay. „Solving maximum fuzzy clique problem with neural networks and its applications“. In: *Memetic Computing* 1.4 (2009), S. 281–290. DOI: 10.1007/s12293-009-0019-6. URL: http://dx.doi.org/10.1007/s12293-009-0019-6.

[Beu16] Cédric Beust. *JCommander*. Version 1.58. 2016. URL: www.jcommander.org/.

[BGS95] Mihir Bellare, Oded Goldreich und Madhu Sudan. „Free Bits, PCPs and Non-Approximability - Towards Tight Results“. In: *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October 1995*. 1995, S. 422–431. DOI: 10.1109/SFCS.1995.492573.

[BH92] Ravi B. Boppana und Magnús M. Halldórsson. „Approximating Maximum Independent Sets by Excluding Subgraphs“. In: *BIT* 32.2 (1992), S. 180–196.

[BHB07] Mauro Brunato, Holger H. Hoos und Roberto Battiti. „On Effectively Finding Maximal Quasi-cliques in Graphs“. In: *Learning and Intelligent Optimization, Second International Conference, LION 2007, Trento, Italy, December 8-12, 2007. Selected Papers*. 2007, S. 41–55. DOI: 10.1007/978-3-540-92695-5_4. URL: http://dx.doi.org/10.1007/978-3-540-92695-5_4.

[Bom+99] Immanuel M Bomze, Marco Budinich, Panos M Pardalos und Marcello Pelillo. „The maximum clique problem“. In: *Handbook of combinatorial optimization*. Springer, 1999, S. 1–74.

[Bon97] Murty U. S. R. Bondy J. A. *Graph Theory with Applications*. 1997. ISBN: 0-444-19451-7.

[Bré79] Daniel Brélaz. „New Methods to Color Vertices of a Graph“. In: *Commun. ACM* 22.4 (1979), S. 251–256. DOI: 10.1145/359094.359101.

[BT90] Luitpold Babel und Gottfried Tinhofer. „A branch and bound algorithm for the maximum clique problem“. In: *ZOR - Meth. & Mod. of OR* 34.3 (1990), S. 207–217. DOI: 10.1007/BF01415983.

[Cai] Shaowei Cai. *DIMACS graph format*. URL: http://lcs.ios.ac.cn/~caisw/Resource/about_DIMACS_graph_format.txt.

[CCJ90] Brent N. Clark, Charles J. Colbourn und David S. Johnson. „Unit disk graphs“. In: *Discrete Mathematics* 86.1-3 (1990), S. 165–177. DOI: 10.1016/0012-365X(90)90358-O.

[Cer+08] Márcia R. Cerioli, Luerbio Faria, Talita O. Ferreira, Carlos A. J. Martinhon, Fábio Protti und Bruce A. Reed. „Partition into cliques for cubic graphs: Planar case, complexity and approximation“. In: *Discrete Applied Mathematics* 156.12 (2008), S. 2270–2278. DOI: 10.1016/j.dam.2007.10.015.

[Cer+11] Márcia R. Cerioli, Luerbio Faria, Talita O. Ferreira und Fábio Protti. „A note on maximum independent sets and minimum clique partitions in unit disk graphs and penny graphs: complexity and approximation“. In: *RAIRO - Theor. Inf. and Applic.* 45.3 (2011), S. 331–346. DOI: 10.1051/ita/2011106.

[Cha+14] Parinya Chalermsook, Sandy Heydrich, Eugenia Holm und Andreas Karrenbauer. „Nearly Tight Approximability Results for Minimum Biclique Cover and Partition“. In: *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*. 2014, S. 235–246. DOI: 10.1007/978-3-662-44777-2_20. URL: http://dx.doi.org/10.1007/978-3-662-44777-2_20.

[CK97] Pierluigi Crescenzi und Viggo Kann. „Approximation on the Web: A Compendium of NP Optimization Problems“. In: *Randomization and Approximation Techniques in Computer Science, International Workshop, RANDOM'97, Bolognna, Italy, July 11-12. 1997, Proceedings*. 1997, S. 111–118. DOI: 10.1007/3-540-63248-4_10.

[CL93] Joseph C. Culberson und Feng Luo. „Exploring the k-coloable landscape with Iterated Greedy“. In: *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*. 1993, S. 245–284.

[CP90] Randy Carraghan und Panos M. Pardalos. „An Exact Algorithm for the Maximum Clique Problem“. In: *Oper. Res. Lett.* 9.6 (Nov. 1990), S. 375–382. ISSN: 0167-6377. DOI: 10.1016/0167-6377(90)90057-C.

[DCC11] Paolo Dell'Olmo, Raffaele Cerulli und Francesco Carrabs. „The maximum labeled clique problem“. In: *Proceedings of the 10th Cologne-Twente Workshop on graphs and combinatorial optimization. Extended Abstracts, Villa Mondragone, Frascati, Italy, June 14-16, 2011*. 2011, S. 146–149.

URL: http://ctw2011.dia.uniroma3.it/ctw_proceedings.pdf#page=158.

[Des+07] Anders Dessmark, Jesper Jansson, Andrzej Lingas, Eva-Marta Lundell und Mia Persson. „On the Approximability of Maximum and Minimum Edge Clique Partition Problems“. In: *Int. J. Found. Comput. Sci.* 18.2 (2007), S. 217–226. DOI: 10.1142/S0129054107004656. URL: http://dx.doi.org/10.1142/S0129054107004656.

[dle13] dlegland. *javaGeom*. Version 0.11.1. 2013. URL: https://github.com/dlegland/matGeom.

[DP11] Adrian Dumitrescu und János Pach. „Minimum Clique Partition in Unit Disk Graphs“. In: *Graphs and Combinatorics* 27.3 (2011), S. 399–411. DOI: 10.1007/s00373-011-1026-1.

[Epp03] David Eppstein. „Small Maximal Independent Sets and Faster Exact Graph Coloring“. In: *J. Graph Algorithms Appl.* 7.2 (2003), S. 131–140. URL: http://www.cs.brown.edu/publications/jgaa/accepted/2003/Eppstein2003.7.2.pdf.

[Fei04] Uriel Feige. „Approximating Maximum Clique by Removing Subgraphs“. In: *SIAM J. Discrete Math.* 18.2 (2004), S. 219–225. DOI: 10.1137/S089548010240415X.

[FGO15] Mathew C. Francis, Daniel Gonçalves und Pascal Ochem. „The Maximum Clique Problem in Multiple Interval Graphs“. In: *Algorithmica* 71.4 (2015), S. 812–836. DOI: 10.1007/s00453-013-9828-6.

[Gav72] Fanica Gavril. „Algorithms for Minimum Coloring, Maximum Clique, Minimum Covering by Cliques, and Maximum Independent Set of a Chordal Graph“. In: *SIAM J. Comput.* 1.2 (1972), S. 180–187. DOI: 10.1137/0201013. URL: http://dx.doi.org/10.1137/0201013.

[GLC04] Andrea Grosso, Marco Locatelli und Federico Della Croce. „Combining Swaps and Node Weights in an Adaptive Greedy Approach for the Maximum Clique Problem“. In: *J. Heuristics* 10.2 (2004), S. 135–152. DOI: 10.1023/B:HEUR.0000026264.51747.7f.

[GLL82] U. I. Gupta, D. T. Lee und Joseph Y.-T. Leung. „Efficient algorithms for interval graphs and circular-arc graphs“. In: *Networks* 12.4 (1982), S. 459–467. DOI: 10.1002/net.3230120410. URL: http://dx.doi.org/10.1002/net.3230120410.

[GM86] Fred Glover und Claude McMillan. „The general employee scheduling problem. An integration of MS and AI“. In: *Computers & OR* 13.5 (1986), S. 563–573. DOI: 10.1016/0305-0548(86)90050-X.

[GPR93] Fred Glover, Mark Parker und Jennifer Ryan. „Coloring by tabu branch and bound“. In: *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*. 1993, S. 285–308.

[Gra+06] Jens Gramm, Jiong Guo, Falk Hüffner und Rolf Niedermeier. „Data Reduction, Exact, and Heuristic Algorithms for Clique Cover“. In: *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments, ALENEX 2006, Miami, Florida, USA, January 21, 2006*. 2006, S. 86–94. DOI: 10.1137/1.9781611972863.9. URL: http://dx.doi.org/10.1137/1.9781611972863.9.

[Hal93] Magnús M. Halldórsson. „A Still Better Performance Guarantee for Approximate Graph Coloring“. In: *Inf. Process. Lett.* 45.1 (1993), S. 19–23. DOI: 10.1016/0020-0190(93)90246-6.

[Hås96] Johan Håstad. „Clique is Hard to Approximate Within $n^{\frac{1}{2}-\varepsilon}$“. In: *37th Annual Symposium on Foundations of Computer Science, FOCS '96, Burlington, Vermont, USA, 14-16 October, 1996*. 1996, S. 627–636. DOI: 10.1109/SFCS.1996.548522.

[HK73] John E. Hopcroft und Richard M. Karp. „An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs“. In: *SIAM J. Comput.* 2.4 (1973), S. 225–231. DOI: 10.1137/0202019.

[HW87] Alain Hertz und Dominique de Werra. „Using tabu search techniques for graph coloring“. In: *Computing* 39.4 (1987), S. 345–351. DOI: 10.1007/BF02239976.

[Kar72] Richard M. Karp. „Reducibility Among Combinatorial Problems“. In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.* 1972, S. 85–103. URL: http://www.cs.berkeley.edu/~luca/cs172/karp.pdf.

[Kel] E. Kellerman. „Determination of keyword conflict“. In: *IBM Technical Disclosure Bulletin* 16 (2), S. 544–546.

[KL06] Dieter Kratsch und Mathieu Liedloff. „An Exact Algorithm for the Minimum Dominating Clique Problem“. In: *Parameterized and Exact Computation, Second International Workshop, IWPEC 2006, Zürich, Switzerland, September 13-15, 2006, Proceedings*. 2006, S. 130–141. DOI: 10.1007/11847250_12. URL: http://dx.doi.org/10.1007/11847250_12.

[KS06] J. Mark Keil und Lorna Stewart. „Approximating the minimum clique cover and other hard problems in subtree filament graphs“. In: *Discrete Applied Mathematics* 154.14 (2006), S. 1983–1995. DOI: 10.1016/j.dam.2006.03.003. URL: http://dx.doi.org/10.1016/j.dam.2006.03.003.

[KSW78] Lawrence T. Kou, Larry J. Stockmeyer und C. K. Wong. „Covering Edges by Cliques with Regard to Keyword Conflicts and Intersection Graphs“. In: *Commun. ACM* 21.2 (1978), S. 135–139. DOI: 10.1145/359340.359346. URL: http://doi.acm.org/10.1145/359340.359346.

[Kuh55] Harold W Kuhn. „The Hungarian method for the assignment problem“. In: *Naval research logistics quarterly* 2.1-2 (1955), S. 83–97.

[Law76] Eugene L. Lawler. „A Note on the Complexity of the Chromatic Number Problem“. In: *Inf. Process. Lett.* 5.3 (1976), S. 66–67. DOI: 10.1016/0020-0190(76)90065-X.

[LH10] Zhipeng Lü und Jin-Kao Hao. „A memetic algorithm for graph coloring“. In: *European Journal of Operational Research* 203.1 (2010), S. 241–250. DOI: 10.1016/j.ejor.2009.07.016.

[LQ10] Chu Min Li und Zhe Quan. „An Efficient Branch-and-Bound Algorithm Based on MaxSAT for the Maximum Clique Problem“. In: *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. 2010. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1611.

[LY94] Carsten Lund und Mihalis Yannakakis. „On the Hardness of Approximating Minimization Problems“. In: *J. ACM* 41.5 (1994), S. 960–981. DOI: 10.1145/185675.306789.

[MM16] Mehdi Rezapoor Mirsaleh und Mohammad Reza Meybodi. „A new memetic algorithm based on cellular learning automata for solving the vertex coloring problem“. In: *Memetic Computing* 8.3 (2016), S. 211–222. DOI: 10.1007/s12293-016-0183-4. URL: http://dx.doi.org/10.1007/s12293-016-0183-4.

[MMT11] Enrico Malaguti, Michele Monaci und Paolo Toth. „An exact approach for the Vertex Coloring Problem“. In: *Discrete Optimization* 8.2 (2011), S. 174–190. DOI: 10.1016/j.disopt.2010.07.005.

[MV15] Arne Meier und Heribert Vollmer. *Komplexität von Algorithmen*. Bd. 4. Mathematik für Anwendungen. Lehmanns Media, 2015. ISBN: 978-3-86541-761-9. URL: www.lehmanns.de/shop/mathematik-informatik/32129287-9783865417619-komplexitaet-von-algorithmen.

[MZ06] Isabel Mèndez-Dìaz und Paula Zabala. „A Branch-and-Cut algorithm for graph coloring“. In: *Discrete Applied Mathematics* 154.5 (2006), S. 826–847. DOI: 10.1016/j.dam.2005.05.022.

[Ora] Oracle. *Java Standard Library*. URL: https://docs.oracle.com/javase/8/docs/api/.

[OS06] Hussein A. Omari und Khair E. Sabri. „New Graph Coloring Algorithms“. In: *Journal of Mathematics and Statistics* 2.4 (2006), S. 439–441. DOI: 10.3844/jmssp.2006.439.441.

[Pat+13] Jeffrey Pattillo, Alexander Veremyev, Sergiy Butenko und Vladimir Boginski. „On the maximum quasi-clique problem“. In: *Discrete Applied Mathematics* 161.1-2 (2013), S. 244–257. DOI: 10.1016/j.dam.2012.07.019.

[PH06] Wayne J. Pullan und Holger H. Hoos. „Dynamic Local Search for the Maximum Clique Problem“. In: *J. Artif. Intell. Res. (JAIR)* 25 (2006), S. 159–185. DOI: 10.1613/jair.1815.

[PMB11] Wayne Pullan, Franco Mascia und Mauro Brunato. „Cooperating local search for the maximum clique problem“. In: *J. Heuristics* 17.2 (2011), S. 181–199. DOI: 10.1007/s10732-010-9131-5.

[Pro12] Patrick Prosser. „Exact Algorithms for Maximum Clique: A Computational Study“. In: *Algorithms* 5.4 (2012), S. 545–587. DOI: 10.3390/a5040545.

[PS12] Imran A. Pirwani und Mohammad R. Salavatipour. „A Weakly Robust PTAS for Minimum Clique Partition in Unit Disk Graphs“. In: *Algorithmica* 62.3-4 (2012), S. 1050–1072. DOI: 10.1007/s00453-011-9503-8.

[Pul06] Wayne J. Pullan. „Phased local search for the maximum clique problem“. In: *J. Comb. Optim.* 12.3 (2006), S. 303–323. DOI: 10.1007/s10878-006-9635-y.

[PZ12] Abraham P. Punnen und Ruonan Zhang. „Analysis of an approximate greedy algorithm for the maximum edge clique partitioning problem“. In: *Discrete Optimization* 9.3 (2012), S. 205–208. DOI: 10.1016/j.disopt.2012.05.002. URL: http://dx.doi.org/10.1016/j.disopt.2012.05.002.

[RGG15] Ryan A. Rossi, David F. Gleich und Assefaw Hadish Gebremedhin. „Parallel Maximum Clique Algorithms with Applications to Network Analysis“. In: *SIAM J. Scientific Computing* 37.5 (2015). DOI: 10.1137/14100018X. URL: http://dx.doi.org/10.1137/14100018X.

[Rid] H.N. de Ridder. *Information System on Graph Classes and their Inclusions*. URL: http://www.graphclasses.org/ (besucht am 18. 11. 2016).

[ron15] ronmamo. *reflections*. Version 0.9.10. 2015. URL: https://github.com/ronmamo/reflections.

[Ros15] James Roseborough. *luaj-jse*. Version 3.0.1. 2015. URL: https://www.luaj.org.

[Seg12] Pablo San Segundo. „A new DSATUR-based algorithm for exact vertex coloring“. In: *Computers & OR* 39.7 (2012), S. 1724–1733. DOI: 10.1016/j.cor.2011.10.008.

[Sim90] Hans Ulrich Simon. „On Approximate Solutions for Combinatorial Optimization Problems“. In: *SIAM J. Discrete Math.* 3.2 (1990), S. 294–310. DOI: 10.1137/0403025. URL: http://dx.doi.org/10.1137/0403025.

[Ste14] Kevin L. Stern. *stern-library*. 2014. URL: www.github.com/KevinStern/software-and-algorithms.

[TS03] Etsuji Tomita und Tomokazu Seki. „An Efficient Branch-and-Bound Algorithm for Finding a Maximum Clique“. In: *Discrete Mathematics and Theoretical Computer Science, 4th International Conference, DMTCS 2003, Dijon, France, July 7-12, 2003. Proceedings*. 2003, S. 278–289. DOI: 10.1007/3-540-45066-1_22.

[Vig16] Sebastiano Vigna. *fastutil*. Version 7.0.13. 2016. URL: http://fastutil.di.unimi.it.

[WH12] Qinghua Wu und Jin-Kao Hao. „Coloring large graphs based on independent set extraction“. In: *Computers & OR* 39.2 (2012), S. 283–290. DOI: 10.1016/j.cor.2011.04.002.

[WH15] Qinghua Wu und Jin-Kao Hao. „A review on algorithms for maximum clique problems“. In: *European Journal of Operational Research* 242.3 (2015), S. 693–709. DOI: 10.1016/j.ejor.2014.09.064.

[WS84] Douglas B. West und David B. Shmoys. „Recognizing graphs with fixed interval number is NP-complete“. In: *Discrete Applied Mathematics* 8.3 (1984), S. 295–305. DOI: 10.1016/0166-218X(84)90127-6. URL: http://dx.doi.org/10.1016/0166-218X(84)90127-6.

[Xue01] Guoliang Xue. „Handbook of Combinatorial Optimization DingZhu Du and Panos M. Pardalos (co-editors), Kluwer Academic Publishers, 1998, Vols. 1-3, ISBN: 0-7923-5285-8“. In: *J. Global Optimization* 19.4 (2001), S. 1–3. DOI: 10.1023/A:1011210425701.

[Zho+14] Zhaoyang Zhou, Chu Min Li, Chong Huang und Ruchu Xu. „An exact algorithm with learning for the graph coloring problem“. In: *Computers & OR* 51 (2014), S. 282–301. DOI: 10.1016/j.cor.2014.05.017.