Phillips-Universität-Marburg
Fachbereich Mathematik und Informatik
AG Algorithmik

Bachelorarbeit zum Thema

Algorithm Engineering für Max Cut mit Cardinality Constraints

zur Erlangung des akademischen Grades eines

Bachelor of Science

im Studiengang

Data Science

Autor: Moritz Groß
Betreuer: Prof. Dr. Christian Komusiewicz
M. Sc. Frank Sommer

Datum der Abgabe: 13. Oktober 2022

Danksagung

Hiermit möchte ich mich bei Herrn Prof. Dr. Komusiewicz und Herrn Sommer für die Betreuung meiner Arbeit, hilfreiche Diskussionen und viel Geduld bei zahlreichen Fragen bedanken.

Zusammenfassung

Beim Graphenproblem Max $(k, n-k)$-Cut wird eine Knotenmenge S der Größe k gesucht, sodass möglichst viele Kanten genau einen Endpunkt in S haben. Die Anzahl dieser Kanten wird *Cutgröße* genannt. Zudem ist Max $(k, n-k)$-Cut ein $\mathcal{NP}$-schweres Graphproblem [1].

Es wird ein exakter Branch-and-bound Algorithmus für Max $(k, n-k)$-Cut entworfen und implementiert. An diesem werden Verbesserungen umgesetzt, welche größtenteils darauf basieren, zu erkennen, ob eine aktuelle Teillösung noch zu einer optimalen Lösung erweiterbar ist. Zum werden wir algorithmische Techniken wie das Vorverarbeiten der Probleminstanz durch einen sogenannten Kern und das Erstellen einer oberen Schranke durch lokale Suche anwenden. Auch werden wir den Funktionswert dynamisch zwischenspeichern, anstatt ihn für jeden Lösungskandidaten komplett neu zu berechnen. Zuletzt modellieren wir Max $(k, n-k)$-Cut auch als Integer Linear Program (ILP) und lösen dieses durch den CPLEX-Solver von IBM.

Wir nutzen 19 geläufige Benchmarkgraphen, um für die Parameterwerte $k \in \{1, \ldots, 30\}$ die Laufzeiten zu evaluieren. Für jede umgesetzte Verbesserung wird ausgewertet, wie sehr die Laufzeit im Vergleich zur vorherigen Version reduziert wurde. Die Experimente zeigen, dass die verschiedenen Verbesserungen sehr unterschiedlich effektiv sind. Durch alle Verbesserungen insgesamt konnte für 4 der 19 getesteten Graphen das innerhalb einer Stunde maximale gelöste k von 1 auf mindestens 12 erhöht werden. Das ILP kann innerhalb einer Stunde nahezu gleich viele Instanzen lösen wie der Suchbaum mit allen Verbesserungen.

Inhaltsverzeichnis

1 Einleitung **1**

2 Definitionen und Grundlagen **3**
2.1 Allgemeines . . . 3
2.2 Graphtheorie . . . 3
2.3 Klassische Komplexitätstheorie . . . 4
2.4 Parametrisierte Komplexität . . . 7

3 Experimentelles Setup **8**
3.1 Erstellen von Lösungen mit Brute-Force-Solver . . . 9

4 Basisalgorithmus **10**
4.1 Suchbaum . . . 10
4.2 Pseudocode . . . 12
4.3 Keine Extension für Blätter erstellen . . . 14
4.4 Auswertung . . . 14

5 Lösen des Optimierungsproblems **16**

6 Annotierte Version von Max $(k, n-k)$-Cut **17**

7 Verbesserungen des Suchbaums **20**
7.1 Obere Schranke . . . 20
7.2 Startlösung durch lokale Suche . . . 23
7.3 Verbesserungen durch Beitrag . . . 27
7.4 Zufriedenstellende Knoten . . . 28
7.5 Unnötige Knoten . . . 30
7.6 Problemkern durch Maximalgrad . . . 31
7.7 Zusammenfassung der Verbesserungen . . . 34

8 ILP-Formulierung **35**
8.1 Einführung . . . 35
8.2 ILP-Formulierung für Max $(k, n-k)$-Cut . . . 35

9 Vergleich von Branch and Bound mit ILP **37**
9.1 Laufzeit . . . 37

10 Fazit und Ausblick **39**

1 Einleitung

Eine zentrale Problemstellung bei der Analyse von Netzwerken ist es, herauszufinden, welche Knotenmengen besonders stark oder schwach mit dem restlichen Graphen verbunden sind. Für einen einzelnen Knoten ist eine einfache Interpretation hiervon bereits gegeben, da der Grad eines Knotens die Anzahl seiner Kanten zu restlichen Graphknoten zählt.

Für beliebige Knotenmengen S kann auf verschiedene Weisen gemessen werden, wie stark diese an den Rest des Graphen angebunden ist. So kann gezählt werden, wieviele Graphknoten mit einem Knoten aus S benachbart sind. Diese Definition heißt *group degree centrality* und wird in Everett et al. [2] eingeführt. Fordert man eine vollständige Abdeckung, so handelt es sich um das Problem Independent Set, beispielsweise verwendet in Corneil et al. [3]. Die andere Interpretation ist es, die Anzahl der ausgehenden Kanten einer Knotenmenge S zu zählen. In Saurabh et al. [4] wird dieses Modell eingeführt mit den Zusatzbedingungen, dass $|S| = k$ gelten muss und S zusammenhängt. Diese Arbeit verfolgt auch die Variante, die Anzahl der ausgehenden Kanten zu zählen, jedoch fordern wir nicht, dass S zusammenhängend sein muss. Leizhen Cai [1] untersucht die Klasse der Graphprobleme, in welchen eine Knotenmenge fester Größe gesucht wird. Dabei ordnet er Max $(k, n-k)$-Cut zusammen mit anderen Graphproblemen hinsichtlich ihrer Komplexität ein.

Nachfolgend sei die Anzahl der Kanten zwischen zwei Mengen A, B von Knoten notiert mit $m(A, B)$. Zudem definieren wir $m(A, A) = m(A)$. Viele der algorithmischen Techniken, welche im Code dieser Arbeit umgesetzt wurden, stammen aus Koana et al. [5]. Dort wird die Zielfunktion $(1-\alpha) \cdot m(S) + \alpha \cdot m(S, V(G) \setminus S)$ verwendet. Im Fall von $\alpha = 1$ handelt es sich hier um die Größe des Cuts von $(S, V(G) \setminus S)$, Max $(k, n-k)$-Cut ist also ein Sondernfall der in Koana et al. [5] behandelten Probleme. Im Fall von $\alpha = 0$ handelt es sich dort um das Problem Clique [6] und dessen Relaxierung Densest Subgraph. Hierdurch lässt sich der Zusammenhang der Probleme erahnen: Für Max $(k, n-k)$-Cut wird die Anzahl der Kanten nach außen, also $V \setminus S$ gezählt, für Densest Subgraph die Anzahl innerhalb. Würde man beide Werte addieren, so erhält man die Summe der Knotengrade innerhalb S, da die obengenannten Definitionen abdeckende Fallunterscheidungen sind.

Nachfolgend beschreiben wir die Problemstellung formal.

Max $(k, n-k)$-Cut
Eingabe: Ein Graph $G = (V, E)$, $k \in \mathbb{N}$ und $t \in \mathbb{Q}$.
Frage: Gibt es eine Menge S aus exakt k Knoten mit $\text{val}(S) := m(S, V \setminus S) \geq t$?

In Abbildung 1 ist ein Graph abgebildet, dessen Knoten farblich kodiert wurden, um S und $V \setminus S$ anzuzeigen.

Die Optimierungsvariante von Max $(k, n-k)$-Cut sucht eine Menge von k Knoten mit maximalem Cut, das heißt $S \subseteq V, |S| = k$. Es kann mehrere Lösungen für das Problem geben, da ein Graph auch mehrere Teilmengen mit gleicher Cutgröße haben kann. Bei

einem Graphen ohne Kanten wäre die Cutgröße für jede Teilmenge gleich 0. Zu beachten ist insbesondere, dass die Lösung von im Allgemeinen nicht aus den k Knoten mit höchsten Graden besteht. Da Kanten innerhalb von S nicht zu $m(S, V \setminus S)$ beitragen, müssen diese von $\sum_{s \in S} \deg(s)$ abgezogen werden.

Das Problem Max $(k, n-k)$-Cut ist $\mathcal{NP}$-vollständig [1]. Für solche Probleme ist es eine geläufige Strategie, sie approximativ und durch Heuristiken zu lösen. In dieser Arbeit werden wir mit Suchbäumen zusätzlich eine Technik kennenlernen, *exakte* Lösungen möglichst schnell zu konstruieren. Die finale Version des Suchbaumalgorithmus wird beispielsweise eine Instanz mit mehr als 50 000 Knoten und $k = 30$ lösen können. Würde man alle Lösungskandidaten einzeln auswerten, so wären dies $\binom{50\,000}{30} \approx 3 \cdot 10^{108}$ Stück. Dieses Beispiel verdeutlicht die Notwendigkeit, dass es auch bei leistungsstarker Hardware unerlässlich ist, algorithmische Techniken zur Beschleunigung einzusetzen. In den letzten Jahren gab es etwa Fortschritte darin, auf einem Kern basierende Algorithmen für Max-Cut zu entwickeln, siehe Ferizovic et al. [7] oder auch Saurabh et al. [8].

Ein anderer Ansatz wird in Abschnitt 8 dargestellt. Dort wird Max $(k, n-k)$-Cut als ILP modelliert. Diese ILP-Formulierung wird dann mit den erzielten Laufzeiten des Suchbaums verglichen.

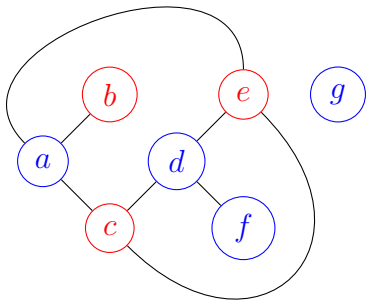


Abbildung 1: Dieser Graph $\mathcal{G}$ besteht aus 7 Knoten und 7 Kanten. Es ist ein Cut zwischen den Knoten $S = \{a, d, f, g\}$ und $V \setminus S = \{b, c, e\}$ eingezeichnet worden. Die Größe des Cuts ist $|\{ab, ac, ae, dc, de\}| = 5$. Für die Probleminstanz $(\mathcal{G}, 4, 3)$ ist das Ergebnis `true`, da wir hier einen Cut der Größe $5 \geq 3$ gefunden haben. Für $t = 10$ wissen wir unmittelbar, dass das Ergebnis `false` sein muss, da der Graph nur 7 Kanten besitzt.

2 Definitionen und Grundlagen

In diesem Abschnitt werden die verwendeten mathematischen Notationen vorgestellt. Dieser Abschnitt ist so gegliedert, dass allgemeinere mathematische Inhalte früher eingeführt werden.

2.1 Allgemeines

Die natürlichen Zahlen bezeichnen wir mit $\mathbb{N}$ und notieren $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$. Für $n \in \mathbb{N}$ definieren wir $[n] := \{1, 2, ..., n\}$.

Mit $\mathbb{R}$ wird die Menge der reellen Zahlen bezeichnet. Die Anzahl der Elemente einer Menge X ist $|X|$. Für zwei Mengen A, B gibt $A \subseteq B$ an, dass A eine nicht notwendigerweise echte Teilmenge von B ist. Wenn A und B disjunkt sind, also $A \cap B = \emptyset$ gilt, schreiben wir die Vereinigung zweier Mengen A, B als $A \dot\cup B$. Durch $A \setminus B$ notieren wir die Differenz von A und B, also $\{a \in A : a \notin B\}$.

Für eine Menge M und $k \in \mathbb{N}_0$ definieren wir die Menge $\binom{M}{k} := \{T \subseteq M : |T| = k\}$ aller Teilmengen von M von Größe k. Diese Notation ist angelehnt an den Binomialkoeffizienten, da $|\binom{M}{k}| = \binom{|M|}{k}$ gilt.

Betrachte folgendes Problem: Für eine endliche Menge M und eine Funktion $f : M \to \mathbb{R}$ ist ein $m \in M$ gesucht, sodass $m = \arg\max f$. Ein Algorithmus ist *brute-force*, wenn er das Problem dadurch löst, dass $f(m)$ für alle Elemente in M berechnet wird und der beste Wert zwischengespeichert wird. Für Max $(k, n-k)$-Cut gilt $M = \binom{V}{k}$ und f ist die Größe des jeweiligen Cuts.

2.2 Graphtheorie

Wir definieren einen *Graph* als $G := (V, E)$, wobei V eine endliche Menge an *Knoten* ist und $E \subseteq \binom{V}{2}$ die Menge der *Kanten* ist. Die Anzahl der Knoten wird mit $n := n$ und die Anzahl der Kanten wird mit $m := |E|$ abgekürzt. Mit V_G bezeichnet wir die Knoten des Graphens, mit E_G die Kanten.

Die *Nachbarschaft* eines Knoten v ist $N_G(v) := \{w \in V | \{w, v\} \in E\}$. Für $v \in V_G$ sei $\deg_G(v) := |N_G(v)|$ der *Grad* von v und $\Delta_G := \max_{v \in V_G} \deg(v)$ ist der maximale Grad des Graphen G. Zwei Knoten v, w sind *benachbart*, wenn $v \in N(w)$ oder äquivalent $w \in N(v)$.

Für $v, w \in V_G$ ist ein *Pfad* eine endliche Folge $(p_1, p_2, \ldots, p_k)$, $p_1 = v$, $p_k = w$ bestehend aus k verschiedenen Knoten, sodass $\{v_i, v_{i+1}\} \in E_G$ für $i \in [k-1]$. Die *Länge* eines Pfades ist die Zahl seiner Kanten, also $k - 1$.

Die *Distanz* $\text{dist}(v, w) \in \mathbb{N}_0$ zwischen zwei Knoten v, w ist die Länge eines kürzesten Pfades von v nach w. Bei $v = w$ gilt $\text{dist}(v, w) = 0$. Außerdem ist $\text{dist}(v, w) = 1$, wenn $\{v, w\} \in E$.

Eine Klasse von Graphen ist eine Menge von Graphen, die alle eine vorgegebene Eigenschaft erfüllen. Eine der am meisten genutzten Klassen sind *Bäume*. Ein Graph $G = (V, E)$ ist genau dann ein Baum, wenn er verbunden ist und $n - 1$ Kanten besitzt. Äquivalent dazu ist, dass zwei Knoten immer durch genau einen Pfad verbunden sind.

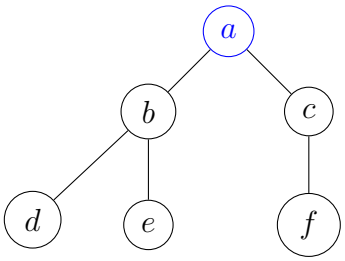


Abbildung 2: Ein Baum aus 6 Knoten, bei dem der Knoten a als Wurzel gekennzeichnet wurde. Die Kinder von a sind $\{b, c\}$, c hat nur f als Kind. Die Blätter des Baumes sind $\{d, e, f\}$. Die Vorfahren von e sind $\{b, a\}$, sein Elternknoten ist b.

Man kann bei Bäumen zudem einen Knoten als *Wurzel* markieren. Hierdurch sind Kanten im Baum orientierbar: Es gilt $|\operatorname{dist}(v, \text{wurzel}) - (w, \text{wurzel})| = 1 \quad \forall \{v, w\} \in E$. Bei jeder Kante ist also einer der Endpunkte näher an der Wurzel. Die Knoten entlang dieses Pfades werden, mit Ausnahme von x selbst, als *Vorfahren* von x bezeichnet. *Nachfahren* von x sind Knoten, von denen x ein Vorfahr ist. Als *Elternknoten* bezeichnen wir den eindeutigen benachbarten Vorfahren eines Knotens, die benachbarten Nachfahren eines Knotens werden auch dessen *Kinder* genannt. Ein *Blatt* ist ein Knoten ohne Nachfahren. Blätter haben immer Grad 1, da sie nur den Elternknoten als Nachbarn besitzen.

Die Menge aller Kanten, welche zwischen X und Y liegen, also in beiden Mengen jeweils einen Endpunkt haben, sei $E_G(X, Y) := \{\{x, y\} \in E_G | x \in X, y \in Y\}$. Wir nutzen die Notation $E_G(X) := E_G(X, X)$ als Abkürzung, um die Kanten innerhalb einer Menge von Knoten zu spezifizieren. Angelehnt daran, dass in der Literatur oft $m := |E_G|$ verwendet wird, definieren wir durch $m_G(X, Y) := |E_G(X, Y)|$ und analog $m_G(X) := |E_G(X)|$ die jeweilige Anzahl von Kanten.

Allgemein kann der tiefgestellte Name des Graphen (bisher G) ausgelassen werden, wenn der Kontext eindeutig ist, etwa bei V statt V_G.

Für eine detaillierte Einführung in die Graphentheorie, siehe Bondy et al. [9].

2.3 Klassische Komplexitätstheorie

Die hier vorgestellten Grundlagen sind entnommen aus Arora et al. [10] und können dort detailliert nachgelesen werden.

In der Regel bezeichnen wir mit Σ eine endliche Menge zur Kodierung von Ein- oder Ausgabe und nennen Σ ein *Alphabet*. Eine *Zeichenkette* ist die Aneinanderreihung von Elementen aus Σ. Mit Σ^* bezeichnen wir die Menge aller Zeichenketten, die sich durch Aneinanderreihung beliebig vieler Elemente aus Σ ergeben. Häufig gilt $\Sigma := \{0, 1\}$ für die binäre Kodierung. Eine Teilmenge der möglichen Wörter $L \subseteq \Sigma^*$ wird *Sprache* genannt.

Ein *Entscheidungsproblem* ist eine Sprache L über einem Alphabet Σ, also $L \subseteq \Sigma^*$. Eine konkrete Eingabe x mit $x \in \Sigma^*$ wird *Instanz* genannt und das Entscheidungsproblem

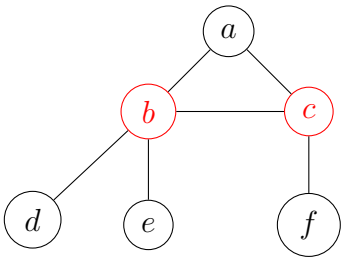


Abbildung 3: Eine Instanz von Vertex Cover mit der Lösung $\{b, c\}$.

lautet „gilt x in L?“.

Ein Problem heißt *entscheidbar*, falls die Fragestellung „gilt x in L?“ für jedes x in endlicher Zeit lösbar ist.

Neben Entscheidungsproblemen gibt es auch (diskrete) *Optimierungsprobleme*, in denen für eine gegebene endliche Menge X und $f : X \to \mathbb{R}$ das Element $x \arg\max f$ gesucht wird. Entscheidungsprobleme der Form $\exists x \in X : f(x) \geq t$ für $t \in \mathbb{R}$ lassen sich also zum Optimierungsproblem der obigen Definition umformen, wenn X und f genutzt werden, und Optimierungsprobleme lassen sich mithilfe des Grenzparameters t zur Entscheidung umformen.

Ein gutes Beispiel für ein Entscheidungsproblem ist:

Vertex Cover (VC)

Eingabe: Ein Graph $G = (V, E)$ und $k \in \mathbb{N}$

Frage: Eine Knotenmenge $S \subseteq V$ mit $|S| \leq k$, sodass jede Kante aus E mindestens einen ihrer beiden Endknoten in S hat.

Hier ist M die Menge aller gültigen Teilmengen S nach Definition 2.3, f ist die Größe von S, und k ist die Entscheidungsgrenze. Wollen wir VC als Optimierungsproblem betrachten, so suchen wir die kleinstmögliche gültige Teilmenge, welche in Abbildung 3 dargestellt ist. Obwohl in 4 auch eine gültige Teilmenge eingezeichnet ist, ist ihre Größe nicht minimal und löst somit nicht das Optimierungsproblem.

Sei I eine Instanz eines beliebigen Berechnungsproblems, dann ist $|I|$ die Anzahl der Zeichen in I. Es handelt sich also insbesondere nicht um die Kardinalität einer Menge, da I ein String ist. Die Laufzeit von Algorithmen wird meist in Abhängigkeit von I untersucht. Im Fall von $\Sigma = \{0, 1\}$ ist dies also die Bitzahl.

In diesem Abschnitt geben wir einen kurzen Überblick über die wichtigsten Konzepte der Komplexitätstheorie. Eine detaillierte Einführung kann im Standardwerk *Computational Complexity: A Modern Approach* [10] gefunden werden.

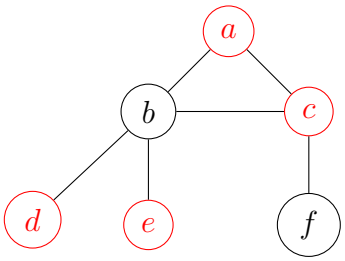


Abbildung 4: Die gleiche Instanz von VERTEX COVER, die Lösungsgröße ist hier jedoch $4 = |\{a, c, d, e\}|$

.

Um das Wachstum der Laufzeit von Algorithmen besser vergleichen zu können, hilft es, feste Summanden und Faktoren zu ignorieren und sich nur auf den asymptotischen Verlauf zu beschränken. Hierzu wird für Funktionen f, g die *Landau-Notation* genutzt, um das Wachstum von g durch f zu beschränken:

$$g \in O(f) \iff \exists c > 0 \quad \exists n_0 > 0 \quad \forall n \geq n_0 : g(n) \leq c \cdot f(n)$$

Die Notation $g \in O(f(n))$ kann also so verstanden werden, dass g asymptotisch nicht schneller als f wächst, also durch f abgeschätzt werden kann. Dies gilt, da ab einem Index n_0 die Funktion g nie ein festes Vielfaches $c \cdot f(n)$ überschreitet.

Gilt zugleich $f \in O(f)$ und $g \in O(g)$, so schreiben wir $f \in \Theta(g)$ und $g \in \Theta(f)$ um auszudrücken, dass f und g asymptotisch gleich schnell wachsen, da beide Funktionen jeweils durch die andere begrenzt sind. Durch Θ wird somit eine Äquivalenzrelation definiert.

Eine *Komplexitätsklasse* ist eine Menge von Sprachen. Eine Sprache L ist in der Komplexitätsklasse $\mathcal{P}$, wenn jede Instanz I von L in polynomieller Zeit in $|I|$ lösbar ist.

Eine Sprache L ist in der Komplexitätsklasse $\mathcal{NP}$, wenn es ein Polynom p und einen Verifizierer V gibt, so dass für jedes $x \in \Sigma^*$ ein Zertifikat $c \in \Sigma^{p(|x|)}$ existiert, so dass der Verifizierer V die Eingabe (x, c) in polynomieller Zeit in $|x|$ akzeptiert, und zwar genau dann wenn $x \in L$.

Durch *Reduktionen* lassen sich Probleme genauer kategorisieren. Ein Entscheidungsproblem $A \subseteq \Sigma^*$ ist in polynomieller Zeit reduzierbar auf ein Entscheidungsproblem $B \subseteq \Sigma^*$, wenn eine Funktion $g : \Sigma^* \to \Sigma^*$ existiert, welche in polynomialer Zeit in $|I|$ berechenbar ist, so dass $I \in A \iff g(I) \in B, \forall I \in \Sigma^*$. Der Algorithmus, der die Funktion g berechnet, wird *Polynomialzeit-Reduktion* genannt.

Eine Sprache B ist $\mathcal{NP}$-schwer, wenn jede Sprache $A \in \mathcal{NP}$ in polynomieller Zeit auf B reduzierbar ist. Eine Sprache B ist $\mathcal{NP}$-vollständig, wenn sie in der Komplexitätsklasse $\mathcal{NP}$ liegt und $\mathcal{NP}$-schwer ist.

Man beachte, dass $\mathcal{P} \subseteq \mathcal{NP}$, da für Probleme in $\mathcal{P}$ der Verifizierer nicht notwendig ist. Zudem ist $\mathcal{P} \subset \mathcal{NP}$ eine weithin anerkannte Hypothese, siehe Arora et al. [10].

2.4 Parametrisierte Komplexität

Für eine detaillierte Einführung in das Gebiet der parametrisierten Komplexität sei für die folgenden Definitionen neben [11] auf die Grundlagenarbeit von Downey und Fellows [12] verwiesen.

Ein *parametrisiertes Problem* ist eine Sprache $L \subseteq \Sigma^* \times \mathbb{N}$. Ein Tupel $(x, k) \in \Sigma^* \times \mathbb{N}$ heißt Instanz I eines parametrisierten Problems, wobei x die Eingabe des Entscheidungsproblems darstellt und k der Parameter ist. Zwei Instanzen I und I' sind *äquivalent*, wenn die Aussage $I \in L \iff I' \in L$ gilt. Eine Menge C aus parametrisierten Problemen ist eine Komplexitätsklasse.

Ein parametrisiertes Problem A ist *fixed-parameter-tractable*, wenn es einen Algorithmus gibt, der $f(k) \cdot \text{poly}(|x|)$ Zeit für jede Instanz $(x, k) \in \Sigma^* \times \mathbb{N}$ benötigt und entscheidet, ob die Instanz in der Sprache liegt. Die entsprechende Komplexitätsklasse wird als $\mathcal{FPT}$ bezeichnet.

Sei $L \subseteq \Sigma^* \times \mathbb{N}$ ein parametrisiertes Problem. Die Reduktion zu einem *Problemkern* heißt, $(I, k) \in L$ durch eine Instanz (I', k') zu ersetzen (genannt *Kern*), sodass

- $k' \leq k$ und $|I'| \leq g(k)$ für eine Funktion g, welche nur von k abhängt
- $(I, k) \in \mathcal{L} \iff (I', k') \in \mathcal{L}$
- Die Reduktion von (I, k) auf (I', k') ist in polynomieller Zeit $T_K(|I|, k)$ berechenbar. Der Wert $g(k)$ wird die Größe des Problemkerns genannt.

3 Experimentelles Setup

Die Experimente sind auf einem Intel(R) Xeon(R) Silver 4116 CPU mit 2.1 GHz, 12 CPUs und 128 GB RAM durchgeführt worden. Im Fall des ILP wurde aus praktischen Gründen zudem ein privates Gerät genutzt, welches als Prozessor einen AMD Ryzen 7 3700U mit Radeon Vega Mobile Gfx, 2.3 GHz besitzt und über 16GB RAM verfügt. Die Leistungsfähigkeit wurde auf die des Xeon-Prozessors hochgerechnet, sodass die Ergebnisses vergleichbar bleiben. Ein beiden Fällen war der verfügbare RAM mit Abstand ausreichend, sodass dies keine Schwierigkeit darstellte.

Der vollständige Quellcode dieser Arbeit ist verfügbar auf GitHub unter folgendem Repository: https://github.com/Morinator/Bachelorarbeit-KCut. Zur Umsetzung der Algorithmen wurde Kotlin 1.6.21 genutzt. Als Graph-Bibliothek wurde JGraphT[1] [13] genutzt. Zum Lösen der ILP-Formulierungen verwenden wir den *CPLEX-Solver* in Version 22.1.0.0 von IBM, indem auf die Bibliothek durch die mitgelieferte Datei *cplex.jar* über das Java Native Interface zugegriffen wurde, siehe die offizielle Dokumentation[2].

[1] https://jgrapht.org/
[2] https://docs.oracle.com/javase/8/docs/technotes/guides/jni/

Tabelle 1: Die sieben rechten Spalten enthalten den größten Wert von k, für welchen die Instanz durch den in der Spalte angegebenen Algorithmus in maximal einer Stunde lösbar war. Für k wurden die Werte $\{1, \ldots, 30\}$ getestet.

Name von G	n	m	Basis	*UB*	*Heur*	*Cont*	*SatNeed*	*Kern*	*ILP*
soc-firm-hi-tech	33	91	13	12	13	**30**	**30**	**30**	**30**
ENZYMES_g295	123	139	6	5	5	**30**	**30**	**30**	**30**
ca-netscience	379	914	4	4	4	26	26	27	**30**
bio-diseasome	516	1 188	3	3	4	28	29	29	**30**
rt-twitter-copen	761	1 029	3	3	3	26	26	26	**30**
road-euroroad	1 174	1 417	3	3	10	**30**	**30**	**30**	**30**
bio-yeast	1 458	1 948	3	3	4	**30**	**30**	**30**	**30**
mammalia-voles	1 686	4 623	3	3	9	**30**	**30**	**30**	**30**
ca-CSphd	1 882	1 740	3	3	3	**30**	**30**	**30**	**30**
inf-openflights	2 939	15 677	2	2	2	22	22	23	**30**
econ-poli	3 915	4 119	2	3	19	**30**	**30**	**30**	**30**
inf-power	4 941	6 594	2	2	14	**30**	**30**	**30**	**30**
ca-Erdos992	5 094	7 515	2	2	2	29	29	**30**	**30**
bio-dmela	7 393	25 569	2	2	2	26	27	**30**	**30**
bio-CE-CX	15 229	245 952	2	2	2	10	10	**18**	5
rec-yelp-user	50 394	229 572	1	2	2	5	5	14	**28**
soc-brightkite	56 739	212 945	1	2	2	4	4	12	**30**
H2O	67 024	1 074 856	1	2	10	11	12	**30**	1
rec-amazon	91 813	125 704	1	2	**30**	**30**	**30**	**30**	**30**

Als Datensatz für die Experimente in Tabelle 1 wurden 19 Graphen von der Webseite https://networkrepository.com [14] genutzt. Hierzu wurden 5 Graphen mit Knotenzahl zwischen 100 und 1 000 gewählt, 9 Graphen mit Knotenzahl zwischen 1 000 und 10 000 und 5 Graphen mit Knotenzahl zwischen 10 000 und 100 000. Die einzelnen Graphen sind in Tabelle 1 aufgelistet. Die Algorithmen wurden für $k \in \{1, \ldots, 30\}$ getestet und abgebrochen, wenn sie nach einer Stunde noch nicht terminierten.

In den rechten 7 Spalten von 1 wird für verschiedene Algorithmen der größte Wert für k abgebildet, für den innerhalb einer Stunde die Lösung gefunden werden konnte. Diese Algorithmen werden in den folgenden Abschnitten eingeführt. Es wird mit dem Algorithmus *Basis* begonnen, welcher dann iterativ verbessert wird zu den Versionen *UB*, *Heur*, *Cont*, *SatNeed* und *Kern*.

3.1 Erstellen von Lösungen mit Brute-Force-Solver

Für zum Testen verwendeten Graphen und Werte für k wurde die optimale Lösung in einer Textdatei abgespeichert. Erzeugt wurden die Ergebnisse, indem mit der Open-Source Bibliothek combinatoricslib[3] über $\binom{V}{k}$ iteriert wurde und für jedes Element der Zielwert durch Algorithmus 1 bestimmt wird. Nebenbei wird die bisher beste Teilmenge in einer Variable gespeichert, welche, nachdem alle $S \in \binom{V}{k}$ evaluiert wurden, das globale Optimum enthält und zurückgegeben wird.

Die Laufzeit von Algorithmus 1 wird dadurch dominiert, dass es $\binom{n}{k} \in \Theta(n^k)$ Durchläufe von Zeile 3 bis 7 gibt. In jedem dieser Durchläufe wird Algorithmus 2 genutzt, was die Laufzeit des Körpers der Schleife dominiert. Insgesamt ergibt sich demnach eine Laufzeit von $O(\binom{n}{k} \cdot k \cdot \Delta)$.

Algorithmus 2 wird auch im später folgenden Algorithmus *Basis* 4 genutzt, um die Cutgröße von Teilmengen zu bestimmen.

Algorithmus 1 SolverBruteforce

Require: Graph G, Teilmengengröße k

```
1: bestSubset ← ∅
2: bestValue ← 0
3: for all S in (V_G choose k) do                ▷ (n choose k) Elemente
4:     x ← val(G, S)                             ▷ Siehe Algorithmus 2
5:     if x > bestValue then                     ▷ Laufzeit insgesamt konstant
6:         bestSubset ← S
7:         bestValue ← x
8: return (bestSubset, bestValue)
```

[3] https://github.com/dpaukov/combinatoricslib

Algorithmus 2 *val*

Require: Graph G, Teilmenge S

```
1: ctr ← 0                                   ▷ Zählt, wieviele Kanten bisher gefunden wurden.
2: for all v in S do
3:     for all w ∈ N(v) do
4:         if w ∉ S then
5:             ctr ← ctr + 1                 ▷ Erhöhe um 1
6: return ctr
```

4 Basisalgorithmus

Der Ablauf des Algorithmus *Basis* folgt einem sogenannten *Suchbaum*, wobei die Knoten des Suchbaumes Zustände des Programms darstellen und die Kanten Übergänge zwischen den Zuständen sind.

In Algorithmus 1 ist beschrieben, wie Instanzen dieses Problems durch Brute-Force-Suche berechnet werden können. Das Aufzählen von $\binom{V}{k}$ kann durch verschiedene Herangehensweisen umgesetzt werden. Eine Möglichkeit ist, die betrachtete Menge V_G in eine beliebige Reihenfolge zu bringen und in einer Liste L_V zu speichern. Damit ist es möglich, durch k verschiedene Integer-Variablen, welche jeweils für eine Position in L_V stehen, alle Kandidaten aufzuzählen, wie es in der für die Implementierung von Algorithmus 1 verwendeten Bibliothek combinatoricslib3[4] getan wird.

Eine andere Umsetzung der Brute-Force-Suche ist die Nutzung eines Suchbaums. Dieser macht es leichter viele Verbesserungen des Algorithmus umzusetzen. Aus diesem Grund verfolgen wir diesen Ansatz im Folgenden weiter.

4.1 Suchbaum

Die Notationen für die Beziehungen der Suchbaumknoten ist zum Teil aus Staus [15] entnommen.

Die Umsetzung des Durchlaufens aller Teilmengen der Größe k geschieht durch einen gewurzelten Baum. In diesem repräsentiert jeder Knoten eine Teilmenge von V_G. Um die Umsetzung dieses Suchbaums technisch beschreiben zu können, werden im Folgenden einige Definitionen eingeführt. Sei B die Menge aller Knoten des Suchbaumes. Für alle $b \in B$ sei definiert:

- T_b ist die Teilmenge von V_G, die von b repräsentiert wird.
- child(b) ist die Menge der Kindknoten von b im Suchbaum.
- depth(b) ist die Anzahl der Kanten im Pfad zwischen der Wurzel des Suchbaumes und b. Daher gilt depth(wurzel) $= 0$.

[4] https://github.com/dpaukov/combinatoricslib3

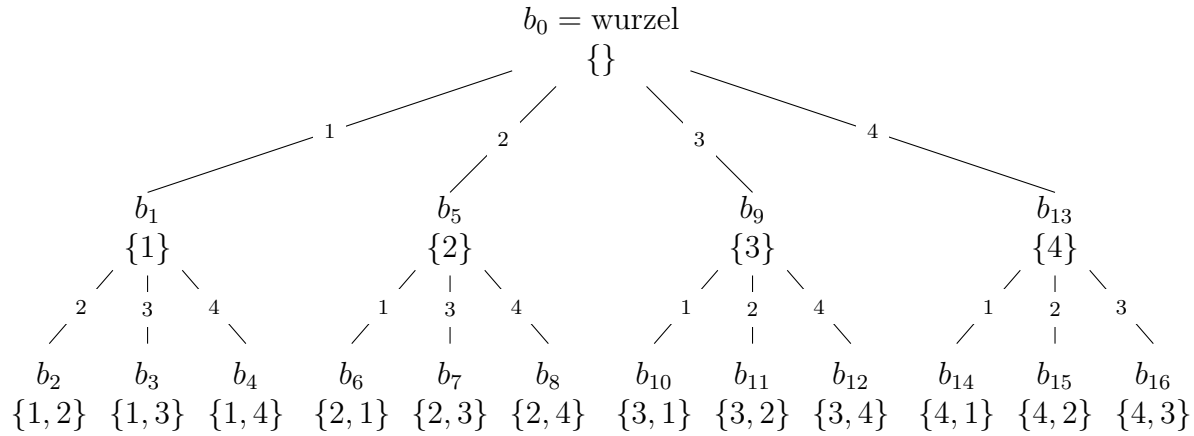


Abbildung 5: Suchbaum für $V = \{1, 2, 3, 4\}$, $k = 2$. Jeder Suchbaumknoten enthält in der ersten Zeile seinen Namen b_i, in der Zeile darunter steht die entsprechende Knotenmenge T_{g_i}. An den Kanten steht, welcher Knoten $\text{vert}(b) \in V_G$ durch die Kante zum darunter liegenden Suchbaumknoten $b \in B$ hinzugefügt wird.

- $\text{parent}(b)$ ist der Elternknoten von b. Für die Wurzel ist $\text{parent}(b)$ nicht definiert.
- $\text{vert}(b)$ ist definiert als $T_b \setminus T_{\text{parent}(b)}$, also der Graphknoten, welcher durch b neu hinzugefügt wurde. $\text{vert}(b)$ ist daher für alle Suchbaumknoten außer der Wurzel definiert.
- $\text{tree}(b)$ ist der Teilbaum, der dadurch entsteht, dass b als Wurzel gewählt wird und nur Knoten mit b als Vorfahr hinzugenommen werden.

Die Wurzel des Suchbaumes repräsentiert die leere Menge. Die restliche Struktur ist rekursiv definiert. Für jeden Graphknoten $v \in V_G \setminus T_b$ wird b ein Kind b' hinzugefügt, sodass $T_{b'} = T_b \cup \{v\}$. Dieser Prozess wird rekursiv auch für alle Blätter b angewendet, für die $\text{depth}(b) < k$ gilt. Es gilt $\text{depth}(b) = |T_b|$ für alle Knoten $b \in B$. Deshalb repräsentieren die Blätter des Suchbaumes Teilmengen von V mit Größe k.

Da der gesamte Suchbaum in einem Thread ausgeführt wird, müssen die Suchbaumknoten sequenziell abgearbeitet werden. Die Reihenfolge, in der das geschieht, hängt davon ab, in welcher Reihenfolge $\text{child}(b)$ für alle Suchbaumknoten b durchlaufen wird. Wir setzen im Folgenden der Einfachheit halber voraus, dass $V_G \subseteq \mathbb{Z}$ gilt, wodurch die Graphknoten einfach nach aufsteigendem Wert hinzugefügt werden, sodass eine einheitliche Reihenfolge definiert ist. Für $b, b' \in B$ mit $\text{parent}(b) = \text{parent}(b')$ gilt also, dass b vor b' besucht wird, wenn $\text{vert}(b) < \text{vert}(b')$ gilt.

In Abbildung 5 wurde ein Suchbaum mit 4 Knoten und $k = 2$ gezeichnet, in dem die Kinder somit jedes mal von links nach rechts durchlaufen werden.

Um $\binom{V_G}{k}$ aufzuzählen ist dieser Algorithmus jedoch nicht optimal, da es verschiedene Knoten b und b' gibt mit $T_b = T_{b'}$, was die Anzahl der durchlaufenen Suchbaumknoten unnötig erhöht. So ist in Abbildung 5 Suchbaumknoten b_6 redundant, da $\{1, 2\}$ schon $\text{vert}(b_2)$ ist.

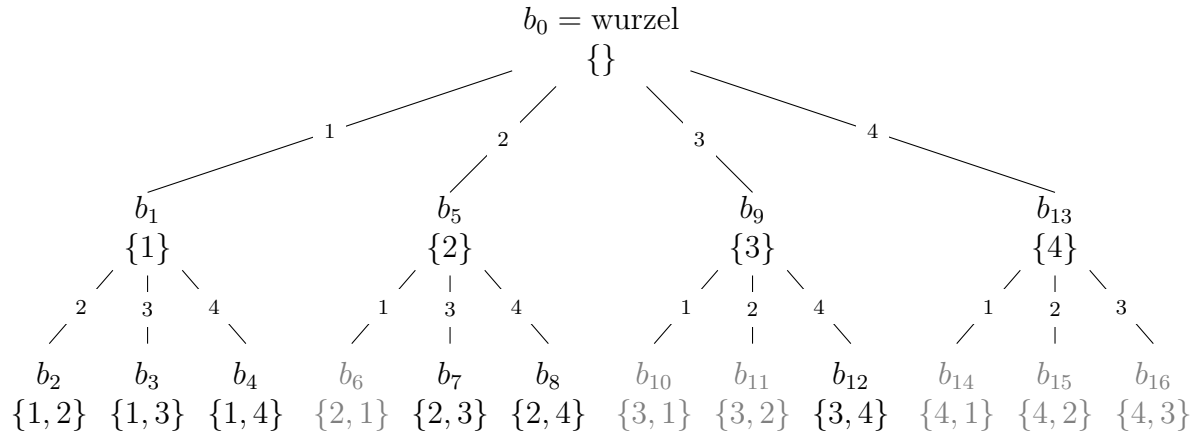


Abbildung 6: Der Suchbaum aus Abbildung 5, jedoch wurden die Blätter, welche während des Ablaufs erneut auftreten, grau eingefärbt. Im Pseudocode wird diese optimierte Version des Suchbaums erzeugt.

Dieses Problem kann dadurch gelöst werden, das wir nicht für jeden inneren Suchbaumknoten b für alle $V_G \setminus T_b$ ein Kind einfügen. Stattdessen wird mit child(b) gesondert verwalten, mit welchen Kindknoten wir noch nicht erkundete Teilmengen von V_G erzeugen können.

4.2 Pseudocode

Im Gegensatz zu Algorithmus 1 erhält Algorithmus 4 den Parameter t, da so wie in der Definition von Max $(k, n-k)$-Cut beschrieben das Entscheidungsproblem gelöst wird.

In der Variablen *ext* ist eine Liste gespeichert, welche als Elemente Liste aus Graphknoten enthält. Diese Listen beinhalten die noch unbesuchten Kinder der aktuell besuchten Suchbaumknoten. In Zeile 1 wird *ext* mit einer Liste initialisiert, die als einziges Element eine Liste der Knoten $V(G)$ enthält. Die erste Liste in *ext* enthält alle Möglichkeiten für den ersten Knoten von T. Wächst T im Folgenden um ein Element, so enthält *ext* zusätzlich eine zweite Liste an Index 2, welche die Möglichkeiten für den zweiten Knoten von T enthält. Allgemein enthält das i-te Element von *ext* eine Liste, welche die möglichen Knoten für das i-te Element von T speichert. Zudem gilt nach jedem Schleifendurchlauf die Invariante $|ext| = |T| + 1$, da immer nur die möglichen Optionen für den als nächstes anstehenden Knoten von T verwaltet werden.

In Zeile 3 wird T mit einer leeren Liste initialisiert, da $T_{\text{wurzel}} = \emptyset$. Es wird eine Liste und keine Menge als Datentyp genutzt, damit die Reihenfolge, in welcher Elemente hinzugefügt werden, ablesbar ist.

Um den Suchbaum zu traversieren wird in Zeile 4 eine Schleife genutzt. Diese ist erst dann beendet, wenn es keine neuen Kandidaten zu untersuchen gibt, also dementsprechend die Liste *ext* leer ist. Innerhalb der Schleife gibt es zwei Optionen:

Branch Es wird im Suchbaum zu einem Kind des aktuellen Knoten gewandert. Daher wird T um einen Eintrag vergrößert. Dies wird dann ausgewählt, wenn T noch nicht volle Größe hat, also $|T| < k$ gilt, und wenn es noch eine neue ungeprüfte Teilmenge $S \supseteq T$, $|S| = k$ gibt. Damit dies der Fall ist, muss die letzte Liste innerhalb von ext noch mindestens $k - |T|$ Knoten enthalten, was die zweite Bedingung ist.

Es wird ein neuer Knoten e aus $ext.last$ in Zeile 6 herausgenommen und T wird um e erweitert. An ext wird eine Kopie der letzten enthaltenen Liste angehangen, nachdem in Zeile 7 aus dieser e entfernt wurde. Dies ist sinnvoll, da e nun nicht mehr zur Erweiterung von T zur Verfügung steht.

In Zeile 8 und 9 werden T und ext jeweils vergrößert.

Backtrack Es wird überprüft, ob im Suchbaum zum Elternknoten gewandert werden soll, in welchem Fall der zuletzt zu T hinzugefügte Knoten wieder entfernt wird. Dieser Schritt ist die Alternative zu *Branch*, es kann also keine Bedingung aus Zeile 5 im Suchbaum-Algorithmus wahr sein.

Zuerst wird kontrolliert, ob die aktuelle Teilmenge die richtige Größe hat, also ob $|T| = k$ gilt, in welchem Fall der aktuelle Suchbaumknoten eine Lösung sein könnte. Wenn die in diesem Fall berechnete Cutgröße mindestens t ist, so kann die Lösung zurückgegeben werden und der Algorithmus terminiert.

Wenn nicht, dann wird im Suchbaum zum Elternknoten gewandert. Dies bedeutet schlicht, dass von T und ext je der letzte Eintrag entfernt wird. Der einzige Fallstrick hierbei ist, dass bei T überprüft werden muss, ob $|T| > 0$. Nach der ersten Iteration der

Algorithmus 3 Suchbaum mit Stack

Require: Teilmengengröße k, Graph G, Zielwert t

```
1:  ext ← Leere Liste von Knotenlisten
2:  Hänge Liste mit Elementen V_G an ext an
3:  T ← leere Liste von Knoten
4:  while ext ist nicht leer do
5:      if |T| < k und |T| + |ext.last| ≥ k then                ▷ Branch
6:          Wähle Element e aus ext.last
7:          Entferne e aus ext.last
8:          Füge e zu T hinzu
9:          Hänge Kopie von ext.last an ext an
10:     else                                                    ▷ Backtrack
11:         if |T| is equal to k then
12:             x ← val(G, T)                                   ▷ Siehe Algorithmus 2
13:             if x ≥ t then return (T, x)
14:         if |T| > 0 then
15:             Entferne letztes Element von T
16:         Entferne letztes Element von ext
17:     return null
```

Schleife tritt $|T| = 0$ erst wieder auf, wenn der aktuelle Zustand die Wurzel des Suchbaums ist, der Algorithmus sich also im letzten Suchbaumknoten befindet. Würde man hier versuchen, noch ein Element aus T zu entfernen, so würde die Java Virtual Machine eine *NoSuchElementException* werfen und der Algorithmus somit ungewollt mit einem Laufzeitfehler terminieren. Um dies zu vermeiden, ist die Bedingung in Zeile 14 notwendig.

Es wird `null` zurückgegeben, wenn keine passende Teilmenge gefunden werden konnte.

4.3 Keine Extension für Blätter erstellen

Theoretischer Hintergrund Sei $b \in B$ ein Knoten des Suchbaums, für den $|T_b| = k$ gilt. Da T_b schon volle Größe hat, müssen keine Kinder von b mehr erzeugt werden. Daher muss für b keine Liste mit den Elementen aus child(b) auf *ext* gepusht werden, da diese ungenutzt bleiben würde. Dies hat zur Folge, dass *ext* auch nicht wieder verkleinert werden muss, wenn der Suchbaum von b zu parent(b) übergeht.

Implementierung Diese Verbesserung kann durch eine einfache Abfrage sowohl im Fall von Branch als auch Backtrack umgesetzt werden.

Im ersten Fall wird nur dann eine neue Liste an *ext* angehangen, wenn die aktuelle Größe von T kleiner als $k-1$ ist, siehe Zeile 21. Gilt nämlich $|T| = k-1$, so ist der aktuelle Suchbaumknoten in der vorletzten Schicht. Das bedeutet, dass der Algorithmus gerade zu einem Blatt springt und für dieses *ext* updaten würde.

Beim Backtracken wird in Zeile 13 überprüft, ob $|T| = k$ gilt. Wenn ja, dann muss von *ext* kein Element entfernt werden, da wir bereits wissen, dass für den aktuellen Suchbaumknoten kein Element in *ext* hinzugefügt wurde.

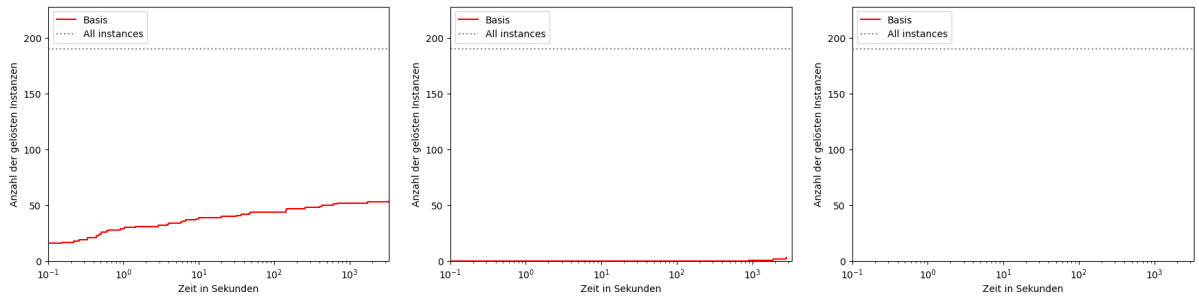
Verwendung Da diese Anpassung des Suchbaums sowohl theoretisch als auch in der Implementierung sehr einfach ist, wird sie im Folgenden immer genutzt. Diese Version nennen wir *Basis*, um sie von nachfolgenden, verbesserten Versionen unterscheiden zu können.

4.4 Auswertung

Abbildung 7 zeigt, wie viele Instanzen mit $k \in 1, \ldots, 10$, $k \in 11, \ldots, 20$ und $k \in 21, \ldots, 30$ der Basisalgorithmus lösen konnte, abhängig von der Laufzeit.

Die Zahl der Instanzen für jedes der drei Intervalle ist $10 \cdot 19 = 190$. Von diesen konnte der Basisalgorithmus nach 1 Sekunde 30, nach 5 Minuten 64 und nach 30 Minuten 69 lösen. Insgesamt ist das etwas mehr als ein Drittel. Von den 190 Instanzen mit $k \in 11, \ldots, 20$ konnten insgesamt gerade mal 3 Instanzen gelöst werden - selbst bei einer erlaubten Laufzeit von einer Stunde. Diese Ergebnisse sind erwartbar, da der Basisalgorithmus alle möglichen Teilmengen der Größe k testet und die Anzahl dieser Teilmengen exponentiell mit k wächst. Für $k \in 21, \ldots, 30$ konnte durch *Basis* keine einzige Instanz gelöst werden.

Die Ergebnisse aus Tabelle 1 passen zu diesen Beobachtungen. Die Spalte *Basis* in Tabelle 1 zeigt für jeden Graphen den größten Wert für k, für den der Basisalgorithmus



(a) $k \in \{1, \ldots, 10\}$ (b) $k \in \{11, \ldots, 20\}$ (c) $k \in \{21, \ldots, 30\}$

Abbildung 7: Anzahl an Istanzen, die je innerhalb der gegebenen Zeit vom Basisalgorithmus gelöst werden konnten. Abbildung a) wertet für k von 1 bis 10 aus, Abbildung b) von 11 bis 20 und Abbildung c) von 21 bis 30.

in der vorgegebenen Zeit eine Lösung finden konnte. Für $k \geq 5$ konnte der Algorithmus nur für die zwei kleinsten Graphen mit $n < 200$ Lösungen finden. Ausschließlich bei dem kleinsten Graphen konnte *Basis* Lösungen bis $k = 13$ finden.

5 Lösen des Optimierungsproblems

Bisher wird Max $(k, n-k)$-Cut nur als Entscheidungsproblem betrachtet, es wird also für einen festen Wert t überprüft, ob ein Cut von Wert mindestens t existiert. Eine andere gängige Problemstellung ist es, für einen gegebenen Graphen den maximalen Cut zu finden. Hierbei wird t also nicht mehr als Parameter benötigt. Das ist durch das Entscheidungsproblem lösbar, indem wir einen Wert t finden, sodass bei Eingabe (G, k, t) das Ergebnis `true` lautet und bei $(G, k, t+1)$ hingegen `false`. Hierdurch wissen wir, dass t maximal ist.

Der Pseudocode hierfür ist in Algorithmus 4 dargestellt. Dieser enthält in Zeilen 1 bis 5 Schritte, welche erst in späteren Abschnitten detailliert erläutert werden. Zeilen 1 und 2 wird der Kern aus Abschnitt 7.6 durchgeführt, Zeilen 3 und 4 setzen die lokale Suche aus 7.2 um. In Zeile 5 wird die obere Schranke bestimmt, welche in Abschnitt 7.1 vorgestellt wird. Eine obere Schranke ist ein Wert $x \in \mathbb{N}$, für den $x \geq \text{val}(S)$ für alle S aus $\binom{V}{k}$ gilt. Die Schleife in Zeile 6 wird so lange wiederholt, bis ein Suchbaum erfolglos ist oder wir die obere Schranke erreicht haben, in welchem Fall Algorithmus 4 terminiert.

Die Gültigkeit der oberen Schranke lässt sich wie folgt erklären: Sei t' die obere Schranke, so wissen wir, dass für $t'+1$ das Entscheidungsproblem negativ ausfällt, weshalb wir in Zeile 7 für $t'+1$ nach keiner Lösung suchen müssen. Daher werden nur neue Suchbäume erstellt, solange die obere Schranke noch nicht erreicht ist, siehe Bedingung in Zeile 6.

Algorithmus 4 Optimierungsalgorithmus

Require: Teilmengengröße k, Graph G, Zielwert t

```
1:  counter ← Map, die jeden Knoten 0 zuordnet      ▷ Nötig für Exklusionsregel im Kern
2:  Wende Problemkern auf G an                      ▷ Siehe Algorithmus 8
3:  S ← beste Menge von mehreren Durchläufen der Heuristik   ▷ Siehe Algorithmus 5
4:  val_S ← val(S)
5:  upperbound ← Top-Ext-Upper-Bound(∅, k)          ▷ Siehe Abschnitt 7.1
6:  while val_S < upperbound do
7:      result ← SuchbaumMaster(k, G, val_S + 1)
8:      if result is null then      ▷ Rückgabewert null signalisiert erfolglose Suche
9:          break
10:     S ← result
11:     val_S ← val(S)
12: return (S, val_S)
```

6 Annotierte Version von Max $(k, n-k)$-Cut

In diesem Abschnitt erweitern wird die Problemdefinition von Max $(k, n-k)$-Cut , basierend auf dem von Koana et al. [5] eingeführten Problem Annotiertes Max $(k,\ n-k)$-Cut . Für die nachfolgenden Definitionen sei also auf dieses Papier verwiesen.

Zu den bisher bestehenden Parametern von Max $(k, n-k)$-Cut speichern wir für jeden Knoten eine Zählerfunktion counter : $V \to \mathbb{N}_0$, die die gelöschten Nachbarn jedes Knotens vermerkt, die nicht in der Lösung enthalten sind. Die Idee von counter ist folgende: Angenommen $\{a, b\} \in E_G$, jedoch wird b nicht in die Lösung aufgenommen und aus dem Graphen entfernt. Dann muss der Knoten b dennoch zum Beitrag von a zum Cut hinzugezählt werden, falls a in S hinzugefügt wird, weshalb counter(a) um 1 inkrementiert wird. Werten wir nun aus, wie viele Kanten a zum Cut beisteuern kann, so muss zu $\deg(a)$ noch counter(a) addiert werden. Zur Erinnerung, $m(A, B)$ ist die Anzahl der Kanten mit einem Endpunkt in A und einem Endpunkt in B. Um die erweiterte Problemstellung definieren zu können, müssen wir zunächst counter formal einführen:

Für eine Menge $S \subseteq V$ sei

- $\text{counter}(S) := \sum_{v \in S} \text{counter}(v)$
- $\text{val}_G(S) := m(S, V \setminus S) + \text{counter}(S)$

Hiermit definieren wir nun eine neue, erweiterte Formulierung des Problems, mit einer erweiterten Version der Zielfunktion val_G.

Annotiertes Max $(k,\ n-k)$-Cut

Eingabe: Ein Graph G, $T \subseteq V$, counter : $V \to \mathbb{N}$, $k \in \mathbb{N}$ und $t \in \mathbb{Q}$.
Frage: Gibt es eine Knotenmenge S aus exakt k Knoten, sodass $T \subseteq S \subseteq V$ und $\text{val}_G(S) = m(S, V \setminus S) + \text{counter}(S) \geq t$?

Eine Knotenmenge S, die diese Anforderungen erfüllt, wird als *Lösung* bezeichnet. Setzen wir $T := \emptyset$ und $\text{counter}(v) := 0 \quad \forall v \in V_G$, so erhalten wir Max $(k, n-k)$-Cut als Spezialfall von Annotiertes Max $(k,\ n-k)$-Cut .

Nun definieren wir den Beitrag eines Knotens v, um bewerten zu können, wie stark sich die Teillösung T dadurch erhöht, dass v zu T hinzugefügt wird. Dadurch kann in verbesserten Versionen des Algorithmus wie in Abschnitt 7.3 vermieden werden, dass die Zielfunktion val_G für jeden Kandidaten komplett neu ausgewertet werden muss. Stattdessen wird während Umformungen der Teillösung T die Änderung des Funktionswertes über den Beitrag angepasst.

Für einen Knoten $v \in S$, sei $\deg^{+c}(v) := \deg(v) + \text{counter}(v)$. Es wird counter$(v)$ addiert, um Knoten zu berücksichtigen, welche aus G bereits entfernt wurden. Wir nennen $\deg^{+c}(v)$ auch *annotierten Grad* von v.

Definition 6.1 (Beitrag eines Knotens)**.** Für eine beliebige Menge von Knoten $T \subseteq V$ wird der *Beitrag* $\text{cont}(v, T)$ von v bezüglich T definiert als:

$$\text{cont}(v, T) := \deg^{+c}(v) - 2|N(v) \cap T|$$

Es gilt $\text{val}(T \cup \{v\}) = \text{val}(T) + \text{cont}(v, T)$ für alle $v \in V \setminus T$. Das lässt sich mit der Gleichung $\deg^{+c}(v) - 2|N(v) \cap T| = |N(v) \setminus T| + \text{counter}(v) - |N(v) \cap T|$ folgendermaßen herleiten:

- Alle Kanten zwischen v und $V \setminus T$ kommen hinzu. Ihre Anzahl ist $\deg(v)^{+c} - |N(v) \cap T|$.
- Wenn v hinzugefügt wird, zählen alle Kanten zwischen T und v nicht mehr zum Funktionswert dazu, daher muss $-|N(v) \cap T|$ wieder abgezogen werden.

Die Summe beider Ausdrücke ergibt den Term in Definition 6.1.

Man kann die Größe von $T \cap N(v)$ in durchschnittlicher Zeit von $O(|T|)$ berechnen, wenn man annimmt, dass die Nachbarn jedes Knotens in einem Hashset abgespeichert sind. Für jedes Element von T kann dann in konstanter Zeit abgefragt werden, ob es in $N(v)$ liegt. Die Laufzeit von cont wird durch diese Operation dominiert, da die Bestimmung von $\deg^{+c}(v)$ konstante Zeit benötigt. Das ist nur möglich, da der Grad eines Knotens in konstanter Zeit ausgelesen werden kann, da HashSets in Kotlin ihre Größe in einem Integer-Feld auslesen lassen. Bei der Wahl einer anderen Datenstruktur ist somit zu beachten, ob sich diese Zugriffszeit ändern würde.

Die Definition von cont ist so gewählt, dass sich der Wert $\text{val}(S)$ einer Knotenmenge S wie folgt berechnet.

Lemma 6.1. Sei G ein Graph und $S := \{v_1, \ldots, v_\ell\} \subseteq V$ eine Menge von Knoten. Dann gilt $val(S) = \sum_{i \in [\ell]} \text{cont}(v_i, \{v_1, \ldots, v_{i-1}\})$

Um mit dem Beitrag cont eines Knoten besser arbeiten zu können, untersuchen wir weitere Eigenschaften der Funktion:

Lemma 6.2. Für $v \in V_G$ und zwei Mengen $X \subseteq Y \subseteq V_G$ gilt $\text{cont}(X, v) \geq \text{cont}(Y, v)$.

Funktionen mit dieser Eigenschaft heißen auch *submodular*. Durch Einsetzen der Definition von cont erhält man $\deg^{+c}(v) - 2|N(v) \cap X| \geq \deg^{+c}(v) - 2|N(v) \cap X|$, was äquivalent ist zu $-2|N(v) \cap X| \geq -2|N(v) \cap X|$, da $\deg^{+c}(v)$ konstant ist. Da der Subtrahend $|N(v) \cup T|$ monoton steigt, wenn T größer wird, muss also $-2|N(v) \cap X| \geq -2|N(v) \cap X|$ gültig sein.

Grob gesprochen haben submodulare Funktionen also abnehmende Erträge insofern, als dass das Hinzufügen eines Knotens v weniger bringt, je größer die aktuelle Teillösung T ist.

Bisher gibt die Verwendung von counter noch keinen Vorteil, da wir keinen Fall definiert haben, wann $\text{counter}(v)$ abgeändert werden darf, wodurch in $\text{val}_G(S)$ der Summand $\text{counter}(S)$ konstant 0 ist. Bisher haben wir durch die Einführung von counter noch keine Verbesserung im Suchbaum erzielen können. Hierzu existieren folgende Reduktionsregeln:

Reduktionsregel 6.3 (Inklusionsregel)**.** Wenn es eine Lösung S mit $v \in S \setminus T$ gibt, dann füge v zu T hinzu. Wenn es einen Knoten $v \in T$ mit counter$(v) > 0$ gibt, dann verringere t um counter(v) und setze counter$(v) := 0$.

Diese Regel wird in Abschnitt 7.4 genutzt.

Reduktionsregel 6.4 (Exklusionsregel)**.** Wenn es eine Lösung S mit $v \notin S$ gibt, dann erhöhe counter(u) für jedes $u \in N(v)$ um eins und entferne v aus G.

Diese Regel wird in Abschnitt 7.5 genutzt

Diese beiden Regeln sind simpel und auch einfach zu implementieren. Die Herausforderung ist es, Fälle festzustellen, in welchen sie gültig anwendbar sind. Diese werden in Abschnitt 7 vorgestellt.

7 Verbesserungen des Suchbaums

Es werden verschiedene Verbesserungen für den Basisalgorithmus des vorherigen Abschnitts entwickelt, welche die Laufzeit des Algorithmus zu verbessern, meist, indem die Größe des Suchbaums reduziert wird. In den folgenden Unterabschnitten wird der Basisalgorithmus um je eine Verbesserungen erweitert. Auch wird die Laufzeit mit den vorangehenden Versionen verglichen.

Weiterhin ist T ist die in Abschnitt 6 eingeführte Teillösung, sodass $S \supseteq T$ gilt. In Algorithmus 3 ist T in der gleichnamigen Variablen gespeichert, obwohl diese eine Liste und keine Menge ist, um die Einfügereihenfolge der Graphknoten verfügbar zu machen.

Im gesamten Abschnitt sei $k' := k - |T|$ und $t' := t - \text{val}(T)$.

7.1 Obere Schranke

Theoretischer Hintergrund

Eine obere Schranke ist eine natürliche Zahl $x \in \mathbb{N}_0$, sodass die optimale Lösung von Max $(k, n-k)$-Cut einen Wert von maximal x hat.

Diese Idee vertiefen wir im Folgenden formaler: Obere Schranken dienen dazu, dass der Algorithmus bestimmte Teile des Baumes auslassen kann. Werden die Kinder eines Baumknotens b betrachtet, kann eine obere Schranke es ermöglichen, dass bestimmte oder sogar alle Kinder von b übersprungen werden können. Dadurch müssen weniger Teilmengen überprüft werden.

Eine *obere Schranke* für $b \in B$ ist ein Wert $x \in \mathbb{N}$ mit der Eigenschaft, dass $x \geq \text{val}(T_{b'})$ für alle Blätter b' aus tree(b). Haben wir nun bereits eine Teilmenge S mit $\text{val}_S \geq x$ gefunden, so muss tree(b) nicht mehr erkundet werden, da für alle Blätter b' aus tree(b) gilt, dass $\text{val}(b') \leq x \leq \text{val}_S$, da die Relation $\leq$ transitiv ist. Es gibt somit keine Beschränkung, wie groß x maximal sein darf, jedoch sind kleinere Werte nützlicher, da $x \leq \text{val}_S$ hier häufiger eintritt.

Wenn wir für tree(b) das Problem Annotiertes Max $(k,\ n-k)$-Cut exakt lösen, erhalten wir eine triviale obere Schranke, welche zudem minimal ist. Jedoch ist in diesem Fall die Laufzeit nicht verringert worden, weshalb wir eine effizientere Möglichkeit brauchen. Als obere Schranke definieren wir daher die *Top-Ext-Upper-Bound*:

In dieser schauen wir uns für die Knoten, welche noch zur aktuellen Teilmenge hinzugefügt werden können, an, wie hoch ihr Beitrag ist. Daran kann abgeschätzt werden, wie viele Kanten noch maximal zum Cut hinzukommen können.

Definition 7.1. Sei $b \in B$ und $k \in \mathbb{N}$. Dann ist die *Top-Ext-Upper-Bound* (teub(b,k)) von b definiert als:

$$\text{teub}(b,k) := \text{val}(T_b) + \max_{R \in \binom{\text{child}(b)}{k'}} \left(\sum_{v \in R} \text{cont}(T, v) \right).$$

Für die Top-Ext-Upper-Bound werden also die $k' = k - |T|$ Knoten aus child(b) gewählt, für welche der Beitrag zu T so groß wie möglich ist. Für diese wird der Beitrag aufsummiert

und zu $\text{val}(T_b)$ addiert.

Im Folgenden zeigen wir, dass die Top-Ext-Upper-Bound tatsächlich eine korrekte obere Schranke für den gesamten Teilbaum $\text{tree}(b)$ ist.

Theorem 7.1. Sei $b \in B$ ein Suchbaumknoten und $k \in \mathbb{N}$. Dann gilt für alle Blätter b' aus $\text{tree}(b)$, dass $\text{val}_G(T_{b'}) \leq \text{teub}(b, k)$ gilt.

Beweis. Sei $R \in \binom{\text{child}(b)}{k'}$ beliebig. Es ist zu zeigen, dass $\text{val}(R \dot\cup T) \leq \text{teub}(T)$ ist. Seien die Elemente von R in einer beliebigen Reihenfolge $r_1, ..., r_{k'}$ sortiert. Dann gilt nach Lemma 6.1, dass $\text{val}(R \dot\cup T) = \text{val}(T) + \sum_{i \in [k']} \text{cont}(T \dot\cup \{r_1, ..., r_{i-1}\}, r_i)$. Nach Definition des Beitrags ist $\text{cont}(T \dot\cup \{r_1, ..., r_{i-1}\}, r_i) \leq \text{cont}(T, r_i)$ für jedes $i \in [k']$. Da $\text{val}(T)$ konstant ist und wir die k' Knoten gerade so wählen, dass die Summe ihrer Beiträge maximal ist, ist die Aussage schon gezeigt. □

Falls $\text{teub}(b, k)$ kleiner als die bisher beste Lösung ist, müssen die Kinder $\text{child}(b)$ nicht mehr betrachtet werden, da die dadurch entstehenden Teilmengen keine bessere Lösung erzeugen können.

Allgemein ist diese Backtracking-Regel effektiver, je mehr Suchbaumblätter ausgewertet wurden, da dann val_S höher ist und somit $x \leq \text{val}_S$ häufiger eintritt. Somit ist die Heuristik aus Abschnitt 7.2 auch hilfreich, wenn sie keine Optimale Lösung liefert, da sie schon früh einen recht hohen Wert für val_S zurückgibt. Zudem ist $\text{tree}(b)$ größer, je näher b an der Wurzel des Suchbaums liegt, wodurch mehr Teilmengen übersprungen werden können.

Verbesserte obere Schranke Damit die Schranke $X := \max_{R \in \binom{\text{child}(b)}{k'}} \left(\sum_{v \in R} \text{cont}(v)\right)$ exakt ist, muss für jeden Knoten $x \in X$ gelten, dass $N(x) \cap T = \emptyset$ und $m(X) = 0$, da ansonsten in beiden Fällen Kanten nicht zum Cut dazuzählen würden. Die Idee liegt nahe, in der oberen Schranke $m(X)$ abzuziehen. Hierdurch wird die obere Schranke jedoch inkorrekt, da $\max_{R \in \binom{\text{child}(b)}{k'}} \left(\sum_{v \in R} \text{cont}(v)\right)$ durch mehrere Teilmengen von $\text{child}(b)$ erfüllt sein kann. Dies ist insbesondere anschaulich für den trivialen Fall, dass cont für alle Elemente aus $\text{child}(b)$ gleich ist. Nun ist es möglich, dass für eine solche Menge X gilt, dass $m(X) > 0$ ist, für eine andere jedoch nicht. Wir wären hierdurch also gezwungen, alle Maximierer von $\max_{R \in \binom{\text{child}(b)}{k'}} \left(\sum_{v \in R} \text{cont}(v)\right)$ durchzuprobieren, um die Korrektheit zu gewährleisten.

Dies zieht mehrere Probleme mit sich: Zum einen ist die Implementierung hiervon recht umständlich. Zudem ist die Laufzeit von $O(\binom{|\text{child}(b)|}{k'})$ sehr schlecht, etwa zu Beginn des Suchbaums, wenn $\text{child}(\text{wurzel}) = V_G$ und $k' = k$ gilt. In Versuchen auf verschiedenen Instanzen von Max $(k, n-k)$-Cut hat sich die Laufzeit auch experimentell stark verschlechtert, sodass es sich nicht nur um ein theoretisch problematisches Resultat handelt, sondern auch in der Praxis die Laufzeit schlechter wird.

Daher verwenden wir diese Version der oberen Schranke vorerst nicht. Eine mögliche Herangehensweisen, um dennoch einen Nutzen hierraus zu ziehen, ist es, die verbesserte obere Schranke nicht in jedem Knoten zu verwenden, sondern nur abhängig von gewissen Bedingungen, etwa dass k' kleiner als ein unbekannter Hyperparameter sein muss. Auch eine randomisierte Variante, dass in jedem Knoten die Regel nur mit einer Chance $P(b) \in [0, 1]$ angewandt wird, ist prinzipiell möglich.

Implementierung und Auswertung

Gesucht ist also ein Algorithmus, welcher aus child(b) die k' Elemente mit den größten Beiträgen findet.

Dies kann dadurch umgesetzt werden, dass eine Liste $D := d_1, ..., d_l$ der annotierten Knotengrade aus child(b) erstellt wird, welche absteigend nach ihrem Beitrag sortiert wird. Die Laufzeit hierzu ist $O(|\operatorname{child}(b)| \cdot \log(|\operatorname{child}(b)|))$. Das Ergebnis $x = \sum_{i=1}^{k'} d_i$ kann in Laufzeit von $O(k')$ berechnet werden, da nur eine Schleife mit k' Iterationen benötigt wird. Somit wird die Gesamtlaufzeit durch die Sortierung von D mit $O(|\operatorname{child}(b)| \cdot \log(|\operatorname{child}(b)|))$ dominiert.

Glücklicherweise steht child(b) schon in sortierter Reihenfolge durch Verbesserung 7.3 bereit, wodurch nur eine Laufzeit von $O(k')$ für das Aufsummieren der ersten k' Elemente dieser Liste entsteht.

Die einzige Stelle, an der child(b) nicht zur Verfügung steht ist in Zeile 5 von Algorithmus 4. An dieser Stelle ist es also einmalig notwendig, die Menge V_G in einer Laufzeit von $O(n \cdot \log n)$ zu sortieren. Dieser Aufwand entsteht jedoch nur einmalig und ist im Vergleich zu den Suchbäumen absolut zu vernachlässigen. Es wäre asymptotisch besser für die Laufzeit, die höchsten k Elemente aus einer Collection mithilfe eines Maxheaps zu extrahieren. Jedoch ist in allen Experimenten der Aufwand von Zeile 5 von Algorithmus 4 so gering, dass keine Unterschied zwischen den Varianten festgestellt werden konnte.

Unabhängig davon, wie die obere Schranke x berechnet wird, wird $x \leq \text{val}_S$ zu Beginn jedes Schleifendurchlaufs ausgewertet, siehe Zeile 8 von Algorithmus 7. Da hierbei an keinem Objekt eine Änderung durchgeführt wird, sondern nur eine Information abgefragt wird, existieren keine Seiteneffekt etwa zu Abschnitt 7.4, wodurch die Reihenfolge ihrer Aufrufe beliebig ist. In Zeile 10 wird dann ein Backtrack durchgeführt, wenn die obere Schranke oder eine andere der Regeln eintritt.

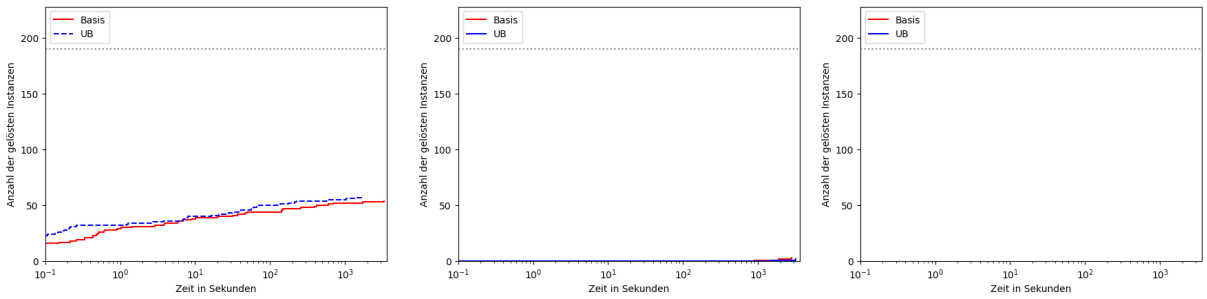
In Abschnitt 4 wird erläutert, wie die obere Schranke für $T := \emptyset$ genutzt wird, um das Optimierungsproblem frühzeitig abbrechen zu können.

Der Algorithmus, der die obigen Verbesserungen beinhaltet, wird im Folgenden *UB* genannt, als Abkürzung für Upper Bound. Wie in Abbildung 8 zu sehen ist, hat sich für $k \in \{1, \ldots, 10\}$ die Anzahl der Instanzen, die *UB* im Vergleich zum Basisalgorithmus 4 lösen kann, von 55 auf 59 erhöht. Dies ist eine Steigerung von 7%.

Für $k \in \{11, \ldots, 20\}$ konnte *Basis* insgesamt 3 Instanzen lösen, durch *UB* 2 Instanzen. Dies entspricht einer Verringerung von etwa 33%. Da 3 beziehungsweise 2 recht kleine Werte sind, ist die Aussagekraft für $k \in \{11, \ldots, 20\}$ recht gering, da insgesamt einfach zu wenige dieser Instanzen gelöst werden können.

Die Effektivität von *UB* scheint nicht nur von der Knotenzahl abzuhängen: Für den Graphen rec-amazon.mtx konnte *Basis* nur $k = 1$ lösen, der Solver *UB* hingegen konnte die Instanz für $k = 2$ in unter 4 Sekunden lösen.

Für $k \in \{21, \ldots, 30\}$ konnte keine einzige Instanz gelöst werden, weshalb für diese Gruppe auch keine Abbildung erstellt wurde.



(a) $k \in \{1, \ldots, 10\}$ (b) $k \in \{11, \ldots, 20\}$ (c) $k \in \{21, \ldots, 30\}$

Abbildung 8: Gelöste Instanzen der Algorithmen *Basis* und *UB* (Upper Bound)

7.2 Startlösung durch lokale Suche

Theoretischer Hintergrund

Unter einer Heuristik verstehen wir einen Algorithmus, der für eine Menge $\mathcal{M}$ und eine Zielfunktion $f : \mathcal{M} \to \mathbb{R}$ ein Element $M \in \mathcal{M}$ sucht, das nicht garantiert das Optimum ist, jedoch erfahrungsgemäß einen hohen Wert $f(M)$ erzeugt. In der kombinatorischen Optimierung ist $\mathcal{M}$ für gewöhnlich endlich. Diese Elemente werden im Folgenden *Kandidaten* genannt, zur Unterscheidung von optimalen Eingaben von f. Es ist hervorzuheben, dass das Ergebnis $M = \arg\max f$ möglich, aber nicht garantiert ist, und M auch nicht innerhalb eines garantierten Näherungsverhältnis zu $\max_{\mathcal{M}} f$ liegen muss, wie es bei Approximationsalgorithmen ist. Der Vorteil von Heuristiken gegenüber exakten Algorithmen liegt in ihrer Einfachheit und schnellen Laufzeit.

Die *lokale Suche* ist eine Familie von Heuristiken, welche alle darauf basieren, einen Kandidaten iterativ zu verbesseren. Ausgehend von einem Kandidaten $K_i \in \mathcal{M}$ wird ein besserer Kandidat $K_{i+1} \in \mathcal{N}(K_i) \subseteq \mathcal{M}$ gesucht wird, wobei die Menge $\mathcal{N}(K_i)$ (genannt *Nachbarschaft*) die Kandidaten enthält, welche durch kleine Umformungen aus K_i entstehen können. Wählen wir eine konkrete dieser Umformungen aus, so nennen wir diese *Schritt*.

Es werden so viele Schritte durchgeführt, bis für den aktuellen Kandidaten K_i keine Verbesserung mehr möglich ist, also wenn $f(K) \leq f(K_i)$ gilt für alle $K \in \mathcal{N}(K_i)$.

Eine solche Folge von Umformungsschritten $K_1 \to K_2 \to \cdots \to K_j$ nennen wir *Durchlauf*. Von einer Heuristik können beliebig viele Durchläufe ausgeführt werden, von welchen dann der beste Durchlauf weiterverwendet wird. Sei die Anzahl der Durchläufe x. Wird x zu hoch gewählt, so dauert die Ausführung der Heuristik zu lang und es wird somit keine Beschleunigung im gesamten Algorithmus mehr erzielt. Ist x zu klein, so ist das Ergebnis der Heuristik durchschnittlich nicht sehr gut, sodass sie keinen Vorteil in ihrer weiteren Verwendung bietet. Um x zu wählen können verschiedene Strategien gewählt werden: Die einfachste Möglichkeit ist es, verschiedene x auszuprobieren und das erfahrungsgemäß beste zu wählen. Da bei zunehmender Größe der Probleminstanz die Laufzeit des exakten Algorithmus für Max $(k, n-k)$-Cut schneller wächst als die der Heuristik, kann es auch

sinnvoll sein, bei größeren Instanzen x zu erhöhen, sodass die Laufzeit von Heuristik und Suchbaum verhältnismäßig bleibt. In diesem Fall kann x von n, m oder anderen Parametern der Probleminstanz abhängen, beispielsweise $\lfloor \log(m \cdot k + 30) \rfloor$. In den Experimenten von Abschnitt 1 wurden konstant 30 Durchläufe der lokalen Suche durchgeführt, da durch ein komplexeres Modell keine Verbesserung beobachtet werden konnte.

Sowohl in Max $(k, n-k)$-Cut als auch in vielen anderen Problemen ist $\mathcal{M}$ nicht explizit in der Problemstellung gegeben, sondern durch $\mathcal{M} := \binom{X}{k}$ definiert. Bei Graphproblemen ist $X := V_G$ häufig, man spricht dann auch von *Cardinality Constrained Optimization Problems* [1]. In diesem Fall ist $\mathcal{N}(K_i)$ oft durch Austauschen von Elementen in K_i definiert. Die Anzahl d der Elemente, die pro Schritt maximal ausgetauscht werden dürfen ist ein vom Nutzer wählrbarer Parameter.

Definition 7.2 (d-swap-Nachbarschaft)**.** Gegeben ist eine endliche Menge X, $k \in [|X|]$. Sei $\mathcal{M} := \binom{X}{k}$. Für $M \in \mathcal{M}$ ist die d-swap-Nachbarschaft definiert als

$$\mathcal{S}_d(M) := \{M' \in \mathcal{M} : |M' \setminus M| \leq d\}$$

Hier sind die Elemente von $\mathcal{M}$ also Teilmengen von X der Größe k. Als Umformungsschritt dürfen maximal d Knoten von M gegen Knoten außerhalb von M ausgetauscht werden. Schon für $d = k$ gilt $SN_d(M) = \mathcal{M}$, weil dadurch jedes Element von M ausgetauscht werden kann. Daher darf d nicht zu hoch gesetzt werden, weil es sich sonst um einen brute-force-Algorithmus handelt und seine Laufzeit somit höher ist als beim verbesserten Suchbaum.

Wahl des ersten Kandidaten Für K_1 kann eine gleichverteilt zufällige Wahl aus $\mathcal{M}$ getroffen werden, das heißt $\mathcal{M}$ wird als Laplace-Raum genutzt. Dieses zufällige Sampling ermöglicht es, in verschiedenen Durchläufen unterschiedliche Bereiche von M zu erkunden. Die Menge $m \in M$ kann erzeugt werden, indem k verschiedene Elemente aus X zufällig gezogen werden, deren Vereinigung dann m bildet.

Finden des Folgekandidaten Eine weitere Entscheidung ist es, ob für K_{i+1} der beste Kandidat aus $\mathcal{N}_d(K_i)$ genommen werden soll oder schon das erste Element aus der Menge $\{x \in \mathcal{N}_d(K_i) : f(x) > f(K_i)\}$, welches gefunden wird.

Die zweite Variante unterschlägt potenzielle bessere Kandidaten, da nicht das Optimum aus $\mathcal{N}_d(K_i)$ gewählt wird. Allerdings ist sie kompakter und einfacher zu implementieren, da während des Durchlaufens von $\mathcal{N}_d(K_i)$ nicht der bisher beste Kandidat des Durchlaufs zwischengespeichert werden muss. Da außerdem direkt beim ersten besseren Kandidaten abgebrochen wird, muss die Iteration von $\mathcal{N}_d(K_i)$ nicht vollständig durchgeführt werden, wodurch Zeit eingespart wird.

Aufgrund obiger Argumentation wird in dieser Arbeit die zweite Variante verwendet. Zudem wird $d = 1$ gewählt, da bereits für $d = 2$ die Menge $\mathcal{N}_d(K_i)$ in der Praxis zu groß ist.

Vor dem exakten Algorithmus eine Heuristik durchzuführen hat zwei Vorteile. Zum einen kann der exakte Algorithmus übersprungen werden, falls wir durch die obere Schranke wissen, dass durch die Heuristik eine exakte Lösung gefunden wurde.

Zudem gilt, dass auch wenn der exakte Algorithmus durchgeführt werden muss, dieser schneller ablaufen kann, wenn durch die Heuristik eine guter Startkandidat S gefunden wurde. Hierdurch können Backtracking-Regeln wie die obere Schranke in Abschnitt 7.1 häufiger genutzt werden. Ansonsten kann erst während des Durchlaufens des Suchbaums ein guter Wert für val_S gefunden werden, sodass zu Beginn die Backtracking-Regeln noch nicht eingesetzt werden können.

Implementierung

Algorithmus 5 Erkunde 1-Swap-Nachbarschaft

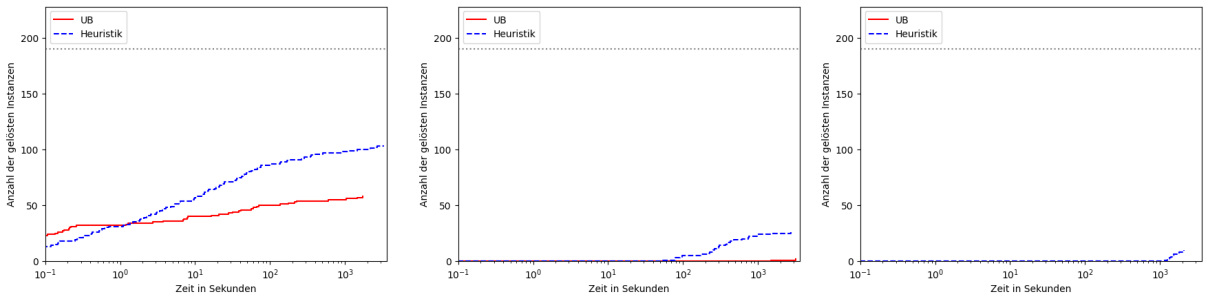
Require: Graph G, Zwischenlösung M

```
1: val ← val(M)
2: for all v_old in M do
3:     Entferne v_old aus M
4:     for all v_new in V_G \ M do
5:         Füge v_new zu M hinzu
6:         if val(M) > val then
7:             Update bisher besten Kandidaten zu M
               return
8:         Entferne v_new aus M
9:     Füge v_old zu M wieder hinzu
```

Pseudocode 5 zeigt, wie die *1-Swap-Nachbarschaft* effizient durchsucht werden kann. Hierbei wird eine Menge M, welche den aktuellen Kandidaten speichert, in alle möglichen Elemente der 1-Swap-Nachbarschaft umgeformt. Hierzu werden alle Möglichkeiten, ein Element aus M zu entfernen und dafür ein anderes zu M hinzuzufügen, durchprobiert.

Hierbei kommt insgesamt $|M|$ Mal der Fall vor, dass $v_{old} = v_{new}$ gilt, also dass das gleiche Element aus m entfernt und wieder hinzugefügt wird. Dies kann dadurch behoben werden, dass vor Zeile 5 überprüft wird, ob $v_{new} \neq v_{old}$ gilt, sodass der unnötige Kandidat $(M \setminus v_{new}) \cup v_{new} = M$ nicht überprüft werden muss. Jedoch entsteht durch den Overhead dieses zusätzlichen Checks ein größerer Aufwand in der Laufzeit als durch das Auslassen des unnötigen Elements eingespart wird. Der Code würde somit nur unnötig komplexer, weshalb auf diese Modifikation verzichtet wird.

Eine weitere potenzielle Verbesserung an Algorithmus 5 ist es, in Zeile 4 die Menge der Graphknoten, welche für v_{new} getestet wird, durch weitere notwendige Bedingungen zu verkleinern. Damit ein besserer Kandidat in Zeile 6 gefunden werden kann, muss $\deg_G(v_{new})$ mindestens so groß sein wie die Änderung von $\text{val}(M)$ durch das Löschen von v_{old}. Obwohl dieser zusätzliche Check korrekt ist, ist in Tests kein signifikanter Unterschied in der



(a) $k \in \{1, \ldots, 10\}$ (b) $k \in \{11, \ldots, 20\}$ (c) $k \in \{21, \ldots, 30\}$

Abbildung 9: Gelöste Instanzen der Algorithmen *UB* (Upper Bound) und *Heur* (Heuristik)

Laufzeit sichtbar gewesen. Daher haben wir die ursprüngliche, kompaktere Version von Algorithmus 5 beibehalten.

Auswertung

In Abbildung 9 wird wieder die Anzahl der gelösten Instanzen in Abhängigkeit von der Laufzeit gezeigt. Mit *Heur* bezeichnen wir den Algorithmus, welcher sowohl die bisherigen Verbesserungen aus Abschnitt 7.1 als auch die des aktuellen Abschnitts enthält. Es ist zu sehen, dass sowohl für $k \in \{1, \ldots, 10\}$ als auch für große Werte von k weitaus mehr Instanzen gelöst werden können.

Für $k \in \{1, \ldots, 10\}$ bis zu einer Laufzeit von etwa einer Sekunde kann *Heur* insgesamt weniger Instanzen lösen. Dies ist jedoch kein Problem, da Unterschiede bei so geringen Laufzeiten in der Praxis vollkommen unbedeutend sind. Etwa zu diesem Zeitpunkt übersteigt die Anzahl der Instanzen die Version *UpperBound* und dieser Abstand wird mit zunehmender Zeit auch immer größer. Insgesamt können durch *Heur* 105 Instanzen innerhalb einer Stunde gelöst werden, durch *UpperBound* nur 59 Instanzen, was einem Zuwachs von etwa 77% entspricht.

Für $k \in \{11, \ldots, 20\}$ ist der Unterschied sogar noch viel größer. Für *Heur* können 27 Instanzen innerhalb einer Stunde gelöst werden, für *UpperBound* hingegen nur 3, was also ein Wachstum um den Faktor 9 ergibt. Diese Effektivität kann zum Teil darauf zurückgeführt werden, dass der Suchbaum direkt übersprungen werden kann, wenn die Heuristik eine Lösung ergibt, welche gleich der oberen Schranke ist, siehe Zeile 6 in Algorithmus 4. Beispielsweise kann der Graph ca-CSphd.mtx für $k \in \{1, 2, 3\}$ je in unter einer Sekunde gelöst werden, weil die Heuristik optimal ist, für $k := 4$ jedoch entsteht ein Timeout, weil die Heuristik keine optimale Lösung finden konnte.

Für $k \in \{21, \ldots, 30\}$ konnten mit *UB* keine einzige Instanz gelöst werden, mit *Heur* hingegen insgesamt 10. Es ist somit ein sehr gutes Zeichen, dass es nun überhaupt Instanzen in dieser Gruppe ist, für welche es kein Timeout gibt.

In Tabelle 1 ist auch zu sehen, dass bei manchen Graphen keine Verbesserung in der Anzahl der gelösten Instanzen stattfand, etwa beim Graphen inf-openflights, bei anderen

Graphen hingegen ein riesiger Sprung stattgefunden hat, beispielsweise von 2 auf 30 bei rec-amazon.

7.3 Verbesserungen durch Beitrag

Sortierung der Kinder nach Beitrag

Die Reihenfolge, in der der Algorithmus den Suchbaum durchläuft, wurde bisher noch nicht betrachtet und ist beliebig. Wir können die Kinder child(b) eines jeden Suchbaumknotens b also in einer Reihenfolge durchlaufen, welche möglichst hilfreich ist, ohne die Korrektheit des Algorithmus zu beeinflussen. Dies ist zunächst ein zusätzlicher Rechenaufwand, wir können hierdurch jedoch auch andere Aspekte im Algorithmus leichter umsetzten.

Eine mögliche Sortierung ist es, die Kindknoten child(b) absteigend nach dem Beitrag $\text{cont}(T_b, c_i), c_i \in \text{child}(b)$ zu sortieren. Dies kann als Greedy-Heuristik aufgefasst werden: Wir hoffen, gute oder sogar optimale Kandidaten $S \in \binom{V_G}{k}$ früh zu entdecken, wodurch zudem val_S früh einen hohen Wert hat und die obere Schranke aus Abschnitt 7.1 effektiver ist. Zudem ist die Hoffnung, dass wenn die Knoten mit großem Beitrag fertig sind, die obere Schranke früher und häufiger angewendet werden kann.

Durch die Sortierung können zudem Knoten mit Betrag unter einer vorgegebenen Grenze leicht identifiziert werden. So lässt sich die Regel für unnötige Knoten aus Abschnitt 7.5 leicht umsetzen.

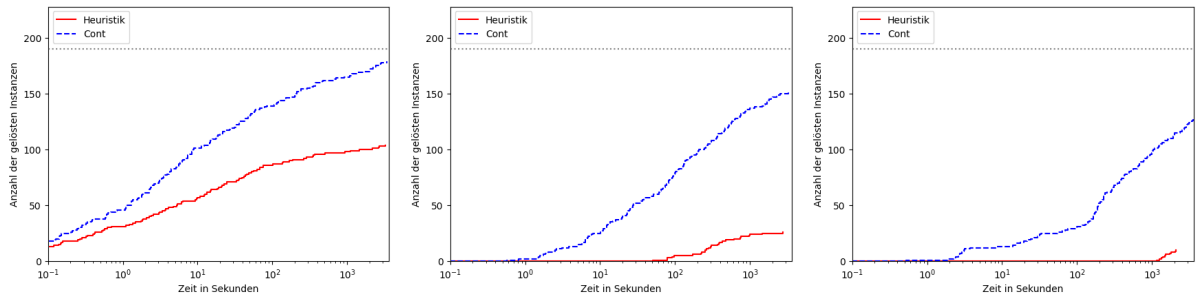
Zwischenspeichern des Funktionswert durch Beitrag

Der naive Ansatz zur Auswertung der Zielfunktion ist es, für jeden Knoten b des Suchbaums den Wert $\text{val}(T_b)$ neu zu berechnen, siehe Algorithmus 2. Dies ist sowohl bei den Blättern nötig, um zu überprüfen, ob eine Lösung gefunden wurde, also auch bei den inneren Suchbaumknoten, um die Regeln für zufriedenstellende oder unnötige Knoten nutzen zu können. Diese werden später eingeführt, siehe jeweils Abschnitt 7.4 und Abschnitt 7.5.

Die Laufzeit für jede Berechnung liegt bei $O(|T_b| \cdot \Delta)$, ist also mit $O(k \cdot \Delta)$ insbesondere für die Blätter des Suchbaums langsam, welche jedoch den Großteil des Suchbaums ausmachen. Effizienter ist es, für beliebige Suchbaumknoten $b \in B$ den Wert $\text{val}(G, T_b)$ mithilfe des Vaterknotens $p := \text{parent}(b)$ zu berechnen. Mit $\text{val}(T_b) = \text{val}(T_p) + \text{cont}(\text{vert}(b), T_p)$ nach Definition 6.1 von cont ist die Laufzeit nur $O(\text{depth}(p))$.

In Zeile 26 des Suchbaums in Algorithmus 7 wird val_T um den Beitrag des neuen Graphknotens erhöht. Hierbei wird zuerst val_T angepasst, und erst in der darauffolgenden Zeile wird T erweitert. Ansonsten würde $\text{cont}(T, e)$ für $e \in T$ ausgewertet werden, was fäschlicherweise 0 ergeben würde.

In Zeile 15 von Algorithmus 7 wird val_T um $\text{cont}(T, e)$ reduziert, nachdem e aus T entfernt wurde. Erneut ist diese Reihenfolge notwendig, da ansonsten $e \in T$ gelten würde, wodurch der Wert von *cont* falsch würde. Die Bedingung $|T| > 0$ ist notwendig, da der Algorithmus ganz zum Schluss seines Ablaufs noch einmal T verkleinern wollen würde,



(a) $k \in \{1, \ldots, 10\}$ (b) $k \in \{11, \ldots, 20\}$ (c) $k \in \{21, \ldots, 30\}$

Abbildung 10: Gelöste Instanzen der Algorithmen *Heur* und *Cont*

obwohl T bereits leer ist, der aktuelle Suchbaumknoten also die Wurzel des Suchbaums ist. Für alle anderen Zeitpunkte des Algorithmus ist $|T| > 0$ immer wahr.

Auswertung

Im Abbildung 10 ist die Anzahl der gelösten Instanzen der Version *Cont* dargestellt. Dies umfasst sowohl die Zwischenspeicherung aus dem aktuellen Abschnitt 7.3 als auch die Sortierung aus Abschnitt 7.3.

Für $k \in \{1, \ldots, 10\}$ beginnt *Cont* erst ab einer Laufzeit von etwa einer Sekunde signifikant besser zu werden. Dies ist kein Problem, da Unterschiede in einem so kleinen Zeitbereich in der Praxis unbedeutend sind. Insgesamt konnte *Cont* für diese Werte von k 179 Instanzen lösen, für *Heur* waren es 104. Somit konnte eine Steigerung von etwa 73% erzielt werden. Durch *Heur* konnten insgesamt 104 Instanzen innerhalb einer Stunde gelöst werden, durch *Cont* konnten 71 Instanzen innerhalb von 12 Sekunden gelöst werden, was einer Beschleunigung um das 300-Fache entspricht.

Für $k \in \{11, \ldots, 20\}$ ist die Verbesserung der Anzahl gelöster Instanzen sehr drastisch. Bis zu einer Laufzeit von etwa einer Sekunde ist die Leistung beider Solver noch vergleichbar, danach beginnt *Cont* viel mehr Instanzen zu lösen. Bei maximaler Laufzeit werden 152 Instanzen gelöst, vorher waren es 27. Das ist eine Steigerung um mehr als das Fünffache.

Auch für die Werte $k \in \{11, \ldots, 20\}$ ist eine starke Verbesserung sichtbar: Innerhalb einer Stunde konnten vorher 10 Instanzen gelöst werden, nun sind es 127, also mehr als das 12-Fache.

7.4 Zufriedenstellende Knoten

Theoretischer Hintergrund

Wir nehmen an, dass wir eine Ja-Instanz von Annotiertes Max $(k,\ n-k)$-Cut betrachten, es also $S \in \binom{V_G}{k}$ gibt mit $\text{val}(S) \geq t$. Dann können wir Knoten $v \in V_G \setminus T$

identifizieren, für die wir mit Sicherheit sagen können, dass es $S' \supseteq T \dot\cup \{v\}$ mit $\text{val}(S') \geq t$ gibt.

Interessant ist auch die Kontraposition dieser Aussage: Wenn es für einen solchen Knoten $v \in V_G$ keine Menge $S' \supseteq T \dot\cup \{v\}$ mit $\text{val}(S') \geq t$ gibt, dann gibt es überhaupt keine Teilmenge $S \in \binom{V_G}{k}$ mit $\text{val}(S) \geq t$. Es muss sich also um eine Nein-Instanz handeln. Hierdurch ist es uns möglich festzustellen, dass T nicht Teil einer gültigen Lösung S sein kann, und wir deswegen im Suchbaum zurückgehen können, um T zu verkleinern. Wir verwenden im Folgenden die Definition aus Koana et al. [5]:

Definition 7.3 (Zufriedenstellender Knoten)**.** Es sei I eine Ja-Instanz von Annotiertes Max $(k,\ n-k)$-Cut . Ein Knoten $v \in V \setminus T$ wird als *zufriedenstellend* bezeichnet, wenn:

$$\text{cont}(v, T) \geq t'/k' + 2 \cdot (k-1)$$

In diesem Fall kann die Inklusionsregel 6.3 auf v angewendet werden.

Implementierung

Sei b ein Suchbaumknoten, sodass $\text{vert}(b)$ zufriedenstellend ist. Wir definieren zudem $p := \text{parent}(b)$. Zudem sei b von allen Kindern von p derjenige Knoten mit dem höchsten Beitrag zu T_p. Da $\text{child}(p)$ bereits wegen Verbesserung 7.3 absteigend nach Beitrag sortiert ist, ist b das erste Kind.

Die Umsetzung hiervon ist wie folgt: Bevor der Suchbaum durchlaufen wird, initialisieren wir einen leeren Stack *sat* aus Booleans, siehe Zeile 4 von Algorithmus 7.

Angenommen, der Algorithmus ist erstmalig im Suchbaumknoten p. Es wurde daher noch keine einziges Kind von p besucht. Ist b zufriedenstellend, so speichern wir für p im Stack *sat* den Wert `true` ab, andernfalls den Wert `false`. Hierzu werten wir in Zeile 24 von Algorithmus 7 aus, ob für b bereits ein Eintrag vorhanden ist. Wenn nein, wird in Zeile 25 von Algorithmus 7 an *sat* angehangen, ob $\text{cont}(v, T) \geq t'/k' + 2 \cdot (k-1)$ gilt. Gelangen wir im Suchbaum zu einem späteren Zeitpunkt erneut zu p, so können wir für p den Wert von *sat* auslesen. Ist dieser `true`, so wissen wir, dass wir direkt zu $\text{parent}(p)$ springen können, da wir keine Lösung innerhalb von $\text{tree}(b)$ finden konnten, weil wir ohne abbrechen bis zu p zurückgewandert sind. Somit wird $\text{child}(p) \setminus \{b\}$ übersprungen. Hat der Eintrag in *sat* allerdings den Wert `false`, so konnte die Regel nicht angewendet werden und der Durchlauf wird ungeändert weitergeführt.

Es ist anzumerken, dass auch wenn für einen Suchbaumknoten die Regel nicht anwendbar ist, dennoch der Wert `false` an *sat* angehangen werden muss. Andernfalls hat *sat* eine inkorrekte Größe, und es kann nicht mehr eindeutig zugeordnet werden, welcher Eintrag zu welchem Suchbaumknoten gehört.

Algorithmus 6 Unnötige Knoten entfernen

Require: Teilmengengröße k, Graph G, Zielwert t
1: **if** $|T| = k$ oder $\text{cont}(T, ext.top.first) \geq \frac{t'}{k'} + 2 \cdot (k-1)$ **then**
2: **return** ▷ *Zufriedenstellender Knoten vorhanden*
3: **for** $child \in ext.top$ von hinten nach vorne **do**
4: **if** $\text{cont}(T, child) \geq \frac{t'}{k'} - 2 \cdot (k-1)^2$ **then return**
5: Entferne letztes Element von $ext.top$ ▷ *Anwendung von Regel*

7.5 Unnötige Knoten

Theoretischer Hintergrund

Ein *unnötiger* Knoten u ist so definiert, dass wir eindeutig wissen, dass sich das Ergebnis des Suchbaums nicht ändert, wenn u ausgelassen wird. Hierzu verwenden wir die Definition aus Koana et al. [5]:

Definition 7.4 (Unnötiger Knoten)**.** Sei I eine Ja-Instanz von Annotiertes Max $(k,\ n-k)$-Cut . Ein Knoten $v \in V \setminus T$ wird als *unnötig* bezeichnet, wenn:

$$\text{cont}(v, T) < t'/k' + 2 \cdot (k-1)^2$$

Nehmen wir an, dass $u \in \text{child}(b)$. Statt $\text{child}(b)$ kann also auch $\text{child}(b) \setminus \{u\}$ durchlaufen werden. Diese Umsetzung unterscheidet sich leicht von Koana et al. [5], da hierzu nicht u aus dem Graphen gelöscht werden muss, da u aus $\text{child}(b)$ entfernt werden kann, ohne den Graphen zu verändern. Somit bleibt u im Graphen erhalten und wird trotzdem nicht zur Teillösung hinzugefügt. Außerdem muss counter somit nicht verwendet werden, da beim Auswerten von val_G der Knoten u im Graphen zur Verfügung steht. Im Kern in Abschnitt 7.6 hingegen werden für counter neue Werte gesetzt und die Implementierung somit näher an Abschnitt 6 gehalten.

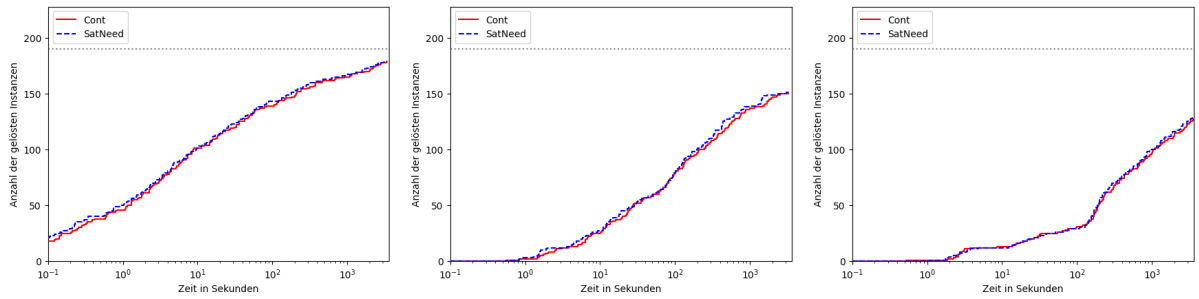
Theorem 7.2. Sei I eine Instanz von Annotiertes Max $(k,\ n-k)$-Cut und sei $v \in V \setminus T$ ein unnötiger Knoten. Dann darf die Exklusionsregel 6.4 auf v angewandt werden.

Implementierung

Der vollständige Algorithmus für Regel 7.2 ist in Abbildung 6 abgebildet.

Damit die Regel für unnötige Knoten angewandt werden kann, darf kein einziger Knoten der aktuellen Extension zufriedenstellend sein. Da die Knoten nach absteigendem Beitrag sortiert sind, muss nur für den ersten Knoten geprüft werden, ob dieser zufriedenstellend ist, siehe Zeile 2 von Algorithmus 6.

Die unnötigen Knoten sind nun alle am Ende der Liste zu finden. Daher werden die Einträge von hinten nach vorne durchgegangen, siehe Zeile 3. Sobald der erste Eintrag gefunden wurde, dessen Beitrag so groß ist, dass er nicht unnötig ist, wissen wir, dass auch alle davor liegenden Einträge unnötig sind, weshalb in Zeile 4 direkt abgebrochen werden kann. Andernfalls kann der Knoten in Zeile 5 entfernt werden.



(a) $k \in \{1, \ldots, 10\}$ (b) $k \in \{11, \ldots, 20\}$ (c) $k \in \{21, \ldots, 30\}$

Abbildung 11: Gelöste Instanzen der Algorithmen *Cont* und *SatNeed*

Auswertung

Der aktuelle Abschnitt 7.5 und der vorhergehende Abschnitt 7.4 werden gemeinsam ausgewertet, da beide Regeln auf der annotierten Problemformulierung aus Abschnitt 6 basieren. Die Version mit diesen und auch allen vorangehenden Verbesserungen wird im Folgenden *SatNeed* genannt.

Wie in Abbildung 11 zu sehen ist, konnten kaum mehr Instanzen gelöst werden, sowohl für $k \in \{1, \ldots, 10\}$ als auch für $k \in \{11, \ldots, 20\}$ und $k \in \{21, \ldots, 30\}$. In Tabelle 1 ist zu sehen, dass bereits in *Cont* alle 30 getesteten Werte für viele Graphen gelöst werden konnten, wie etwa soc-firm-hi-tech oder rec-amazon. Da dort also keine Steigerung mehr zu erwarten ist, spricht dies dafür, dass die Ergebnisse von *SatNeed* nicht so schlecht sind, wie es unmittelbar scheint.

7.6 Problemkern durch Maximalgrad

Theoretischer Hintergrund

Noch bevor auch nur ein einziger Suchbaum durchlaufen wird, kann eine Instanz (G, k) der Optimierungsvariante von Annotiertes Max $(k,\ n-k)$-Cut durch einen Problemkern verkleinert werden.

Für Annotiertes Max $(k,\ n-k)$-Cut ist die aktuelle Teillösung in T abgespeichert, siehe Abschnitt 6. Für T definieren wir den Wert $\Delta_{\overline{T}} := \max_{v \in V_G \setminus T} \deg(v)$, um die Knotengrade außerhalb T kompakt bezeichnen zu können. Damit gilt insbesondere $\Delta = \max_{v \in V_G} \deg(v) \geq \Delta_{\overline{T}} \quad \forall T \subseteq V_G$.

Mithilfe von $\Delta_{\overline{T}}$ kann nun die Reduktionsregel für den Kern definiert werden:

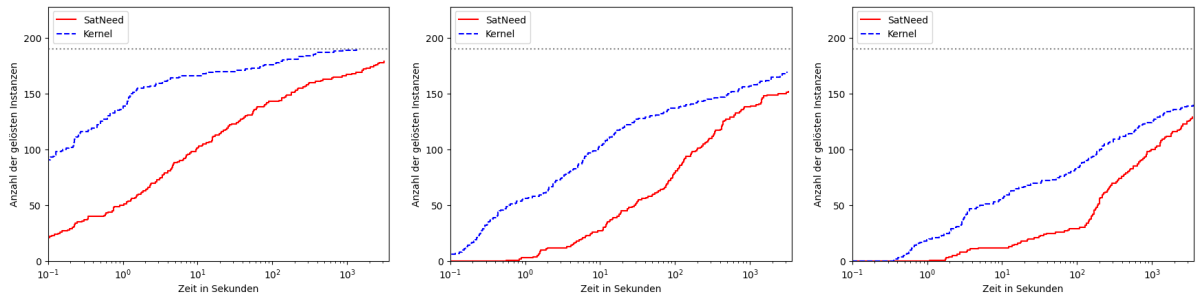
Reduktionsregel 7.3 (Problemkern durch Maximalgrad)**.** Sei I eine Instanz von Annotiertes Max $(k,\ n-k)$-Cut. Wenn es mindestens $(\Delta_{\overline{T}} + 1) \cdot (k-1) + 1$ Knoten gibt, welche besser als v hinsichtlich T sind, dann wende auf v die Exklusionsregel 6.4 an.

Für die theoretische Herleitung dieser Regel verweisen wir auf Koana et al. [5].

Algorithmus 7 Kern

Require: Teilmengengröße k, Graph G, Zielwert t
1: $T \leftarrow$ leere Liste von Knoten
2: $val_T \leftarrow 0$
3: $ext \leftarrow$ Leerer Stack, welche Listen von Knoten enthält
4: $sat \leftarrow$ leere Liste von Booleans
5: Hänge an ext eine Liste an, die V_G in absteigender Reihenfolge nach $cont$ enthält
6: Wende Regel für unnötige Knoten auf $ext.top$ an
7: **while** ext ist nicht leer **do**
8: $back_{bound} \leftarrow$ `true` wenn die obere Schranke von T nicht größer als val_S ist.
9: $back_{sat} \leftarrow$ `true` wenn sat für den aktuellen Suchbaumknoten gesetzt ist und den Wert *true* hat.
10: **if** $|T| \geq k$ or $|T| + |ext.top| < k$ or $back_{bound}$ or $back_{sat}$ **then** ▷ *Backtrack*
11: **if** $|T|$ is equal to k and $\text{val}(T) \geq t$ **then**
12: **return** $(T, \text{val}(T))$ als Lösung
13: **if** $|T| < k$ **then**
14: Entferne letztes Element von ext
15: **if** $|T| > 0$ **then**
16: Entferne letzten Knoten v von T
17: $val_T \leftarrow val_T - \text{cont}(T, v)$
18: **else** ▷ *Branch*
19: Wähle den erste Knoten v aus $ext.top$
20: Entferne v aus $ext.top$
21: **if** $|T| < k - 1$ **then**
22: Hänge Kopie von $ext.top$ an ext an
23: Sortiere $ext.top$ absteigend nach Beitrag
24: **if** sat noch keinen Wert für aktuellen Suchbaumknoten enthält **then**
25: Hänge $\text{cont}(T, v) \geq \frac{t'}{k'} + 2 \cdot (k - 1)$ an sat an
26: $val_T \leftarrow val_T + \text{cont}(T, v)$
27: Hänge v an T an
28: Wende Regel für unnötige Knoten auf $ext.top$ an
29: **return** *null*

Da $T \subseteq V_G$ beliebig ist, kann diese Regel in beliebigen Suchbaumknoten b angewandt werden. Hierbei entsteht jedoch das Problem, dass bei Verlassen von tree(b) die Änderungen durch die Regel wieder rückgängig gemacht werden müssen. Außerdem führt ein Anwenden in jedem Suchbaumknoten dazu, dass einzelne Anwendungen kaum Effekt haben und die meisten Aufrufe nur Laufzeit benötigen. Daher wenden wir sie nur zu Beginn des Suchbaums, also für $T := \emptyset$, an. Der Vollständigkeit halber bleibt $T \subseteq V_G$ im Folgenden dennoch beliebig.



(a) $k \in \{1, \ldots, 10\}$ (b) $k \in \{11, \ldots, 20\}$ (c) $k \in \{21, \ldots, 30\}$

Abbildung 12: Gelöste Instanzen der Algorithmen *SatNeed* und *Kern*

Implementierung

Algorithmus 8 Wende Kern an

Require: Graph G, Teilmengengröße $k \in \mathbb{N}$

1: $V_{sorted} \leftarrow V_G$ absteigend sortiert nach Beitrag zu T
2: $x \leftarrow (\Delta_{\overline{T}} + 1) \cdot (k - 1) + 1$
3: Die ersten x Elemente von V_{sorted} werden übersprungen, auf die restlichen wird die Exklusionsregel 6.4 angewandt.
4: **return** `true` wenn mindestens ein Knoten entfernt wurde, sonst `false`.

Die Umsetzung der Reduktionsregel in Algorithmus 8 soll erschöpfend angewandt werden. Dass heißt, solange es einen Knoten im Graphen gibt, auf den Reduktionsregel 7.3 anwendbar ist, sollte dies getan werden. Daher ist zu beachten, dass nach einer Ausführung von Algorithmus 8 der Maximalgrad von $\Delta_{\overline{T}}$ sinken kann, da eventuell ein Knoten gelöscht wird, welcher zum Maximalgrad beiträgt. Daher liefert Algorithmus 8 einen Boolean zurück, ob eine Änderung am Graphen durchgeführt wurde. Solange das Ergebnis hiervon `true` ist, kann Algorithmus 8 erneut aufgerufen werden. Erst bei Ergebnis `false` ist die Anwendung des Kerns somit abgeschlossen.

Zu Beginn in Zeile 1 von Algorithmus 8 wird V_G absteigend sortiert nach $\text{cont}(\emptyset, v)$, sodass darauf folgend in Zeilen 2 und 3 leicht am Ende der sortierten Liste diejenigen Knoten identifiziert werden können, auf welche die Exklusionsregel angewandt werden soll. Zuletzt wird in Zeile 4 zurückgeliefert, ob eine Änderung an der Instanz durchgeführt wurde, was für die erschöpfende Anwendung genutzt wird.

Auswertung

Der Algorithmus, welcher den Kern und alle vorangehenden Verbesserungen enthält, wird *Kern* genannt. Da der Kern die letzte hinzugefügte Verbesserung ist, ist *Kern* also der vollständige und auch schnellste Solver, wie unsere Experimente zeigen. Abbildung 12 vergleicht die gelösten Instanzen von *Kern* und *SatNeed*.

Für $k \in \{1, \ldots, 10\}$ ist eine sehr große Beschleunigung zu sehen. *Kern* kann bereits in 100 Sekunden fast alle Instanzen lösen, *SatNeed* benötigt hierzu etwa eine Stunde.

Auch für $k \in \{11, \ldots, 20\}$ hat eine große Verbesserung stattgefunden. Bei circa 10 Sekunden ist die Differenz der Anzahl gelöster Instanzen am größten und fängt dann an, sich zu verringern, was daran liegt, dass schon für *SatNeed* viele Graphen bis zu $k = 30$ vollständig gelöst werden konnten, siehe Tabelle 1. In Graphen, welche vorher schlechte Ergebnisse zu beobachten waren, konnten teilweise massive Verbesserungen erzielt werden, wie etwa für soc-brightkite, bei welchem das maximale k von 4 auf 12 gesteigert werden konnte, was also eine Verdreifachung ist.

Die Gruppe $k \in \{21, \ldots, 30\}$ in (c) hat für die gesamte Stunde eine Verbesserung von 129 auf 140 Instanzen erzielt, was einer Verbesserung von 8% entspricht. Vor allem jedoch im Zeitabschnitt von 10 bis 100 Sekunden konnten viel mehr Instanzen gelöst werden. So ist ist bei 100 Sekunden für *Kern* der y-Wert bei 84, für *SatNeed* jedoch nur 30.

Insgesamt hat das Anwenden des Problemkerns also einen großen positiven Einfluss auf die Laufzeit des Algorithmus. Ein weiterer Vorteil ist, dass er sehr isoliert vor der Ausführung des Suchbaums angewandt wird und die Implementierung des Kerns dadurch sehr modular und einfach zu testen ist.

7.7 Zusammenfassung der Verbesserungen

Der Algorithmus mit allen Verbesserungen, die bisher vorgestellt wurden, heißt *Kern*. Algorithmus 7 zeigt dessen Pseudocode. Nachfolgend sind alle Unterschiede zwischen *Basis* und *Kern*:

- Es wird eine obere Schranke genutzt, um innerhalb des Suchbaums backtracken zu können und frühzeitig festzustellen, ob der aktuelle Lösungskandidat optimal ist, siehe Abschnitt 7.1.
- Es wird lokale Suche als Heuristik verwendet, um früh einen recht guten Kandidaten zu finden, siehe Abschnitt 7.2.
- Mithilfe der Definition des Betrags eines Knotens zur aktuellen Teil-Lösung aus Abschnitt 6.1 wird *ext* in jeder Schicht greedy nach dem Beitrag sortiert. Außerdem kann der Funktionswert der aktuellen Teilmenge über den Suchbaum hinweg durch §cont mit recht geringem Rechenaufwand aktualisiert werden.
- Es werden die Regeln für zufriedenstellende und nutzlose Knoten umgesetzt, welche in Abschnitt 6 eingeführt wurden.
- Vor Durchlauf des Suchbaums wird die Probleminstanz durch den Kern aus Abschnitt 7.6 verkleinert.

In Pseudocode 7 wird der finale Algorithmus *Kern* gezeigt.

8 ILP-Formulierung

8.1 Einführung

Eine exakte Lösung für Max $(k, n-k)$-Cut kann auch dadurch gefunden werden, dass das Problem durch ein Integer Linear Programm (ILP) modelliert wird, also ein mathematisches Optimierungsproblem mit ganzzahligen Variablen, linearen Nebenbedingungen und einer linearen Zielfunktion.

Ein ILP besteht immer aus folgenden Teilen:

- Variablen $x_1, ..., x_n$ mit $x_i \in W \subseteq \mathbb{Z} \quad \forall i \in [n]$, wobei oft $W = \{0, 1\}$ ist.
- Eine Funktion f zur Minimierung oder Maximierung, welche linear in den Variablen x_i ist. Auch ist anzumerken, dass f nicht von allen x_i abhängen muss, sondern auch nur eine Teilmenge der Variablen nutzen kann.
- Linearen Nebenbedingungen, die die Menge der möglichen Lösungen einschränken.

Schon für $W = \{0, 1\}$ sind ILPs $\mathcal{NP}$-vollständig [16]. Die $\mathcal{NP}$-Vollständigkeit hat zur Folge, dass ILPs sehr viele Probleme modellieren können.

Die meisten Graphprobleme lassen sich leicht als ILP umsetzen, da Entscheidungen als binäre Variable (also Wertebereich $\{0, 1\}$) dargestellt werden können. Möchte man etwa eine Teilmenge $M \subseteq V(G)$ wählen, die eine bestimmte Eigenschaft optimiert, wie zum Beispiel bei Max $(k, n-k)$-Cut oder auch Vertex Cover, so kann für jeden Knoten $v_i \in V(G)$ eine Variable $x_i \in \{0, 1\}$ darstellen, ob der Knoten v_i in der Lösung enthalten ist.

8.2 ILP-Formulierung für Max $(k, n-k)$-Cut

In der folgenden Formulierung von Max $(k, n-k)$-Cut als ILP wird für jeden Knoten $v \in V$ eine binäre Entscheidungsvariable x_v eingeführt. Wir modellieren $v \in S$ durch $x_v = 1$ und $v \in V \setminus S$ wird modelliert durch $x_v = 0$.

Der schwierigere Teil ist nun, die Zielfunktion zu modellieren, da diese von den Kanten im Graphen abhängt und nicht unmittelbar aus S abzulesen ist. Hierzu wird für jede Kante $\{v, w\} \in E$ die binäre Entscheidungsvariable $y_{v,w}$ definiert. Würden wir basierend auf diesem Stand nun die Zielfunktion $\sum_{\{v,w\} \in E} y_{v,w}$ maximieren wollen, so hätten wir unmittelbar den Optimalwert m, da es keine Einschränkung gibt, wann $y_{v,w} = 1$ erlaubt ist. Dies ist nur dann möglich, wenn $x_v \neq x_w$ gilt, die beiden Endpunkte der Kante also auf verschiedenen Seiten des durch S induzierten Cuts liegen. Dies ist gleichbedeutend zu $XOR(x_v, x_w)$, da beide Variablen den Wertebereich $\{0, 1\}$ besitzen.

Dies kann erreicht werden, indem pro $y_{v,w}$ je zwei Ungleichungen definiert werden.

- $y_{v,w} \leq x_v + x_w \quad \forall \{v, w\} \in E$

  Diese Bedingung erreicht, dass wenn $x_v = 0$ und $x_w = 0$ auch $y_{v,w} \leq 0 + 0 = 0$ gilt. Da $y_{v,w} \in \{0, 1\}$ gilt, ist folglich $y_{v,w} = 0$.

- $y_{v,w} \leq 2 - x_w - x_w \quad \forall \{v, w\} \in E$

  Diese Bedingung erreicht, dass wenn $x_v = 1$ und $x_w = 1$ auch $y_{v,w} \leq 2 - 1 - 1 = 0$ gilt. Da $y_{v,w} \in \{0, 1\}$ gilt, ist folglich $y_{v,w} = 0$.

Um die gewünschte Größe k von S sicherzustellen, müssen wir zudem die Bedingung $\sum_{v \in V} x_v = k$ erstellen, welche den Lösungsraum einschränkt. Pro Knoten und pro Kante gibt es jeweils genau eine Variable, insgesamt existieren dann also $n + m$ Variablen. Wir sehen, dass pro Kante zwei Nebenbedingungen genutzt werden, und zusätzlich die Bedingung $\sum_{v \in V} x_v = k$ hinzugefügt wird. Somit gibt es insgesamt $2m + 1$ Nebenbedingungen, wobei $2m$ Nebenbedingungen jeweils nur drei Variablen verwenden. Dass sowohl die Zahl der Variablen und der Nebenbedingungen nur linear wächst ist wünschenswert und spricht für die Güte dieser ILP-Modellierung von MAX $(k, n-k)$-CUT .

Die volle formale Spezifikation des ILPs lautet also:

Variablen

- $x_v \forall v \in V(G)$
- $y_{v,w} \forall \{v, w\} \in E(G)$

Zielfunktion

- $\max \sum_{\{v,w\} \in E} y_{v,w}$

Nebenbedingungen

- $\sum_{v \in V} x_v = k$
- $y_{v,w} \leq x_v + x_w \quad \forall \{v, w\} \in E$
- $y_{v,w} \leq 2 - x_v - x_w \quad \forall \{v, w\} \in E$

Die Formulierung von MAX-CUT als ILP wird häufig genutzt und kommt auch als Übung in Vorlesungen zum Algorithmendesign vor. Die hier genutzte Abwandlung zu MAX $(k, n-k)$-CUT fügt nur noch die Nebenbedingung $\sum_{v \in V} x_v = k$ hinzu.

9 Vergleich von Branch and Bound mit ILP

In diesem Abschnitt werden die ILP-Formulierung aus Abschnitt 8 und der selbst implementierte Suchbaumalgorithmus miteinander bezüglich ihrer Laufzeit verglichen. Für den Suchbaum wird nur die Version *Kern* aus Abschnitt 7.6 genutzt, da diese alle Verbesserungen aus den vorherigen Abschnitten beinhaltet.

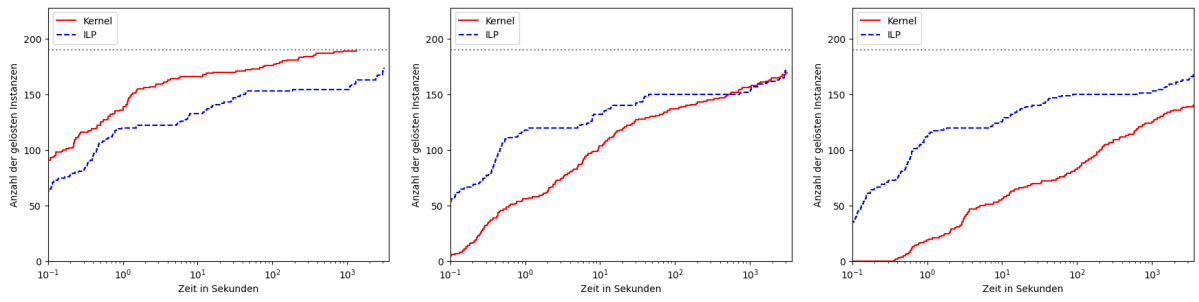
9.1 Laufzeit

Abbildung 13 zeigt in Teil (a), dass für den Wertebereich $k \in \{1, \ldots, 10\}$ der Algorithmus *Kern* bereits in etwa 25 Minuten alle $10 \cdot 19 = 190$ Instanzen lösen konnte, während für das ILP nur 176 Instanzen innerhalb einer Stunde gelöst werden konnten. In Tabelle 1 ist die Ursache ersichtlich: Für die Graphen *bio-CE-CX* und *H2O* konnten für k nur Werte bis 5 beziehungsweise 1 jeweils innerhalb einer Stunde gelöst werden. Für *Kern* haben diese beiden Graphen auch teilweise recht viel Zeit benötigt, jedoch konnten maximale Werte von 18 und 30 für k erreicht werden.

Für das Intervall $k \in \{11, \ldots, 20\}$ in Teil (b) ist zu sehen, dass durch *Kern* 171 Instanzen gelöst werden konnten und für das ILP 173 Instanzen, also ein nahezu gleiches Ergebnis. Im Zeitraum von unter 10 Sekunden konnten durch das ILP zwischenzeitig mehr als doppelt so viele Instanzen gelöst werden, jedoch hat sich bis etwa 100 Sekunden dieser Vorsprung wieder nahezu ausgeglichen.

Der größte Unterschied der Laufzeit zwischen beiden Solvern ist beim Intervall $k \in \{21, \ldots, 30\}$. Hier konnten durch das ILP insgesamt 169 Instanzen gelöst werden, durch *Kern* hingegen 141. Das IKP konnte innerhalb einer Stunde also 20% mehr Instanzen lösen. Zum Zeitpunkt von 2 Sekunden ist das Ergebnis noch viel schlechter für *Kern*: Durch das ILP konnten zu diesem Zeitpunkt bereits 121 Instanzen gelöst werden, durch *Kern* hingegen nur 27, womit das ILP um den Faktor ~ 4.5 besser ist.

Für die Graphen *bio-CE-CX* und *H2O* war der Algorithmus *Kern* verhältnismäßig schnell, da diese beiden Graphen recht sparse sind und einen niedrigen Maximalgrad besitzen. So ist der maximale Grad für den Graphen *H2O* nur 36, was im Vergleich zu seinen



(a) $k \in \{1, \ldots, 10\}$ (b) $k \in \{11, \ldots, 20\}$ (c) $k \in \{21, \ldots, 30\}$

Abbildung 13: Gelöste Instanzen der Algorithmen *Kern* und *ILP*

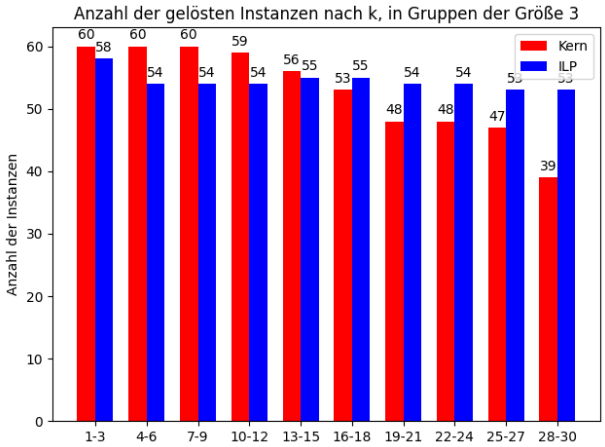


Abbildung 14: Gelöste Instanzen der Algorithmen *Kern* und *ILP*

67 024 Knoten sehr gering ist. Aus diesem Grund konnte die Instanz auf einen sehr kleinen Kern reduziert werden, weshalb alle $k \in \{1, \ldots, 30\}$ lösbar waren.

In Abbildung 14 ist ablesbar, wie viele Instanzen von *Kern* und das ILP innerhalb einer Stunde lösbar waren in Abhängigkeit von k. Da die insgesamt 30 verschiedenen Werte von k schwierig in die horizontale Achse einzutragen sind, wurden die Werte von k in Gruppen der Größe 3 betrachtet. Für kleine Werte von k ist der Algorithmus *Kern* leicht besser. Bei etwa $k = 16$ ist die Schwelle, an der das ILP gleichzieht und der Vorsprung des ILPs vergrößert sich weiter für steigende k. In der Gruppe $k \in \{28, 29, 30\}$ kann das ILP 53 Instanzen lösen, *Kern* hingegen nur 39, das ILP kann also 36% mehr Instanzen lösen.

10 Fazit und Ausblick

Durch die Experimente konnte gezeigt werden, dass der Basisalgorithmus aus Abschnitt 4 durch die Verbesserungen aus Abschnitt 7 stark beschleunigt wird. So konnten für den Basisalgorithmus nur insgesamt drei Instanzen mit $k > 10$ gelöst werden, für *Kern* hingegen 141. Vor allem die Heuristik aus Abschnitt 7.2, die Nutzung des Beitrags eines Knotens in Abschnitt 7.3 und der Kern aus Abschnitt 7.6 haben die Laufzeit jeweils signifikant senken können.

Die Regeln für zufriedenstellende und unnötige Knoten, welche durch die abgewandelte Problemstellung Annotiertes Max $(k,\ n-k)$-Cut eingeführt wurden und in Abschnitten 7.4 und 7.5 getestet werden, konnten keinen großen Nutzen vorweisen. Eine sinnvolle Idee für zukünftige Forschung wäre es, den Grund hierfür genauer herauszufinden. Eine Möglichkeit ist es, dass Fälle, in denen beide Regeln anwendbar sind, schon durch zuvor eingeführte Regeln abgedeckt werden. Auch kann es sein, dass die Regeln auf nicht getesteten Instanzen hilfreich sind und somit eine nicht repräsentative Menge von Graphen zum Testen genutzt wurde. Die 19 getesteten Graphen haben alle die Eigenschaft, dass wenige Knoten einen hohen Grad und viele Knoten einen kleinen Grad haben. In der Zukunft könnten Graphen getestet werden, deren Knotengräde recht einheitlich sind.

In Abschnitt 8 wurde eine Formulierung von Max $(k, n-k)$-Cut als ILP präsentiert. Aktuell verwendet dieses $n+m$ Variablen und $2 \cdot m+1$ Nebenbedingungen. Möglicherweise kann eine kleiner ILP-Formulierung gefunden werden, was möglicherweise auch eine geringere Laufzeit zur Folge hätte. Auch ist der Problemkern aus Abschnitt 7.6 prinzipiell auch für ILPs umsetzbar. Gerade für Graphen wie *bio-dmela* oder *H20*, in denen das ILP schlecht abschneidet, wäre die in der Praxis erreichte Beschleunigung interessant.

Für den Bereich $k \in \{1, \ldots, 13\}$ ist Algorithmus *Kern* leicht schneller, oberhalb dieser Grenze ist das ILP schneller. Für $k \in \{28, 29, 30\}$ konnte das ILP insgesamt 36% mehr Instanzen lösen. Für kleine Werte von k ist also meist *Kern* zu bevorzugen, für große Werte von k hingegen scheint das ILP sinnvoller.

Obwohl durch die Verbesserungen aus Abschnitt 7 der naive Suchbaum sehr stark beschleunigt werden konnte, konnten innerhalb einer Stunde durch das ILP minimal mehr Instanzen gelöst werden.

Zudem könnten auch manche Parameter an den bestehenden Verbesserungen untersucht werden. So wird für die lokale Suche in Abschnitt 7.2 nur die 1-Swap-Nachbarschaft untersucht. Hier könnten auch größere Nachbarschaften in Betracht gezogen werden. Auch ist der Kern in Abschnitt 7.6 abhängig vom maximalen Grad des Graphen. Hier wäre die Evaluation von Kernen in Abhängigkeit von anderen Graphparametern ein womöglich vielversprechender Ansatz.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit mit dem Titel “Algorithm Engineering für Max Cut mit Cardinality Constraints” selbstständig und ohne unerlaubte fremde Hilfe angefertigt, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und die den verwendeten Quellen und Hilfsmitteln wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Ort, Datum Unterschrift

Literatur

[1] L. Cai, "Parameterized complexity of cardinality constrained optimization problems," *Comput. J.*, vol. 51, no. 1, pp. 102–121, 2008.

[2] M. G. Everett and S. P. Borgatti, "The centrality of groups and classes," *The Journal of Mathematical Sociology*, vol. 23, no. 3, pp. 181–201, 1999.

[3] D. G. Corneil and L. K. Stewart, "Dominating sets in perfect graphs," *Discret. Math.*, vol. 86, no. 1-3, pp. 145–164, 1990.

[4] S. Saurabh and M. Zehavi, "Parameterized Complexity of Multi-Node Hubs," in *13th International Symposium on Parameterized and Exact Computation (IPEC 2018)* (C. Paul and M. Pilipczuk, eds.), vol. 115 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 8:1–8:14, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[5] T. Koana, C. Komusiewicz, A. Nichterlein, and F. Sommer, "Covering many (or few) edges with k vertices in sparse graphs," in *Proceedings of the 39th International Symposium on Theoretical Aspects of Computer Science (STACS 22)* (P. Berenbrink and B. Monmege, eds.), vol. 219 of *LIPIcs*, pp. 42:1–42:18, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[7] D. Ferizovic, D. Hespe, S. Lamm, M. Mnich, C. Schulz, and D. Strash, "Engineering kernelization for maximum cut," in *2020 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, arXiv, 2019.

[8] S. Saurabh and M. Zehavi, "$$(k, n-k)$$ (k , n - k) -max-cut: An $$\mathcal{O}^*(2^p)$$ O ∗ (2 p) -time algorithm and a polynomial kernel," *Algorithmica*, vol. 80, no. 12, pp. 3844–3860, 2018.

[9] J. A. Bondy and U. S. R. Murty, *Graph Theory*. Graduate Texts in Mathematics, Springer, 2008.

[10] S. Arora and B. Barak, *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

[11] R. Niedermeier, *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.

[12] R. G. Downey and M. R. Fellows, *Parameterized Complexity*. Monographs in Computer Science, Springer, 1999.

[13] D. Michail, J. Kinable, B. Naveh, and J. V. Sichi, "Jgrapht - A java library for graph data structures and algorithms," *ACM Transactions on Mathematical Software*, vol. 46, no. 2, pp. 16:1–16:29, 2020.

[14] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Association for the Advancement of Artificial Intelligence*, 2015.

[15] L. Staus, "Algorithm engineering für group closeness centrality," Bachelorarbeit, Philipps-Universität Marburg, 2021.

[16] R. 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, USA* (R. E. Miller and J. W. Thatcher, eds.), The IBM Research Symposia Series, pp. 85–103, Plenum Press, New York, 1972.