Technische Universität Berlin
Fachbereich Algorithmik und Komplexitätstheorie

Bachelorarbeit

Im Studiengang Informatik

Algorithm Engineering für das Auffinden dichter Teilgraphen in dünnen Graphen

eingereicht von Kolja Stahl
Matrikelnummer: 325372

eingereicht am 7.6.2013

Betreuer Prof. Dr. Rolf Niedermeier
Dr. Christian Komusiewicz
Manuel Sorge

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Datum, Unterschrift

Abstract

CLIQUE ist ein wichtiges und viel studiertes NP-schweres Problem. Eingabe ist ein Graph G und eine Lösungsgröße k und gesucht wird eine Clique mit mindestens k Knoten. Für einige Anwendungen ist die Cliquen-Definition jedoch zu restriktiv. μ-Cliquen stellen eine Relaxierung der Cliquen-Definition dar. Dabei muss die Dichte des Graphs mindestens μ sein. Die μ-Clique muss nicht zwingend verbunden sein.
In dieser Arbeit werden zwei Algorithmen zum Finden von verbundenen μ-Cliquen einer gegebenen Mindestgröße implementiert und mit den Methoden des Algorithm Engineering optimiert. Das Entscheidungsproblem für μ-Cliquen ist NP-schwer.
Den Kern der Arbeit bildet die „Quasi-Vererbbarkeit“ und der Δ-Algorithmus. Wir zeigen zunächst, dass die Quasi-Vererbbarkeit auch auf verbundene μ-Cliquen übertragen werden kann, sofern $\mu \geq 0{,}5$ ist. Im Anschluss wird die Funktionsweise des Δ-Algorithmus erläutert. Der Δ-Algorithmus ist ein Suchbaumalgorithmus, dessen exponentieller Laufzeitanteil nur vom Maximalgrad Δ abhängt. Es werden mögliche Optimierungen für den Δ-Algorithmus dargelegt und evaluiert. Die Optimierungen beinhalten das Vorsortieren der abzuarbeitenden Knoten, sowie untere und obere Schranken. Zum Schluss wird beschrieben, wie die Algorithmen parallelisiert werden können und diese Parallelisierungen experimentell ausgewertet.

Danksagung

Ich möchte mich bei meinen Betreuern Prof. Dr. Rolf Niedermeier, Dr. Christian Komusiewicz und Manuel Sorge für ihre Unterstützung und Ausdauer bedanken.

Inhaltsverzeichnis

1 Einführung

Viele Probleme in der Praxis lassen sich mit Hilfe von Graphen modellieren. Graphen eignen sich dazu Beziehungen, Abfolgen und Abhängigkeiten zu repräsentieren und sind somit oft ein fester Bestandteil von Planungs- und Optimierungsproblemen. Ein *Graph* $G = (V, E)$ ist ein Tupel, bestehend aus einer Menge V von *Knoten* und einer Menge $E \subseteq \{\{u, v\} \mid u, v \in V \wedge u \neq v\}$ von *Kanten*. Knoten $v_1, v_2 \in V$ sind *benachbart*, wenn eine Kante $\{v_1, v_2\} \in E$ existiert.
Cliquen stehen umgangssprachlich für ein eng verknüpftes soziales Gefüge. Mitglieder einer Clique sind in irgend einer Form miteinander verbunden, sei es aufgrund von gemeinsamen Erfahrungen, Interessen oder anderen Begebenheiten. In der Graphentheorie ist eine *Clique* ein kompletter Graph, d.h. jeder Knoten einer Clique ist zu jedem anderen Knoten benachbart. Soziale Netzwerke können zum Beispiel als Graphen dargestellt werden, indem man die Menschen als Knoten und das miteinander Bekanntsein als Kanten auffasst. Durch das Auffinden oder Identifizieren solcher Cliquen innerhalb eines Graphen erhofft man sich z.B. Rückschlüsse auf Eigenschaften oder Funktionen einzelner Knoten innerhalb dieser Clique, ziehen zu können. „Community Detection“ spielt eine große Rolle in der Soziologie, Biologie und Informatik [9].

CLIQUE:
Eingabe: Graph $G = (V, E)$ und nichtnegative Ganzzahl k
Frage: Existiert eine Clique in G mit mindestens k Knoten?

CLIQUE ist ein wichtiges und viel studiertes Problem, das zu den „21 klassischen NP-vollständigen Problemen“ gehört. Die NP-Vollständigkeit wurde von Richard M. Karp 1972 bewiesen [12]. CLIQUE findet u.a. Anwendung in der Bioinformatik und in der Analyse sozialer Netzwerke. Für einige Anwendungen ist die Cliquen-Definition zu restriktiv. Ein Beispiel sind Fehlerbehaftete Netzwerke. μ-Cliquen stellen eine Relaxierung der Cliquen-Definition dar: μ-Cliquen sind Graphen mit Dichte mindestens μ. Wobei $0 < \mu \leq 1$ ist und die μ-Clique nicht zwingend verbunden sein muss. Die *Dichte (Density)* eines Graphen berechnet sich durch $2|E|/|V|(|V| - 1)$.

μ-CLIQUE:
Eingabe: Graph $G = (V, E)$ und nichtnegative Ganzzahl k
Frage: Existiert eine Knotenmenge $S \subseteq V$ mit $|S| \geq k$ und $G[S]$ ist eine μ-Clique?

Die Anwendungen von μ-CLIQUE sind vielfältig. Die größten Anwendungsgebiete sind Data Mining [17], Bioinformatik [7] und soziale Netzwerke [22].

Ein konkretes Anwendungsszenario ist das Zuweisen von Funktionen an bisher unbekannte Proteine in Protein-Protein-Interaktions Netzwerken. Die Idee ist, dass sich hinter *μ-Cliquen* funktionale Gruppen verbergen [7]. Clique-basiertes Data Mining kann dazu benutzt werden verwandte Gene zu finden, bzw. zu extrahieren [17] oder auch um Communities innerhalb von sozialen Netzwerken zu identifizieren, was z.B. zielgerichtetes Marketing oder spezifische Content-Schaltung ermöglicht [18][22].
Wir beschäftigen uns mit dem Entscheidungsproblem für verbundene μ-Cliquen, d.h. entscheiden zu können, ob es bei gegebenem Graph $G = (V, E)$, einer Lösungsgröße k und einer Mindestdichte μ, eine μ-Clique gibt. Dass die μ-Cliquen verbunden sein sollen, hat primär praktische Gründe. Die meisten Strukturen, die man versucht zu finden, machen intuitiv nur Sinn, wenn sie verbunden sind; wie im Falle von Community Detection [9].

Wir beginnen die Arbeit mit dem Vorstellen einiger, für das Problem und die Problemlösung, notwendigen Begriffe, dem Benennen von Details zur Implementierung und einer Einordnung in die bestehende Literatur. Im Anschluss wird der h-Index-Algorithmus von Komusiewicz und Sorge [13] vorgestellt.
Der Kern der Arbeit beschäftigt sich mit der „Quasi-Vererbbarkeit“ und dem Δ-Algorithmus. Wir zeigen, dass die Quasi-Vererbbarkeit auch auf verbundene μ-Cliquen übertragen werden kann, sofern $\mu \geq 0,5$ ist (siehe Abschnitt 2). Daraus resultierende Ideen und Ansätze werden bei der Optimierung des Δ-Algorithmus benutzt. Der Δ-Algorithmus hat eine Laufzeit von $O(2^{2k} \cdot (\Delta - 1)^k \cdot (n + m))$ und nutzt aus, dass die maximale Anzahl an verbundenen Teilgraphen durch den Maximalgrad nach oben hin beschränkt ist. Zunächst werden wir die Funktionsweise des Δ-Algorithmus erläutern und dann einige Optimierungen vorstellen. Dazu gehören Vorsortierungen der abzuarbeitenden Knoten und untere, sowie obere Schranken. Die Optimierungen werden miteinander verglichen und experimentell ausgewertet. Schlussendlich wird eine Parallelisierung für den Δ- und den h-Index-Algorithmus vorgestellt und ein Ausblick auf weitere mögliche Ansatzpunkte gegeben.

1.1 Begrifflichkeiten

Im Folgenden werden nur einfache ungerichtete Graphen betrachtet. Die *Nachbarschaft* $N(v)$ eines Knotens $v \in V$ ist die Menge von Knoten zu denen v *benachbart* ist. Der *Grad* eines Knotens $v \in V$ wird mit $\deg(v)$ bezeichnet und entspricht der Anzahl der Nachbarn dieses Knotens, das heisst $\deg(v) = |N(v)|$. Ein *induzierter Teilgraph* $G[V']$ eines Graphen $G = (V, E)$ ist der Graph mit Knotenmenge $V' \subseteq V$ und Kantenmenge $E' = \{\{u, v\} | \{u, v\} \in E \wedge u \in V' \wedge v \in V'\}$. Der *Maximalgrad* eines Graphen wird mit Δ bezeichnet, der *Minimalgrad* mit δ. Ein *Cut-Knoten* in einem verbundenen Graphen G ist ein Knoten, dessen Löschung G in mindestens zwei Zusammenhangskomponenten zerfallen lässt. In einer *Zusammenhangskomponente* ist jeder Knoten von jedem anderen Knoten über einen Pfad erreichbar. Zusammenhangskomponenten sind maximal bezüglich ihrer Anzahl von Knoten. Graphen sind verbunden, wenn sie aus genau einer Zusammenhangskomponente bestehen. Dünne (engl.: *sparse*) Graphen sind Graphen mit

Tabelle 1.1: DIMACS Instanzen

Instanz	Knoten	Kanten	δ	Δ	Dichte	h-Index	d	c
brock800_4	800	207643	481	565	0.64	514	485	1
c-fat500-10	500	46627	185	188	0.37	187	185	1
hamming8-4	256	20864	163	163	0.63	163	163	1
keller4	171	9435	102	124	0.64	106	102	1
p_hat1500-1	1500	284923	157	614	0.25	456	252	1

einer geringen Dichte, bzw. einer, im Verhältnis zur Knotenanzahl, geringen Anzahl an Kanten.

1.2 Implementierungsdetails und Experimenteaufbau

Die Algorithmen sind in Haskell implementiert und mit dem Glasgow Haskell Compiler (GHC) in der Version 7.4.1 kompiliert worden. Die Testrechner besitzen 4-Kern Intel Xeon Prozessoren mit 3.6GHz, 68GB Ram und Debian Linux.
Die Knoten werden intern als Integer repräsentiert. Der Graph ist eine „Map“ optimiert für Integer-Keys. Sie bildet die Knoten auf eine Menge (Integer-Sets) ab, die alle Knoten von $N(v)$ enthält. Lookups in dieser Tabelle haben die Zeit $O(min(n, W))$, wobei W für die Anzahl an Bits der verwendeten Integer steht (32 oder 64), n für die Anzahl der Elemente (Knoten) [19]. Die Integer-Sets basieren auf einer ähnlichen Implementierung, wobei für die wichtigsten, von uns verwendeten Operationen, wie Schnitt- und Differenzmengenbildung eine Zeit von $O(n + m)$ angegeben ist; n, m steht für die Größe der Sets [3]. Die Datenstrukturen sind Teil der Standardbibliothek von Haskell und basieren auf „Patricia-Trees“.
Als Testinstanzen wurden Erdős [2], DIMACS [1] und biologische Instanzen (Protein-Protein-Interaktionsnetzwerke) benutzt (siehe Graphtabelle im Anhang). Die biologischen Instanzen stammen aus BioGRID [21]. Die Erdős Graphen sind Kollaborationsnetzwerke. Teilnehmer in dem Netzwerk werden als Knoten dargestellt. Beziehungen zwischen Teilnehmern werden mit einer Kante dargestellt. DIMACS Instanzen dienen als Test- und Benchmarkgraphen. Die biologischen Instanzen bestehen aus Protein-Protein-Netzwerken, wie sie in der Bioinformatik verwendet werden und eignen sich zur Analyse von globalen Netzwerkeigenschaften und Funktionen von Proteinen.
Die Mehrheit der Experimente wurden auf einer Auswahl (s. Tabellen 1.1 und 1.2) von Graphen (s. Graphtabelle im Anhang) durchgeführt. Beginnend mit $k = 2$ wird k erhöht, solange die Instanz lösbar ist, der Algorithmus also in dem vorgegebenen Zeitlimit terminiert. Es wurde ein festes Zeitlimit von 20 Minuten pro k pro Instanz gesetzt und $\mu = 0.7$ gewählt.

Tabelle 1.2: Biologische Instanzen

Instanz	Knoten	Kanten	δ	Δ	Dichte	h-Index	d	c
acker-schmalwand-all	5704	12627	1	438	7.76e-4	43	12	128
acker-schmalwand-pc	1907	2870	1	437	1.57e-3	22	9	84
Human-all	14771	67297	1	8649	6.16e-4	106	19	51
Worm-all	3613	6828	1	524	1.04e-3	35	10	73

1.3 Warum Haskell?

Haskell ist eine rein funktionale Programmiersprache mit statischer Typisierung. Das bedeutet, dass Haskell frei von Seiteneffekten ist. Es gibt also keine Zustandsänderungen, die das Ergebnis einer Funktion beeinflussen können. Bei gleichbleibenden Eingaben gibt es immer die selben Ergebnisse. Das verhindert eine Reihe von schwer aufzufindenen Bugs, die durch Zustandsänderungen entstehen und ermöglicht es dem Compiler Optimierungen zu tätigen, wie beispielsweise das Zwischenspeichern von Ergebnissen. Da pure Berechnungen sich nicht gegenseitig beeinflussen können, ist die Reihenfolge der Berechnungen egal, was Nebenläufigkeit und Parallelisierung begünstigt und vereinfacht. Es ermöglicht auch die Evaluierung von Ausdrücken „lazy“ stattfinden zu lassen. Das heisst, dass Ausdrücke erst evaluiert werden, wenn sie auch tatsächlich gebraucht werden. Haskell ist standardmäßig *lazy*. Das ermöglicht vielfach elegante Lösungen, die auf den ersten Blick ineffzient wirken. Ein Beispiel ist min = head . sort. Der Punkt steht für die Komposition von Funktionen, head liefert das erste Element einer Liste, sort sortiert eine Liste ($O(n \log n)$) in aufsteigender Reihenfolge. Die neue Funktion min liefert das kleinste Element einer Liste. Aufgrund der Lazy-Evaluierung hat sie eine Laufzeit von $O(n)$, es wird also nicht die gesamte Liste sortiert. Die Art der Evaluierung erschwert allerdings stellenweise die Analyse des Platzbedarfs von Algorithmen und fördert „space leaks“. *Space leaks* bezeichnen ein stetiges Anhäufen sogenannter *Thunks*, d.h. ausgelagerter, bzw. noch nicht stattgefundener Berechnungen, die den Speicher belasten. Dem kann entgegen gewirkt werden, indem man Ausdrücke explizit strikt evaluiert und somit die Bildung dieser Thunks verhindert. Mithilfe von Profilern und dem Anwenden einiger Grundregeln, bzw. gewonnener Intuition ist das gut in den Griff zu kriegen.
Haskell ist schnell [16], wird nativ compiliert und verfügt über eine Reihe guter und nützlicher Werkzeuge, vor allem für das Profiling und Testen von Programmen. Haskell benutzt „Garbage Collection“.
Ich habe Haskell gewählt, weil es die Sprache ist, in der ich mich derzeit am wohlsten fühle und in der ich am produktivsten bin. Wenn der Compiler den Code akzeptiert, funktioniert er in der Regel auch, was das „Refactoring“ großer Code-Stücke stark vereinfacht. Die Community ist sehr hilfsbereit und mittlerweile vergleichsweise groß. Es gibt gute Bücher und viele wissenschaftliche Papiere; was auch darin begründet ist, das Haskell primär eine akademische Programmiersprache ist. Die Praxistauglichkeit von Haskell konnte ich bereits im vergangenen Jahr in einem Projekt als Teil meiner Arbeit als studentische Hilfskraft unter Beweis stellen. Darüber hinaus sind einige Eigenschaften von Haskell für

das Problem von Nutzen: Der h-Index-Algorithmus (siehe Abschnitt 1.5) macht starken Gebrauch von dynamischer Programmierung und sowohl der h-Index-Algorithmus, als auch der Δ-Algorithmus (siehe Abschnitt 2) sind gut parallelisierbar (siehe Abschnitt 4).

1.4 Literaturüberblick

In den letzten Jahren gab es vermehrt Forschungsbemühungen bezüglich μ-Cliquen, was sicherlich auch dadurch begründet werden kann, dass mit dem Aufkeimen sozialer Netzwerke ein großes neues Anwendungsgebiet entstanden ist, die Bioinformatik immer mehr an Bedeutung gewinnt und die schiere Masse an Daten *Data Mining* in den Vordergrund rücken lässt.
Die Forschungsbemühungen beinhalten in der Regel exakte Algorithmen [15], Heuristiken und Reduktions-, bzw. Pruningregeln [4] zur Beschleunigung der Algorithmen. Einige Ideen und Ansätze sollen nun in aller Kürze vorgestellt werden.
Der exakte kombinatorische Algorithmus von Mahdavi Pajouh et al. [15] ist ein branch-and-bound Algorithmus für MAXIMUM QUASI-CLIQUE. Es wird eine Tiefensuche (engl.: depth-first-search) benutzt um schnell an eine erste mögliche Lösung zu kommen. Die „Branching-Regel“ besagt, das ein Knoten mit minimalem Grad gelöscht oder der Lösung hinzugefügt wird. Der Fall in dem der Knoten gelöscht wird, wird zuerst betrachtet. Die Schranken bestehen aus einer unteren und einer oberen Schranke. Die untere Schranke (engl.: lower bound) entspricht der bisher größten gefundenen μ-Clique, die obere Schranke (engl.: upper bound) der maximal zu erreichenden Kantendichte.
Patillo et al. [20] zeigen in ihrem Papier, das MAXIMUM QUASI-CLIQUE NP-vollständig ist und das die Eigenschaft Vererbung (engl: „heredity“), die in Algorithmen für MAXIMUM CLIQUE erfolgreich ausgenutzt werden konnte, nicht auf μ-Cliquen übertragbar ist. Es existiert jedoch die Eigenschaft der „Quasi-Vererbbarkeit“ (engl: quasi-heredity). Wenn ein Graph eine Dichte von mindestens μ hat, greift „quasi-inheritance“, sprich für jeden Graphen mit mindestens 2 Knoten, gibt es einen Knoten, ohne den der Graph immernoch eine μ-Clique ist. Dies ist möglich in dem der Knoten mit dem niedrigsten Grad entfernt wird. Diese Idee wird von uns im Abschnitt 2 aufgegriffen und auf verbundene μ-Cliquen übertragen. Der daraus resultierende Ansatz besteht darin, jeweils Knoten der Lösung hinzuzufügen, die die μ-Clique bewahren.
Abello et al. [4] befassen sich mit dem Finden von μ-Cliquen in großen dünnen Graphen, die u.U. nicht vollständig in den Arbeitsspeicher passen oder wo nur die Knotenmenge in den Arbeitsspeicher passt. Das macht es notwendig, das der Graph so schnell wie möglich abgebaut und zerlegt wird. Die Daten stammen aus Telekommunikations-Applikationen (wie Telefonanrufe), Internetdaten (z.B. URL Links) und geografischen Informationssystemen. Die Annahme ist, dass der zugrundeliegende ungerichtete Teilgraph sehr dünn ist und einen kleinen Durchmesser hat. Es ist dann möglich mit einer Breitensuche Zusammenhangskomponenten zu identifizieren. Damit gefundene maximale 1-Cliquen können dann als Ausgangspunkt (Seeds) für die weitere Suche verwendet werden. Weiterhin können Knoten entfernt werden, die nicht zu einer besseren Lösung beitragen können (diese Knoten heißen dann „peelable“). Im Fall von $\mu = 1$ sind das alle Knoten $v \in V$

mit $\deg(v) < k$. Da diese Löschung den Grad der anderen Knoten beeinflusst, wird sie rekursiv fortgesetzt. Wenn $\mu < 1$ ist und es bekannt ist, das eine μ-Clique existiert mit mindestens k Knoten, können alle Knoten $v \in V$ entfernt werden, aufsteigend im Grad sortiert, mit $\deg(v) < \mu \cdot k$ und $nv \in N(v).\deg(nv) < \mu \cdot k$, da diese nicht zu einer besseren Lösung beitragen können. Diese Knoten werden als μk-peelable bezeichnet. Sie stellen mit „greedy randomized adaptive search procedure“ (GRASP) noch einen Suchansatz vor. Dabei wird die Suche von mehreren Knoten aus gestartet. Es wird dann in der Konstruktionsphase eine mögliche Lösung produziert, die im Anschluss durch eine lokale Suche optimiert wird. Die Lösung wird Schritt für Schritt zusammengesetzt. In jedem Schritt gibt es eine Liste von Knoten die der Lösung hinzugefügt werden können. Daraus wird per Zufall ein Knoten gewählt.
Diese Arbeit bezieht sich auf das Papier [13] von Komusiewicz und Sorge. Sie untersuchen die Komplexität von Densest-k-Subgraph (DkS): Gegeben ein Graph G und eine Ganzzahl k wird ein Teilgraph mit exakt k Knoten gesucht, dessen Kantenanzahl maximal ist. Und zeigen, dass das Problem „fixed-parameter tractable“, bzgl. der Parameter Maximalgrad Δ und h-Index, ist, wenn eine konstante Mindestdichte μ gefordert wird. Ein Problem ist *fixed-parameter tractable*, wenn der exponentielle Laufzeitanteil auf ein Parameter beschränkt werden kann. Das daraus resultierende Problem ist das bereits beschriebene μ-Clique. Diese Arbeit implementiert zwei der von Komusiewicz und Sorge beschriebenen Suchbaumalgorithmen. Der exponentielle Laufzeitanteil der Algorithmen hängt einmal vom Maximalgrad Δ und einmal vom h-Index ab. Das Hauptaugenmerk liegt auf dem Δ-Algorithmus.

1.5 h-Index-Algorithmus

Wir beschreiben zunächst den h-Index-Algorithmus von Komusiewicz und Sorge [13] zum Lösen des Entscheidungsproblems für μ-Cliquen. Der exponentielle Laufzeitanteil ist dabei alleinig vom h-Index abhängig. Der *Hirsch-Index* (kurz: h-Index) ist ein Maß für den Einfluss und die Relevanz wissenschaftlicher Papiere, Zeitschriften und Institutionen und dient dazu, die wissenschaftliche Leistung eines Forschers zu quantifizieren [10].
In der Graphtheorie dient der h-Index als ein Dichtemaß für Graphen. Ein Graph hat einen *h-Index* von h, wenn es h Knoten gibt, die mindestens h Nachbarn haben und die verbleibenden $|V| - h$ Knoten maximal h Nachbarn haben [8]. Der h-Index bietet sich für parametrisierte Algorithmen an, da die meisten in der Praxis interessanten, bzw. relevanten Graphen einen h-Index haben, der deutlich kleiner als der Maximalgrad Δ ist. Abgesehen von *regulären Graphen*, das heisst Graphen, für die $\delta = \Delta$ gilt $h < \Delta$.
Es soll nun kurz die Funktionsweise des Algorithmus erläutert werden. Die Annahme ist, dass es eine μ-Clique der Größe k gibt, die durch die Knotenmenge S induziert wird. Die μ-Clique setzt sich zusammen aus verschiedenen verbundenen Teilgraphen und einer Teilmenge H' der h-Index Menge (H). Die *h-Index Menge* ist die Menge der h Knoten, die einen Grad von mindestens h haben. Sie bildet den Ausgangspunkt des Algorithmus. Das Element $H_S = H \cap S$ erhält man durch Probieren aller Möglichkeiten. Um Teilgraphen nicht doppelt zu zählen, werden sie farblich markiert. Dazu werden die Knoten $v \in V \setminus H$

Algorithm 1 BYHINDEX

Data: Graph $G = (V, E)$, Integer k, Double μ
Result: Dichte der gefundenen μ-Clique oder $\perp$, wenn es keine μ-Clique gibt

function BYHINDEX(G, k, μ)
  bilde H mit Knoten der h-Index Menge
  for all $H_S \in \mathcal{P}(H)$ **do**
    bilde C bestehend aus $k - |H_S|$ Farben
    färbe die Knoten in $G[V \setminus H]$ zufällig mit Farben aus C
    for all $C' \in \mathcal{P}(C)$ **do** ▷ bilde Tabelle D: Farbmenge → Anzahl Kanten
      Eintrag in D für C' und Kantenanzahl der größten verbundenen μ-Clique, die genau die Farben aus C' hat; Kanten zu H_S werden berücksichtigt
    end for
    for all $C' \in \mathcal{P}(C)$ **do** ▷ bilde Tabelle T: Farbmenge → Anzahl Kanten
      Eintrag in T für C': $max\{D(C'), \underset{C'' \subset C'}{max}\{T(C'') + T(C' \setminus C'')\}\}$
    end for
    kanten ← Anzahl Kanten in H_S + T(C)
    dichte ← $\frac{2 \cdot kanten}{k(k-1)}$
    if *dichte* $\geq \mu$ **then**
      return *dichte*
    end if
  end for
  return $\perp$
end function

mit $k - |H_S|$ Farben zufällig eingefärbt. Sei C die Menge dieser Farben. Es wird für jedes Element C' der Potenzmenge $P(C)$ der Farbmenge C, die größtmögliche μ-Clique gebildet und das Ergebnis in einer Tabelle D abgelegt. Der Algorithmus greift auf einen leicht angepassten Δ-Algorithmus zurück, der die größte „farbenfrohe“ μ-Clique liefert. D.h. für jede Farbe der korrespondierenden Farbteilmenge C' gibt es maximal einen Knoten in der μ-Clique mit dieser Farbe. Um Mehrfachberechnungen zu verhindern, wird *dynamische Programmierung* verwendet. Dafür werden Zwischenergebnisse in einer Tabelle abgespeichert, auf die später zurückgegriffen werden können. Die eigentliche Lösung wird schrittweise zusammengesetzt. Das findet mithilfe der Tabelle T statt.
Aufgrund der Konstruktion des Algorithmus kann nun nicht mehr garantiert werden, dass die gefundene μ-Clique verbunden ist.
Der Implementierung zugrunde liegende Pseudocode ist in Algorithmus 1 zu sehen. Die experimentelle Auswertung ist in Kapitel 4 zu finden, dort wird auch eine mögliche Parallelisierung für den h-Index-Algorithmus vorgestellt.
Im nächsten Abschnitt beschäftigen wir uns mit der Quasi-Vererbbarkeit.

2 Eigenschaft der Quasi-Vererbbarkeit für μ-Cliquen

Als *vererbbar* (engl. hereditary) bezeichnet man Grapheigenschaften, die auch für alle induzierten Teilgraphen gelten. Für MAXIMUM-CLIQUE konnte erfolgreich ausgenutzt werden, dass die Eigenschaft „Clique sein“ vererbbar ist, um schnellere und effektivere Algorithmen zu entwerfen [11]. Diese Eigenschaft fehlt μ-Cliquen mit $\mu < 1$, da die Teilmengen einer μ-Clique nicht zwangsläufig wieder μ-Cliquen sind [14]. Interessanterweise ist es aber möglich die Cliquendefinition für μ-Cliquen aufzuweichen: Eine Grapheigenschaft Π heisst *quasi-vererbbar*, wenn es für jeden Graph $G = (V, E)$ mit Eigenschaft Π einen Knoten $v \in V$ gibt, sodass der induzierte Teilgraph $G[V \setminus \{v\}]$ ebenfalls die Eigenschaft Π besitzt.
Wir führen nun ein Lemma ein, das wir für die *Quasi-Vererbbarkeit* von μ-Cliquen brauchen. Das folgende Lemma ist eine Erweiterung von Lemma 2 aus [20].

Lemma 1. *Sei $G = (V, E)$ ein Graph der Dichte μ und $v \in V$ mit $|V| > 2$. Der Graph $G[V \setminus \{v\}]$ hat Dichte mindestens μ, genau dann wenn* $\deg(v) \leq \frac{2|E|}{|V|}$.

Beweis. Sei μ die Dichte des Ausgangsgraphen $G[V]$ und μ' die Dichte des Graphen $G[V \setminus \{v\}]$. Dann erhalten wir folgende Äquivalenz.

$$
\begin{aligned}
& \mu \leq \mu' \\
\Leftrightarrow \quad & \frac{2|E|}{|V|(|V|-1)} \leq \frac{2(|E| - \deg(v))}{(|V|-1)(|V|-2)} \\
\Leftrightarrow \quad & \frac{2|E|}{|V|} \leq \frac{2(|E| - \deg(v))}{(|V|-2)} \\
\Leftrightarrow \quad & 2|E||V| - 4|E| \leq 2|E||V| - 2\deg(v)|V| \\
\Leftrightarrow \quad & -4|E| \leq -2\deg(v)|V| \\
\Leftrightarrow \quad & 2|E| \geq \deg(v)|V| \\
\Leftrightarrow \quad & \frac{2|E|}{|V|} \geq \deg(v)
\end{aligned}
$$

□

Demnach hat der Graph $G[V \setminus \{v\}]$ mindestens die Dichte μ genau dann wenn der Grad von v maximal so groß wie der Durchschnittsgrad $2|E|/|V|$ von G ist. Es folgt nun leicht, dass die Eigenschaft μ-Clique *quasi-vererbbar* ist, da man den Knoten mit

dem kleinsten Grad entfernen kann und nach Lemma 1 nicht unter eine Dichte von μ fällt. Das funktioniert aber nur, solange man sich nicht explizit für *verbundene* μ-Cliquen interessiert, d.h. μ-Cliquen, die aus genau einer Zusammenhangskomponente bestehen, wie es später bei unseren Algorithmen der Fall sein wird. Es kann sein, dass alle Knoten deren Grad unter dem Durchschnitt liegt, Cut-Knoten sind. Die Löschung dieser Knoten hätte zur Folge, dass sich die Dichte verringert oder der Zusammenhang zerstört wird. Wir nennen einen Knoten *löschbar*, wenn die Löschung des Knotens den Kantendurchschnitt des Graphs nicht senkt, das heisst dass der Grad des Knotens nicht über dem Durchschnittsgrad liegt.
Im Folgenden wird ein Gegenbeispiel konstruiert, das zeigt, dass verbundene μ-Cliquen zu sein mit $0 < \mu < 0{,}5$ im Allgemeinen nicht *quasi-vererbbar* ist.

Satz 1. $\forall \epsilon > 0\ .\ \exists \mu > 0 : 0{,}5 - \mu < \epsilon$ *und „verbundene* μ*-Clique“ ist nicht quasi-vererbbar.*
$\forall \epsilon > 0\ .\ \exists \mu > 0 : \mu < \epsilon$ *und „verbundene* μ*-Clique“ ist nicht quasi-vererbbar.*

Beweis. Wir geben eine Konstruktion für einen Graphen mit zwei Parametern p und k an, sodass die Dichte des Graphen $\mathrm{f}(k, p)$ ist, für eine unten definierte Funktion f. Wir zeigen, dass man aus dem konstruierten Graphen keinen Knoten löschen kann ohne, dass der Graph in mindestens zwei Zusammenhangskomponenten zerfällt oder die Dichte unter μ sinkt, wenn man p und k genügend groß wählt. Weiterhin zeigen wir, dass für große Werte von p oder k die Dichte $\mathrm{f}(p, k)$ beliebig nah an 0 oder 0,5 herankommt.
In Frage kommen nur Graphen die einen Cut-Knoten haben, der den niedrigsten Grad im Graphen hat und in denen die verbleibenden Knoten einen Mindestgrad haben, der über dem Durchschnitt liegt; sonst greift o.g. Lemma 1 und man kann einfach den Knoten mit dem niedrigsten Grad löschen um wieder eine μ-Clique zu erhalten. Das erreichen wir durch zwei gleichgroße Cliquen mit je k Knoten, die über einen Pfad der Länge p verbunden sind (siehe Abbildung 2.1). Die Anzahl der Kanten setzt sich zusammen aus $\binom{k}{2}$ Kanten je Clique und $p+1$ Kanten auf dem Pfad. Die Dichte der μ-Clique berechnet sich dann wie folgt:

$$\mathrm{f}(k, p) = \frac{2(k^2 - k + 1 + p)}{(2k + p)(2k + p - 1)}.$$

Sei P die Menge der Knoten auf dem Pfad, mit $|P| \geq 1$ und $\forall v \in P : \deg(v) = 2$. Jeder Knoten auf dem Pfad ist ein Cut-Knoten. Damit die Cut-Knoten den niedrigsten Grad im Graphen haben, muss gelten: $\forall v \in V \setminus P : \deg(v) > 2 \Rightarrow \deg(v) \geq 3$. Das heisst diese Eigenschaft ist erfüllt für alle $k \geq 4$. Die Knoten in den Cliquen haben offensichtlich einen Grad von mindestens $k - 1$. Dieser Grad muss über dem Durchschnittsgrad $\frac{2|E|}{|V|}$ liegen:

$$\frac{2|E|}{|V|} \Leftrightarrow \frac{2(k^2 - k + 1 + p)}{2k + p} < k - 1 \Leftrightarrow 2 + 3 \cdot p < k \cdot p \Leftrightarrow \frac{2}{p} + 3 < k$$

Demnach liegt für $k \geq 6$ der Grad der Knoten $v \in V \setminus P$ für beliebige Pfadlängen über dem Graphdurchschnitt und es sind nach Lemma 1 keine Knoten aus den Cliquen löschbar. Damit verbleiben zur Löschung nur die Knoten auf dem Pfad, die nach Konstruktion

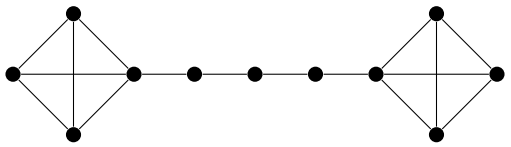


Abbildung 2.1: Eine verbundene 0,29-Clique, die nicht die Eigenschaft der *Quasi-Vererbbarkeit* besitzt.

einen Grad haben, der kleiner als der Durchschnittsgrad ist, wodurch der Graph in zwei Zusammenhangskomponenten zerfallen würde und keine „verbundene μ-Clique“ mehr vorliegt. Es folgt, dass keine Knoten aus der konstruierten μ-Clique löschbar sind falls $\mu = \mathrm{f}(k, p)$.

Indem wir p fixieren, ergibt sich dann

$$\lim_{k\to\infty} \mathrm{f}(k,3) = \lim_{k\to\infty} \frac{2\cdot(k^2-k+4)}{(2k+3)(2k+3-1)} = \lim_{k\to\infty} \frac{k^2\cdot(2-\frac{2}{k}+\frac{8}{k^2})}{k^2\cdot(4+\frac{10}{k}+\frac{6}{k^2})} = 0{,}5,$$

dass die maximale Dichte für einen auf diese Weise konstruierten Graphen gegen 0,5 konvergiert. Die Konstruktion muss monoton steigend sein, d.h. $\mathrm{f}'(k,3) > 0$.

$$\mathrm{f}'(k,3) = \left(\frac{2\cdot(k^2-k+4)}{(2k+3)(2k+3-1)}\right)' = \frac{28k^2-40k+68}{(4k^2+10k+6)^2} > 0$$

Nach Voraussetzung ist $k \geq 4$ und damit $\mathrm{f}(k,3)$ streng monoton steigend.
Wir haben gezeigt, dass wir mithilfe unserer Konstruktion in der Lage sind, Graphen zu konstruieren die eine Dichte haben, die beliebig nah an 0,5 rankommt.

Es bleibt zu zeigen, dass auch Graphen konstruierbar sind, deren Dichte gegen 0 tendiert. Dazu fixieren wir k:

$$\lim_{p\to\infty} \mathrm{f}(4,p) = \lim_{p\to\infty} \frac{2\cdot(13+p)}{(8+p)(7+p)} = \lim_{p\to\infty} \frac{p^2\cdot(\frac{26}{p^2}+\frac{2}{p})}{p^2\cdot(\frac{56}{p^2}+\frac{15}{p}+1)} = 0.$$

Hierfür konvergiert die Dichte gegen 0. Es lassen sich also Graphen konstruieren mit $0 < \mu < 0{,}5$ und „verbundene μ-Clique“ ist nicht *quasi-vererbbar*. □

Wir beweisen im Folgenden, dass die Eigenschaft der *Quasi-Vererbbarkeit* für verbundene μ-Cliquen mit $0{,}5 \leq \mu \leq 1$ gilt.

Satz 2. *Die Eigenschaft „verbundene μ-Clique“ ist quasi-vererbbar, sofern* $0{,}5 \leq \mu \leq 1$.

Beweis. Wir zeigen: Für jede μ-Clique mit $0{,}5 \leq \mu \leq 1$ gibt es einen löschbaren Knoten v, d.h. es gilt $\deg(v) \leq \frac{2|E|}{|V|}$ und v ist kein Cut-Knoten. Sei also im folgenden G eine verbundene μ-Clique. Wir betrachten zwei unterschiedliche Fälle:
Fall 1: Es existiert kein Cut-Knoten, dann lösche einen Knoten mit dem niedrigsten Grad.

Nach Lemma 1 hat der entstehende Graph Dichte mindestens μ. Nach der Voraussetzung des Falls ist er zudem verbunden.
Fall 2: Es gibt einen Cut-Knoten v und $G[V \setminus \{v\}]$ hat mehrere Zusammenhangskomponenten $V_1, \ldots, V_e, e \geq 2$, wobei die kleinste Zusammenhangskomponente, sei diese V_1, höchstens $\frac{n}{2} - 1$ Knoten hat, mit $n = |V|$. Das entspricht gleichzeitig dem maximal möglichen Grad von Knoten in V_1. Der durchschnittliche Knotengrad in G ist

$$\frac{2E}{n} = \mu \cdot (n-1).$$

Knoten sind löschbar, solange sie den Kantendurchschnitt nicht verringern. Daraus folgt für alle $\mu \geq 0{,}5$ mit

$$\frac{n}{2} - 1 < \frac{n-1}{2} \leq \mu \cdot (n-1),$$

dass $\forall v \in V_1 . \deg(v) < \mu \cdot (n-1)$ und somit da nicht alle Knoten in V_1 Cut-Knoten sein können, dass ein löschbarer Knoten in V_1 existiert, der kein Cut-Knoten ist. □

Die Eigenschaft der *Quasi-Vererbbarkeit* könnte benutzt werden, um im aktuellen Suchbaumknoten abzubrechen, sobald keine μ-Clique mehr vorliegt.

3 Algorithm Engineering für den Δ-Algorithmus

Der Δ-Algorithmus von Komusiewicz und Sorge [13] ist ein exakter Algorithmus, mit exponentieller Laufzeit, zum Lösen des μ-Cliquen-Entscheidungsproblems für verbundene μ-Cliquen. Dabei hängt der exponentielle Laufzeitanteil nur vom Maximalgrad Δ ab.
Im Folgenden werden die Funktionsweise (3.1) des Algorithmus, sowie mögliche Optimierungen (3.2) beschrieben und evaluiert.

3.1 Funktionsweise

Gegeben ein Graph $G = (V, E)$, eine Ganzzahl k (Lösungsgröße) und eine Mindestdichte μ ist die Idee: Um Entscheiden zu können, ob eine μ-Clique der Größe k existiert, können alle verbundenen Teilgraphen der Größe k aufgezählt werden. Sobald die Dichte eines Teilgraphs die Mindestdichte μ übersteigt, haben wir unsere μ-Clique gefunden und es wird abgebrochen. Die Aufzählung geschieht ausgehend von jedem Knoten $v \in V$. Wir nennen v den *Pivotknoten*. Um doppeltes Betrachten von Teilgraphen zu verhindern, werden Knoten, die bereits als Pivotknoten gedient haben aus dem Graph entfernt; sie können dann nicht Teil einer μ-Clique, und somit der Lösung, sein.
In Algorithmus 2 ist der Pseudocode des Δ-Algorithmus zu finden. Die abgebildete Funktion wird mit jedem Knoten $v \in V$ aufgerufen bis eine μ-Clique gefunden worden ist oder alle Knoten abgearbeitet worden sind. Trifft letzteres zu, existiert keine verbundene μ-Clique mit k Knoten. Der aktive Knoten wird aus P gewählt. Der erste aktive Knoten ist der Pivot-Knoten. Die Pivotmenge P wird dann Schritt für Schritt mit Knoten aus der Nachbarschaft des aktiven Knotens erweitert. Dazu verzweigen wir in alle Möglichkeiten einen Nachbarn hinzuzunehmen (siehe Zeile 16 in Algorithmus 2). Bis auf im ersten Fall gibt es maximal $\Delta - 1$ Nachbarn, weil der aktive Knoten P entstammt und somit bereits einen Nachbarn in P hat. Das stellt sicher, dass das Ergebnis verbunden ist. Um nicht nur Teilgraphen mit einem maximalen Durchmesser von 2 zu betrachten, wird eine weitere Verzweigung hinzugefügt. In diesem Zweig wird der aktive Knoten inaktiv gesetzt (siehe Zeile 24) und der Menge X hinzugefügt. X beinhaltet demnach Knoten die bereits aktiv waren, es gilt $X \subseteq P$. Wir wählen den neuen aktiven Knoten aus $P \setminus X$ (siehe Zeile 9-15). Die maximale Größe der Mengen P und X ist k. Angenommen wir unterscheiden nur in die Fälle in denen wir entweder P oder X modifizieren, dann kann die Anzahl der benötigten Suchbaumknoten mit 2^{2k} abgeschätzt werden. Wir haben weiterhin maximal $(\Delta - 1)^k$ Möglichkeiten Nachbarn hinzuzufügen. Jeder einzelne Schritt kann in $O(n + m)$ Zeit abgearbeitet werden; dass entspricht dem Aufwand der Schnitt-

und Differenzmengenbildung (z.B. Zeile 16 und 17). Es ergibt sich schlußendlich eine Gesamtlaufzeit von $O(2^{2k} \cdot (\Delta - 1)^k \cdot (n + m))$.

Algorithm 2 BYDELTA

Data: Graph $G = (V, E)$, Integer k, Double μ, Pivot-Set P, Set ehemals aktiver Knoten X, aktiver Knoten a

Result: Dichte der gefundenen μ-Clique oder $\perp$, wenn es keine gibt

```
1:  function BYDELTA(G, k, μ, P, X, a)
2:      d ← DICHTE(G, P)   ▷ berechnet Dichte: 2|E_P|/|P|(|P| − 1), E_P = Kanten in P
3:      if |P| = k ∧ d ≥ μ then
4:          return d
5:      else if |P| = k ∧ d < μ then
6:          return ⊥
7:      end if
8:                  ▷ an dieser Stelle kommen die Schranken zur Anwendung (siehe 3.2.2)
9:      if a = ⊥ then
10:         if P \ X = ∅ then
11:             return ⊥
12:         else
13:             ersten Knoten aus P \ X nehmen und als neuen aktiven Knoten a setzen
14:         end if
15:     end if
16:     for all v ∈ N(a) \ P do
17:         if N(v) ∩ X = ∅ then
18:             dichte ← BYDELTA(G, k, μ, P ∪ {v}, X, a)
19:             if dichte ≠ ⊥ then
20:                 return dichte
21:             end if
22:         end if
23:     end for
24:     dichte ← BYDELTA(G, k, μ, P, X ∪ {a}, ⊥)
25:     if dichte ≠ ⊥ then
26:         return dichte
27:     end if
28: end function
```

3.2 Optimierungen und Beschleunigungen

Wir haben verschiedene Ansatzpunkte für Optimierungen und Beschleunigungen, die auf unterschiedliche Weise Eigenschaften von μ-Cliquen ausnutzen. Unsere Vorgehensweise besteht darin, schrittweise Stellschrauben hinzuzufügen.
Erster Ansatzpunkt ist die Graphzerlegung. Wir zerlegen zunächst den Eingabegraph in seine Zusammenhangskomponenten. Da wir uns für verbundene μ-Cliquen interessieren,

können wir die Zusammenhangskomponenten verwerfen, die eine Knotenanzahl haben, die unterhalb von k liegt. Über die Kantenanzahl kann genauso argumentiert werden: Damit eine μ-Clique existieren kann, muss der Graph eine Mindestkantenanzahl von

$$\frac{2 \cdot |E|}{k \cdot (k-1)} \geq \mu = |E| \geq \frac{\mu \cdot k \cdot (k-1)}{2}$$

haben.
Weiterhin konzentrieren wir uns zunächst darauf, ob eine μ-Clique der Größe $\geq k$ existiert und nicht darauf, ob diese zudem größtmöglich ist. Aus diesem Grund kann man vor dem Ausführen des eigentlichen Algorithmus eine Heuristik (siehe Abschnitt 3.2.3) vorschalten, die, sofern sie ein Ergebnis liefert, das tatsächliche Ausführen des Suchbaumalgorithmus überflüssig macht. Die Heuristik wird daher als untere Schranke für die Lösungsgröße k benutzt.
Das Pendant dazu, die oberen Schranken, können wir direkt im Algorithmus anwenden. Die Schranken ermöglichen es uns ganze Berechnungsbäume wegzuschneiden. Sie werden im Abschnitt 3.2.2 vorgestellt.
Aber zuvor gibt es noch eine weitere Stellschraube. Der Algorithmus wird mit einem Pivot-Knoten gestartet von dem ausgehend Teilgraphen der Größe $\leq k$ betrachtet werden (siehe Algorithmus 2). Unser Ziel ist es mit geeigneten Pivot-Knoten zu starten, um möglichst schnell zu einer Lösung zu kommen.

3.2.1 Sortierung

Wir können an zwei Stellen im Algorithmus (s. Zeile 1 und Zeile 16 in Algorithmus 2) Einfluss auf die Reihenfolge der abzuarbeitenden Knoten nehmen. Ziel ist es, Knoten zu priorisieren, die schnell zu einer Lösung führen könnten.
Da die meisten Graphen die betrachtet werden *dünn* sind, existiert eine Großzahl an Knoten mit einem sehr geringen Grad, die, so die Annahme, eine eher untergeordnete Rolle in der Lösungsfindung spielen. Die Knoten werden deshalb im Grad absteigend vorsortiert. Bei Graphen die sehr dünn sind, kann es sich auch anbieten nach *Degeneracy-Ordnung* zu sortieren. Die Degeneracy-Ordnung ist eine Reihenfolge von Knoten, die wir erhalten, wenn wir solange einen Knoten mit dem niedrigsten Grad entfernen bis der Graph leer ist. Bei dieser Sortierung werden Knoten die einen hohen Grad haben, der aufgrund von vielen niedriggradigen Nachbarn zustande kommt, nach hinten versetzt. Für *reguläre* Graphen, d.h. Graphen für die $\delta = \Delta$ gilt, wird die Sortierung ausgesetzt. Wir benutzen den Algorithmus aus [6] zur Berechnung der Degeneracy und der Degeneracy-Ordnung.
Es ergeben sich eine Reihe von Konfigurationsmöglichkeiten.

- **einfach:** der reine Suchbaumalgorithmus ohne weitere Optimierungen
- **Grad** $\downarrow$: Knoten werden im Grad absteigend sortiert
- **degeneracy** $\downarrow$: Knoten werden entsprechend einer Degeneracy-Ordnung absteigend sortiert

Tabelle 3.1: Auswirkungen der Vorsortierung
∅ k : arith. Mittel über höchstes k der Instanz
∅ Suchbaumknoten: arith. Mittel über Anzahl d. benötigten Suchbaumknoten pro k pro Instanz
∅ Zeit (s): arith. Mittel über benötigte Zeit pro k pro Instanz
Zeitlimit 20 Minuten, $\mu = 0{,}7$

DIMACS Instanzen

Konfiguration	∅ k	∅ Suchbaumknoten	∅ Zeit (s)
einfach	60,8	61.745.170	75,77
Grad ↓	80,2	6.064.156	15,6
degeneracy ↓	59,6	10.536.592	24,83
score	80,2	6.064.156	16,22

biologische Instanzen

Konfiguration	∅ k	∅ Suchbaumknoten	∅ Zeit (s)
einfach	7	241.970.927	69,87
Grad ↓	8,5	96.307.580	76,43
degeneracy ↓	17,25	18.291.322	36,59
score	17,25	18.291.322	36,65

- **score:** sortiert je nach Graph nach Grad oder Degeneracy-Ordnung

Welche Sortierung bei *score* gewählt wird, ist abhängig von der Differenz zwischen der degeneracy des Graphen und dem Maximalgrad Δ. Eine hohe Diskrepanz begünstigt das Sortieren nach Degeneracy-Ordnung. Es wird nach Degeneracy-Ordnung sortiert, sofern die degeneracy des Graphs kleiner als $\frac{\Delta}{10}$ ist. Der Wert hat sich durch Probieren als günstig erwiesen. Die Sortierung nach score entspricht in der Regel bei den DIMACS Instanzen der Sortierung nach Grad und bei den biologischen Instanzen der Sortierung nach degeneracy. In Zeile 16 in Algorithmus 2 werden die Knoten zusätzlich noch mit der Anzahl an Nachbarn in P gewichtet. Ziel ist es, in Anlehnung an die Quasi-Vererbbarkeit, die Dichte der Pivot-Menge P hochzuhalten. Der damit erzielte Gewinn ist im Vergleich zu den anderen Sortierungen aber sehr gering. In Kombination mit den anderen Sortierungen benötigt man im Schnitt ca. 1% weniger Suchbaumknoten.
Die Auswirkungen der Vorsortierung auf DIMACS Instanzen und biologische Instanzen (s. Tabelle 3.1) werden nun kurz dargestellt. Wir betrachten nur Lösungsgrößen k, für die es eine Lösung gab. Man kann in Tabelle 3.1 sehen, dass die Anzahl der zu betrachtenden Suchbaumknoten durch die Vorsortierung im Falle von score um über 90% gesenkt werden konnte, bei einem gleichzeitigen Anstieg von k. Die Laufzeit hat sich für biologische Instanzen annähernd halbiert, für DIMACS Instanzen wurde nur noch ein Fünftel der Zeit benötigt. Auffällig ist, dass sich die Sortierung nach degeneracy negativ auf die Lösungsgröße der DIMACS Instanzen auswirkt. In Abbildung 3.2.1 ist das Verhalten der

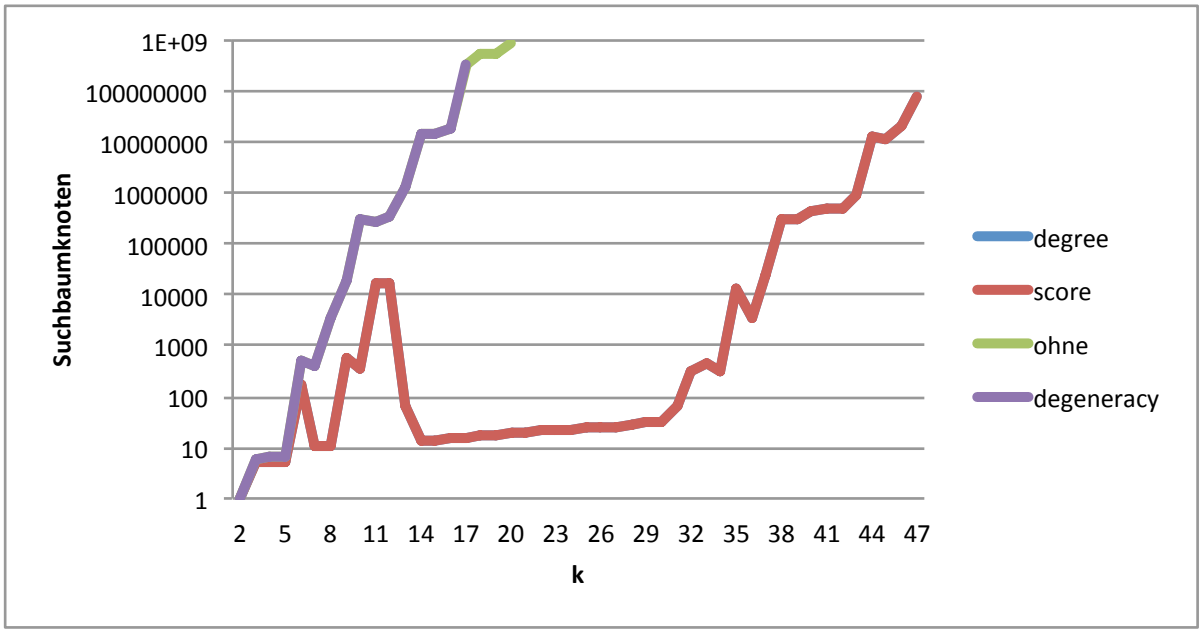


Abbildung 3.1: Sortierung bei „keller4“: Anzahl an Suchbaumknoten pro k

Sortierungen für die DIMACS Instanz „keller4“ zu sehen. Die Sortierung nach score und Grad liefern ein identisches Ergebnis. Interessant ist der Vergleich zwischen der Variante ohne Sortierung und der Sortierung nach degeneracy. Bis $k = 17$ liefern beide, was die Suchbaumknoten angeht, das selbe Ergebnis. Ohne Sortierung werden dann aber in der Folge weitere μ-Cliquen für größere Lösungsgrößen k gefunden. Bei „keller4“, wie bei vielen anderen DIMACS Instanzen auch, ist $d = \delta$. Das heisst, die Degeneracy d entspricht dem Minimalgrad. Sie ist demnach kein Unterscheidungsmerkmal. Die Knotenreihenfolge wird beibehalten. Da zum Sortieren *merge sort* benutzt wird, liegt die benötigte Zeit für den bereits sortierten, wie den unsortierten Fall, in $O(n \cdot \log n)$. Das schlechtere Ergebnis ist die Folge des erhöhten Zeitaufwands durch die Sortierung.
Bei den biologischen Instanzen ist zu beobachten, dass für kleine k die Sortierung nach Degree besser sein kann. Die Unterschiede sind aber zu vernachlässigen. Es handelt sich zumeist nur um wenige tausend Suchbaumknoten und 1-4 Sekunden Laufzeit. Umso stärker sich k der Degeneracy annähert, desto eindeutiger sind die Ergebnisse für die Sortierung nach Degeneracy. Für die Instanz „acker-schmallwand-all“ greift die Sortierung am Besten. Bis $k = 25$ werden pro k nur $k - 1$ Suchbaumknoten benötigt. Für $k = 26$ gibt es im Zeitlimit von 20 Minuten kein Ergebnis.
Wir konnten zeigen, dass die Vorsortierung zu deutlichen Verbesserungen führt und dass es sinnvoll sein kann, in Abhängigkeit der vorliegenden Graphen, verschiedene Strategien zu verfolgen. Wenn nicht anders gekennzeichnet, wird die Sortierung nach score nun immer benutzt.
Im nächsten Abschnitt werden die oberen Schranken vorgestellt.

3.2.2 Theoretische obere Schranken

Obere Schranken ermöglichen es vorzeitig festzustellen, ob der Teil des Suchbaums, in dem wir uns aktuell befinden, noch zu dem gewünschten oder zu einem besseren Ergebnis führen kann. Im Folgenden werden einige obere Schranken für die Anzahl der noch hinzufügbaren Kanten vorgestellt. Die Schranken können in jedem Suchbaumknoten angewendet werden (s. Zeile 8 in Algorithmus 2).

In jedem Aufruf von Algorithmus 2 suchen wir die Knotenmenge $S \supset P$ mit $k = |S|$ für die $G[S]$ eine μ-Clique ist, das heisst wir müssen noch $k' := k - |P|$ Knoten der Pivotmenge P hinzufügen. Damit $G[S]$ eine μ-Clique ist, muss für die Anzahl der Kanten in $G[S]$ gelten:

$$e_s \geq \frac{\mu \cdot k(k-1)}{2}.$$

D.h. wir brauchen mindestens

$$e' \geq \frac{\mu \cdot k(k-1)}{2} - e_p$$

weitere Kanten; e_p entspricht der Anzahl an Kanten in $G[P]$.

Die einfachste Schranke ist auch gleichzeitig die am wenigsten scharfe Schranke. Die Graphstruktur oder die Charakteristika der Knoten der Pivotmenge haben keine Auswirkungen auf die Berechnung. Es wird angenommen, dass die Knoten, die noch hinzugefügt werden, eine Clique bilden und mit allen Knoten der Pivotmenge verbunden sind. Für $P = \emptyset$ entspricht die Abschätzung der Kantenanzahl einer Clique mit k Knoten. Die Schranke berechnet sich wie folgt:

$$\sum_{i=|P|}^{k-1} i. \tag{3.1}$$

Es ist möglich auf die Graphstruktur zurückzugreifen, um so die Schranken enger zu gestalten. Die Knoten der Pivotmenge lassen wir erneut außen vor. Bisher haben wir angenommen, dass die Knoten, die noch hinzugefügt werden können mit allen Knoten der Pivotmenge P verbunden sind. Für $\Delta < |P|$ ist diese Abschätzung zu großzügig. Die erste Verfeinerung wählt daher das Minimum aus Δ und $|P|$ zur Bestimmung der Kantenanzahl, die ein Knoten beitragen kann:

$$\sum_{i=|P|}^{k-1} \min\{\Delta, i\}. \tag{3.2}$$

Da wir es selten mit regulären Graphen zu tun haben, d.h. Graphen in denen alle Knoten den selben Grad haben, kann man die Schranke möglicherweise einengen, in dem man die größten k' verbleibenden Knotengrade betrachtet. Sei $D = (\Delta_1, \Delta_2, ..., \Delta_{k'})$ die Liste

der Knotengrade des Graphs in absteigender Reihenfolge sortiert. Dann lässt sich die Schranke unter Berücksichtigung von D wie folgt berechnen:

$$\sum_{i=|P|}^{k-1} \min\{\Delta_i, i\}. \tag{3.3}$$

Die Liste D kann fortwährend aktualisiert werden, in dem die Grade der Knoten die bereits P hinzugefügt worden sind oder die nicht Teil der Lösung sein können, entfernt werden. Somit können ggf. kleinere Knotengrade nachrücken, wodurch die Schranke im Verlauf schärfer wird.
Die nächsten beiden Schranken beziehen die Pivotmenge P in die Betrachtung mit ein. Da die Knoten in P als Teil der Lösung feststehen, wissen wir definitiv, wie viele Nachbarn die verbleibenden Knoten in P haben. Wir ermitteln deshalb die Anzahl an Nachbarn in P für Knoten $v \in V \setminus P : |N(v) \cap P|$. Sei $N_p = (n_p^1, n_p^2, ..., n_p^{k'})$ die entsprechende Liste in absteigender Reihenfolge sortiert. Für die Anzahl der Kanten, die die verbleibenden Knoten untereinander haben, gehen wir von Cliquen aus. Es ergibt sich dann:

$$\sum_{i=1}^{k'} n_p^i + \binom{k'}{2}. \tag{3.4}$$

Wie bei Schranke (3.2) kann es sich anbieten, nur die maximale Anzahl an Nachbarn n_p in P in die Berechnung einzubeziehen, um Rechenaufwand zu sparen:

$$k' \cdot n_p + \binom{k'}{2}. \tag{3.5}$$

Wir verbessern nun abschliessend die Abschätzung für den $\binom{k'}{2}$, bzw. $V \setminus P$ Teil der Schranke. Die letzte Schranke berechnet die maximale Anzahl an Kanten in einem Graph nach Turàn [5]. Die Idee ist, dass die verbleibenden Knoten $V \setminus P$ eingefärbt werden und die maximale Anzahl an möglichen Kanten über die entstehenden Partitionen berechnet werden.
Ein Graph ist χ-*färbbar*, für eine positive ganze Zahl χ, wenn die Knoten des Graphen mit χ Farben eingefärbt werden können, sodass kein Knoten zu einem anderen Knoten mit gleicher Färbung benachbart ist. Ein χ-färbbarer Graph enthält keine Clique mit $\chi + 1$ Knoten. Unter allen Graphen mit n Knoten, die keine Clique der Größe $\chi + 1$ besitzen, sind die Turàn-Graphen die dichtesten Graphen [5]. Die Größe der Partitionen entspricht in etwa n/χ, das heisst der Graph ist komplett χ-partit und jeder Knoten kann maximal n/χ Nachbarn in einer der anderen Partitionen haben. Es gibt insgesamt χ-Partitionen, daher $n \cdot (\chi - 1) \cdot n/\chi \cdot 1/2$ mögliche Kanten. Uns stehen k' Knoten für die Kantenbildung zur Verfügung, mit $k' := k - |P|$. Es ergibt sich dann für die Anzahl an Kanten:

$$\sum_{i=1}^{k'} n_p^i + (1 - \frac{1}{\chi}) \cdot \frac{k'^2}{2}. \tag{3.6}$$

Tabelle 3.2: Auswirkungen der oberen Schranken
∅ k : arith. Mittel über höchstes k der Instanz
∅ Suchbaumknoten: arith. Mittel über Anzahl d. benötigten Suchbaumknoten pro k pro Instanz
∅ Zeit (s): arith. Mittel über benötigte Zeit pro k pro Instanz
Zeitlimit 20 Minuten, $\mu = 0{,}7$

DIMACS Instanzen

Schranke	∅ k	∅ Suchbaumknoten	∅ Zeit (s)
keine	80,2	6.064.156	16,22
Schranke (3.1)	86,4	4.354.833	18,80
Schranke (3.2)	86,4	4.354.833	19,23
Schranke (3.3)	86,4	4.354.833	19,00
Schranke (3.4)	96,2	379.829	13,52
Schranke (3.5)	90,0	1.100.957	16,79
Schranke (3.6)	87,4	9.017	29,01

biologische Instanzen

Schranke	∅ k	∅ Suchbaumknoten	∅ Zeit (s)
keine	17,25	18.291.322	36,65
Schranke (3.1)	18,25	1.002.528	4,19
Schranke (3.2)	18,25	1.002.528	4,33
Schranke (3.3)	18,25	1.002.528	4,26
Schranke (3.4)	19,25	1.008.561	17,9
Schranke (3.5)	18,5	436.253	8,76
Schranke (3.6)	19,25	32	1,96

Die Kanten zwischen $G[P]$ und den verbleibenden Knoten $V \setminus P$ werden hier wie in Schranke (3.4) abgeschätzt.
Für das Färben wird eine „Greedy-Coloring-Heuristik“ benutzt. Die Knoten werden im Grad absteigend sortiert und dann der Reihe nach abgearbeitet. Für jeden Knoten wird die nächste freie Farbe gewählt. Farben sind frei, wenn kein Knoten aus der Nachbarschaft mit dieser Farbe eingefärbt ist. Sollten alle Farben belegt sein, wird eine neue Farbe hinzugefügt. Die Farben sind in der Reihenfolge ihrer Erstellung, von alt nach jung, sortiert.
Die Schranken wurden in Kombination mit den besten Ergebnissen bei der Vorsortierung getestet. Die Ergebnisse sind in Tabelle 3.2 zu sehen. Wir betrachten nur Lösungsgrößen k, für die der Algorithmus innerhalb des Zeitlimits terminierte. Wir konnten in der Regel Verbesserungen verzeichnen. Auch wenn die Zeit für die DIMACS Instanzen im Schnitt leicht anstieg, konnte die Anzahl der Suchbaumknoten teilweise deutlich reduziert werden. Die gefundenen Lösungsgrößen nahmen zu. Für die biologischen Instanzen sank die Zeit

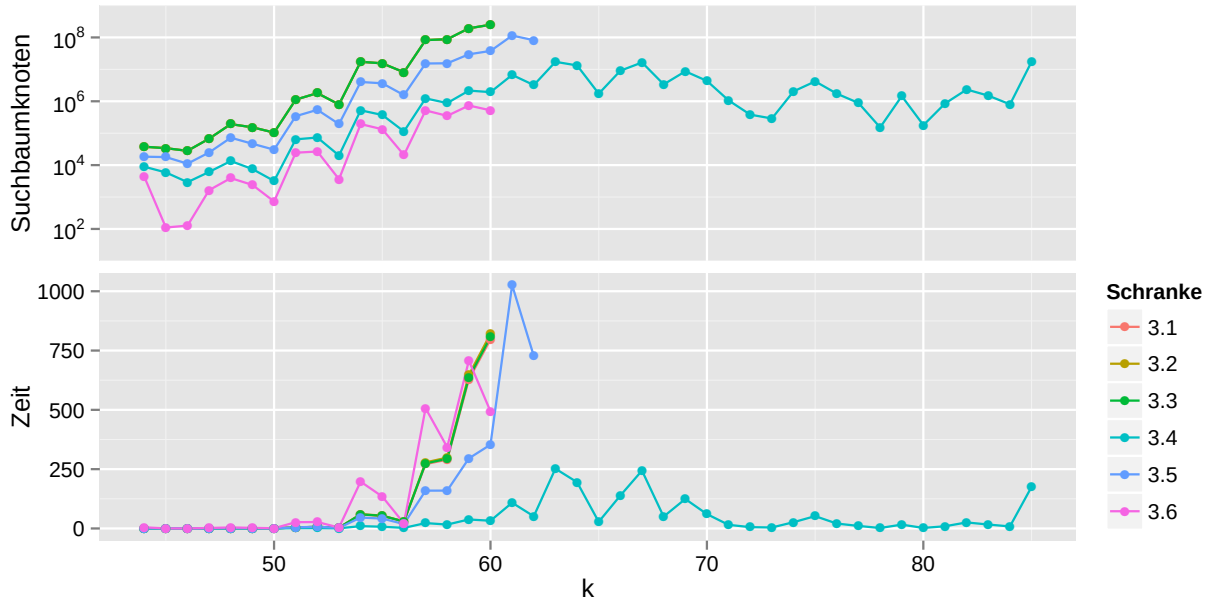
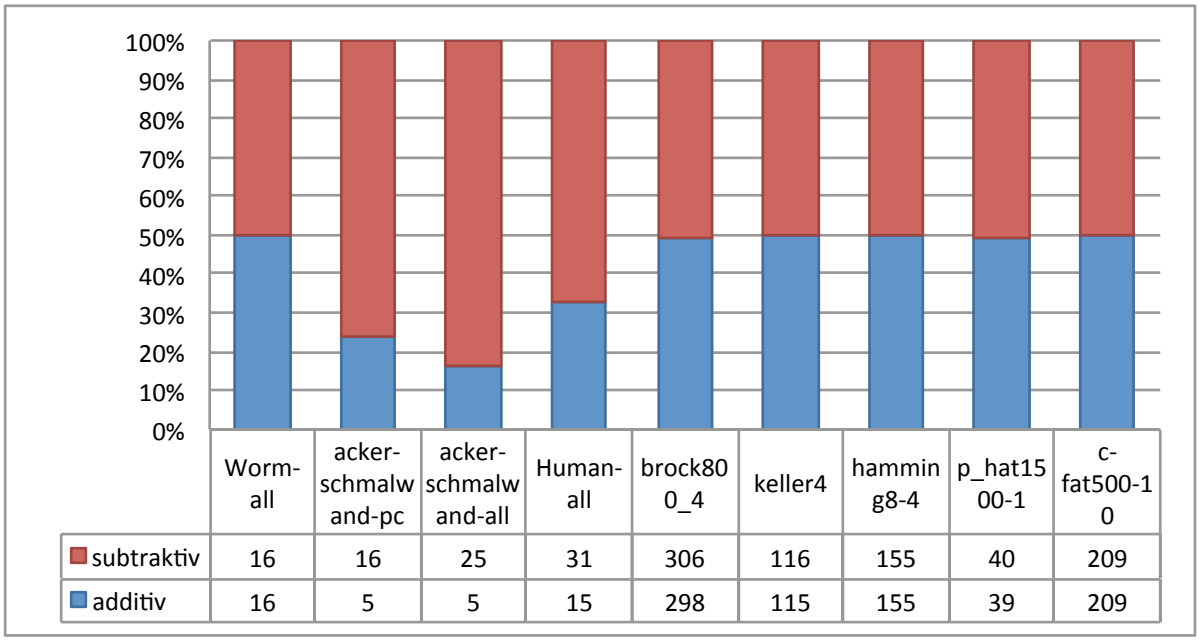


Abbildung 3.2: Schranken - Instanz „keller4“: Suchbaumknoten und Zeit (s) pro k

um bis zu 94%. Das mit Abstand beste Ergebnis, hinsichtlich der Suchbaumknoten, brachte die Schranke (3.6) ein, mit nur 32 Suchbaumknoten pro k. Für die DIMACS Instanzen fällt sie jedoch deutlich hinter der Schranke (3.4) zurück. Es werden zwar wiederum weniger Suchbaumknoten benötigt, allerdings wurden auch weitaus weniger μ-Cliquen im Zeitlimit gefunden. In Abbildung 3.2 wird die Zeit in Sekunden, die die Schranken pro k benötigt haben, für die Instanz „keller4“ dargestellt. Die Schranke (3.6) ist die teuerste Schranke, das heisst, für einen Suchbaumknoten wird im Vergleich zu den anderen Schranken mehr Zeit benötigt. Durch die Abschätzung der Kanten zwischen P und $V \setminus P$ braucht sie mindestens so lange wie Schranke (3.4). Hinzu kommt dann noch die Zeit für die Greedy-Coloring-Heuristik. Die Unterschiede zwischen Schranke (3.4) und (3.5) lassen sich damit erklären, dass die Hauptarbeit bei beiden Schranken identisch ist. Der eingesparte Rechenaufwand bei Schranke (3.5) kann die damit verbundene schlechtere Abschätzung der Kantenanzahl nicht kompensieren. Da die Maximalgrade der Testinstanzen im Vergleich zu k deutlich größer sind, greifen die Schranken (3.2) und (3.3) nicht. Damit verhalten sie sich genauso wie Schranke (3.1).
Es hat sich gezeigt, dass es auch hier wieder sinnvoll ist, verschiedene Strategien zu verfolgen. Für die DIMACS Instanzen ist die Schranke (3.4) die Beste, für die biologischen Instanzen die Schranken (3.4) und (3.6). Beide zusammen zu schalten ist wenig hilfreich, da die Schranke (3.6) im weitesten Sinne eine Verbesserung der Schranke (3.4) ist. Eine Kombination mit den verbleibenden Schranken würde also keine Verbesserung bringen. Das ist auch in Abbildung 3.2 zu sehen. Die Einsparungen bei Schranke (3.2) und (3.5) haben nicht den gewünschten Erfolg gebracht. Die Instanz „keller4“ ist repräsentativ, für das Verhalten der Schranken.
Im nächsten Abschnitt beschäftigen wir uns mit unserer letzten Stellschraube, die unteren



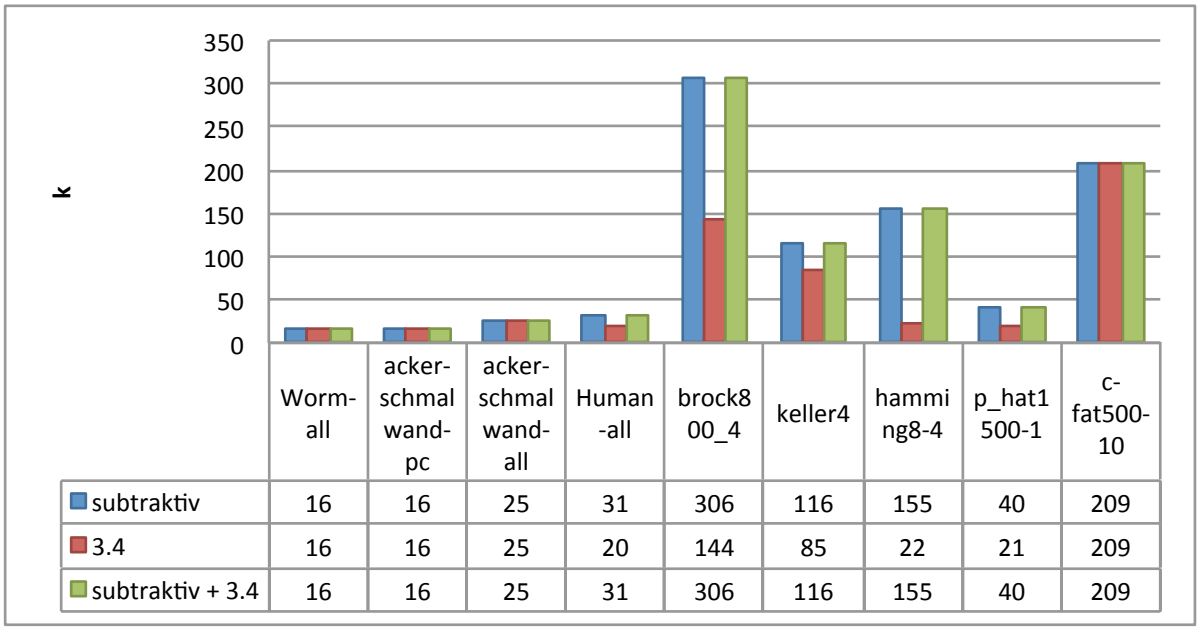
	Worm-all	acker-schmalwand-pc	acker-schmalwand-all	Human-all	brock800_4	keller4	hamming8-4	p_hat1500-1	c-fat500-10
subtraktiv	16	16	25	31	306	116	155	40	209
additiv	16	5	5	15	298	115	155	39	209

Abbildung 3.3: Heuristik Vergleich: Lösungsgröße k
100% entspricht der Summe beider Lösungsgrößen k
die Lösungsgröße entspricht dem größten k, für das eine μ-Clique gefunden worden ist

Schranken.

3.2.3 Untere Schranken

Wir setzen Heuristiken zur Problemlösung und Beschleunigung der Algorithmen ein. *Heuristiken* versuchen schnell zu einer möglichst guten Lösung zu kommen. Die Abweichung von der optimalen Lösung kann allerdings beliebig stark sein. Da wir das Entscheidungsproblem für μ-Cliquen lösen, akzeptieren wir auch Ergebnisse, die unter Umständen nicht optimal sind: Die Heuristiken können den Algorithmen vorgeschaltet werden. Liefern sie ein valides Ergebnis, das heisst sie finden eine μ-Clique der größe k, wird der Algorithmus gar nicht ausgeführt.
Unsere beiden Ansätze sollen nun kurz vorgestellt werden.
Die *additive Heuristik* fügt schrittweise Knoten zu einer Pivotmenge P hinzu. Sobald $|P| = k$ ist, gibt sie P aus, falls $G[P]$ dicht genug ist, ansonsten nichts. Zu Beginn wird der höchstgradige Knoten der Pivotmenge P hinzugefügt, er dient als Ausgangspunkt für das weitere Vorgehen. Es wird nun nach und nach die Pivotmenge mit einem Knoten $v \in V \setminus P$ erweitert, welcher die meisten Nachbarn in P hat, also die Kantenanzahl innerhalb der Pivotmenge P maximiert.
Unser zweiter Ansatz geht den entgegengesetzten Weg und baut den Graph Stück für Stück ab. Die *subtraktive Heuristik* löscht immer den Knoten mit dem niedrigsten Grad, bis nur noch k Knoten übrig sind. Da wir uns für verbundene μ-Cliquen interessieren,



	Worm-all	acker-schmalwand-pc	acker-schmalwand-all	Human-all	brock800_4	keller4	hamming8-4	p_hat1500-1	c-fat500-10
subtraktiv	16	16	25	31	306	116	155	40	209
3.4	16	16	25	20	144	85	22	21	209
subtraktiv + 3.4	16	16	25	31	306	116	155	40	209

Abbildung 3.4: Vergleich untere und obere Schranke: Lösungsgröße k

muss noch sichergestellt werden, dass der zu löschende Knoten kein Cut-Knoten ist. In diesem Fall wird der Knoten zunächst übergangen und sofern notwendig, ein Knoten mit einem höheren Grad probiert. Das Vorgehen orientiert sich an der Quasi-Vererbbarkeit. Um Zeit zu sparen, kann es sich lohnen diesen Schritt zu überspringen und erst am Ende zu prüfen, ob das vorliegende Ergebnis verbunden ist. Oftmals lösen sich zwischenzeitig entstehende Probleme von selbst. Die Lösungsgröße k kann bei beiden Ansätzen voneinander abweichen.
In Abbildung 3.3 sind die Heuristiken im direkten Vergleich hinsichtlich ihrer erreichten Lösungsgröße k zu sehen. Bei 50% hätten beide Heuristiken die selbe Lösungsgröße erreicht. Es hat sich gezeigt, dass der subtraktive Ansatz für biologische Instanzen deutlich bessere Ergebnisse liefert. Bei den DIMACS Instanzen ist die Situation etwas ausgeglichener. Die additive Heuristik benötigt im Schnitt weniger als 0,02 Sekunden pro k, für die subtraktive Heuristik liegt die Zeit bei 0,2 Sekunden für die DIMACS Instanzen und 1,4 Sekunden bei den biologischen Instanzen. Aufgrund des Ergebnisses werden wir uns nur noch mit dem subtraktiven Ansatz beschäftigen.
Abschließend vergleichen wir die bisher beste Kombination unserer Stellschrauben, bestehend aus der Sortierung nach score und der Schranke (3.4), mit der subtraktiven Heuristik. In Abbildung 3.4 sehen wir die Ergebnisse hinsichtlich der Lösungsgröße k.
Die untere Schranke liefert für die DIMACS Instanzen die deutlich besseren Ergebnisse. Besonders auffällig ist das Ergebnis für die Instanzen „brock800_4“ und „hamming8-4“. „hamming8-4“ ist ein regulärer Graph, dort wird die Sortierung nach score ausgesetzt. Beide Instanzen haben hohe Maximalgrade und sind relativ dicht; also eher schwere Graphen. Bei den biologischen Instanzen sind die Ergebnisse bzgl. der Lösungsgröße bei 3 von 4 Instanzen identisch. Nur bei der Instanz „Human-all“, der größte Graph der

biologischen Instanzen, ist die obere Schranke schlechter. Weiterhin ist zu sehen, dass in der Kombination aus unterer und oberer Schranke keine besseren Ergebnisse innerhalb des Zeitlimits erzielt werden konnten.

3.2.4 Fazit

Wir haben uns mit verschiedenen Stellschrauben beschäftigt und sie miteinander verglichen. Die tatsächliche Wahl der Stellschrauben ist am Ende abhängig von dem Eingabegraph und den weiteren Parametern. Die Ergebnisse auf unseren Testinstanzen legen Nahe, dass die Sortierung nach score grundsätzlich verwendet werden sollte, genauso wie die subtraktive Heuristik. Bei den oberen Schranken fällt die Wahl auf Schranke (3.4).
Ausgehend von dem reinen Suchbaumalgorithmus konnten wir die größte gefundene Lösungsgröße von im Schnitt 60,8 für DIMACS Instanzen auf 165,2 verbessern. Gleichzeitig wurden die im Schnitt benötigten Suchbaumknoten von 61745170 auf 561 und die Zeit von durchschnittlich 75,77 auf 0,22 Sekunden gesenkt. Bei den biologischen Instanzen hat sich die durchschnittliche Lösungsgröße mehr als verdreifacht, von 7 auf 22,25. Die Reduktion ist bei den Suchbaumknoten noch deutlicher. Brauchte man zuvor noch 241970927 bei durchschnittlich 69,87 Sekunden pro k, sind es nun noch 251971 und 1,65 Sekunden.
Die vorgestellte Quasi-Vererbbarkeit können wir nicht im Suchbaum anwenden. In Verbindung mit dem Δ-Algorithmus führt das zu einem inkorrekten Ergebnis. Wir sind zwar in der Lage deutlich schneller μ-Cliquen zu finden, aber aufgrund der Funktionsweise des Δ-Algorithmus wäre es nicht möglich festzustellen, ob es *keine* μ-Clique für eine Lösungsgröße k gibt. Das Problem besteht darin, dass der Algorithmus die Teilgraphen in einer bestimmten Reihenfolge aufbaut. Teilgraphen werden nicht doppelt betrachtet. Es kann also passieren, dass wir einen Knoten v der Pivotmenge P hinzufügen und die Dichte von $G[P]$ kurzzeitig unter μ fällt. In diesem Fall würde man mit Anwendung der Quasi-Vererbbarkeit abbrechen, auch wenn es die Möglichkeit gibt, in einem späteren Branching die Dichte wieder auf mindestens μ zu erhöhen. Das könnte dazu führen, dass wir unter Umständen existierende μ-Cliquen nicht finden.
Im letzten Kapitel schauen wir uns eine mögliche Parallelisierung für den Δ- und den h-Index-Algorithmus an.

4 Beschleunigung durch Parallelisierung

Die vorgestellten Algorithmen eignen sich hervorragend zur Parallelisierung. Wir haben jeweils große, voneinander unabhängige, Berechnungsstränge oder Teilbäume, die für sich genommen genug Arbeit schaffen, um den eventuell entstehenden Overhead durch Synchronisation oder Verwaltung nicht ins Gewicht fallen zu lassen.
Die gewählte Implementierungssprache Haskell ist für solche Zwecke bestens geeignet, da sie *pure* ist, d.h. seiteneffektfreies Programmieren und das Benutzen von persistenten Datenstrukturen forciert, sowie über äußerst leichtgewichtige Threads verfügt. Für Haskell stehen eine Reihe von verschiedenen Ansätzen für Nebenläufigkeit und Parallelisierung zur Verfügungm, wie z.B. deterministischer Parallelismus. Die Entscheidung fiel auf „Software Transactional Memory“ (STM), da wir ein Minimum an Synchronisation benötigen und es das vorzeitige Abbrechen der Berechnungen erheblich erleichtert.
Bei dem h-Index-Algorithmus gibt es viele Ansatzpunkte. Am Besten lassen sich das Erstellen und Füllen der Tabellen parallel ausführen. Wir wählen die „Lookup“-Tabellen T. Jede Tabelle steht für sich allein und wir haben keine Abhängigkeiten zu bereits erstellten, oder noch kommenden Tabellen. Das Modell sieht vor, dass es gemeinsame Speicherbereiche gibt, über die die Synchronisation stattfindet. Dafür wird eine „Queue“ an Jobs zur Verfügung gestellt, die das einzige für die Berechnung einer Tabelle T notwendige bereitstellt: Eine „Seed“-Knotenmenge, die der Potenzmenge von H entspringt. Diese „Queue“, bestehend aus Knotenmengen, wird fortwährend gefüllt und von „Worker“-Threads abgearbeitet.
Da wir ein Entscheidungsproblem lösen, reicht uns ein valides Ergebnis. Für ein solches Ergebnis wird ein Container bereit gestellt. Sobald dieser gefüllt worden ist, werden alle noch laufenden Berechnungen sofort abgebrochen und das Ergebnis zurückgeliefert.
Bei dem Δ-Algorithmus kann man im Prinzip genauso vorgehen, hier entspricht die „Seed“-Knotenmenge den Knoten des jeweiligen Graphs.

Wir vergleichen die unoptimierte Version des Δ-Algorithmus mit dem h-Index-Algorithmus. Es hat sich als sinnvoll herausgestellt den Allokationsbereich für den Garbage Collector zu vergrößern, wenn wir die parallele Variante ausführen. Das führt dazu, dass der Garbage Collector seltener ausgeführt werden muss; dadurch verschlechtert sich aber ggf. das Cache-Verhalten. Der Testrechner besitzt 4 Kerne mit jeweils 2 Threads pro Kern. Für jeden Kern stehen 10MB L3-Cache zur Verfügung. Der Allokationsbereich wurde daher von den standardmäßigen 512KB auf 10MB erhöht.
Der h-Index-Algorithmus wurde aufgrund der Zufallskomponente 3-mal ausgeführt, und jeweils die beste Wertung genommen. Je nach dem wie gut die Einfärbungen sind,

Tabelle 4.1: Auswirkungen der Parallelisierung
∅ k : arithmetische Mittel über höchstes k der Instanz
∅ Zeit (s): arith. Mittel von benötigter Zeit pro k pro Instanz
Zahl hinter dem Algorithmus entspricht der Anzahl an Threads
Zeitlimit 20 Minuten, $\mu = 0{,}7$

DIMACS Instanzen

Algorithmus	∅ k	∅ Zeit (s)
delta-1	60,8	127,62
delta-4	60,6	89,5
delta-8	60,2	74,24
h-Index-1	12	164,93
h-Index-4	13	191,58
h-Index-8	13	188,34

biologische Instanzen

Algorithmus	∅ k	∅ Zeit (s)
delta-1	7	233,39
delta-4	8,25	198,9
delta-8	9	156,8
h-Index-1	9	175,77
h-Index-4	10,25	171,8
h-Index-8	10	171,36

können die Ergebnisse sehr unterschiedlich ausfallen. Bei der Evaluierung der Instanz „acker-schmalwand-pc“ war beispielsweise folgendes zu beobachten:
Durchlauf 1: Lösung gefunden für $k = 12$ in 493.97 Sekunden
Durchlauf 2: Lösung gefunden für $k = 12$ in 31.56 Sekunden
Durchlauf 3: keine Lösung für $k = 12$ gefunden (Zeitlimit: 1200 Sekunden)
Das gewählte Beispiel fällt deutlich aus dem Rahmen. Bei anderen Instanzen waren die Geschwindigkeitsunterschiede in der Regel im Bereich Faktor 3 bis 5 anzusiedeln. Nichtsdestotrotz waren auch dort, im betrachteten Zeitraum, unterschiedliche Lösungsgrößen zu finden.
Die Parallelisierung des Δ-Algorithmus ermöglicht es von mehreren Knoten aus gleichzeitig zu starten, in der Hoffnung schneller zu einer Lösung zu kommen. Schlimmstenfalls brauchen alle Startknoten in etwa gleichlang oder der erste Startknoten ist ohnehin der schnellste, sodass die Mehrarbeit keine Zeitersparnis bringt und es durch den entstehenden Overhead oder architekturbedingt sogar zu einer Verlangsamung kommt.
In Tabelle 4.1 ist zu sehen, dass mit einer Steigerung der Threadanzahl die Ergebnisse in der Regel verbessert werden konnten. Bei den DIMACS Instanzen war „delta-4“ bei einer, „delta-8“ bei zwei Instanzen im Hinblick auf die Lösungsgröße schlechter als „delta-1“. Die betroffenen Instanzen konnte „delta-1“ gerade noch im Zeitlimit lösen. Die eingebüßte Zeit lässt sich durch den entstehenden Overhead und der zugrundeliegenden Architektur erklären. Die Architektur des Testrechners erlaubt es bei der Benutzung von einem Kern automatisch in den Turbomodus zu schalten, wodurch die Variante „delta-1“ effektiv mit 200MHz mehr arbeitet. Dazu kommt, dass der Garbage Collector von Haskell nicht *concurrent* ist. Wenn der Garbage Collector tätig wird, müssen alle Threads gestoppt werden. Die Phasen verlängern sich mit steigender Threadanzahl. Es entsteht weiterhin Overhead durch die Kommunikation und dem erhöhten *Scheduling*-Aufwand. Bei „delta-8“ kommt hinzu, dass sich 2 Threads einen Cache und eine Speicheranbindung teilen. In Abbildung 4 wird die Leistung exemplarisch anhand der Instanz „hamming8-4“ gezeigt.

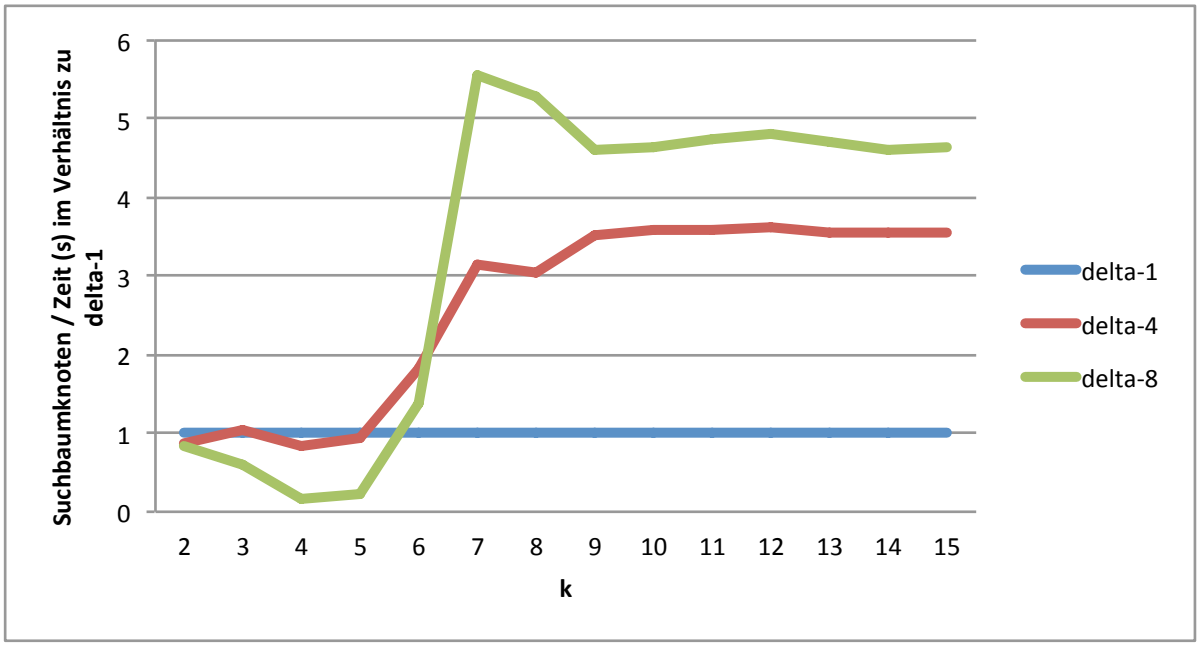


Abbildung 4.1: Anzahl an Suchbaumknoten pro Zeit (s) pro k im Verhältnis zu „delta-1“ für die Instanz „hamming8-4“

„hamming8-4“ bietet sich an, weil die Laufzeiten ab $k = 6$ etwas länger sind. Bis dahin ist der Aufwand zu gering; die Algorithmen benötigen weniger als 0.01 Sekunden. Wir sind in der Lage mit steigender Threadanzahl die geleistete Arbeit zu steigern. Der zusätzliche Gewinn durch 8 Threads fällt im Vergleich zu 4 Threads deutlich geringer aus. Das war aufgrund der Architektur auch zu erwarten. Nichtsdestrotrotz ist „delta-4“ mit ca. 3,5-facher Arbeit knapp unterhalb des theoretischen Maximums. In Tabelle 4.2 wurde das arithmetische Mittel der jeweiligen Leistung pro Instanz genommen. Im Schnitt fällt der Leistungszuwachs durch 8 Threads noch geringer aus.
Für die DIMACS Instanzen bleibt festzuhalten, dass der h-Index-Algorithmus deutlich schlechter ist. Das liegt primär an dem sehr großen h-Index der DIMACS Instanzen. Der Ansatz eignet sich vor allem für biologische Instanzen, wie in Tabelle 4.1 zu sehen ist. Die bessere durchschnittliche Lösungsgröße von „hindex-4“ kann der Zufallskomponente zugeschrieben werden.

Tabelle 4.2: ∅ K/Z: Anz. Suchbaumknoten pro Zeit (s) pro k im Verhältnis zu „delta-1“

DIMACS Instanzen	
Algorithmus	∅ K/Z
delta-1	1
delta-4	3,26
delta-8	4,25

biologische Instanzen	
Algorithmus	∅ K/Z
delta-1	1
delta-4	3,41
delta-8	4,1

5 Abschluss und Ausblick

Wir haben zwei Algorithmen zum Lösen des Entscheidungsproblems für μ-Cliquen vorgestellt und gezeigt, dass die Quasi-Vererbbarkeit auch auf verbundene μ-Cliquen übertragen werden kann; sofern $\mu \geq 0{,}5$ ist. Im Abschnitt 4 haben wir die Grundversionen der beiden Algorithmen, inklusive einer Parallelisierung, miteinander verglichen. Die Experimente haben gezeigt, dass der h-Index-Algorithmus auf den biologischen Instanzen konkurrenzfähig ist (s. Tabelle 4.1). Den Algorithmus haben wir bisher außen vor gelassen, d.h. es ist noch Spielraum für Verbesserungen. Das Hauptaugenmerk dieser Arbeit lag auf dem Δ-Algorithmus (s. Abschnitt 3). Mithilfe verschiedenster Stellschrauben konnte die Performance und das Ergebnis gegenüber dem einfachen Suchbaumalgorithmus deutlich verbessert werden. Die experimentelle Auswertung ergab, dass die beste Kombination der Stellschrauben aus der Vorsortierung nach score, der Schranke (3.4) und der subtraktiven Heuristik besteht.
Im Folgenden findet eine abschließende experimentelle Auswertung statt. Wir haben die beste Variante des Δ-Algorithmus mit $\mu = 0{,}5, \mu = 0{,}7$ und $\mu = 0{,}9$ auf allen Graphen (siehe Graphtabelle im Anhang) getestet. Das Zeitlimit beträgt 20 Minuten. Für die jeweiligen μ wurden Graphen weggelassen, deren Dichte über μ liegt, weil es sonst zu einfach ist. Die Ergebnisse sind in Tabelle 5.1 zu finden.
Das Bild, dass sich in den vorangegangenen Experimenten abgezeichnet hat, wird bestätigt. Die optimierte Version ist in allen Belangen deutlich überlegen. Vor allem bei den biologischen Instanzen, die von Hauptinteresse sind, können starke Verbesserungen verzeichnet werden. Beide Algorithmen sind in der Lage Instanzen zu lösen. Die beiden größten gelösten Instanzen sind „c-fat500-1“ und „acker-schmalwand-all.txt“ mit $k = 16$ für $\mu = 0{,}9$. Das heisst es konnte gezeigt werden, dass für die beiden Instanzen keine μ-Clique mit $k \geq 16$ und $\mu \geq 0{,}9$ existiert. Für „c-fat500-1“ hat der optimierte Δ-Algorithmus 232634194 Suchbaumknoten und 899,44 Sekunden gebraucht. Mit 4 Threads war die Instanz nach 275,33 Sekunden gelöst, mit 8 Threads nach 232,1 Sekunden. Die Ergebnisse der parallelisierten Version sind bzgl. der Lösungsgröße mit einer Ausnahme identisch zu der Single-Thread-Variante und wurden deshalb der Übersicht halber weggelassen. Die Ausnahme bildet die Instanz „p_hat700-1“. Für $\mu = 0{,}9$ konnte mit 8 Threads, im Zeitlimit, eine μ-Clique der Größe $k = 12$ gefunden werden; die anderen kamen im selben Zeitraum nur bis $k = 11$.
Wir haben im Abschnitt 1.4 einige Papiere vorgestellt. Die Vergleichbarkeit ist nicht ganz gewährleistet, aufgrund der unterschiedlichen Testrechner und des anders gearteten Problems: MAXIMUM QUASI-CLIQUE. Für $\mu = 1$ sollten wir aber zumindestens die Ergebnisse mit dem Branch-and-Bound Algorithmus von [15] vergleichen können. Der Branch-and-Bound Algorithmus wurde auf einem Intel Xeon E5430 mit 2.66GHz und 16GB RAM getestet; das Zeitlimit beträgt eine Stunde. Zum Vergleich: Unsere

Tabelle 5.1: abschließende Auswertung

∅ k : arith. Mittel über höchstes k der Instanz
∅ Knoten: arith. Mittel über Anzahl d. benötigten Suchbaumknoten pro k pro Instanz
∅ Zeit: arith. Mittel über benötigte Zeit [in Sekunden] pro k pro Instanz
✓ : Anzahl gelöster Instanzen

DIMACS Instanzen

	unoptimiert				optimiert			
μ	∅ k	∅ Knoten	∅ Zeit	✓	∅ k	∅ Knoten	∅ Zeit	✓
0,5	155,25	7.135.436	8,19	0	182,46	374	0,08	0
0,7	99,05	18.120.298	24,3	1	231,56	665.313	4,28	2
0,9	11,4	43.582.887	41,47	2	98,84	662.637	6,23	4

biologische Instanzen

	unoptimiert				optimiert			
μ	∅ k	∅ Knoten	∅ Zeit	✓	∅ k	∅ Knoten	∅ Zeit	✓
0,5	7,76	36.798.848	16,98	8	17	154.314	1,69	9
0,7	5,35	46.920.053	11,86	10	12,05	3.349	0,74	11
0,9	4,53	8.695.179	2,81	11	8,61	424.623	3,86	15

Erdős Graphen

	unoptimiert				optimiert			
μ	∅ k	∅ Knoten	∅ Zeit	✓	∅ k	∅ Knoten	∅ Zeit	✓
0,5	8,8	45.555.403	22,44	0	22,66	3.082	0,5	0
0,7	7,4	268.666.665	94,77	0	13	1.191.678	6,5	0
0,9	6,4	319.413.336	105,42	0	9,5	202.254	1,84	6

Experimente wurden auf einem 4-Kern Intel Xeon Prozessor mit 3.6GHz und 68GB Ram ausgeführt. Die getesteten Instanzen waren zumeist deutlich kleiner als unsere. Es wurden exemplarisch ein paar DIMACS Instanzen verglichen. Die Ergebnisse sind in Tabelle 5.2 abgebildet. Die Unterschiede in der Laufzeit sollten nicht alleine auf die Rechenleistung zurückzuführen sein. Abgesehen davon, dass die Laufzeit bei „c-fat200-2“ etwas länger ist, sind die Ergebnisse mindestens gleich gut. Bei „keller4“ ist die gefundene Lösungsgröße deutlich größer. Für $k = 11$ wurde die μ-Clique bereits nach 11 Sekunden gefunden.
Zum Schluss wird noch ein Ausblick gegeben.
Es ist noch Spielraum für Verbesserungen. Für den Δ-Algorithmus sollte versucht werden die Greedy-Coloring-Heuristik der Turàn-Bound (Schranke 3.6) zu beschleunigen. Ein Ansatz wäre die Knoten in umgekehrter degeneracy Reihenfolge abzuarbeiten. Ziel ist es, nicht mehr als $d + 1$ Farben zu haben. Da die Schranke sehr teuer ist, bestünde die Möglichkeit sie nicht in jedem Suchbaumknoten auszuführen. Dann würde sich ei-

Tabelle 5.2: Algorithmen Vergleich
Zeit (s): Zeit [in Sekunden] bis zum Terminieren, wenn „-“: nicht terminiert
k : größte gefundene μ-Clique

	Branch-and-Bound		Δ-Algorithmus	
Instanz	Zeit (s)	k	Zeit (s)	k
„p_hat300-1“	1772,37	8	926,7	8
„keller4“	-	9	-	11
„brock200_2“	-	12	-	12
„c-fat200-2“	2,07	24	29,85	24

ne Kombination mit der Schranke 3.4 anbieten. Auch wenn es nichtmehr aufgetreten ist, da die Sortierung nach Degeneracy für die DIMACS Instanzen nicht benutzt wird, sollte man den Fall das $\delta = d$ ist, abfangen, um nicht umsonst zu sortieren. Die vielen Sortierungsschritte kann man eventuell umgehen, in dem man geordnete Adjazenzlisten benutzt.
Bei der Parallelisierung gibt es einige Verbesserungsmöglichkeiten. Vieles ist durch die Werte, die man für die Analyse sammelt komplizierter. Ohne diese wäre es möglich den Δ-Algorithmus völlig ohne Seiteneffekte auskommen zu lassen, was ggf. zu besseren Optimierungen seitens des Compilers führt. Wenn man sich auf den Fall spezialisiert, dass die Threadanzahl gleich der Anzahl der Kerne ist, kann man die OS Threads direkt einem Kern zuteilen, um die Ressourcen besser ausnutzen zu können und ein Umverteilen zu unterbinden. Ein anderer Ansatz der Parallelisierung für den Δ-Algorithmus wäre auch denkbar. Wenn wir davon ausgehen, dass wir durch unsere Vorauswahl mit dem besten Pivotknoten starten, ist es sinnvoll, diesen möglichst schnell abzuarbeiten. Das heisst man würde den Algorithmus parallel auf seine Nachbarn ansetzen. Beim h-Index-Algorithmus könnte man deterministischen Parallelismus verwenden, da beim Füllen der Tabellen keinerlei Kommunikation notwendig ist. Schlussendlich kann es hilfreich sein, den Graph effizienter abzubauen. Sobald wir einen Knoten abgearbeitet haben, wird dieser aus dem Graph entfernt. Sollte der Graph dadurch in mehrere Zusammenhangskomponenten zerfallen, könnte man solche, die nicht Teil der Lösung sein können, direkt entfernen. Ähnlich wie bei den Knoten, arbeitet man dann mit einer „Queue“ von Graphen.
Implementierungstechnisch sollte man noch mehr die Stärken von Haskell ausnutzen und mehr Kontrolle an den Compiler bzw. das Typsystem abgeben. Einige Bugs hätten vermieden werden können. Zum Beispiel sind Knoten zur Zeit keine eigenständigen Typen, sondern nur Aliasse für die dahinterstehenden Integer. Das ermöglicht es mit Datenstrukturen zu arbeiten, die speziell auf Integer ausgelegt sind, was jedoch zu Lasten der Typsicherheit geht.

Literaturverzeichnis

[1] DIMACS Clique Benchmark Graphen. `http://dimacs.rutgers.edu/Challenges/`. Abgerufen: 11-2008.

[2] Erdős Graphen. `http://vlado.fmf.uni-lj.si/pub/networks/data/`.

[3] Komplexitätsangabe IntSet. `http://hackage.haskell.org/packages/archive/containers/0.5.2.1/doc/html/Data-IntSet.html`.

[4] James Abello, MauricioG.C. Resende, and Sandra Sudarsky. Massive quasi-clique detection. In Sergio Rajsbaum, editor, *LATIN 2002: Theoretical Informatics*, volume 2286 of *Lecture Notes in Computer Science*, pages 598–612. Springer Berlin Heidelberg, 2002.

[5] Martin Aigner and Günter M. Ziegler. *Proofs from THE BOOK (3. ed.)*. Springer, 2004.

[6] Vladimir Batagelj and Matjaz Zaversnik. An O(m) algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.

[7] Dongbo Bu, Yi Zhao, Lun Cai, Hong Xue, Xiaopeng Zhu, Hongchao Lu, Jingfen Zhang, Shiwei Sun, Lunjiang Ling, Nan Zhang, Guojie Li, and Runsheng Chen. Topological structure analysis of the protein–protein interaction network in budding yeast. *Nucleic Acids Research*, 31(9):2443–2450, 2003.

[8] David Eppstein and Emma S. Spiro. The h-index of a graph and its application to dynamic subgraph statistics. *J. Graph Algorithms Appl.*, 16(2):543–567, 2012.

[9] Santo Fortunato. Community detection in graphs. *CoRR*, abs/0906.0612, 2009.

[10] Jorge E. Hirsch. An index to quantify an individual's scientific research output. *Proceedings of the National Academy of Sciences of the United States of America*, 102(46):16569–16572, 2005.

[11] Sun-Yuan Hsieh, Chin-Wen Ho, Tsan sheng Hsu, Ming-Tat Ko, and Gen-Huey Chen. Characterization of efficiently solvable problems on distance-hereditary graphs. In *ISAAC*, pages 257–266, 1998.

[12] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, 1972.

[13] Christian Komusiewicz and Manuel Sorge. Finding dense subgraphs of sparse graphs. In *Proceedings of the 7th International Symposium on Parameterized and Exact Computation (IPEC'12)*, volume 7535 of *LNCS*, pages 242–251. Springer, 2012.

[14] Sven Kosub. Local density. In *Network Analysis*, pages 112–142, 2004.

[15] Foad Mahdavi Pajouh, Zhuqi Miao, and Balabhaskar Balasundaram. A branch-and-bound approach for maximum quasi-cliques. *Annals of Operations Research*, pages 1–17, 2012.

[16] Geoffrey Mainland, Roman Leshchinskiy, Simon Peyton Jones, and Simon Marlow. Haskell beats C using generalized stream fusion. In submission, 2012.

[17] Tsutomu Matsunaga, Chikara Yonemori, Etsuji Tomita, and Masaaki Muramatsu. Clique-based data mining for related genes in a biomedical database. volume 10, page 205, 2009.

[18] Nina Mishra, Robert Schreiber, Isabelle Stanton, and RobertE. Tarjan. Clustering social networks. In Anthony Bonato and FanR.K. Chung, editors, *Algorithms and Models for the Web-Graph*, volume 4863 of *Lecture Notes in Computer Science*, pages 56–67. Springer Berlin Heidelberg, 2007.

[19] Chris Okasaki and Andrew Gill. Fast mergeable integer maps. In *In Workshop on ML*, pages 77–86, 1998.

[20] Jeffrey Pattillo, Alexander Veremyev, Sergiy Butenko, and Vladimir Boginski. On the maximum quasi-clique problem. *Discrete Applied Mathematics*, 161(1-2):244–257, 2013.

[21] Chris Stark, Bobby-Joe Breitkreutz, Teresa Reguly, Lorrie Boucher, Ashton Breitkreutz, and Mike Tyers. BioGRID: a general repository for interaction datasets. *Nucleic Acids Research*, 34(suppl 1):D535–D539, 2006.

[22] Shengqi Yang, Bin Wu, Haiyan Long, and Bai Wang. Commtrend: An applied framework for community detection in large-scale social network. *Fuzzy Systems and Knowledge Discovery, Fourth International Conference on*, 2:139–143, 2009.

A.1 Legende

c : Komponenten
d : degeneracy
$\varnothing$ k : arith. Mittel über höchstes k der Instanz
$\varnothing$ Suchbaumknoten: arith. Mittel über Anzahl d. benötigten Suchbaumknoten pro k pro Instanz
$\varnothing$ Zeit (s): arith. Mittel über benötigte Zeit pro k pro Instanz

A.2 Graphtabelle - Übersicht über verschiedene Graph-Parameter

biologische Instanzen

Instanz	Knoten	Kanten	δ	Δ	Dichte	h-Index	d	c
acker-schmalwand-all	5704	12627	1	438	7.76e-4	43	12	128
acker-schmalwand-pc	1907	2870	1	437	1.57e-3	22	9	84
acker-schmalwand-p	1872	2828	1	437	1.61e-3	22	9	82
Ecoli-all	67	54	1	28	2.44e-2	3	1	13
Ecoli-pc	31	29	1	27	6.23e-2	2	1	2
Ecoli-p	31	29	1	27	6.23e-2	2	1	2
HIV1-all	317	332	1	244	6.62e-3	6	2	1
HIV1-pc	76	79	1	21	2.77e-2	6	2	2
HIV1-p	64	63	1	21	3.12e-2	5	2	2
HIV2-all	5	4	1	2	0.4	2	1	1
HIV2-pc	4	3	1	2	0.5	2	1	1
HIV2-p	4	3	1	2	0.5	2	1	1
Human-all	14771	67297	1	8649	6.16e-4	106	19	51
Human-pc	12385	45159	1	8560	5.88e-4	81	16	42
Human-p	12342	44739	1	8560	5.87e-4	81	16	39
Worm-all	3613	6828	1	524	1.04e-3	35	10	73
Worm-pc	157	153	1	10	1.24e-2	8	5	39
Worm-p	157	153	1	10	1.24e-2	8	5	39

DIMACS

Instanz	Knoten	Kanten	δ	Δ	Dichte	h-Index	d	c
brock200_1	200	14834	130	165	0.74	145	134	1
brock200_2	200	9876	78	114	0.49	99	84	1
brock200_4	200	13089	112	147	0.65	128	117	1
brock400_2	400	59786	274	328	0.74	294	278	1
brock400_4	400	59765	275	326	0.74	294	277	1
brock800_2	800	208166	472	566	0.65	516	486	1
brock800_4	800	207643	481	565	0.64	514	485	1
c-fat200-1	200	1534	14	17	7.71e-2	17	14	1
c-fat200-2	200	3235	32	34	0.16	33	32	1
c-fat200-5	200	8473	83	86	0.42	85	83	1
c-fat500-1	500	4459	17	20	3.57e-2	20	17	1
c-fat500-2	500	9139	35	38	7.33e-2	38	35	1
c-fat500-5	500	23191	92	95	0.19	94	92	1
c-fat500-10	500	46627	185	188	0.37	187	185	1
hamming10-4	1024	434176	848	848	0.82	848	848	1
hamming6-4	64	704	22	22	0.34	22	22	1
hamming8-4	256	20864	163	163	0.63	163	163	1
johnson16-2-4	120	5460	91	91	0.76	91	91	1
johnson32-2-4	496	107880	435	435	0.87	435	435	1
johnson8-2-4	28	210	15	15	0.55	15	15	1
johnson8-4-4	70	1855	53	53	0.76	53	53	1
keller4	171	9435	102	124	0.64	106	102	1
keller5	776	225990	560	638	0.75	565	560	1
p_hat1500-1	1500	284923	157	614	0.25	456	252	1
p_hat1500-2	1500	568960	335	1153	0.50	759	504	1
p_hat1500-3	1500	847244	912	1330	0.75	1050	929	1
p_hat300-2	300	21928	59	229	0.48	148	98	1
p_hat300-3	300	33390	168	267	0.74	208	180	1
p_hat700-1	700	60999	75	286	0.24	207	117	1
p_hat700-2	700	121728	157	539	0.49	352	235	1
p_hat700-3	700	183010	408	627	0.74	486	426	1

ERDÖS

Instanz	Knoten	Kanten	δ	Δ	Dichte	h-Index	d	c
ERDOS-97-1	433	1314	1	41	1.40e-2	22	9	3
ERDOS-97-2	5482	8972	1	257	5.97e-4	48	9	11
ERDOS-98-1	445	1381	1	42	1.39e-2	22	9	4
ERDOS-98-2	5816	9505	1	273	5.62e-4	49	9	12
ERDOS-99-1	454	1417	1	42	1.37e-2	22	9	5
ERDOS-99-2	6094	9939	1	276	5.35e-4	50	9	11