Heuristiken für das Entfernen von verbotenen Teilgraphen

Paul Walger

15. Juni 2016

Inhaltsverzeichnis

Zusammenfassung

1 Einleitung

Graphen sind eine bildliche Darstellung von Beziehungen zwischen Objekten. Dabei stellt man die Objekte als Kreise (auch Knoten genannt) und die Beziehungen als Linien, die diese Kreise verbinden (auch Kanten genannt) [21].

Graphen können eine Vielzahl von Problemen und Szenarien modellieren. Zum Beispiel können die Knoten Städte und die Kanten Zugverbindungen zwischen diesen Städten sein und somit wird der Graph das Zugnetzwerk von diesen Städten modellieren.

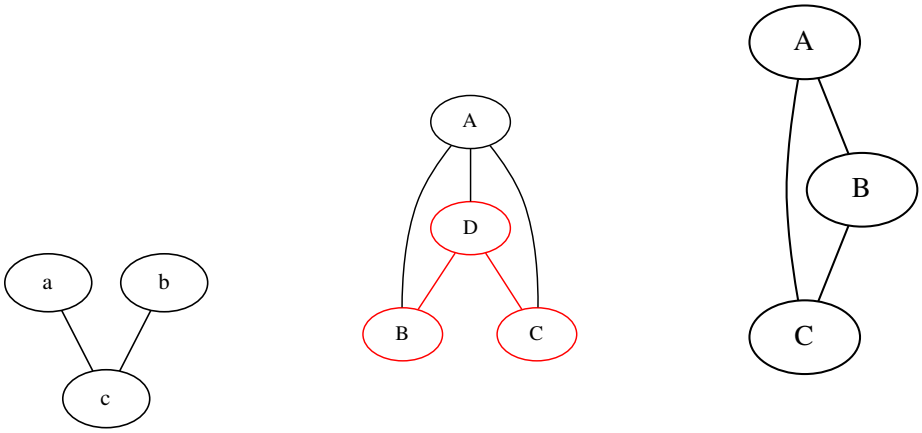
1.1 Motivation

Wird nun ein System von Objekten mittels einem Graphen modelliert, hat der Graph ebenso wie das modellierte System gewisse Eigenschaften.

Eine solche Eigenschaft wäre, dass wenn ein Knoten u mit einem anderen Knoten v verbunden ist, so ist u auch mit allen Nachbarn von v verbunden. Diese Graphen werden Cluster-Graphen genannt. Diese bestehen aus einer oder mehreren Komponenten, in welcher jeder Knoten mit jedem Knoten verbunden ist. Diese Cluster-Graphen lassen sich aber auch dadurch charakterisieren, dass sie gewisse Strukturen nicht besitzen und zwar dass sie in diesem Fall nicht den Graphen P_3 als einen induzierten Graphen haben. Der P_3 Graph ist in der Abbildung 1a zu sehen. Einen P_3 als induzierten Teilgraphen zu haben, bedeutet, dass man jedem Knoten aus dem P_3 einen Knoten aus dem Graphen zuordnen kann, sodass gilt, wenn zwei Knoten im P_3 durch eine Kante verbunden sind, so sind auch die entsprechenden Knoten in dem Graphen durch eine Kante verbunden und umgekehrt.

Erklären lässt sich das an der Abbildung 1. In dem Graphen G_1 in der Abbildung 1b sehen wir, dass der Teilgraph mit den Knoten B,D und C ein von P_3 induzierter Subgraph ist, weil wir den Knoten a dem Konten B, den Knoten b dem Knoten c und den Knoten c dem Konten D zuordnen können und die vorhin geforderte Eigenschaft gilt. So sehen wir, dass im Graphen P_3 a und c verbunden ist und im G_1 auch B und D verbunden ist. Auch ist B und C in Graphen G_1 nicht verbunden und a und b ebenso. Man kann überprüfen, dass die Voraussetzung für jede beliebiges Paar von Knoten geben ist und so ist der rot markierte Teilgraph ein von P_3 induzierte Teilgraph.

Der Graph G_2 (Abbildung 1c)hat keinen P_3 als induzierten Teilgraphen, weil wir nicht die Knoten vom P_3 zu Knoten in diesem Graphen zu zuordnen können, sodass unsere Voraussetzung erfüllt bleibt. Wenn wir zum Beispiel, die Zuordnung vom Knoten a zum Knoten A, b zu B und c zu C nehmen, sehen wir das Problem, dass wir eine Kante zwischen A und C haben aber



(a) Ein P_3 Graph

(b) Graph G_1 mit einem von P_3 induzierten Teilgraphen.

(c) Graph G_2 ohne einen induzierten P_3

Abbildung 1: Cluster-Graphen

keine Kante zwischen a und b. Somit ist das keine gültige Zuordnung. Man kann aber auch nachvollziehen, dass egal wie die Zuordnung gewählt wird, es nicht möglich ist die Bedingung zu erfüllen.

Deshalb ist der Graph G_2 ein Cluster-Graph und hat eben diese besondere Eigenschaft, die auf zwei Weisen beschreiben werden kann: 1. Der Graph hat eine oder mehrere Komponenten wo jeder Knoten mit jedem Knoten verbunden ist und 2. Der Graph hat kein P_3 als einen induzierten Teilgraphen. Die erste Charakterisierung ist eine mehr natürliche und wir könnten zu einer solchen Beschreibung aus praktischen Beobachtungen über unsere Daten kommen, während die zweite eine eher mathematische ist und nicht immer intuitiv, aber dennoch sehr nützlich ist, weil wir eine Eigenschaft sehr leicht formulieren können.

In dieser Arbeit werden wir uns die Methoden anschauen, welche einen Graphen so modifizieren, so dass er eine bestimmte Eigenschaft hat. Viele Eigenschaften lassen sich, wie in dem vorherigen Beispiel mittels einer Charakterisation durch verbotenen Teilgraphen beschreiben.

1.2 Definitionen

1.2.1 Notationen und Definitionen

Mit Graphen sei im Folgenden stets ein ungerichteter, einfacher Graph gemeint. Wenn nicht anders angegeben ist $G = (V, E)$ ein Graph, V die Menge seiner Knoten und E die Menge seiner Kanten.

Die Menge der Konten des Graphen G ist $V(G)$ ist und die Menge der Kanten des Graphen G heißt $E(G)$. Die Nachbarschaft vom Knoten u heißt $N(u)$, während $N^*(u)$ ist die Nachbarschaft von u mit u inklusive ist.

Sei $G = (V, E)$ ein Graph und $S \subseteq V$ eine beliebige Knotenmenge von V. Dann ist $G[S] = (S, E \cap \{\{u, v\} \mid u \in S \wedge w \in S\})$ der durch S induzierte Subgraph von G.

Seien $H = (V_H, E_H)$ und $G = (V, E)$ zwei Graphen. Ein Subgraph-Isomorphismus von H nach G ist eine Funktion $f : V_H \to V$ sodass wenn $(u, v) \in E_H$, dann gilt auch $(f(u), f(v)) \in E$. f ist ein induzierter Subgraph-Isomorphismus, wenn es auch gilt, dass wenn $(u, v) \notin E_H$, dann auch $(f(u), f(v)) \notin E$.

Seinen G und F Graphen, dann ist der Graph G F-frei, wenn es nicht F als induzierten Subgraphen enthält. Für eine Menge $\mathcal{F}$ von Graphen, heißt G $\mathcal{F}$-frei, wenn G F-frei ist für jeden $F \in \mathcal{F}$.

TODO: Siehe [10] für eine Definition von FPT und Kernel.

Ein ungerichteter Graph G heißt Wald, wenn er keinen Zyklus enthält. Ist der Graph G zudem zusammenhängend, dann heißt er auch Baum. Jede Zusammenhangkomponente eines Waldes ist daher ein Baum.

1.2.2 Problemstellung

Um Heuristiken für das Entfernen von verbotenen Teilgraphen entwickeln zu können, ist zuerst das Problem selbst zu definieren. Es ist unter dem Namen $\mathcal{F}$-Free Edge Editing bekannt.

> $\mathcal{F}$-Free Edge Editing
> **Eingabe:** Graph G, natürliche Zahl k, Menge von Graphen $\mathcal{F}$
> **Frage:** Können wir in G höchstens k Änderungen machen, so dass G keinen induzierten Teilgraphen aus $\mathcal{F}$ enthält?
> **Parameter:** k

1.3 Anwendungsbeispiele

In diesem Abschnitt werden wir auf einige Anwendungsfälle eingehen, warum man Graphen mit bestimmten Eigenschaften haben will.

1.3.1 Maximum Clique auf Co-Graphen

Um das Problem der Maximum Clique auf einem Graphen G zu finden, so ist es offensichtlich, dass wenn der Graph zwei Zusammenhangkomponenten hat, dann können wir Maximum Clique auf den beiden Komponenten lösen und das Maximum davon ist das Ergebnis für den Graphen G. Somit können wir das Problem also in kleinere Probleme zerlegen. Außerdem gilt es, dass das Finden einer Clique in einem Graphen G äquivalent zum Finden einer stabilen Menge in dem Komplementgraphen von G ist. Wir können also unser Problem weiter zerlegen, indem wir das Komplement von jeder Zusammenhangkomponente nehmen und es wieder in ihre Zusammenhangkomponenten zerlegen. Dabei gilt dass, eine maximale stabile Menge die Vereinigung aller maximalen stabilen Mengen der Zusammenhangkomponente ist.

Eine solche Vorgehensweise ist offensichtlich sehr attraktiv um ein schweres Problem, wie Maximum Clique es ist, zu lösen. Dieses Vorgehen hat jedoch ein Problem, der Graph G keine Zusammenhangkomponenten hat und der Komplementgraphen von G

Dann funktioniert unsere Vorgehensweise nicht mehr. Aber es gibt eine Klasse von Graphen, die complement reducible graphsöder kurz Co-Graphen, die so definiert worden ist, dass es nicht zu dem eben beschriebenen Problem kommt. Diese Klasse von Graphen ist auch durch charaktereriesiert, dass sie keinen P_4 als induzierten Teilgraphen enthalten [22]. Weil es nun bei dieser Klasse von Graphen nie dazu dem Problem kommt, können wir Maximum Clique in linearer Zeit auf solchen Graphen lösen, während auf allgemeinen Graphen dieses Problem NP-Vollständig ist.

1.3.2 Soziale Netzwerke

(P_4, C_4)-freie Graphen modellieren eine familiäre Struktur oder eine Gemeinschaft die hierarchisch organisiert ist. Diese Graphen werden auch trivial perfekte Graphen oder quasi-threshold Graphen genannt[28].

Die letztere Bezeichnung kommt von Charakterisierung dieser Graphen als baumartige Strukturen. Ein solcher gerichteter Graph lässt sich als der transitive Abschluss eines Waldes sehen[8]. Dieser Graph muss gerichtet sein, weil es sonst eine Clique wird.

In der Abbildung 2 ist solch ein Quasi-Threshold Graph zu sehen. Die durchgehenden Kanten zeigen den Wald, in diesem Fall ist es ein Wald mit genau einem Baum. Der dunkle-graue Knoten ist die Wurzel des Baumes. Dabei stellen gestrichelten Kanten den transitiven Abschluss dar.

Solche Graphen modellieren eine große

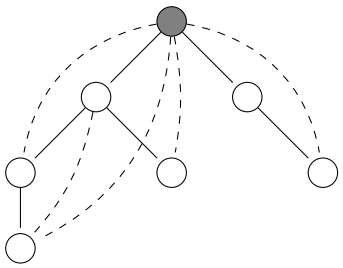
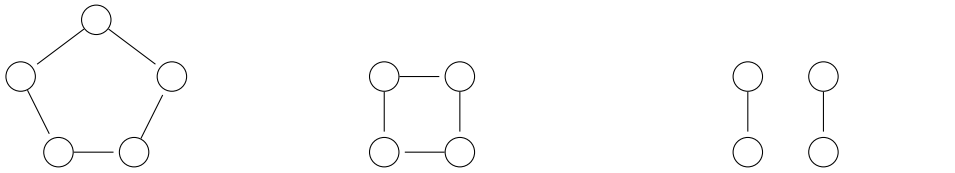


Abbildung 2: Ein Quasi-



(a) C_5 (b) C_4 (c) $2K_2$

Abbildung 3: Die verbotenen Graphen in einem Split-Graphen

Anzahl von Netzwerken. Unter anderem auch soziale Strukturen, wie eine Hierarchie in einer Organisation. Sei die Knoten Personen und die Kanten stellen die Wege dar, auf denen die Befehle fließen. Fast alle haben einen Vorgesetzten (bis auf die Wurzel) und nehmen Befehle vom ihm an, was durch die Baumstruktur modelliert wird. Aber sie hören auch auf die Befehle von dem Vorgesetzen des Vorgesetzen, was durch den transitiven Abschluss modelliert wird[22].

Ähnlich dazu sind (P_5, C_5)-freie Graphen die auch soziale Strukturen modellieren und dafür geeignet sind Gemeinschaften zu identifizieren [26].

1.3.3 Protein Interaktionsnetzwerke

Im Abschnitt 1.1 haben wir Cluster-Graphen betrachtet, wo jede Zusammenhangkomponente eine Clique ist. Split-Graphen sind ähnliche Graphen, aber bestehen nur aus einer Clique und einigen Knoten, die an der Clique anhängen. Formell definiert heißt es, dass man den Menge der Knoten des Graphens in zwei Mengen V_1 und V_2 teilen kann, sodass G[V_1] ein stabile Menge und eine G[V_2] eine Clique ist. Split-Graphen sind $(2K_2, C_4, C_5)$-freie Graphen. Ein Split-Cluster Graph ist ein Graph wo jede Zusammenhangkomponente ein Split-Graph ist [9].

Diese Struktur modelliert gut Protein-Protein Interaktion Netzwerke. In Körper vom Menschen oder anderen Lebewesen gibt es eine Vielzahl von Proteinen, die einen Zweck haben, aber diesen erfüllen sie meistens nicht alleine, sondern in dem sie Proteinkomplexe bilden. Diese Protein-Komplexe bestehen aus einer großen Anzahl von Proteinen und sind nicht einfach zu untersuchen. Ein Weg um solche Proteinkomplexe zu identifizieren besteht darin, herauszufinden welche Proteine mit welche Proteinen überhaupt inter-

agieren. So kann man die diese Proteine als Knoten ansehen und wenn es eine Interaktion zwischen diesen Proteinen gibt, diese als eine Kante ansehen. Ein Proteinkomplex hat einen Kern, wo jedes Protein mit jedem Protein interagiert und Anhängsel, einzelne Proteine, die nur mit dem Kern interagieren. Damit modellieren Split-Cluster-Graphen die Struktur von Proteinkomplexen und können dazu verwendet werden um Protein-Komplexe zu bestimmen.

1.4 Annäherung

Jedoch haben die Eingabedaten typischerweise Fehler oder sind unvollständig. So können in einem Cluster-Graphen bestimmte Kanten fehlen, obwohl sie dazu gehören sollten. Oder in Split-Cluster-Graphen fehlen Informationen zu Interaktionen zu gewissen Paaren von Proteinen. Weil die Daten nun fehlerhaft sind, stellt sich die Frage, wie viele und welche Änderung müssen am Graphen getan werden, damit er die erwünschte Struktur hat. Wenn man nun die Struktur durch eine Menge $\mathcal{F}$ von verbotenen Graphen charakterisiert, nennt sich das Problem $\mathcal{F}$-Free Edge Editing.

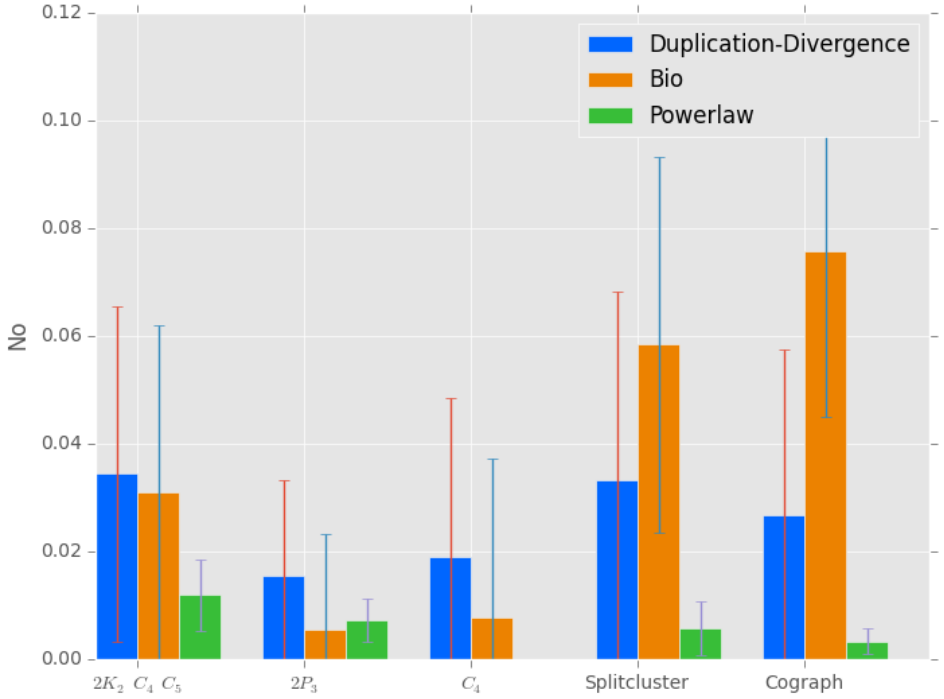
Nun ist $\mathcal{F}$-Free Edge Editing für die meisten $\mathcal{F}$ sehr aufwendig zu lösen. Nun ist es möglich dieses Problem nicht perfekt zu lösen indem wir die minimale Anzahl von Änderungen finden, sonder einfach eine möglichst kleine Anzahl von Änderungen.

Was bringt es aber, wenn man nur eine Annäherung hat? Wenn der zugrunde liegende Graph eine gewisse Struktur hat, kann auch mit 2-mal soviel Änderungen wie es nötig, eine ganz andere Struktur aufgedeckt oder eher erzeugt werden. Auch wenn wenn in den Experimenten festgestellt wurde, dass die Heuristik um 1.7x größere Menge von zu änderenden Kanten zurückgibt, wie kann diese Heuristiken sinnvoll verwenden?

Ersten soll es die Frage klären, wie gut eine allgemeine Heuristik ist im Vergleich zu den auf spezifische Anwendungsfälle zugeschnittenen Heuristiken. Lohnt es sich überhaupt für spezifische Heuristiken zu entwickeln?

Auch ein möglicher Anwendungsfall wäre, dass man eine Menge von Graphen hat, die man noch nicht analysiert hat. Wenn man nun die Heuristiken mit vielen möglichen verbotenen Teilgraphen auf den Eingabegraphen laufen lässt und dann vergleicht, kann man sehen, welche Graphen welche Strukturen eher besitzen.

Um diesen Anwendungsfall zu demonstrieren: Wir testen 3 Graphen die aus verschiendenen Datensätzen kommen.



1.5 Ähnliche Arbeiten

Implicit Hitting Set hilft hier leider nicht viel.[20]

Approximation von H-Free Editing für montone graphen eigenschaften: $o(n^2)$ ist effizient, aber $O(n^{2-\epsilon})$ ist NP-Hard.[4]

$\mathcal{F}$-Free Edge Editing ist NP-Schwer für die meisten $\mathcal{F}$. Man es exakt mit FPT lösen[10], für die meisten Instanzen von $\mathcal{F}$ gibt es keine Polynomiale Kernel.

1.6 Überblick

Im Abschnitt 2 werden die Ansätze vorstellt um dieses Problem zu lösen. Dann werden wir im Abschnitt 3 betrachten wie die Tests aussehen und auf welchen Modellen diese Algorithmen auf ihre Tauglichkeit getestet wurden. Im Abschnitt 4 wird auf Details der Umsetzung von diesen Algorithmen eingegangen. Darauf folgend werden im Abschnitt 5 die Resultate vorgestellt und diskutiert, während im Abschnitt 5.4 unsere Lösungen mit anderen bisherigen Lösungsansätzen verglichen werden.

2 Algorithmen

In diesem Abschnitt werden die verschiedenen Heuristik-Ansätze für $\mathcal{F}$-Free Edge Editing vorgestellt. Die im nachfolgenden beschriebenen Algorithmen basieren alle auf dem folgenden Prinzip: Suche einen validen Graphen, welcher die verbotenen Subgraphen nicht enthält. Wiederhole dies mehrmals und gebe dann die Differenz zwischen dem besten validen Graphen und dem Eingabegraphen aus.

Dieses Prinzip ist im Algorithmus 1 zu sehen. Dort ist G_{input} der Eingabegraph, `F` die Menge der verbotenen Teilgraphen und i_{max} die Angabe wie oft der Algorithmus wiederholt werden soll. Dabei steht SolveAlgo für einen der Algorithmen, die wir in den folgenden Abschnitten betrachten werden.

Algorithm 1 Genereller Aufbau

```
1: function Solve(G_input, F, i_max)
2:     G_best ← (∅, ∅)
3:     for i = 1 to i_max do
4:         G_valid ← solveAlgo(G_input, F)
5:         if #(diff(G_best, G_input)) < #(diff(G_valid, G_input)) then
6:             G_best ← G_valid
7:         end if
8:     end for
9:     print diff(G_input, G_best)
10: end function
```

Da alle Ansätze diese Schritte enthalten und sich nur darin unterschieden, wie der valide Graph mittels SolveAlgo gefunden wird, wird folgend nur dieser Aspekt betrachtet.

Die entwickelten Ansätze sind in 3 große Gruppen zu unterteilen. Der Backward-Ansatz nimmt den Graphen und ändert ihn solange, bis ein gültiger Graph entsteht. Der Forward-Ansatz fängt mit einem leeren oder vollen Graphen an, und ändert solange Konten, bis man möglichst nahe an dem Eingabegraphen ist. Der Grow-Reduce-Ansatz kombiniert diese beiden Ansätze, indem es prinzipielle zwei Stufen gibt. In der Grow-Stufe werden mögliche Kanten eingefügt und in der Reduce-Stufe werden Kanten entfernt.

2.1 Backward-Ansatz

Der Backward-Ansatz nimmt den Graphen und ändert ihn solange, bis ein gültiger Graph entsteht.

2.1.1 RandomFlip

Das ist der einfachste Algorithmus. Solange der Graph verbotene Subgraphen hat, dann versuche eine Kante in dem Graphen zu ändern.

Algorithm 2 RandomFlip

```
1: function RandomFlip(G, F)
2:     while G is not valid do
3:         for every Graph f ∈ F  do
4:             forbiddenSubgraph ← findFS(G, f)
5:             while forbiddenSubgraph ≠ ∅ do
6:                 change the edge in G corresponding to an random edge in
   forbiddenSubgraph
7:                 forbiddenSubgraph ← findFS(G, f)
8:             end while
9:         end for
10:    end while
11:    return graph
12: end function
```

2.1.2 RandomFlipUnchanged

Der RandomFlipUnchanged-Algorithmus ist wie der RandomFlip-Algorithmus, aber bereits editierte Kanten werden mit einer geringeren Wahrscheinlichkeit geändert. Auch hat es für kleine Graphen ein Konvergenzkriterium. Dieses Konvergenzkriterum besteht darin, dass nach für jede Änderung, die Anzahl der verbotenen Subgraphen gezählt wird und die nur dann ausgeführt wird, wenn die Anzahl der verbotenen Subgraphen dadurch weniger wird.

Der allgemeine Vorgehensweise wird im Algorithmus 3 beschrieben. Der große Unterschied zu zum RandomFlip sehen wir ab Zeile 7. Dort wird eine Kante zufällig ausgewählt, welche geändert werden soll, aber eine bereits geänderte Kante wird 4-mal so selten ausgewählt, wie eine noch nicht geändertete Kante (siehe Zeile 11).

2.2 Forward-Ansatz

Die Forward-Ansätze zeichnet sich dadurch aus, dass wir mit einem Graphen beginnen, der die selben Knoten wie der Eingabegraph hat, aber keine Kanten. Dieser Graph ist somit valide, weil er keine verbotenen Subgraphen ent-

Algorithm 3 RandomFlipUnchanged

```
1: function RandomFlipUnchanged(G, F)
2:     while G is not valid do
3:         for Graph f ∈ F  do
4:             forbiddenSubgraph ← findFS(G, f)
5:             while forbiddenSubgraph != ∅ do
6:                 foundEdge ← ∅
7:                 while true do
8:                     e ← random edge from forbiddenSubgraph
9:                     prob ← 1 / #(E(f))
10:                    if e is already visited then
11:                        prob ← prob / 4
12:                    end if
13:                    if random number from [0,1] > prob then
14:                        foundEdge ← e
15:                        break
16:                    end if
17:                    flip e in graph
18:                end while
19:                forbiddenSubgraph ← findFS(graph, f)
20:            end while
21:        end for
22:    end while
23:    return graph
24: end function
```

hält. Dies ist ein Vorteil gegenüber den Top-Bottom-Ansätzen, da es möglich ist immer einen validen Graphen zu haben und somit jederzeit terminieren.

2.2.1 Extend

Zu sehen die Vorgehensweise in Algorithmus 4. Er fängt mit einem Graphen an, der die selben Knoten wie der Eingabegraph hat, aber keine Kanten (Zeile 2). Dann wird versucht jede Kante einzufügen, die auch im originalen Graphen `input` vorhanden war (Zeile 4). Wenn es einen invaliden Graphen erzeugt, also dass es nun einen verbotenen Teilgraphen im `graph` gibt, dann wird die Änderung rückgängig gemacht (Zeile 5). Wenn nun keine Änderung in einem Durchlauf gemacht wurden, dann bricht der Algorithmus ab (Zeile 7) und gibt den erzeugen Graphen zurück.

Ein beispielsweiser Ablauf für den Extend-Algorithmus, wo P_3 der verbotene Teilgraph ist ist in der Abbildung 4 zu sehen. Dabei wird der Graph `graph` dargestellt und eine gestrichelte Kante gibt an, wenn die Kante in dem Graphen `input` vorhanden ist, aber nicht in dem Graphen `input`. Wenn die Kante rot ist, dann wurde versucht die Kante einzufügen, aber es hat eine P_3 erzeugt, wenn die Kante grün ist, dann wurde die Kante eingefügt, ohne dass ein P_3 erzeugt wurde.

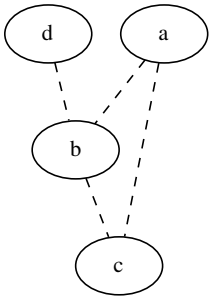
Es ist zu sehen, dass dieser Algorithmus keine Kanten hinzufügt, die nicht in dem Eingabe-Graphen nicht vorhanden waren.

Algorithm 4 Extend

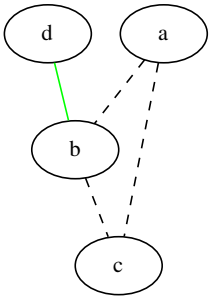
```
1: function ExtendSolve(G, F)
2:     graph ← (V(G), ∅)
3:     while true do
4:         for each Edge e ∈ Difference(graph,G) do
5:             try to flip e, revert if it produces an invalid graph
6:         end for
7:         break if there was no change
8:     end while
9:     return graph
10: end function
```

2.3 GRASP

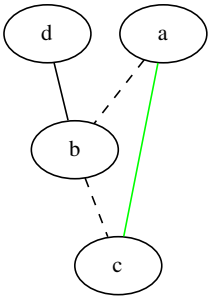
Nach dem experimentellen Untersuchen der Forward und Backward-Ansätze wurden deren Stärken und Schwächen deutlich. So kam auch die Idee diese beiden Ansätze zu kombinieren. Nach dem Planen und der Umsetzung einer



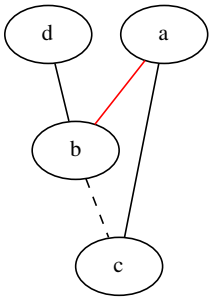
(a) Startgraph



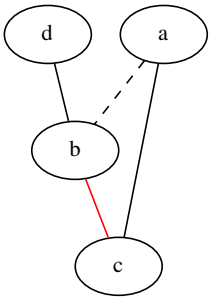
(b) Kante (d,b) wird erfolgreich hinzugefügt



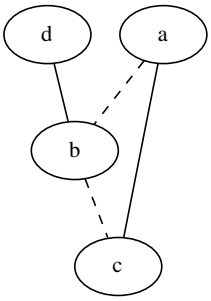
(c) Kante (a,c) erfolgreich wird hinzugefügt



(d) Kante (a,b) kann nicht hinzugefügt werden



(e) Kante (b,c) kann nicht hinzugefügt werden



(f) Resultierender Graph

Abbildung 4: Beispielweiser Ablauf des Extends

solchen kombinierten Heuristik, hat die Recherche ergeben, dass es eine sehr ähnliche Art von Algorithmen bereits beschrieben wurde und unter dem dem Namen GRASP (Greedy Randomized Adaptive Search Procedure) [14] [5] bekannt ist. Es ist eine iterative Vorgehensweise wobei jede Iteration aus zwei Phasen besteht:

- **Konstruktions-Phase.** Es wird mit einem leeren Graphen angefangen und nach und nach werden Knoten/Kanten nach einem bestimmten Kriterium hinzugefügt.
- **Lokale Suche.** Da die Lösung, die in der Konstruktions-Phase generiert wurde, meistens nicht optimal ist, auch nicht in der einfachen Nachbarschaft. Deswegen wird in der Lokalen Suche versucht die Lösung zu verbessern, indem man nach und nach die derzeitige Lösung durch eine bessere Lösung in der Nachbarschaft ersetzt.

Dieser GRASP Ansatz wurde auch schon erfolgreich für das Biclustering-Problem verwendet [13].

Unsere Ansätze unterscheidet sich darin von den GRASP-Algorithmen, dass wir nicht in jeder Iteration einen neuen Graphen Konstruieren, sondern auf dem bereits konstruieren Graphen weiter arbeiten.

2.3.1 Grow-Reduce

Der Grow-Reduce-Ansatz sieht wie folgt aus: Begonnen wird mit einem Graphen, der die selben Knoten wie der Eingabegraph hat, aber keine Kanten. Dann wird in jeder Iteration ein Knoten und seine Umgebung hinzugefügt und durch lokale Suche werden alle neu entstandenen verbotenen Subgraphen wieder entfernt. Dies ist im Algorithmus 5 zu sehen.

Die Grow-Phase ist mit der Konstruktion-Phase des GRASP-Ansatzes vergleichbar und die Reduce-Phase mit der lokalen Suche.

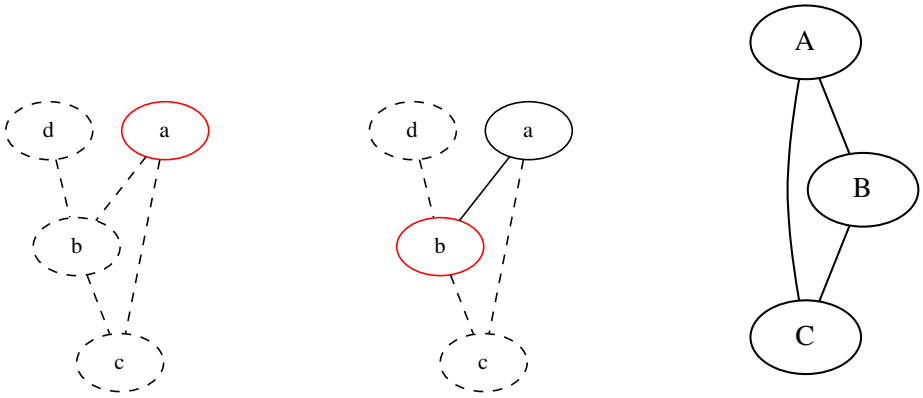
2.3.2 Explored-Grow-Reduce

Der Explored-Grow-Reduce-Ansatz ist dem Grow-Reduce-Ansatz ähnlich, bis auf das, es in der Grow-Phase nur die Kanten zu Knoten hinzufügt, die bereits erforscht sind.

Dieser Unterschied wird in der Abbildung 5 verdeutlich, wo nur die Grow Schritte visualiert wurden, ohne die Reduce-Phase, um es zu vereinfachen.In Abbildung 5a ist der Anfangstatus zu sehen. Die gestrichelten Kanten, sind Kanten die Graphen vorhanden sind, aber noch nicht hinzugefügt worden sind und es wird der Knoten a hinzugefügt, weil aber keine anderen Knoten

Algorithm 5 GrowReduce

```
1: function GrowReduceSolve(G, forbidden)
2:     graph ← (V(input), ∅)
3:     nodes ← order(V(G))
4:     for node ∈ nodes do
5:         for neighbor ∈ N(node) do                    ▷ Grow Phase
6:             Add Edge (node, neighbor) to graph
7:         end for
8:         for f ∈ forbidden do                         ▷ Reduce Phase
9:             forbiddenSubgraph ← findFS(graph,f)
10:            while forbiddenSubgraph != ∅ do
11:                edge ← random Edge from forbiddenSubgraph
12:                count ← #(findAllFS(graph,f))
13:                flip edge in graph
14:                countAfter ← #(findAllFS(graph,f))
15:                if countAfter ≥ count then
16:                    flip edge in graph
17:                end if
18:                forbiddenSubgraph ← findFS(graph, f)
19:            end while
20:        end for
21:    end for
22:    return graph
23: end function
```



(a) Knoten a wird hinzugefügt

(b) Knoten b wird hinzugefügt

(c) Knoten c wird hinzugefügt

Abbildung 5: Beispielweise Grow-Phase

bisher hinzugefügt worden sind, wird werden keine Kanten hinzugefügt. In Abbildung 5b wird der Knoten b hinzugefügt und weil a auch schon hinzugefügt wurde, wird auch die Kante (a, b) hinzugefügt. Aber weder (a, c) noch (a, b) werden hingefügt, weil c noch nicht erforscht wurde. In Abbildung 5c wird der Knoten c hinzugefügt und somit auch die Kanten (a, b) und (a, c).

2.3.3 Sortierung der Knoten

Sowohl in dem Grow-Reduce als auch dem Explorer-Grow-Reduce Algorithmus werden die Knoten mittels der Funktion ORDER sortiert und dann eine nach der anderen abgearbeitet. Nun kann man die Knoten nach unterschiedlichen Kriterien sortieren. Es wurden 3 Sorterierungsfunktionen verwendet:

- Zufällig. Das ist einfachste und offensichtliches Sortierungsfunktion. Wenn keine Informationen über die Struktur der Daten vorhanden sind, dann ist das zufällige Abarbeiten der Knoten naheliegend.
- Anzahl der Nachbarn. Die Knoten werden nach der Anzahl der Nachbarn im Eingabegraphen sortiert.
- Anzahl von verbotenen Teilgraphen. Die Knoten werden danach sortiert, wie oft der Knoten in einem verbotenen Teilgraphen vorkam.

Algorithm 6 ExploredGrowReduce

```
1: function ExploredGrowReduceSolve(input, forbidden)
2:     graph ← (V(input), ∅)
3:     nodes ← order(V(input))
4:     explored ← ∅
5:     for node ∈ nodes do
6:         for neighbor ∈ N(node) do                          ▷ Grow Phase
7:             if neighbor ∈ explored then
8:                 Add Edge (node, neighbor) to graph
9:             end if
10:        end for
11:        explored ← explored ∪ {node}
12:        for f ∈ forbidden do                               ▷ Reduce Phase
13:            forbiddenSubgraph ← findFS(graph,f)
14:            while forbiddenSubgraph != ∅ do
15:                edge ← random Edge from forbiddenSubgraph
16:                count ← #(findAllFS(graph,f))
17:                flip edge in graph
18:                countAfter ← #(findAllFS(graph,f))
19:                if countAfter ≥ count then
20:                    flip edge in graph
21:                end if
22:                forbiddenSubgraph ← findFS(graph, f)
23:            end while
24:        end for
25:    end for
26:    return graph
27: end function
```

Bei den letzten beiden Sortierunsgmöglichenkeiten gibt es sowohl die aufsteigende und absteigende Variante.

2.4 Lineare Programmierung

2.4.1 Lineare Optimierung

Bei der linearen Optimierung wird eine lineare Zielfunktion minimiert bzw. maximiert, wobei sie durch lineare Gleichungen und Ungleichungen beschränkt ist.

2.4.2 Das Model des Graphen

Wir nutzen binäre Variablen e_{uv}, wobei $u, v \in V$ sind und $u < v$ gilt. Dabei ist $e_{uv} = 1$ genau dann wenn, die kante u, v ein Teil des Lösungsgraphen ist.

Wir minimieren

$$\sum_{u,v \in V} \begin{cases} e_{u,v} & \{u, v\} \in E \\ -e_{u,v} & \{u, v\} \notin E \end{cases}$$

Dies ist die Zielfunktion `objective` im Algorithmus 7.

Da alle möglichen Bedingungen hinzuzufügen, welche bei alle verbotenen Subgraphen ausschließen würden, viel zum umfangreich wäre, werden die Bedingungen iterative dort hinzugefügt, wo es einen verbotenen Teilgraphen gibt (siehe Zeile 5). Dann wird der Problem gelöst(siehe Zeile 16) und und die Änderungen auf den Graphen übertragen(siehe Zeile 17) . Dann wird wieder nach alle verboteten Subgraphen gesucht(siehe Zeile 4). Dies wird solange wiederholt bis es keine mehr gibt. Nun ist die minimale Anzahl von Änderungen gefunden. Dieses Vorgehen ist im Algorithmus 7 zu sehen.

3 Aufbau der Test

3.1 Datensätze

Folgende Datensätze wurden verwendet. Da verschiedene Mengen von verbotenen Teilgraphen monotonen Grapheneigenschaften zugeordnet werden können und jeder Datensatz von Graphen und jede Methode zufällige Graphen zu erzeugen, charakteristische Eigenschaften hat, ist es notwendig verschiedene Datensätze zu verwenden und verschiedenen Methoden zur Erzeugung von zufälligen Graphen. Ingesammt wurde 5 verschiedene Methoden zur Erzeugung von zufälligen Graphen verwendet und 3 Datensätze. Wir brachten zuerst die zufälligen Graphen.

Algorithm 7 F-Free BLP

```
1: function SOLVEBLP(graph, forbidden)
2:     constraints ← ∅
3:     for graph f ∈ forbidden do
4:         while FINDFS(graph, f) != ∅ do
5:             for each graph M ∈ FINDEFS(graph, f) do
6:                 contstraint ← 0
7:                 for each {u, v} ∈ E(M) do          ▷ Add all Constraints
8:                     if {u, v} ∈ E(graph) then
9:                         contstraint += 1 - e_uv
10:                    else
11:                        contstraint += e_uv
12:                    end if
13:                end for
14:                constraints ← constraints ∪ { constraint }
15:            end for
16:            variables ← LPSOLVE(constraints, objective)
17:            for e_u,v ∈ variables do          ▷ Apply solution to the graph.
18:                if e_u,v = 1 then
19:                    Set edge (u, v) in graph
20:                else
21:                    Remove edge (u, v) in graph
22:                end if
23:            end for
24:        end while
25:    end for
26:    return graph
27: end function
```

3.1.1 Barabási–Albert

Für den Datensatz `barabasi albert` wurde das Barabási–Albert Modell verwendet, welches ein zufälliges skalenfreies Netz erzeugt[3]. Skalenfrei bedeutet hier, dass die Knotengrad einer Potenzverteilung folgt. Es gibt also viel mehr Konten die einen geringeren Grad haben als Knoten mit einem hohen Anzahl von Nachbarn.

Es wurden 56 Graphen generiert mit Knotenanzahl in $\{10, 20, 30 \ldots 140\}$, mit dem Parameter $m \in \{1, 3, 5, 7\}$, welcher die Anzahl der der Kanten definiert, die zu bereits bestehenden Knoten erstellt werden.

3.1.2 Erdős-Rényi

Für den Datensatz `ER` wurde das Erdős-Rényi Modell verwendet[15] [6], wo jede Kante eine fixe Wahrscheinlichkeit hat zu existieren oder nicht zu existieren.

Es wurden dabei 54 Graphen generiert, mit einer Knotenanzahl $n \in \{10, 20 \ldots, 90\}$ und den folgenden Wahrscheinlichkeiten: $\frac{1}{10}$, $\frac{2}{10}$, $\frac{5}{20}$, $\frac{4}{10}$, $\frac{5}{10}$, $\frac{8}{10}$.

3.1.3 Duplication-Divergence

Für den Datensatz `duplication divergence` wurde das Duplication Divergence Modell verwendet[17], welches Interaktionsnetzwerke zwischen Proteinen modelliert. Ein Beispiel ist in Abbildung 6 zu sehen.

Dabei gibt es in jeder Iteration bei der Erstellung eines solchen zufälligen Graphen zwei Phasen. Die erste ist die Duplikations-Phase, wo ein zufälliger Knoten u genommen und dupliziert wird zu v. Dann beginnt die Divergence-Phase, wo zu jedem Nachbarn von u mit gewissen Wahrscheinlichkeit p eine Kante zu v hinzugefügt wird. Falls keine Kanten hinzugefügt wurde, dann wird v wieder gelöscht. Dies wird n-mal wiederholt

Es wurden mit diesem Modell 54 Graphen generiert mit $n \in \{10, 20 \ldots, 90\}$. Für die Wahrscheinlichkeiten p wurden folgenden Werte verwendet $\frac{1}{10}$, $\frac{2}{10}$, $\frac{5}{20}$, $\frac{4}{10}$, $\frac{5}{10}$, $\frac{8}{10}$.

3.1.4 Newman-Watts-Strogatz

Für den Datensatz `newman_watts_strogatz` wurde das Newman-Watts-Strogatz Modell verwendet[24], welches Kleine-Welt-Graphen erzeugt mit geringen durchschnittlichen Knotendistanzen und einem hohen Clusterkoeffizienten.

Dabei wird zuerst ein Ring von n Knoten erstellt. Dann wird jeder Knoten mit k von seinen nächsten Nachbarn verbunden (oder mit $k - 1$, wenn k

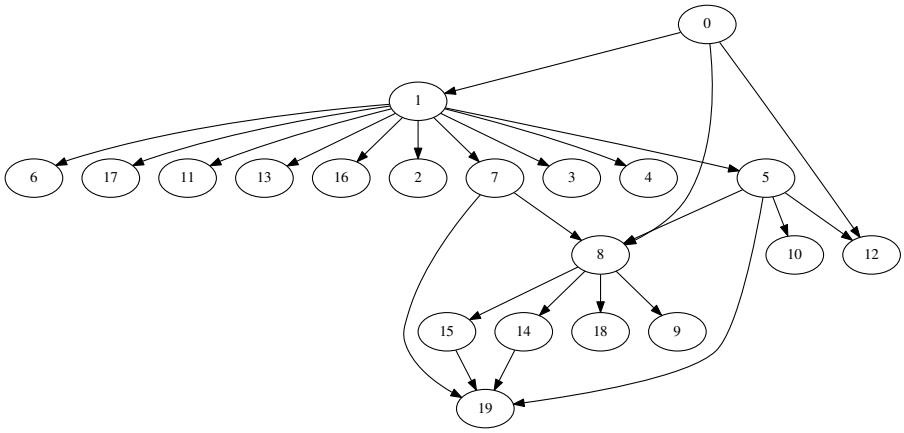


Abbildung 6: Ein beispielhafter Duplication-Divergence Graph mit $n = 20$ und $p = 0{,}4$

ungerade ist). Dann werden Abkürzungen erzeugt indem, man für jede Kante (u, v) in dem zugrundeliegenden n-Ring mit den k nächsten Nachbarn: Füge mit der der Wahrscheinlichkeit p eine neue Kante (u, w) ein, wobei w ein zufällig gewählter Knoten ist.

Es wurden mit diesem Model 144 Graphen generiert mit einer Knotenanzahl $n \in \{10, 20 \ldots, 90\}$, $m \in \{2, 4, 6, 8\}$ und der Wahrscheinlichkeit $p \in \{\frac{2}{10}, \frac{4}{10}, \frac{6}{10}, \frac{8}{10}\}$.

3.1.5 Powerlaw-Baum

Für den Datensatz `powerlaw`, wurde ein Modell verwendet, dass einen Baum erzeugt, dessen Knotengrad einer Potenzverteilung folgt. Ein Beispiel ist in Abbildung 7 zu sehen. Es wurden mit diesem Model 30 Graphen generiert mit einer Knotenanzahl zwischen $n \in \{10, 15, 20 \ldots, 155\}$.

3.1.6 Netzwerke

Für den Datensatz `UCINetworkDataRepository` wurden 9 reale Graphen verwendet, bereitgestellt von der University of California.

- Der Graph `karate` ist ein soziales Netzwerk von Freundschaften zwischen 34 Mitgliedern eines Karate-Clubs in einer US-Universität in 1970 [29].

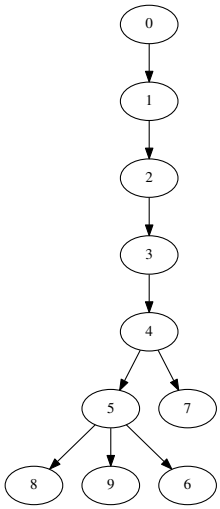


Abbildung 7: Ein Powerlaw-Baum mit $n = 10$

- Der Graph `polbooks.paj` ist ein Netzwerk von Bücher über die aktuelle US Politik, die von dem Onlinehänder Amazon.com verkauft wurdem. Kanten repräsentieren häufiges Kaufen von den beiden Büchern von dem selben Käufer [1].+
- Der Graph `football` ist ein Netzwerk von amerikanischen Footballspielen im Herbst 2000 [16].
- Der Graph `power.paj` ist ein Netzwerk, dass die Topologie des "Western States Power Gridïn der Vereinigten Staaten wiederspieglt [27].
- Der Graph `adjnoun` ist ein Netzwerk von häufigen Adjektiven und Nomen in dem Roman "David Copperfield"von Charles Dickens [23].
- Der Graph `lesmiserables` ist ein Netwerke von Figuren, die in dem Roman Les Misérables"von Victor Hugo, zur gleichen Zeit auftreten [18].
- Der Graph `celegansneural.paj` welches das neurale Netzwerk von Caenorhabditis elegans [27]. Es ist ein Fadenwurm, welcher gerne als Modellorganismus studiert wird. Jeder erwachsene C. elegans hat genau 302 Nervenzellen.
- Der Graph `dolphins` ist ein soziales Netzwerk von 62 Dolphinen die in einer Gemeinschaft in der Nähe von Neuseeland leben [19].

- Der Graph `polblogs` ist ein Netzwerk von Hyperlinks zwischen Webblogs in 2005, die sich mit auf US Politik beschäftigten [2].

3.1.7 Squenzähnlichkeit von Proteinen

Der Datensatz `bio1` umfasst 147 Graphen und der `bio2` umfasst 350 Graphen. Beide sind Grahen die Squenzähnlichkeit von Proteinen modellieren [25] [7].

3.2 Optimale Lösung

Um die Qualität der Lösung eines heuristischen Ansatzes bewerten zu können, ist es sehr gut die optimale Lösung zu wissen. Es gibt verschiedene Ansatz wie das Problem zu lösen sein, wir haben uns jedoch für die lineare Optimierung entschieden. Der BLP-Alogoritmus versucht eine optimale Lösung zu finden. Dies kann jedoch sehr lange dauern, da die Komplexität des Problemes exponentiell. In der Tabelle Tabelle 1 sehen wir, dass für viele verbotene Teilgraphen wir für alle Instanzen eine optimale Lösung berechnen konnten, aber es gibt schwierige verbotene Teilgraphen wie (P_5, triangle) wo man wir für die meisten Instanzen keine optimale Lösung haben.

Tabelle 1: Nicht gelöste Instanzen für den Datensatz *bio*2

Verbotener Teilgraph	Anzahl	Prozent
$2P_3$	0	0%
C_4	0	0%
claw	0	0%
paw	4	1%
triangle	11	3%
splitcluster	26	7%
(P_4, C_4)	37	10%
(x172, triangle)	129	36%
(P_5, triangle)	309	86%

4 Implementation

4.1 Repräsentation vom Graphen

Die Graphen werden in einer Adjazenzmatrix gespeichert.
TODO: MORE

4.2 Das Finden von induzierten Subgraphen

Das Finden von induzierten Subgraphen ist der Teil der Algorithmen, der am zeitaufwändigsten ist und auf häufigsten aufgerufen wird. Deswegen ist hier die Wahl von dem richtigen Ansatz sehr wichtig.

4.2.1 Vergleich von Algorithmen fürs Finden von Ismorphen Subgraphen

In diesem Abschnitt wollen wir drei Ansätze bezüglich ihrer Geschwindigkeit vergleichen. Um das ganze zu vereinfachen, testen wir Anwendungsfälle eines Ansatzes mit dem verbotenen Teilgraphen P_3.

- Der erste ist VFLib. Es ist ein Bibliothekt, die mehrere Algorithmen jeweils für das Graphen Isomorphismus Problem, Graphen-Subgraph Isomorphismus Problem und das Graphen Monomorphimus Problem implementiert und zählt zu den schnellsten Implementationen [12].
- Der zweite Ansatz nutzt die Boost-Implementation von VF2, welches VFLib auch implementiert.
- Im dritten Ansatz wird die Suche nach dem P_3 selbst möglichst effizient implementiert.

Hierbei vergleichen wir nicht die verschiedenen Algorithmen für das Graph-Subgraph Isomorphismus Problem, weil VFLib und Boost den selben grundlegenden Ansatz verwenden und die eigene Implentieren gar kein Graph-Subgraph Isomorphismus Algorithmus ist. Vielmehr geht es darum, herausfinden wie groß ungefähr der Overhead ist und wo es sich lohnt zu optimieren.

Die drei Anwendungsfälle wären:

- **Alle Finden**: Finde und gebe alle verbotenen Teilgraphen zurück
- **Alle Zählen**: Zählen wie viele verbotene Teilgraphen der Graph hat
- **Einen zurückgeben**: Finde einen verbotenen Teilgraphen und gebe ihn zurück.

In der Tabelle 2 werden die Resultate anzeigt, dabei gibt die Zeit an wie lange es gebraucht hat die Aufgabe auf allen 147 Graphen von dem Datensatz `bio1` auszuführen. In der Zeit ist die Zeit für das Einlesen der Graphen nicht mit einbegriffen. Die Werte unter der Zeit, sind Angaben, wie viel mal langsamer der Ansatz als der schnellste Ansatz ist. Zu sehen ist, dass die Naive P3-Suche"viel schneller ist, aber es ist zu bebachten, dass VFLib

Verfahren	Alle finden	Alle zählen	Einen zurückgeben
VFLib	1,73s (2,4x)	0,87s (22x)	0,0124s (77x)
Boost	3,04s (4,2x)	1,68s (42x)	0,00102s (6,4x)
Naive P3-Suche	0,73s	0,04s	0,00016s

Tabelle 2: Benchamrk

und Boost allgemeine Ansätze sind, während unse nur für den Graphen P_3 funktioniert.

Zu beobachten ist auch, dass VFLib bei dem „Alle finden“ und „Alle zählen“ deutlich schneller ist, aber beim „Einen zurückgeben“ weitaus langsamer ist. Dies hat mit dem Benchmark selbst zu tun. Weil dort die Algorithmen sehr schnell sind, wurde es 100-mal auf jedem Graphen ausgeführt. Dann wurde die Zeit durch 100 dividiert. An sich ist es auch kein Problem, aber der Unterschied liegt in der Tatsache, dass die Datenstruktur von von VFLib unveränderlich ist und wir bei jedem Aufruf einen neuen Graphen für VFLib erstellen, während bei Boost wir den Graphen nicht immer wieder neu erstellen müssen. Auch wenn man den Benchmark anders gestalten könnte, wurde er absichtlich so gestaltet, weil es in den Heuristiken eine reale Anwendung ist: Oft wird wiederholt eine Änderung gemacht und dann nach einem verbotenen Teilgraphen gesucht. Hier wäre der Boost-Ansatz also weitaus besser, weil man nach einer jeder Änderung nicht einen neuen Graphen konstruieren müsste.

Die Schlussfolgerung von diesem Benchmark ist, dass die VFLib und Boost Algorithmen schnell genug sind und höchstens eine Optimierung von „Alle zählen“ eine Option wäre, die auch vielversprechend wäre.

5 Auswertung

5.1 Backward-Ansatz

5.2 Forward-Ansatz

5.3 Grow-Reduce-Ansatz

5.4 Vergleich mit anderen Heuristiken

5.4.1 Cluster-Editing

Die 2K-Heuristik basiert auf einem Kernel für das Cluster-Editing-Problem, welches maximal 2K Knoten liefert [11]. Wenn man dort eine Bedingung für

die Reduktionsregel abschwächt, kommt eine gute Heuristik für das Cluster-Editing-Problem heraus. Dabei wird die Bedingung mit jedem Durchlauf abgeschwächt.

Algorithm 8 2K Heuristik

```
1: function SOLVE2K(g :: Gewichteter Graph)
2:     x ← 1,0
3:     while g has a P3 do
4:         for each knoten u ∈ g do
5:             if 2 · x · costClique(g, u) + x · costCut(g, u) < #(N(u)) then
6:                 for each {a, b} mit a ∈ N(u), b ∈ N(u) ∧ a ≠ b do
7:                     MERGE(a,b)
8:                 end for
9:             end if
10:        end for
11:        x ← 0, 99 · x − 0, 01
12:    end while
13:    return graph
14: end function
15: function COSTCLIQUE(graph :: Gewichteter Graph, u :: Kante)
16:    cost ← 0
17:    for each {a, b} mit a ∈ N*(u), b ∈ N*(u) ∧ {a, b} ∉ graph do
18:        cost += | w({a, b}) |
19:    end for
20:    return cost
21: end function
22: function COSTCUT(graph :: Gewichteter Graph, u :: Kante)
23:    cost ← 0
24:    for each {a, b} mit a ∈ N*(u), b ∉ N*(u) ∧ {a, b} ∈ graph do
25:        cost += w({a, b})
26:    end for
27:    return cost
28: end function
```

Tabelle 3: Vergleich der durchschnittlichen Lösungsgröße

Datensatz	Aprox2k mean	Aprox2k std	ExploredGrowReduce[1] mean	ExploredGrowReduce[1] std
bio1 [2]	**43.44**	79.81	133.59	249,62
bio2 [3]	**34.11**	19.79	113.81	166,01
duplication-divergence [4]	155.17	217.38	**92.30**	115,22
newman-watts-strogatz [5]	165.22	149,86	**152.34**	127,51
albert-barabasi [6]	324.30	316,32	**248.84**	226,87
binomial [7]	**493.17**	545,19	554.61	670,91

5.4.2 Quasi-Threshold-Editing

In [8] wurde ein neuer schneller und auch für große Graphen geeigneter Algorithmus entwickelt für das Quasi-Threshold Editing Problem. Quasi-Threshold Graphen, auch bekannt als trivial perfekte Graphen lassen sich auch als (P_4, C_4) - freie Graphen charakterisieren.

In Tabelle 4 wird die Lösungsqualität von dem Quasi-Thresold-Mover und unserem Algorithmus ExploredGrowReduce verglichen. In der Tabelle 5 wird die durschnittliche Größe der Lösungen verglichen. Der Quasi-Threschold-Mover ist unserem Ansatz weit überlegen.

Tabelle 4: Vergleich der durchschnittlichen Lösungsqualität

Datensatz	ExploredGrowReduce[8] mean	ExploredGrowReduce[8] std	Quasi-Threshold-Mover mean	Quasi-Threshold-Mover std
bio1 [9]	1.61x	1.30	**1.05x**	0.14
bio2 [10]	1.69x	1.03	**1.05x**	0.10
duplication-divergence [11]	1.42x	0.33	**1.04x**	0.05
newman-watts-strogatz [12]	1.38x	0.22	**1.06x**	0.05

Tabelle 5: Vergleich der durchschnittlichen Lösungsgröße

[1] Siehe Algorithmus 6 mit 5 Iterationen
[2] Testergebnis: 2016-04-20 17:29:08
[3] Testergebnis: 2016-04-20 17:31:40
[4] Testergebnis: 2016-04-20 17:36:40
[5] Testergebnis: 2016-04-20 17:36:57
[6] Testergebnis: 2016-04-20 17:38:17
[7] Testergebnis: 2016-04-20 18:17:08
[8] Siehe Algorithmus 6 mit Standard-Parametern
[9] Testergebnis: 2016-04-20 12:46:35
[10] Testergebnis: 2016-04-20 12:53:45
[11] Testergebnis: 2016-04-20 13:09:36
[12] Testergebnis: 2016-04-20 13:12:44

	ExploredGrowReduce		Quasi-Threshold-Mover	
Datensatz	mean	std	mean	std
bio1	52.25	150.25	**20.88**	46.51
bio2	23.12	19.91	**13.89**	9.64
duplication-divergence	73.87	117.35	**51.46**	80.23
newman-watts-strogatz	146.48	128.42	**103.38**	87.99

6 Zusammenfassung

6.1 Zukünftige Forschungsmöglichkeiten

6.1.1 Besser Sortierung von Grow-Reduce

6.1.2 Besser Konvergenzkriterium

6.1.3 Obere Schranken

6.1.4 Obere Schranken

Literatur

[1] Books about us politics. `http://networkdata.ics.uci.edu/data.php?d=polbooks`.

[2] Lada A. Adamic and Natalie Glance. The political blogosphere and the 2004 u.s. election: Divided they blog. In *Proceedings of the 3rd International Workshop on Link Discovery*, LinkKDD '05, pages 36–43, New York, NY, USA, 2005. ACM.

[3] Reka Albert and Albert-Laszlo Barabasi. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74:47, 2002.

[4] Noga Alon and Uri Stav. Hardness of edge-modification problems. *Theoretical Computer Science*, 410(47):4920–4927, 2009.

[5] Lucas Bastos, Luiz Satoru Ochi, Fábio Protti, Anand Subramanian, Ivan César Martins, and Rian Gabriel S. Pinheiro. Efficient algorithms for cluster editing. *Journal of Combinatorial Optimization*, 31(1):347–371, 2014.

[6] V. Batagelj and U. Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3):36113, 2005.

[7] Sebastian Böcker, Sebastian Briesemeister, Quang Bao Anh Bui, and Anke Truß. A fixed-parameter approach for weighted cluster editing. In *APBC*, pages 211–220. Citeseer, 2008.

[8] Ulrik Brandes, Michael Hamann, Ben Strasser, and Dorothea Wagner. Fast quasi-threshold editing. *CoRR*, abs/1504.07379, 2015.

[9] Sharon Bruckner, Falk Hüffner, and Christian Komusiewicz. A graph modification approach for finding core-periphery structures in protein interaction networks. *Algorithms for Molecular Biology*, 10:16, 2015.

[10] Leizhen Cai. Fixed-parameter tractability of graph modification problems for hereditary properties. *Information Processing Letters*, 58(4):171–176, 1996.

[11] Jianer Chen and Jie Meng. A 2k kernel for the cluster editing problem. *J. Comput. Syst. Sci.*, 78(1):211–220, 2012.

[12] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(10):1367–1372, 2004.

[13] Gilberto F de Sousa Filho, F Lucidio dos Anjos, Luiz Satoru Ochi, Fábio Protti, Rio Tinto-PB-Brazil, and Joao Pessoa-PB-Brazil. Metaheuristic grasp for the bicluster editing problem. 2012.

[14] Thomas A Feo and Mauricio GC Resende. Greedy randomized adaptive search procedures. *Journal of global optimization*, 6(2):109–133, 1995.

[15] Edgar N Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, 1959.

[16] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *PNAS*, 99(12):7821–7826, June 2002.

[17] Iaroslav Ispolatov, PL Krapivsky, and A Yuryev. Duplication-divergence model of protein interaction network. *Physical review E*, 71(6):061911, 2005.

[18] D. E. Knuth. *The Stanford GraphBase: a platform for combinatorial computing*. ACM Press New York, NY, USA, 1993.

[19] David Lusseau, Karsten Schneider, Oliver J Boisseau, Patti Haase, Elisabeth Slooten, and Steve M Dawson. The bottlenose dolphin community of doubtful sound features a large proportion of long-lasting associations. *Behavioral Ecology and Sociobiology*, 54(4):396–405, 2003.

[20] Erick Moreno-Centeno and Richard M. Karp. The implicit hitting set approach to solve combinatorial optimization problems with an application to multigenome alignment. *Operations Research*, 61(2):453–468, 2013.

[21] James Nastos. (p5 , not-p5)-free graphs. Master's thesis, University of Alberta, Spring 2006.

[22] James Nastos and Yong Gao. Familial groups in social networks. *Social Networks*, 35(3):439–450, 2013.

[23] M. E. J. Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical Review E*, 74:036104, 2006.

[24] M. E. J. Newman and D. J. Watts. Renormalization group analysis of the small-world network model. *Physics Letters A*, 263(4-6):341–346, 1999.

[25] Sven Rahmann, Tobias Wittkop, Jan Baumbach, Marcel Martin, Anke Truss, and Sebastian Böcker. Exact and heuristic algorithms for weighted cluster editing. In *Comput Syst Bioinformatics Conf*, volume 6, pages 391–401. Citeseer, 2007.

[26] Philipp Schoch. Editing to (p5, c5)-free graphs - a model for community detection? Bachelor's thesis (Studienarbeit), Karlsruher Institut für Technologie, October 2015.

[27] D. J. Watts and S. H. Strogatz. Collective dynamics of "small-world" networks. *Nature*, 393:440–442, 1998.

[28] E. S. Wolk. A note on "The comparability graph of a tree". *Proceedings of the American Mathematical Society*, 16:17–20, 1965.

[29] Wayne Zachary. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research*, 33:452–473, 1977.