Heuristiken für das Entfernen von verbotenen Teilgraphen

Paul Walger (349968)

5. Juli 2016

Technische Universität Berlin, Fakultät IV
Algorithmik und Komplexitätstheorie

Gutachter:	Prof. Dr. Rolf Niedermeier
	Prof. Dr. Stephan Kreutzer
Betreuer:	Dr. Falk Hüffner
	Dr. Christian Komusiewicz

Zusammenfassung:
Graphen so zu modifizieren, dass sie gewisse induzierte Teilgraphen nicht mehr enthalten, ist bis auf gewisse triviale Fälle NP-vollständig. Dieses Problem ist unter dem Namen $\mathcal{F}$-Free Edge Editing bekannt und hat als Eingabe einen Graphen und eine Menge von verbotenen Graphen. Für einige Mengen von verbotenen Teilgraphen wurde bereits gute Heuristiken entwickelt. In dieser Arbeit werden Heuristiken für die Lösung des Problemes erarbeitet, indem drei verschiedene Ansätze entwickelt werden, welche $\mathcal{F}$-Free Edge Editing für beliebige $\mathcal{F}$ lösen. Diese werden auf ihre Güte und Geschwindigkeit auf zufälligen und realen Datensätzen von Graphen getestet. Dafür wurde $\mathcal{F}$-Free Edge Editing mittels Binary-Integer-Programming gelöst, um optimale Lösungen zu bestimmen. Außerdem werden diese allgemeinen Heuristiken mit spezifischen Heuristiken verglichen.

Abstract:
To modify graphs in such a way, that they will not contain certain induced subgraphs is mostly NP-complete. This problem is known as $\mathcal{F}$-Free Edge Editing and gets a graph and a set of forbidden graphs and answers with the minimal amount of changes required, so that the graph will not contain the forbidden subgraphs. For certain sets auf $\mathcal{F}$ good heuristics already exists. But in this paper, we will develop 3 different heuristics for general sets of forbidden subgraphs. We will test the quality and performance of the heuristics. To make this possible, we solved $\mathcal{F}$-Free Edge Editing using binary integer programming to obtain optimal solutions. We also compare our general heuristics to other specific heuristics.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den

. . . . . .

Unterschrift

Inhaltsverzeichnis

1 Einleitung

Graphen sind eine bildliche Darstellung von Beziehungen zwischen Objekten. Damit können sie eine Vielzahl von Problemen und Szenarien modellieren [25].

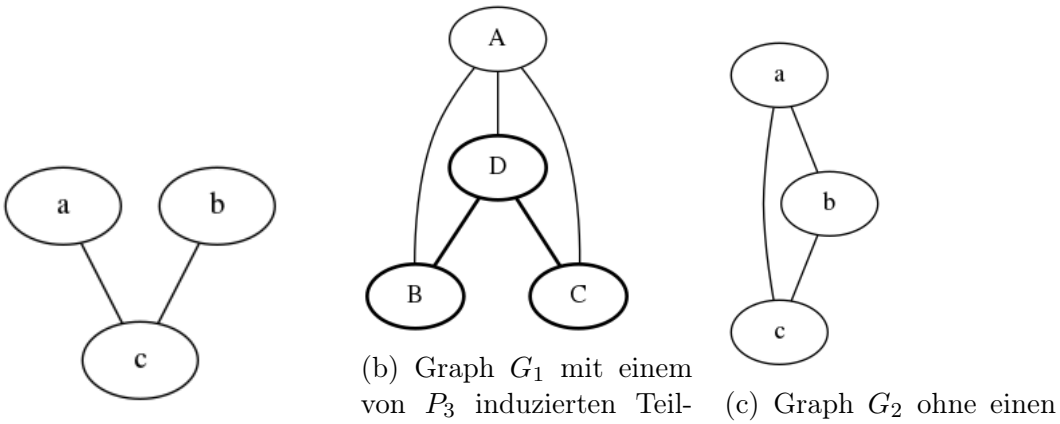
1.1 Motivation

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

Eine solche Eigenschaft wäre, dass wenn ein beliebiger Knoten im Graphen u mit einem anderen Knoten v verbunden ist, so ist u auch mit allen Nachbarn von v verbunden. Graphen mit dieser Eigenschaft 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 nicht den Graphen P_3 als einen induzierten Teilgraphen 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, dass 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, dass wenn zwei Knoten im Graphen verbunden sind, so sind auch die Knoten im P_3 verbunden.

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 vom P_3 induzierter Teilgraph ist, weil wir den Knoten a aus dem P_3 Graphen dem Knoten B im Graphen G_1, den Knoten b dem Knoten C und den Knoten c dem Knoten D zuordnen können, sodass die vorhin geforderte Eigenschaft gilt. So sehen wir, dass im Graphen P_3 die Knoten a und c verbunden sind und im G_1 auch die Knoten B und D verbunden ist. Auch ist B und C in Graphen G_1 nicht verbunden und a und b im P_3 Graphen ebenso. Man kann überprüfen, dass die Voraussetzung für jedes beliebige Paar von Knoten geben ist und so ist der fett markierte Teilgraph in G_1 ein von P_3 induzierter Teilgraph.

Der Graph G_2 in der Abbildung 1c hat keinen P_3 als induzierten Teilgraphen, weil wir nicht die Knoten vom P_3 zu Knoten in diesem Graphen zuordnen können und dabei unsere Voraussetzung erfüllt bleibt. Wenn wir also 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



(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

C haben, aber 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 vorhin geforderte Eigenschaft zu erfüllen.

Deshalb ist der Graph G_2 ein Cluster-Graph und hat eben diese besondere Eigenschaft, die auf zwei Weisen beschrieben werden kann:

- der Graph hat eine oder mehrere Komponenten wo jeder Knoten mit jedem Knoten verbunden ist
- 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 klar formulieren können.

In dieser Arbeit werden wir uns Methoden erarbeiten, welche einen Graphen so modifizieren, sodass er bestimmte Eigenschaften hat und zwar Eigenschaften, welche sich mittels einer Charakterisation durch verbotene Teilgraphen beschreiben lassen.

1.2 Definitionen

1.2.1 Notationen

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

Menge seiner Knoten und E die Menge seiner Kanten.

Die Menge der Knoten des Graphen G ist $V(G)$ und die Menge der Kanten des Graphen G heißt $E(G)$. Die Nachbarschaft von dem Knoten u heißt $N(u)$, während $N^*(u)$ die Nachbarschaft von u inklusive u ist. Ein Pfad der Länge i wird als P_i bezeichnet. Ein Kreis der Länge i wird als C_i bezeichnet.

Eine Kante k in einem Graphen G zu kippen (oder auf engl. flip) heißt, dass man die Kante aus dem Graphen G entfernt, wenn $k \in E(G)$ oder, dass man die Kante k in der Graphen G einfügt, wenn $k \notin E(G)$.

Sei $G = (V, E)$ ein Graph und $S \subseteq V$ eine beliebige Knotenteilmenge von V. Dann ist $G[S] = (S, E \cap \{\{u, v\} \mid u \in S \wedge w \in S\})$ der durch S induzierte Teilgraph von G. Die Degeneracy eines Graphen G ist das kleinste $k \in \mathcal{N}$, sodass jeder induzierte Teilgraph von G einen Knoten mit k oder weniger Nachbarn hat.

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 $(u, v) \in E_H$ gilt, genau dann wenn $(f(u), f(v)) \in E$.

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

Die Kantendifferenz $\delta(G_1, G_2) = E(G_1)\Delta E(G_2)$ zwischen zwei Graphen G_1 und G_2 ist definiert als die symmetrische Differenz der Kantenmengen der beiden Graphen[1]. Die Editierdistanz zwischen zwei Graphen definieren wir hier als $\Delta(G_1, G_2) = |\delta(G_1, G_2)|$. Die normierte Editierdistanz $\delta^*(G_1, G_2)$ zwischen zwei Graphen G_1 und G_2 ist $\Delta^*(G_1, G_2) = \frac{\Delta(G_1, G_2)}{|E(G_2)|^2}$.

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

In Tabelle 1 sind die in dieser Arbeit erwähnten Graphen-Klassen mit ihren Charakterisationen durch verbotene Teilgraphen aufgelistet.

1.2.2 Problemstellung

Das Problem, für das in dieser Arbeit Heuristiken entwickelt werden, kann wie folgt definiert werden:

> $\mathcal{F}$-Free Edge Editing
> **Eingabe:** Graph G, natürliche Zahl k, Menge von Graphen $\mathcal{F}$
> **Frage:** Können wir in G höchstens k Änderungen machen, sodass

[1] Dies ist möglich, weil wir nur Graphen vergleichen, die die selben Knoten haben

Graphklasse	**Charakterisation**
Threshold	$P_4, C_4, \overline{C_4}$
Cograph	P_4
Cluster	P_3
Split	$C_4, C_5, \overline{C_4}$
Splitcluster	C_4,C_5,P_5, Bowtie, Necktie
Quasi-Threshold	P_4, C_4
Claw-Free	$K_{1,3}$

Tabelle 1: Charakterisation einiger Graphenklassen durch verbotene Teilgraphen

G keinen induzierten Teilgraphen aus $\mathcal{F}$ enthält?
Parameter: k

In dieser Arbeit werden wir uns darauf beschränken, dass $\mathcal{F}$ eine endliche Menge ist. Dabei ist jeder Graph in $\mathcal{F}$ kein kantenloser Graph ist. Des weiteren gibt es das F-Free Edge Editing, wo F keine Menge von Graphen, sondern nur ein Graph ist.

Ähnliche Probleme:

$\mathcal{F}$-Free Edge Deletion/Completion
Eingabe: Graph G, natürliche Zahl k, Menge von Graphen $\mathcal{F}$
Frage: Können wir in G höchstens k Kanten löschen/entfernen machen, sodass G keinen induzierten Teilgraphen aus $\mathcal{F}$ enthält?
Parameter: k

$\mathcal{F}$-Free Vertex Deletion/Completion
Eingabe: Graph G, natürliche Zahl k, Menge von Graphen $\mathcal{F}$
Frage: Können wir in G höchstens k Knoten löschen/entfernen machen, sodass G keinen induzierten Teilgraphen aus $\mathcal{F}$ enthält?
Parameter: k

1.3 Anwendungsbeispiele

In diesem Abschnitt werden auf einige Anwendungsfälle eingegangen, warum Graphen mit bestimmten Eigenschaften gewünscht sind.

1.3.1 Soziale Netzwerke

Eine familiäre Struktur oder eine Gemeinschaft, die hierarchisch organisiert ist, wird durch (P_4, C_4)-freie Graphen modelliert. Diese Graphen werden auch trivial perfekte Graphen oder Quasi-Threshold Graphen genannt [33].

Die letztere Bezeichnung kommt von der Charakterisierung dieser Graphen als baumartige Strukturen. Ein solcher gerichteter Graph lässt sich als der transitive Abschluss eines Waldes sehen [9]. Dieser Graph muss gerichtet sein, weil es sonst eine Clique wird.

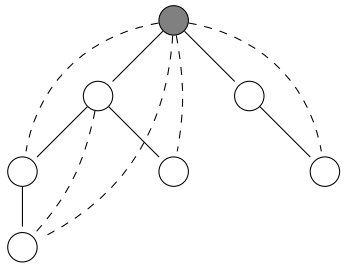


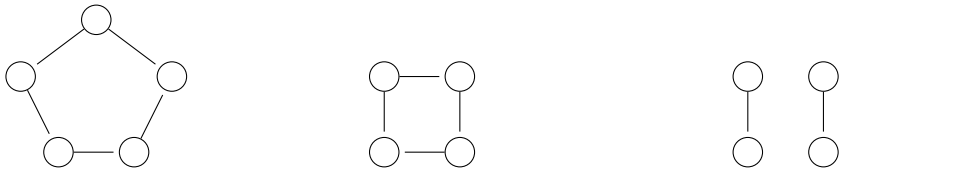
Abbildung 2: Ein Quasi-Threshold Graph

In der Abbildung 2 ist 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 Anzahl von Netzwerken. Unter anderem auch soziale Strukturen, wie eine Hierarchie in einer Organisation. Dabei modellieren 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 von dem Vorgesetzten 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[26].

Ähnlich dazu sind (P_5, C_5)-freie Graphen die auch soziale Strukturen modellieren und dafür geeignet sind Gemeinschaften zu identifizieren [31].

1.3.2 Maximum Clique auf Co-Graphen

Um das Problem der Maximum Clique, welches die größte Clique in einem Graphen findet, auf einem Graphen G zu lösen, kann man wie folgt vorgehen. Es ist offensichtlich, dass wenn der Graph zwei Zusammenhangskomponenten hat, dass es dann möglich ist Maximum Clique auf den beiden Komponenten lösen und das Maximum davon ist das Resultat für den Graphen G. Somit kann das Problem also in kleinere Probleme zerlegt werden. Außerdem gilt es, dass das Finden einer Clique in einem Graphen G äquivalent zum Finden einer stabilen Menge in dem Komplementgraphen von G ist. Es ist also möglich, das Problem weiter zu zerlegen, indem das Komple-



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

Abbildung 3: Die verbotenen Graphen in einem Split-Graphen

ment von jeder Zusammenhangskomponente genommen wird und es wieder in seine Zusammenhangskomponenten zerlegt wird. Dabei gilt, dass eine maximale stabile Menge die Vereinigung aller maximalen stabilen Mengen der Zusammenhangskomponente ist [25].

Eine solche Vorgehensweise ist offensichtlich sehr attraktiv um ein schweres Problem, wie MAXIMUM CLIQUE es ist, zu lösen. Dieses Vorgehen kommt jedoch in eine Sackgasse, wenn der Komplementgraphen nicht mehr zerlegt werden kann, weil er keine Zusammenhangskomponenten mehr hat. Dann funktioniert diese Vorgehensweise nicht mehr.

Aber es gibt eine Klasse von Graphen, die „complement reducible graphs "oder kurz Co-Graphen, die so definiert worden ist, dass es nicht zu dem eben beschriebenen Sackgasse nicht kommen kann. Das bedeutet, dass MAXIMUM CLIQUE in linearer Zeit auf solchen Graphen lösbar ist, während auf allgemeinen Graphen MAXIMUM CLIQUE NP-vollständig ist. Diese Klasse von Co-Graphen ist auch durch charakterisiert, dass sie keinen P_4 als induzierten Teilgraphen enthalten [26].

1.3.3 Protein-Interaktionsnetzwerke

In dem 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 die Menge der Knoten des Graphen in zwei Mengen V_1 und V_2 teilen kann, sodass G$[V_1]$ ein stabile Menge und G$[V_2]$ eine Clique ist. Split-Graphen lassen sich als $(2K_2, C_4, C_5)$-freie Graphen beschrieben. Dazu ist ein Splitcluster-Graph ein Graph wo jede Zusammenhangkomponente ein Split-Graph ist [10].

Diese Struktur modelliert gut Protein-Protein-Interaktionsnetzwerke. 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 Proteinkomplexe 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 interagiert. So kann man 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 Splitcluster-Graphen die Struktur von Proteinkomplexen und können dazu verwendet werden um Proteinkomplexe zu bestimmen[10].

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 Splitcluster-Graphen fehlen Informationen zu Interaktionen von 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 der Graph 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 [11], weil es NP-vollständig ist. Deswegen ist es attraktiv dieses Problem nicht optimal zu lösen, indem die minimale Anzahl von Änderungen gefunden wird, sonder einfach eine möglichst kleine Anzahl von Änderungen.

Was bringt es aber, wenn man nur eine Annäherung hat? Vorallem 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 $\mathcal{F}$ Heuristiken zu entwickeln? Oder können allgemeine Heuristiken eine guten Dienst leisten?

1.5 Ähnliche Arbeiten

1.5.1 Komplexität von $\mathcal{F}$-Free

Einige Edge Editing/Deletion/Completion Probleme wurde in [27] untersucht. Weitere Resultate zur der Komplexität wurden in [11] erbracht. Einige von den Ergebnissen sind in der Tabelle 2 dargestellt.

Für F-Free Edge Editing, F-Free Edge Completion und F-Free Edge Deletion hat [?] folgende Ergebnisse gefunden:

- F-Free Edge Deletion ist in polynomieller Zeit lösbar, wenn F ein Graph mit höchstens einer Kante ist.

Graphklasse	Edge Completion	Edge Deletion	Edge Editing
Threshold	NPC	NPC	?
Cograph	NPC	NPC	NPC
Cluster	P	NPC	NPC
Split	NPC	NPC	P
Quasi-Threshold	NPC	NPC	?
Triangle-Free	P	NPC	NPC

Tabelle 2: Komplexitäts-Ergebnisse für einige Graphenklassen. Dabei steht NPC für ein NP-vollständiges Problem, P für ein polynomiales Problem und ? für ein offenes Problem.

- F-Free Edge Completion ist in polynomieller Zeit lösbar, wenn F ein Graph mit höchstens einer Nicht-Kante ist.
- F-Free Edge Editing ist in polynomieller Zeit lösbar, wenn F ein Graph mit höchstens zwei Knoten ist.

Bezüglich der NP-Vollständigkeit hat es folgende Ergebnisse:

- F-Free Edge Deletion ist genau dann NP-vollständig, wenn F ein Graph mit mindestens zwei Kanten ist.
- F-Free Edge Completion ist genau dann NP-vollständig, wenn F ein Graph mit mindestens zwei Nicht-Kanten ist.
- F-Free Edge Editing ist genau dann NP-vollständig, wenn F ein Graph mit mindestens drei Knoten ist.

Ähnliche Fragen sind für $\mathcal{F}$-Free Edge Deletion/Completion/Editing noch offen [**?**].

1.5.2 Approximationen

Nach [12] gibt es für $\mathcal{F}$-Free Edge Editing parametrisierbare Algorithmen. Jedoch gibt es nach [22] für die meisten $\mathcal{F}$ keine polynomiale Kernel für $\mathcal{F}$-Free Edge Editing.

1.5.3 Hitting-Set

Es wurden Greedy-Alogrithmen für das Hitting-Set Problem entwickelt [24], welches sich wie folgt definieren lässt [20]: Gegeben ist ein Universum U und eine Menge Γ von Teilmengen von U. Gesucht ist eine Teilmenge H von U,

sodass jede Menge in Γ ein Element aus H enthält, dazu soll die Mächtigkeit der Menge kleiner sein als eine positive ganze Zahl k.

Wenn die Menge $\mathcal{F}$ endlich ist, dann kann das $\mathcal{F}$-Free Vertex Deletion Problem leicht als d-Hitting Set formuliert werden, wobei die Konstante d die größte Anzahl von Knoten in einem Graphen in $\mathcal{F}$ ist [22]. Damit gäbe es polynomiale Kernel für $\mathcal{F}$-Free Vertex Deletion nach [**?**]. Nach [8] ist es offen, ob es für F-Free Edge Deletion analog auch polynomiale Kernel gibt.

Es scheint eine Ähnlichkeit zu dem F-Free Edge Editing zu bestehen, aber leider lässt sich das F-Free Edge Editing nicht leicht als ein Hitting-Set-Problem formulieren. Die Schwierigkeit beim Editing Problem ist, dass auch Kanten hinzugefügt werden können. Wenn wir nun das Universum U als alle möglichen Kanten nehmen und Γ die Menge der verbotenen Teilgraphen ist, dann wäre zwar H die Menge der Änderungen, doch diese Änderungen könnten neue verbotene Teilgraphen erzeugen und somit den erzeugten Graphen nicht mehr F-frei machen.

1.6 Überblick

Im Kapitel 2 werden die Algorithmen für die Heuristiken vorgestellt. Dann werden wir im Kapitel 3 betrachten wie die Experimente aussehen und auf welchen Modellen diese Algorithmen auf ihre Güte getestet wurden und wie die Details der Implementierung aussehen. Darauf folgend werden im Kapitel 4 die Resultate vorgestellt, diskutiert und mit anderen Heuristiken 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 Teilgraphen 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, $\mathcal{F}$ 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 in den folgenden Abschnitten vorgestellt werden.

Algorithmus 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 Δ(G_input, G_best) < Δ(G_input, G_valid)| then
6:             G_best ← G_valid
7:         end if
8:     end for
9:     return δ(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 im Folgenden nur dieser Aspekt betrachtet.

In dem folgenden Abschnitt werden zwei Funktionen verwendet: findAllFS und findFS. Diese sind eine Implementation des Algorithmus zum Finden von Subgraph-Isomorphien, wie er in Abschnitt 3.3 beschrieben ist. Dabei gibt findFS(G,$\mathcal{F}$) eine Subgraph-Isomorphie f zurück, die Knoten aus dem verbotenen Teilgraphen auf Knoten aus dem Eingabegraphen G abbildet. Die Funktion findAllFS(G,$\mathcal{F}$) findet alle induzierten Teilgraphen in G für jedes $F \in \mathcal{F}$ und gibt somit eine Menge von Subgraph-Isomorphien zurück.

Dabei ist $Bild(f)$ die Menge der Knoten, die in dem Eingabegraphen den verbotenen Teilgraphen induzieren. Das Bild und der Definitionsbereich der Funktion f ist leer, wenn der Eingabegraph keinen von $\mathcal{F}$ induzieren Teilgraphen hat.

Im Algorithmus 3 sehen wir eine Funktion, die eine Graphen darauf testet, ob er $\mathcal{F}$-frei ist. Eine Funktion, die eine zufällige Kante in dem Eingabegraphen zurück gibt, wobei die Kante in dem

Algorithmus 2 Prüfung ob ein Graph valide ist

```
1: function isValid(G_input, F)
2:     if |findAllFS(G_input, F)| = 0 then
3:         return TRUE
4:     else
5:         return FALSE
6:     end if
7: end function
```

Algorithmus 3 Zufällige Kante aus dem Eingabegraphen

```
1: function randomEdge(f)
2:     u ← random node from Bild(f)
3:     v ← random node from Bild(f) \ {u}
4:     return (u, v)
5: end function
```

Die entwickelten Ansätze sind in 3 große Gruppen zu unterteilen. Der Backward-Ansatz nimmt den Eingabegraphen und ändert ihn solange, bis ein gültiger Graph entsteht. Der Forward-Ansatz fängt mit einem leeren oder vollen Graphen an, und ändert solange Knoten, bis man möglichst nahe an dem Eingabegraphen ist. Der Grow-Reduce-Ansatz kombiniert diese beiden Ansätze, indem es 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 Teilgraphen hat, dann versuche eine Kante in dem Graphen zu ändern.

Algorithmus 4 RandomFlip

```
1: function RandomFlip(G, F)
2:     while forbidden ← findFS(G, F) do
3:         edge ← randomEdge(forbidden)
4:         flip egde in G
5:     end while
6:     return G
7: end function
```

2.1.2 RandomFlipUnchanged

Der RandomFlipUnchanged-Algorithmus ist ähnlich zu dem RandomFlip-Algorithmus, aber bereits editierte Kanten werden mit einer geringeren Wahrscheinlichkeit geändert.

Der allgemeine Vorgehensweise wird in dem Algorithmus 5 beschrieben. Den großen Unterschied zu zum RandomFlip sehen wir ab Zeile 3. 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 geänderte Kante (siehe Zeile 7).

Algorithmus 5 RandomFlipUnchanged

```
1: function RandomFlipUnchanged(G, F)
2:     while forbidden ← findFS(G, F) do
3:         while true do
4:             e ← randomEdge(forbidden
5:             maxEdges ← (|Bild(forbidden)| 2)
6:             if e is already visited then
7:                 prob ← 1 / 4 · maxEdges
8:             else
9:                 prob ← 1 / maxEdges
10:            end if
11:            if random number from [0,1] > prob then
12:                flip e in G
13:                break
14:            end if
15:        end while
16:    end while
17:    return graph
18: end function
```

2.2 Forward-Ansatz

Der Forward-Ansatz 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 Teilgraphen enthält. Dies ist ein Vorteil gegenüber den Backward-Ansätzen, da es möglich ist immer einen validen Graphen zu haben und somit jederzeit terminieren.

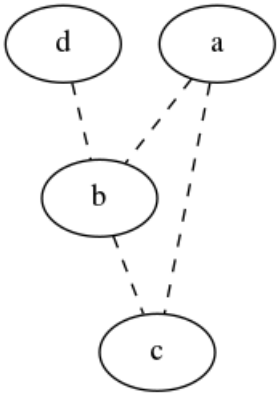
Zu sehen ist die Vorgehensweise in dem Algorithmus 6. Er fängt mit einem Graphen an, der die selben Knoten wie der Eingabegraph hat, aber keine Kanten (Zeile 2). Dann wird versucht jede Kante einzufügen, die auch im originalen Graphen G vorhanden war (Zeile 4). Wenn es einen invaliden Graphen erzeugt, alsodass 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 10) 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 G vorhanden ist, aber nicht in dem Graphen `graph`. Dabei stellen fette Kanten dar, dass sie einzufügen versucht wurde.

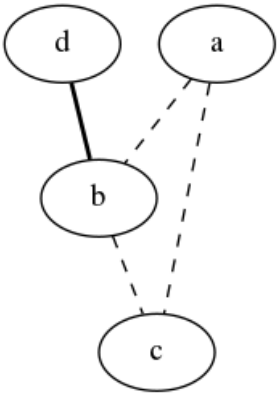
Es ist zu sehen, dass dieser Algorithmus keine Kanten hinzufügt, die nicht in dem Eingabe-Graphen vorhanden waren.

Algorithmus 6 Extend

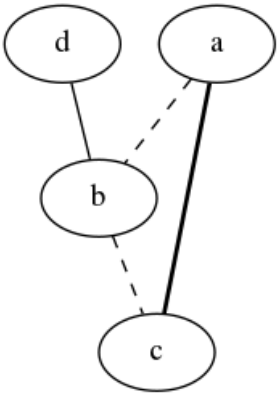
```
1: function ExtendSolve(G, F)
2:     graph ← (V(G), ∅)
3:     while true do
4:         for each Edge e ∈ δ(graph,G) do
5:             flip e in graph
6:             if not isValid(G, F) then
7:                 flip e in graph
8:             end if
9:         end for
10:        break if there was no change
11:    end while
12:    return graph
13: end function
```



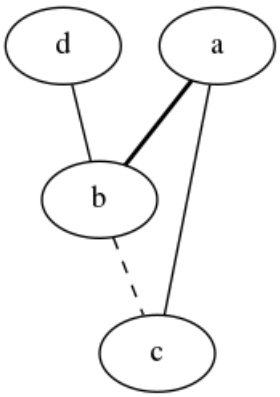
(a) Startgraph



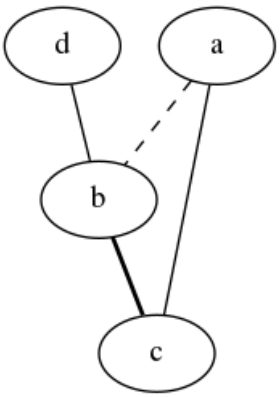
(b) Kante (d,b) wird erfolgreich hinzugefügt



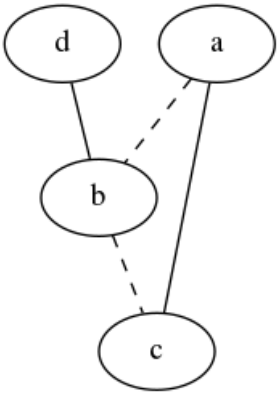
(c) Kante (a,c) erfolgreich wird hinzugefügt



(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-Algorithmus

2.3 GRASP-Ansatz

Nach dem experimentellen Untersuchen der Forward und Backward-Ansätze wurden deren Stärken und Schwächen deutlich. So entstand auch die Idee diese beiden Ansätze zu kombinieren. Nach dem Planen und der Umsetzung einer solchen kombinierten Heuristik, hat die Recherche ergeben, dass eine sehr ähnliche Art von Algorithmen bereits beschrieben wurde und unter dem dem Namen GRASP (Greedy Randomized Adaptive Search Procedure) [16, 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 es werden iterativ 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 bereits erfolgreich für das Biclustering-Problem verwendet [15].

Unsere Ansätze unterscheidet sich darin von den GRASP-Algorithmen, dass wir nicht in jeder Iteration einen neuen Graphen konstruieren, sondern auf dem bereits konstruierten 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 Nachbarschaft hinzugefügt und durch lokale Suche werden alle neu entstandenen verbotenen Teilgraphen wieder entfernt. Dies ist im Algorithmus 7 zu sehen. Die Grow-Phase ist mit der Konstruktion-Phase des GRASP-Ansatzes vergleichbar und die Reduce-Phase mit der lokalen Suche.

Sortierung:
Sowohl in Grow-Reduce als auch in Explorerd-Grow-Reduce Algo 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 Sortierfunktionen verwendet:

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

Bei den letzten beiden Sortiergmöglichkeiten gibt es sowohl die aufsteigende als auch die absteigende Variante.

Lokale Suche: Bei der lokalen Suche oder der Reduce-Phase wird ein verbotener Teilgraph gesucht und eine zufällige Kante in diesem geändert. Dabei wird ein Kovergenzkriterium verwendet, welches besagt, dass nach der Änderung der Anzahl der verbotenen Teilgraphen kleiner werden soll. Wenn dieses nicht erfüllt wird, dann wird die Änderung wieder rückgängig gemacht.

Zeitlimit: Da bei der Entwicklung festgestellt wurde, dass eine Iteration duch alle Knoten zeitlich sehr umfangreich ist, wurde die Tatsache ausgenutzt, dass in jeder Iteration wir einen validen Graphen haben. Das bedeutet, dass man nach jeder Iteration abbrechen und den Graphen zurückgeben könnte. Dies wäre jedoch suboptimal, deswegen wird der schnelle Extend-Algorithmus vor der Rückgabe der Graphs nochmal auf den Graphen angewandt und dann erst wird der Graph zurückgegeben. Die Entscheidung, nach welcher Iteration abzubrechen ist, hängt von dem Zeitlimit ab, wie lange das Programm zu laufen hat.

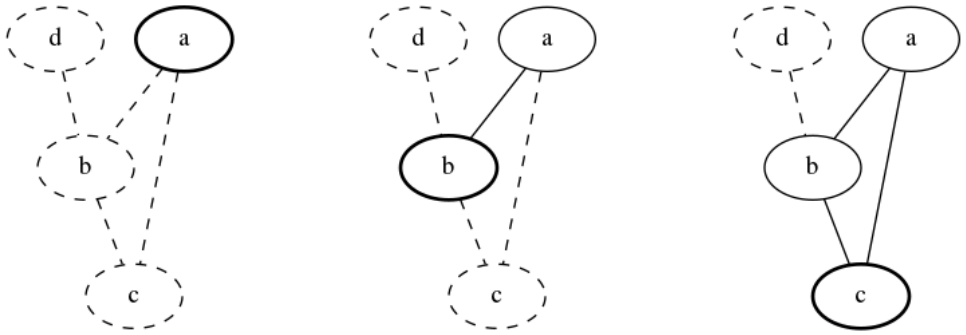
2.3.2 Explored-Grow-Reduce

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 visualisiert wurden, ohne die Reduce-Phase, um die Erklärung zu vereinfachen. In der Abbildung 5a ist der Anfangstatus zu sehen. Die gestrichelten Kanten sind Kanten, die im Eingabegraphen vorhanden sind, aber noch nicht hinzugefügt worden sind. In der Abbildung 5a wird nun der Knoten a hinzugefügt, weil aber keine anderen Knoten bisher hinzugefügt worden sind, werden auch keine Kanten hinzugefügt. In Abbildung 5b wird der Knoten b hinzugefügt und weil a auch schon hinzugefügt wurde, wird auch die

Algorithmus 7 GrowReduce

```
1: function GrowReduceSolve(G, F)
2:     graph ← (V(G), ∅)
3:     nodes ← order(V(G))
4:     for node ∈ nodes do
5:         for neighbor ∈ N(node) do                          ▷ Grow
6:             Add edge (node, neighbor) to graph
7:         end for
8:         while forbidden ← findFS(graph, F) do              ▷ Reduce
9:             edge ← randomEdge(forbidden)
10:            count ← |(findAllFS(graph,F)|
11:            flip edge in graph
12:            countAfter ← |(findAllFS(graph,F)|
13:            if countAfter ≥ count then
14:                flip edge in graph
15:            end if
16:        end while
17:    end for
18:    return solveExtend(graph, F)
19: end function
```



(a) Knoten a wird hinzugefügt (b) Knoten b wird hinzugefügt (c) Knoten c wird hinzugefügt

Abbildung 5: Beispielweise Grow-Phase

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

Im Grow-Reduce-Ansatz werden im ersten Schritt auch die Kanten zu b und c hinzugefügt werden. Dies wäre im Beispiel vom P_3 als verbotenen Teilgraphen aber suboptimal, denn dann wäre der Teilgraph mit den Knoten a,b und c ein vom P_3 induzierter Teilgraph.

2.4 Ganzzahlige lineare Optimierung

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

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 maximieren

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

Dies ist die Zielfunktion `objective` im Algorithmus 9.

Da alle möglichen Bedingungen hinzuzufügen, welche alle verbotenen Teilgraphen ausschließen würden, viel zum umfangreich wäre, werden die Bedingungen iterativ dort hinzugefügt, wo es einen verbotenen Teilgraphen gibt. Dann wird das Problem mittels Binary-Linear-Programming gelöst und die Änderungen auf den Graphen übertragen. Dann wird wieder nach allen ver-

Algorithmus 8 ExploredGrowReduce

```
1: function ExploredGrowReduceSolve(G, F)
2:     graph ← (V(G), ∅)
3:     nodes ← order(V(G))
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:                                                           ▷ Reduce Phase
12:        while forbidden ← findFS(graph, F) do
13:            edge ← randomEdge((forbidden)
14:            count ← |(findAllFS(graph,F)|
15:            flip edge in graph
16:            countAfter ← |(findAllFS(graph,F)|
17:            if countAfter ≥ count then
18:                flip edge in graph
19:            end if
20:        end while
21:    end for
22:    return solveExtend(graph, F)
23: end function
```

botenen Teilgraphen gesucht. Dies wird solange wiederholt bis es keine mehr gibt. Damit ist die minimale Anzahl von Änderungen gefunden.

Algorithmus 9 F-Free BIP

```
1: function solveBIP(G, F)
2:     constraints ← ∅
3:     while not isValid(G, F) do
4:         for f ∈ findAllFS(G, F) do
5:             c ← 0
6:             for each {u, v} mit u ∈ Bild(f) ∧ v ∈ Bild(f) ∧ u ≠ v do
7:                 if {u, v} ∈ E(G) then
8:                     c += 1 - e_uv
9:                 else
10:                    c += e_uv
11:                end if
12:            end for
13:            constraints ← constraints ∪ { c }
14:        end for
15:        variables ← BIPSolve(constraints, objective)
16:        for e_u,v ∈ variables do          ▷ Apply solution to the graph.
17:            if e_u,v = 1 then
18:                Set edge (u, v) in G
19:            else
20:                Remove edge (u, v) in G
21:            end if
22:        end for
23:    end while
24:    return G
25: end function
```

3 Aufbau der Experimente

3.1 Datensätze

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 verschiedene Methoden zur Erzeugung von zufälligen Graphen. Insgesamt wurde 5 verschiedene Methoden zur Erzeugung von zufälligen Graphen verwendet und 3 Datensätze. Wir brachten zuerst die zufälligen Graphen und dann einige Graphen basierend auf realen Modellen.

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 Knoten, 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[17, 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[19], 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

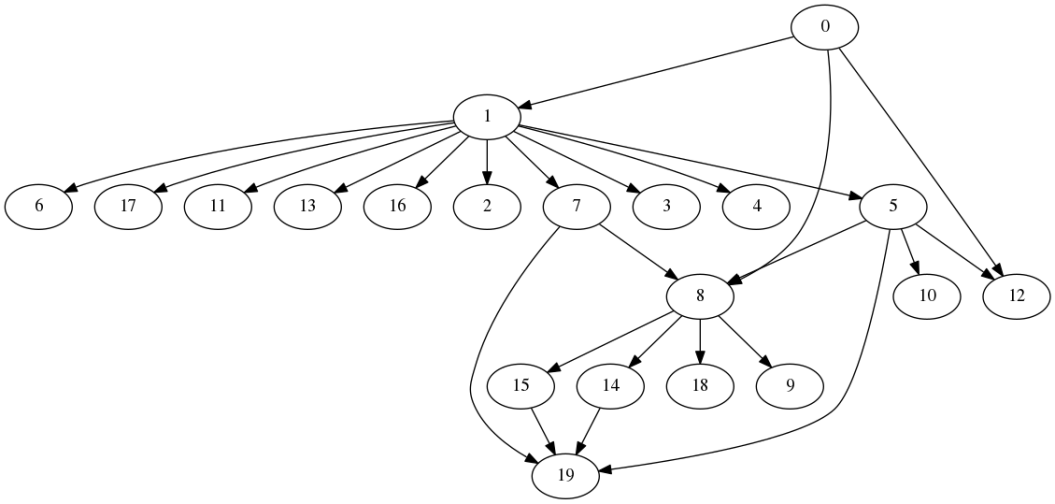


Abbildung 6: Ein beispielhafter Duplication-Divergence Graph mit $n = 20$ und $p = 0{,}4$

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 folgende 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[29], welches Kleine-Welt-Graphen erzeugt mit einer geringen durchschnittlichen Knotendistanz und einem hohen Clusterkoeffizienten.

Dabei wird zuerst ein Kreis von n Knoten erstellt. Dann wird jeder Knoten mit k von seinen nächsten Nachbarn verbunden (oder mit $k-1$, wenn k ungerade ist). Dann werden Abkürzungen erzeugt, indem man für jede Kante (u, v) in dem zugrunde liegenden n-Kreis mit den k nächsten Nachbarn, folgendes tut: 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 $n \in \{10, 20, \ldots, 90\}$, $k \in \{2, 4, 6, 8\}$ und der Wahrscheinlichkeit $p \in \{\frac{2}{10}, \frac{4}{10}, \frac{6}{10}, \frac{8}{10}\}$.

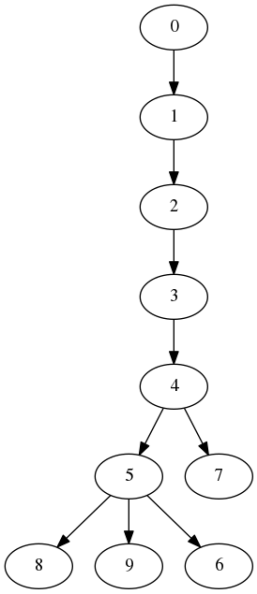


Abbildung 7: Ein Powerlaw-Baum mit $n = 10$

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 [34].
- Der Graph `polbooks` 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 [18].

- Der Graph `power` ist ein Netzwerk, dass die Topologie des "Western States Power Gridïn der Vereinigten Staaten wiederspieglt [32].
- Der Graph `adjnoun` ist ein Netzwerk von häufigen Adjektiven und Nomen in dem Roman "David Copperfield"von Charles Dickens [28].
- Der Graph `lesmiserables` ist ein Netwerke von Figuren, die in dem Roman Les Misérables"von Victor Hugo, zur gleichen Zeit auftreten [21].
- Der Graph `celegansneural` welches das neurale Netzwerk von Caenorhabditis elegans [32]. 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 [23].
- 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 Graphen die Squenzähnlichkeit von Proteinen modellieren [30, 7].

3.2 Details der Experimente

Um nachvollziehbare, fehlerarme und detaillierte Resultate der Experimente zu erhalten, wurde eine Umgebung implementiert, die automatisch die Experimente laufen lässt, die Daten sammelt und diese auswertet.

Dafür werden Experimente in einer Konfigurationsdatei definiert, indem folgende Informationen dort aufgelistet werden: Datensätze, $\mathcal{F}$, Algorithmen und die maximale Laufzeit. Diese Information wird genutzt, um dann die Experimente automatisch laufen zu lassen. Dabei wird bei jedem Experiment unter anderem die Laufzeit, der maximaler Speicherverbrauch, die Anzahl der Änderungen, der resultierende Graph und Informationen, um das Verhalten des Algorithmus nach zu vollziehen, gesammelt.

Um die Qualität der Lösung eines heuristischen Ansatzes bewerten zu können, ist es notwendig die optimale Lösung zu wissen. Es gibt verschiedene Ansätze wie das Problem zu lösen sein, wir haben uns jedoch für die binäre lineare Optimierung entschieden. Der BIP-Algorithmus versucht eine

optimale Lösung zu finden. Dies kann jedoch sehr lange dauern, da das Problem NP-vollständig ist. In der Tabelle 3 sehen wir, dass für einige verbotene Teilgraphen wir für alle Instanzen eine optimale Lösung berechnen konnten, aber es gibt schwierige verbotene Teilgraphen wie (P_5, triangle) wo wir für die meisten Instanzen keine optimale Lösung berechnen konnten.

Tabelle 3: Nicht gelöste Instanzen für den Datensatz `bio2`

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%

Des weiteren wurden für jeden Graphen in den Datensätzen folgende Variablen bestimmt, die Anzahl der Knoten, die Anzahl der Kanten, die minimale / maximale / durchschnittliche Anzahl von Nachbarn, der minimale / maximale / durchschnittliche Clustering-Koeffzient, Degeneracy und Dichte.

Die Resultate der Experimente kann man mittels einer webbasierten Benutzeroberfläche interaktiv untersuchen. So ist es beispielsweise möglich die Auswirkung der Dichte der Graphen auf die Laufzeit in einem Linendiagramm darstellen zu lassen. Oder die Auswirkung der der durchschnittlichen Dichte auf die normierte Lösungsgröße. Diese Benutzeroberfläche für Resultate von den Experimenten ist unter `http://metaxy.github.io/f-free-browser-data/` zu finden.

3.3 Implementierungsdetails

3.3.1 Allgemein

Die Algorithmen wurden in der Sprache C++14 implementiert. Für die Repräsentation der Graphen wurde eine einfache Adjazenzmatrix verwendet. Der Code ist unter [**?**] zu findenFür das Binary-Integer-Programming wurde Gurobi 6.5 verwendet [**?**].

Für das Finden von verboten Teilgraphen haben wir Bibliotheken verwendet, die das Graphen-Subgraph Isomorphismus Problem implementieren. Da das Finden von induzierten Teilgraphen der zeitaufwendigste Teil der Algorithmen ist, ist hier die Wahl von dem richtigen Ansatz sehr wichtig. Hier

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 P_3-Suche	0,73s	0,04s	0,00016s

Tabelle 4: Laufzeitvergleich von VFLib, Boost und Naive P3-Suche

werden drei Ansätze bezüglich ihrer Geschwindigkeit verglichen, wobei der P_3 der verbotene Teilgraph ist.

- Der Ansatz erste ist VFLib. Es ist ein Bibliothek, die mehrere VF2-Algorithmen [14] jeweils für das Graphen-Isomorphismus-Problem, Graphen-Subgraph-Isomorphismus-Problem und das Graphen-Monomorphimus Problem-implementiert und zählt zu den schnellsten Implementationen.
- 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 ob eine Optimierung lohnenswert wäre.

Die drei unteschiedlichen Anwendungsfälle sind:

- **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 Tabelle 4 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 P_3-Suche"viel schneller ist, aber es ist auch zu beachten, dass VFLib und Boost

allgemeine Ansätze sind, während die naive P_3-Suche 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 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 vielversprechende Option wäre.

So auch wurde bei der Implementierung der Algorithmen sowohl VFLib als auch Boost verwendet. Wenn es darum ging immer wieder einen verbotenen Graphen zurückzugeben, dann wurde Boost verwendet und es darum ging vor allem häufig die Anzahl der verbotenen Teilgraphen zählen oder alle verbotenen Teilgraphen zu zählen, dann wurde VFLib verwendet.

$\mathcal{F}$	RandomFlip	RandomFlipUnchanged
Splitcluster	253	17
Cluster	293	54
Co-Graph	251	12
x172 triangle	320	31
Quasi-Threshold	250	15
Split	254	10
K_{14} Paw	266	31

Tabelle 5: Anzahl nicht gelöster Instanzen auf dem `bio2` Datensatz

4 Auswertung

In diesem Abschnitt werden wir die Auswertung der Tests vornehmen, indem wir zuerst die Algorithmen betrachten, welche einige Instanzen nicht lösen konnten. Dann werden wir die Lösungsqualität und den Zeitaufwand der Heuristiken untersuchen. Zum Schluss wird die Wahl der Sortierfunktion für die Grow-Reduce-Ansätze untersucht. Die Experimente wurden auf einem Intel 3Ghz Dual-Core (BX80637G2030) mit 8 GB RAM ausgeführt. Die Resultate der Tests sind unter [**?**] zu finden. Auch können sie interaktiv unter folgender URL untersucht werden: `http://metaxy.github.io/f-free-browser-data/`.

4.1 Nicht gelöste Instanzen

Die beiden Backward-Ansätze RandomFlip und RandomFlipUnchanged haben im Gegensatz zu den anderen Algorithmen viele Instanzen, die sie in der vorgegebenen Zeit nicht lösen konnten. In der Tabelle 5 ist die Anzahl der nicht gelöste Instanzen auf den `bio2` Daten bei einer maximalen Laufzeit von 10s zu sehen. Zu beachten ist, dass der Datensatz `bio2` nur 350 Graphen hat.

Die Ursache für die hohe Anzahl der ungelösten Instanzen bei RandomFlip liegt darin, dass bei RandomFlip der Algorithmus nicht nach kurzer Zeit terminieren muss. Es wird jedes Mal eine zufällige Kante/Nicht-Kante aus dem ersten gefundenen verbotenen Teilgraphen genommen. So kann es sein, dass wenn die minimale Anzahl der Änderungen am Graphen, um ihn $\mathcal{F}$-frei zu machen, sehr hoch ist. Dann passiert es, dass der Algorithmus lange braucht um überhaupt eine Lösung zu finden. So ist in der Abbildung 8 zu sehen, dass mit größerer Knotenanzahl immer mehr Instanzen von RandomFlip nicht gelöst werden.

So ist RandomFlipUnchanged deutlich besser, schafft es bei einer größeren Knotenanzahl nicht alle Instanzen zu lösen. Die Idee, dass man eine einmal

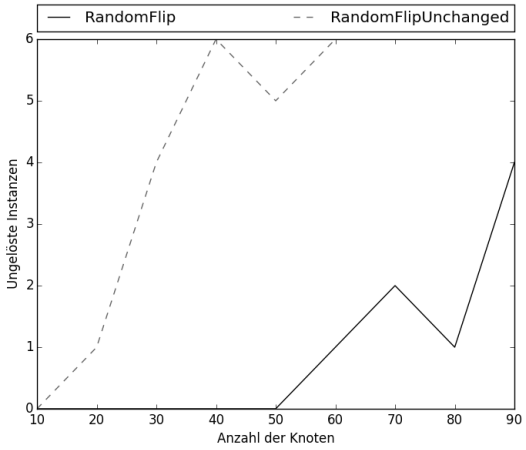


Abbildung 8: Anzahl der nicht gelösten Instanzen auf dem `duplication_divergence` Datensatz für Splitcluster

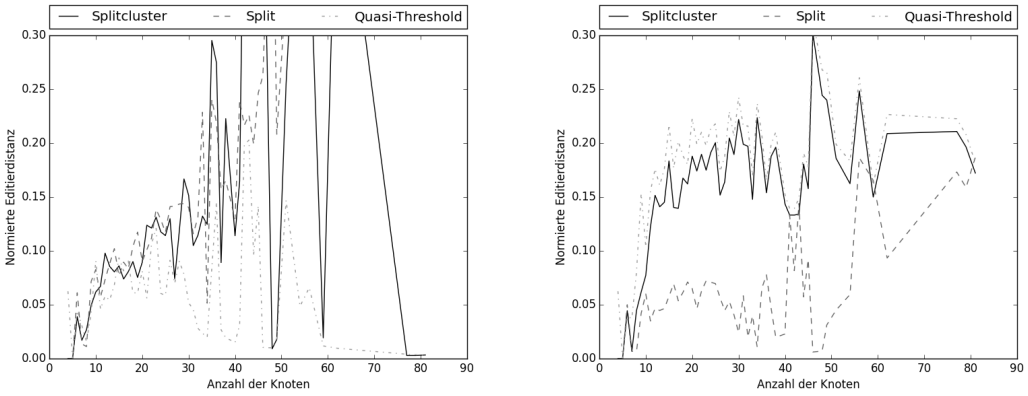
geänderte Kante mit einer geringer Wahrscheinlichkeit wieder ändert, hat also den RandomFlip-Algorithmus verbessert. Damit sind die Algorithmen RandomFlip und RandomFlipUnchanged sehr begrenzt nützlich und zwar nur für kleine Graphen.

4.2 Lösungsqualität

In der Abbildung 9 sind die durchschnittlichen normierten Lösungsgrößen der Algorithmen auf dem `bio2`-Datensatz dargestellt. RandomFlip wurde nicht einbezogen, weil es so viele ungelöste Instanzen hat. Dabei wurden ungelöste Instanzen mit der normierten Lösungsgröße von $0,5$ eingerechnet. Der Extend-Algorithmus kann bei kleinen Graphen nicht mal mit dem RandomFlipUnchanged-Algorithmus mithalten, außer bei den Split-Graphen. Wenn man Abbildung 9c mit 9d vergleicht, ist zu sehen, dass der ExploredGrowReduce-Ansatz dem GrowReduce-Ansatz bezüglich der Lösungsgröße überlegen ist. Alles in allem ist der ExploredGrowReduce-Algorithmus bezüglich der Lösungsqualität der Beste.

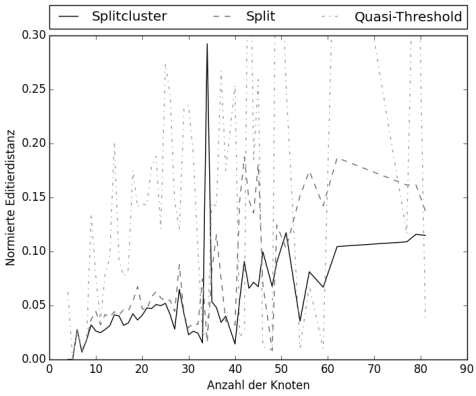
4.3 Laufzeit

In der Abbildung 10 sind zwei Linieniendiagramme, welche die Laufzeit von Extend, GrowReduce und ExploredGrowReduce darstellen. Zu sehen ist, dass sowohl mit der Kantenanzahl, als auch mit der Knotenanzahl die Laufzeit wächst. Dabei ist aber zu beachten, dass sowohl GrowReduce als auch ExploredGrowReduce ein Abbruchkriterium haben. Das heißt, dass wenn sie

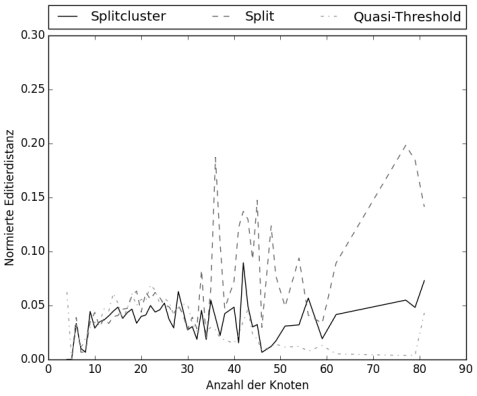


(a) RandomFlipUnchanged

(b) Extend

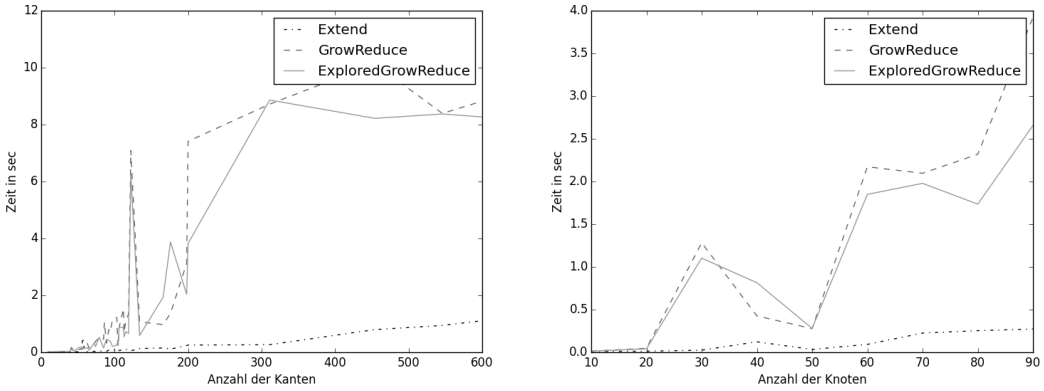


(c) GrowReduce



(d) ExploredGrowReduce

Abbildung 9: Durchschnittlich normierte Lösungsgröße der Algorithmen auf dem `bio2`-Datensatz



(a) Kantenanzahl-Zeit-Diagramm (b) Knotenanzahl-Zeit-Diagramm

Abbildung 10: Durchschnittliche Laufzeit von Extend, GrowReduce und ExploredGrowReduce auf dem `duplication_divergence`-Datensatz für Split-cluster

es nicht schaffen durch alle Knoten zu iterieren, dass sie vorzeitig abbrechen und dann noch den Extend-Algorithmus laufen lassen und dann das Ergebnis zurückgeben. In diesem Experiment war das Zeitlimit 10 Sekunden.

Zu sehen ist auch, dass die Kantenanzahl einen größeren Effekt auf die Laufzeit hat als die Knotenanzahl. Auch ist der Extend-Algorithmus wesentlich schneller als die Grow-Reduce-Ansätze.

4.4 Sortierungsverfahren

In der Tabelle 6 ist zu sehen, dass die Wahl der Sortierfunktion relevant ist für die Lösungsgröße. Es wurden 4 Datensätze ausgewählt, um zu zeigen, dass die Sortierfunktion erheblichen Einfluss hat und dies hängt auch von dem $\mathcal{F}$ und dem Datensatz ab. So ist zu erkennen, dass auf dem `bio2`-Datensatz bei unterschiedlichen Sortierfunktionen am besten sind. So ist bei Split-Cluster-Graphen die Hits-Sortierfunktion die Beste und bei Split-Graphen die absteigende Nachbar-Sortierfunktion die Beste. Auch ist die beste Sortierungsfunktion abhängig vom Datensatz, auf dem `bio1` ist die absteigende Nachbar-Funtion und auf dem `bio2` die aufsteigende Hits-Funktion. Auch ist interessant, dass bei (X_{172}, Triangle), die Random-Funktion und die auf- und absteigenden Nachbar-Funktionen ähnlich schlecht sind, während Hits-Funktionen sowohl absteigend als auch aufsteigend weitaus besser sind. Hinzufügen ist hier aber, dass die Hits Sortierungsfunktionen erheblich langsamer sind. Diese Zusammenhänge würde sich lohnen genauer zu untersuchen,

$\mathcal{F}$	Datensatz	R	N ASC	N DESC	H ASC	H DESC
Splitcluster	`bio2`	37,5	34,0	28,7	**26,6**	44,6
($2K_2$, C_4, C_5)	`bio2`	50,2	67,1	**32,8**	45,0	55,8
Splitcluster	`bio1`	87,6	85,8	**40,7**	67,4	93,2
(X_{172}, Triangle)	`barabasi`	234,0	232,4	237,2	157,5	**154,3**

Tabelle 6: Durchschnittliche Größe der Lösung beim ExploredGrowReduce in Abhängigkeit von der Sortierungsfunktion. Dabei steht R für die zufällige Sortierung, N die Sortierung nach der Anzahl der Kanten, H die Sortierung nach der Anzahl von verbotenen Teilgraphen. ASC steht für aufsteiegend und DESC für absteigend.

waren jedoch im Umfang dieser Arbeit nicht möglich.

4.5 Vergleich mit anderen Heuristiken

In diesem Abschnitt, wird ExploredGrowReduce mit zwei verschiedenen speziellen Heuristiken verglichen. Zuerst wird mit einer Heuristik für das Cluster Editing verglichen und dann mit einer Heuristik für das Quasi-Threshold Editing.

4.5.1 Cluster-Editing

Die 2K-Heuristik basiert auf einem Kernel für das Cluster-Editing-Problem, welches maximal 2K Knoten liefert [13]. 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. Dabei wurde ein gewichteter Graph verwendet, bei dem es möglich ist zwei Knoten zu vereinigen und durch die entsprechende Anpassung der Gewichte die Strukturinformation beizubehalten. Das Vorgehen ist in dem Algorithmus 10 zu sehen. Für einen Knoten u sind folgende Funktionen definiert, wobei $w(u, v)$ die Gewichtung der Kante (u, v) in dem Graphen ist:

$$costClique(u) = \sum_{\{a,b\}|a\in N^*(u),b\in N^*(u),\{a,b\}\notin E(G)} |\, w(\{a,b\})\,| \tag{1}$$

$$costCut(u) = \sum_{\{a,b\}|a\in N^*(u),b\notin N^*(u),\{a,b\}\in E(G)} w(\{a,b\}) \tag{2}$$

Algorithmus 10 2K-Heuristik

```
1: function SOLVE2K(G)
2:     x ← 1,0
3:     while G has a P3 do
4:         for each knoten u ∈ E(G) do
5:             if 2 · x · costClique(u) + x · costCut(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 G
14: end function
```

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

Tabelle 7: Vergleich der durchschnittlichen Lösungsgröße

In der Tabelle 7 wird die Lösungsgröße zwischen dem 2K-Algorithmus und dem ExploredGrowReduce verglichen. Zu sehen ist, dass der 2K-Algorithmus auf den realen Daten signifikant besser ist.

4.5.2 Quasi-Threshold Editing

In [9] wurde ein neuer schneller und auch für große Graphen geeigneter Algorithmus entwickelt für das Quasi-Threshold Edge Editing Problem. Quasi-Threshold Graphen, auch bekannt als trivial-perfekte Graphen. Diese lassen sich auch als (P_4, C_4)-freie Graphen charakterisieren.

In Tabelle 8 wird die Lösungsqualität von dem Quasi-Threshold-Mover und unserem Algorithmus ExploredGrowReduce verglichen. Da für den Vergleich der Lösungsqualität die minimale Distanz zum $\mathcal{F}$-freien Graphen be-

Graph	**ExploredGrowReduce**	**Quasi-Threshold-Mover**
`karate`	26	21
`polbooks`	363	226
`power`	5071	2806
`football`	372	259
`lesmiserables`	147	60
`adjnoun`	391	283
`celegansneural`	2035	1499
`dolphins`	103	73
`polblogs`	-	12607

Tabelle 10: Vergleich von der Lösungsgröße auf dem Datensatz `UCINetworkDataRepository`

kannt sein muss, wird in der Tabelle 9 die durchschnittliche Größe der Lösungen verglichen, um auszuschließen, dass der Quasi-Threshold-Mover nur auf den einfachen Instanzen besser ist. Zu sehen ist, dass der der Quasi-Threshold-Mover um 70% konsistent besser löst.

Tabelle 8: Vergleich der durchschnittlichen Lösungsqualität

	ExploredGrowReduce[3]		Quasi-Threshold-Mover	
Datensatz	mean	std	mean	std
bio1	1,61x	1,30	**1,05x**	0,14
bio2	1,69x	1,03	**1,05x**	0,10
duplication-divergence	1,42x	0,33	**1,04x**	0,05
newman-watts-strogatz	1,38x	0,22	**1,06x**	0,05

Tabelle 9: Vergleich der durchschnittlichen Lösungsgröße

	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

[3]Siehe Algorithmus 8 mit Standard-Parametern

5 Zusammenfassung

In dieser Arbeit wurde gezeigt, dass es möglich ist, gute Heuristiken für $\mathcal{F}$-Free Edge Editing zu finden. Dafür wurden 3 Ansätze entwickelt. Der Forward-Ansatz modifiziert den Graphen so lange bis er $\mathcal{F}$-frei ist. Der Backward-Ansatz baut den Graphen aus einem leeren Graphen auf, bis er möglichst nahe an dem Eingabegraphen ist und dabei $\mathcal{F}$-frei bleibt. Der Grow-Reduce-Ansatz kombiniert diese beiden Ansatz und baut iterativ den Graphen auf und dann modifiziert ihn durch lokale Suche, sodass er $\mathcal{F}$-frei wird. Der Forward-Ansatz zeichnet sich durch gute Lösungen für kleine Graphen aus, während der Backward-Ansatz sich durch eine schnelle Laufzeit auszeichnet. Die besten Lösungen liefern aber der ExploreGrowReduce-Algorithmus, ist jedoch auf großen Graphen nicht sehr schnell. Der Vergleich mit anderen Heuristiken hat ergeben, dass die speziellen Heuristiken bessere Ergebnisse liefern, aber diese Lösungen durchschnittlich nur um 70% besser sind.

5.1 Ausblick

Besser Sortierung von Grow-Reduce
Ein wichtiges Detail bei der Qualität der Lösungen von GrowReduce und ExploredGrowReduce ist die Sortierungsfunktion der Knoten am Anfang. In dieser Arbeit wurden drei sehr einfache Sortierungsfunktionen verwendet. Man könnte untersuchen, ob es bessere allgemeine Sortierfunktionen gibt. Auch wären spezielle Sortierungsfunktionen, die vom $\mathcal{F}$ abhängen, sehr interessant.

Obere und untere Schranken
Obere und untere Schranken für die Lösungsqualität zu untersuchen, wäre interessant.

Heuristiken für Subgraph-Isomorphismen
Da viel Zeit bei der Suche nach den verbotenen Teilgraphen investiert wird, wäre zuerst eine Vereinfachung der Graphens durch das schnelle Lösen mittels der Verwendungen einer Heuristik für die Suche für die Teilgraphen [**?**] optimal. Und dann könnte man mittels einen von diesen hier vorgestellten Algorithmen eine Lösung endgültig bekommen, wo sichergestellt ist, dass der resultierende Graph $\mathcal{F}$-frei ist. Dies hätte vor allem einen Vorteil bei großen Graphen, wo die hier vorgestellten Algorithmen langsam sind.

Besseres Konvergenzkriterium

In einigen der Algorithmen wurden Konvergenzkriterien verwendet, um zu entscheiden ob eine Änderungen der Kante sinnvoll ist oder nicht. Dies wurde so implementiert, dass die Anzahl der verbotenen Teilgraphen vor und nach der Änderung gezählt wurde, was jedoch sehr zeitaufwändig ist. Man könnte es verbessern, indem man auch hier eine Abschätzung oder eine Heuristik für das Subgraphen-Isomorphismen Problem verwendet.

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] Hans L Bodlaender, Leizhen Cai, Jianer Chen, Michael R Fellows, Jan Arne Telle, and Dániel Marx. Open problems in parameterized and exact computation-iwpec 2006. *UU-CS*, 2006, 2006.

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

[10] 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.

[11] Pablo Burzyn, Flavia Bonomo, and Guillermo Durán. Np-completeness results for edge modification problems. *Discrete Applied Mathematics*, 154(13):1824–1844, 2006.

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

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

[14] 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.

[15] 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.

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

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

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

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

[20] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

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

[22] Stefan Kratsch and Magnus Wahlström. Two edge modification problems without polynomial kernels. *Discrete Optimization*, 10(3):193–199, 2013.

[23] 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.

[24] 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.

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

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

[27] A. Natanzon, R. Shamir1, and R. Sharan2. Complexity classification of some edge modification problems. *DISCRETE APPLIED MATHEMATICS*, 113(1):109–128, 2001.

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

[29] 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.

[30] 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.

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

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

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

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