Heuristiken für das Entfernen von verbotenen Teilgraphen

Paul Walger

21. April 2016

Inhaltsverzeichnis

1 Einleitung

Man kann einen Graphen als eine bildliche Darstellung von Beziehungen zwischen Objekten vorstellen. Dabei stellt man sie als Kreise (auch Knoten genannt) mit einem Namen dar und die Beziehungen als als Linien die diese Kreise verbinden (auch Kanten genannt). [20] Eine genauere Definition ist im Abschnitt 1.3.1 zu finden.

Diese Graphen können eine Vielzahl von Problemen und Szenarien modellieren. Zum Beispiel könnten sie Knoten Städte und die Kanten Verbindungen zwischen diesen sein und somit wird der Graph das Netzwerk von diesen Städten modellieren. Weitere Bespiele sind im dem Abschnitt 1.2 zu finden.

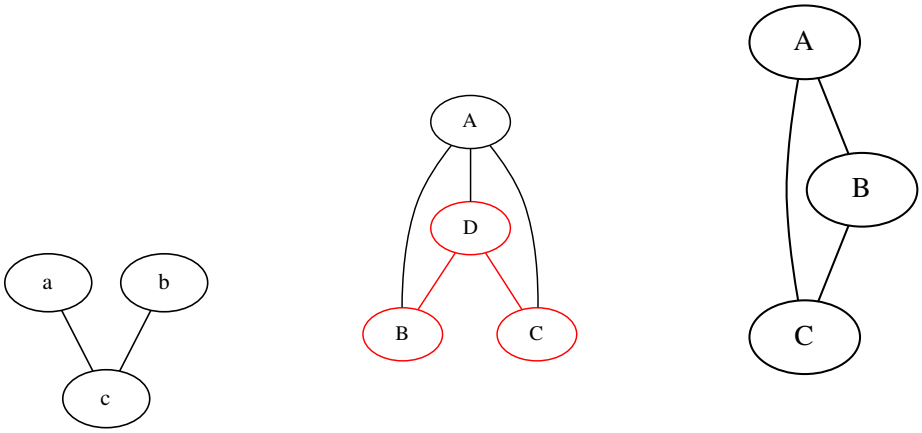
In diesem Abschnitt werden wir die Fundamente legen und diese Arbeit motivieren. Im Abschnitt 2 werden die Ansätze vorstellt um dieses Problem zu lösen. Dann werden wir im Abschnitt 3 betrachten wie die Tests aussehen und auf welchen Modellen diese Algorithmen auf ihre Tauglichkeit getestet wurden. Im Abschnitt 4 wird Details der Umsetzung von diesen Algorithmen eingegangen. Darauf folgend werden im Abschnitt 5 die Resultate vorgestellt und diskutiert, während im Abschnitt 6 unsere Lösungen mit anderen bisherigen Lösungsansätzen verglichen werden.

1.1 Motivation

Wenn man nun ein Model eines Problemes erstellt hat, hat dieser entstandene Graph 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 kennt man Cluster-Graphen. Diese bestehen einer oder mehren Komponenten, wo jeder Knoten mit jedem Knoten verbunden ist. Diese Cluster-Graphen lassen sich aber auch dadurch charakterisieren, dass sie keinen P_3 als einen induzierten Teilgraphen haben, was bedeutet, dass man jedem Knoten aus dem P_3 einen Knoten aus der 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 der Abbildung 1a ist ein P_3 zu sehen. Es ist einfach ein Graphen mit 3 Knoten wo eine Knoten mit zwei anderen verbunden ist.

In Graphen in der Abbildung 1b sehen wir, dass der Teilgraph B,D,C ein von P_3 induzierter Subgraph ist, weil wir Konten 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. Weil im Graphen 1a a und c



(a) Ein P_3

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

(c) Ein Graph ohne einen induzierten P_3

Abbildung 1: Cluster-Graphen

verbunden ist, so muss auch B und D verbunden sein. Weil B und C in Graphen 1b nicht verbunden sind, so darf auch a und b nicht durch eine Kante verbunden sein. Man kann überprüfen, dass es auch für die Voraussetzung für jede beliebiges Paar von Knoten geben ist und so ist der rot markierte Teilgraph ein von P_3 induzierte Teilgraph.

Im Graphen 1c haben wir keinen von P_3 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 Kanten zwischen A und C haben aber keine Kante zwischen a und b. Somit ist das keine gültige Zuordnung. Man kann aber auch nachvollziehen, dass egal wie wir die Zuordnung wählen, wir es nicht schaffen eine solche Zuordnung zu finden.

Deshalb ist der Graph in 1c ein Cluster-Graph und hat eben diese besondere Eigenschaft, die auf zwei Weisen beschreiben 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 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 als Methoden anschauen, wie man einen Graphen möglichst schnell so modifizieren kann, so dass er eine bestimmte Eigenschaft hat. Und um diese Eigenschaft zu beschrieben, verwenden wir die Charakterisation durch verbotenen Teilgraphen.

1.2 Anwendungsbeispiele

In diesem Abschnitt werden wir auf einige Anwendungsfälle eingehen, warum man Graphen so umformen möchte, dass sie bestimmte Eigenschaften haben.

1.2.1 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, dass Ergebnis für den Graphen G. Hiermit können wir das Problem also in kleinere Probleme zerlegen. Außerdem gilt es, dass das Finden einer Clique in einem Graphen äquivalent zum Finden der stabilen Menge in der Komplementgraphen ist. Wir können also unser Problem weiter zerlegen indem wir das Komplement von jeder Zusammenhangkomponenten nehmen und es wieder in seine seine Zusammenhangkomponenten zerlegen. Dabei gilt die maximale stabile Menge die Vereinigung aller maximalen stabilen Mengen der Zusammenhangkomponenten ist.

Eine solche Vorgehensweise ist offesichtlich sehr attraktiv um eine solch schweres Problem, wie Maximum Clique es ist, zu lösen. Dieses Vorgehen hat jedoch ein Problem, wenn das Komplement eines Graphen zusammmenhängend ist und keine Zusammenhangskomponenten hat. Wir definieren, dass G einen complement reducible graph, oder kurz Co-Graph, wenn so etwas in dem Graphen nie auftritt.[20]

Die Co-Graphen-Klasse hat also sehr wünschenswerte Eigenschaften, wie dass man Maximum Clique in linearer Zeit lösen kann. Diese Klasse ist eine sehr gut studierte Klasse, die in vielen Anwendungen auftritt.

Co-Graphen lassen sich auch durch charakterisieren, dass sie keinen P_4 als induzierten Subgraphen enthalten. [21]

1.2.2 Soziale Netzwerke

(P_4, C_4)-freie Graphen modellieren eine soziale Struktur. [21]

Ähnlich dazu sind (P_5, C_5)-freie Graphen die auch soziale Strukturen modellieren und dafür geeignet sind Gemeinschaften zu identifizieren. [25]

1.2.3 Protein interaction networks

$(2K_2, C_4, C_5)$-freie Graphen haben gewisse Vorteile für die Untersuchung von Interaktionsnetzwerken von Proteinen [9].

1.2.4 Bicluster Editing

[12] [18]

1.3 Definitionen

1.3.1 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.

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

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

Sei $H = (V_H, E_H)$ 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 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 $(f(u), f(v)) \notin E$.

Ein Graph G ist F-frei, wenn es nicht F als induzierten Subgraphen enthält. Für eine Menge $\mathcal{F}$ von Graphen, heißt G $\mathcal{F}$-frei, wenn G für jeden $F \in \mathcal{F}$ F-frei ist. Siehe [10] für eine Defintion von FPT und Kernel.

1.3.2 Problemstellung

Um Heuristiken für das Entfernen von verbotenen Teilgraphen entwickeln zu können, ist zuerst ein Problem zu definieren. Es ist unter dem Namen F-Free Edge Editing[1] bekannt.

> $\mathcal{F}$-Free Edge Editing
> **Eingabe:** Graph G, natürliche Zahl k

[1] oder H-Free (Edge) Editing

Frage: Können wir von G höchstens k Kanten entfernen, so dass G keinen induzierten Teilgraphen aus $\mathcal{F}$ enthält?
Parameter: k

$\mathcal{F}$-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.

Die hier vorgestellten Heuristiken.

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

1.4 Ähnliche Arbeiten

Implicit Hitting Set hilft hier leider nicht viel.[19]

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

2 Algorithmen

In diesem Abschnitt werden die verschiedenen Ansätze an die das Problem beschrieben. Die im nachfolgenden beschriebenen Algorithmen basieren alle auf dem folgenden Prinzip: Suche einen validen Graphen, welcher die verbotenen Subgraphen nicht enthält. Wiederhole dies `iterations`-mal und dann geben die Differenz zwischen besten validen Graphen und dem Eingabegraphen aus.

Dieses Prinzip ist im Algoritmus 1 zu sehen. Dabei steht SolveAlgo für einen der Algorithmen, die wir in den folgenden Abschnitten betrachten werden.

Da alle Ansätze diesen Schritte enthalten und sich nur in dem unterschieden, wie der valide Graph gefunden wird, wird folgend nur dieser Aspekt betrachtet.

Die entwickelten Ansätze sind in 3 große Gruppen zu unterteilen. Der Top-Bottom-Ansatz nimmt den Graphen und ändert ihn solange, bis ein gültiger Graph entsteht. Der Bottom-Top-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 unterschiedliche Stadien gibt...

Algorithm 1 Genereller Aufbau

```
1: function SOLVE(graph, forbidden, iterations)
2:     best ← (∅, ∅)
3:     for i = 1 to iterations do
4:         valid ← SOLVEALGO(graph, forbidden)
5:         if #(DIFF(best, graph)) < #(DIFF(valid, graph)) then
6:             best ← valid
7:         end if
8:     end for
9:     print DIFF(graph, best)
10: end function
```

2.1 Top-Bottom

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

2.1.1 RandomChange

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

Algorithm 2 RandomChange

```
1: function RANDOMCHANGESOLVE(graph, forbidden)
2:     for Graph f ∈ forbidden do
3:         forbiddenSubgraph ← FINDFS(graph, f)
4:         while forbiddenSubgraph ≠ ∅ do
5:             change a random edge ∈ forbiddenSubgraph
6:             forbiddenSubgraph ← FINDFS(graph, f)
7:         end while
8:     end for
9:     return graph
10: end function
```

2.1.2 Random

Es ist wie RandomChange[2], aber bereits editierte Kanten werden mit einer geringeren Wahrscheinlichkeit geändert. Auch hat es für kleine Graphen ein

[2] Algorithmus 2

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 RandomChange sehen wir ab Zeile 6. Dort wählen wird die Kante ausgewählt welche geändert werden soll, aber eine bereits geänderte Kante bekommt eine Wahrscheinlichkeit zugewiesen die 4-mal kleiner ist(siehe Zeile 10).

Algorithm 3 Random

```
 1: function StateRandom2Solve(graph, forbidden)
 2:     for Graph f ∈ forbidden  do
 3:         fforbiddenSubgraph ← findFS(graph, f)
 4:         while forbiddenSubgraph != ∅ do
 5:             foundEdge ← ∅
 6:             while true do
 7:                 e ← random edge from forbiddenSubgraph
 8:                 prob ← 1 / #(E(f))
 9:                 if e is already visited then
10:                     prob ← prob / 4
11:                 end if
12:                 if random number from [0,1] > prob then
13:                     foundEdge ← e
14:                     break
15:                 end if
16:                 flip e in graph
17:             end while
18:             forbiddenSubgraph ← findFS(graph, f)
19:         end while
20:     end for
21:     return graph
22: end function
```

2.2 Bottom-Top

Die Bottom-Top-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 enthält. Dies ist ein Vorteil gegenüber den Top-Bottom-Ansätzen, da es mög-

lich ist immer einen validen Graphen zu haben und somit jederzeit terminieren.

2.2.1 Extend

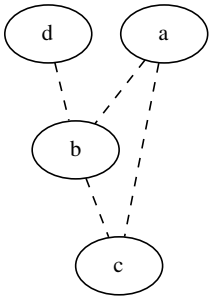
Das ist der einfachste Algorithmus aus der Klasse der Bottom-Top-Ansätze. Zu sehen die Vorgehensweise in Algorithmus 4. Er fängt mit einem Graphen an, der die selben Knoten wie der Eingabegraph hat, aber keine Kanten (Zeile 2). Dann wird versucht jede Kante einzufügen, die auch im originalen Graphen `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 2 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.

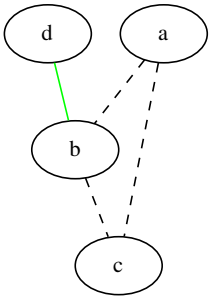
Es ist zu sehen, dass dieser Algorithmus keine Kanten hinzufügt, die nicht in dem Eingabe-Graphen nicht vorhanden waren.

Algorithm 4 Extend

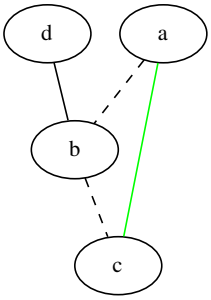
```
1: function ExtendSolve(input, forbidden)
2:     graph = (V(input), ∅)
3:     while true do
4:         for each Edge e ∈ Difference(graph,input) do
5:             try to flip e, revert if it produces an invalid graph
6:         end for
7:         break if there was no change
8:     end while
9: 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



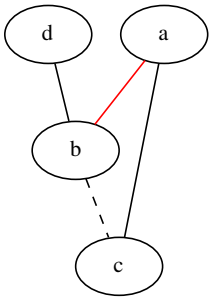
(a) Startgraph



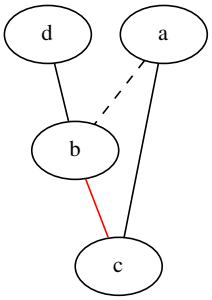
(b) Kante (d,b) wird erfolgreich hinzugefügt



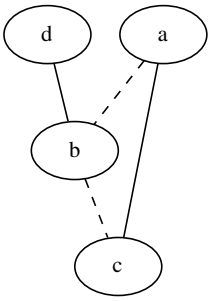
(c) Kante (a,c) erfolgreich wird hinzugefügt



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



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



(f) Resultierender Graph

Abbildung 2: Beispielweiser Ablauf des Extends

2.3 GRASP

2.3.1 Grow-Reduce

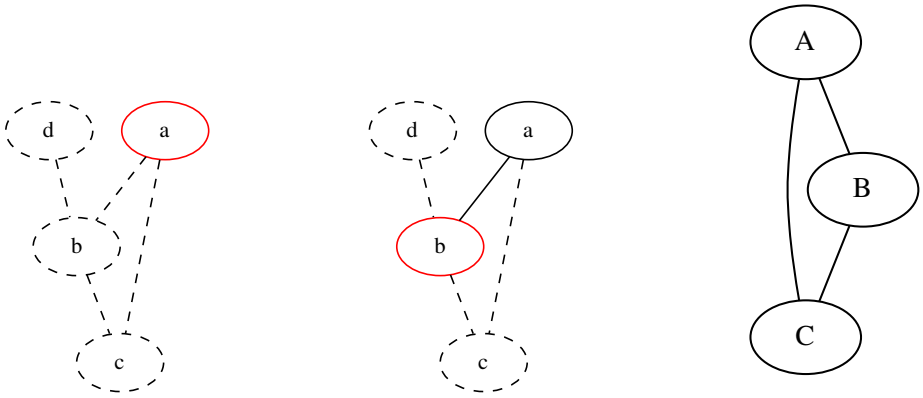
Der Grow-Reduce-Ansatz ist ein Art von einer greedy randomized adaptive search procedure(GRASP).GRASPH beschreiben Siehe [5] Der Grow-Reduce-Ansatz sieht wie folgt aus: Begonnen wird mit einem Graphen, der die selben Knoten wie der Eingabegraph hat, aber keine kanten. Dann wird in jeder Iteration ein Knoten und seine Umgebung hinzugefügt und durch lokale Suche werden alle neu entstandenen verbotenen Subgraphen wieder entfernt. Die ist im Algorithmus 5 zu sehen.

Algorithm 5 GrowReduce

```
1:  function GrowReduceSolve(input, forbidden)
2:      graph ← (V(input), ∅)
3:      nodes ← randomOrder(V(input))
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
```

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



(a) Knoten a wird hinzugefügt

(b) Knoten b wird hinzugefügt

(c) Knoten c wird hinzugefügt

Abbildung 3: Beispielweise Grow-Phase

bereits erforscht sind.

Dieser Unterschied wird in der Abbildung 3 verdeutlich, wo nur die Grow Schritte visualiert wurden, ohne die Reduce-Phase, um es zu vereinfachen.In Abbildung 3a 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 bisher hinzugefügt worden sind, wird werden keine Kanten hinzugefügt. In Abbildung 3b 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 3c wird der Knoten c hinzugefügt und somit auch die Kanten (a, b) und (a, c).

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.

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
```

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.

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 zwischen 10 und 150, wobei der Parameter m, welcher die Anzahl der der Kanten definiert, die zu bereits bestehenden Knoten erstellt werden, zwischen 1 und 8 war.

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
```

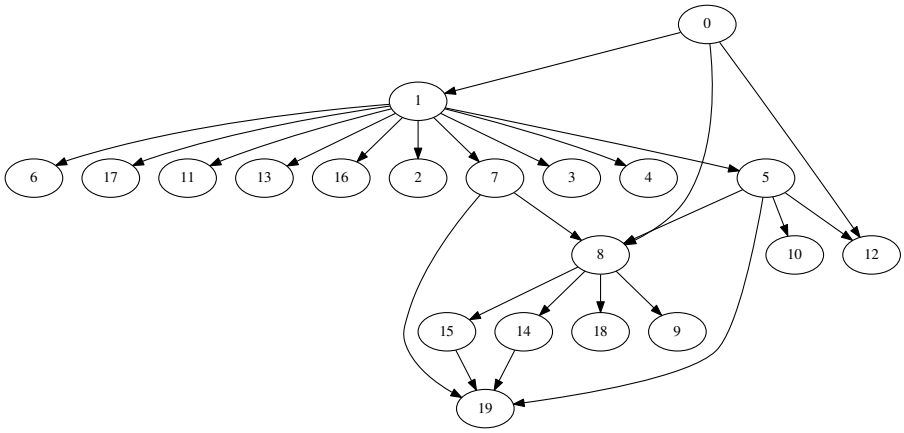


Abbildung 4: Ein beispielhafter Duplication-Divergence Graph mit $n = 20$ und $p = 0{,}4$

3.1.2 Erdős-Rényi

Für den Datensatz `binomial` wurde das Erdős-Rényi Modell verwendet[13] [6], wo jede Kante eine fixe Wahrscheinlichkeit hat zu existieren oder nicht zu existieren.

Es wurden dabei 54 Graphen generiert, mit einer Knotenanzahl zwischen 10 und 100 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[15], welches Interaktionsnetzwerke zwischen Proteinen modelliert. Ein Beispiel ist in Abbildung 4 zu sehen.

Dabei gibt es in jeder Iteration bei der Erstellung eines solchen zufälligen Graphen zwei Phase. 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 zwischen 10 und 100. Für die Wahrscheinlichkeiten p wurden folgenden Werte verwendet $\frac{1}{10}$, $\frac{2}{10}$, $\frac{5}{20}$, $\frac{4}{10}$, $\frac{5}{10}$, $\frac{8}{10}$.

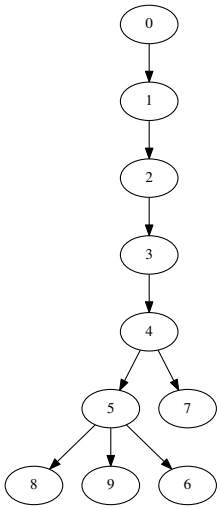


Abbildung 5: Ein Powerlaw-Baum mit $n = 10$

3.1.4 Newman-Watts-Strogatz

Für den Datensatz `newman_watts_strogatz` wurde das Newman-Watts-Strogatz Modell verwendet[23], welches einen Kleine-Welt-Graphen erzeugt mit kurzen durchschnittlichen Pfaden 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 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 zufälliger existierender Knoten ist.

Es wurden mit diesem Model 144 Graphen generiert mit einer Knotenanzahl(n) zwischen 10 und 100, m zwischen 2 und 8 und der Wahrscheinlichkeit p zwischen 0,2 und 0,8.

3.1.5 Powerlaw-Baum

Für den Datensatz `powerlaw`, wurde ein Modell verwendet, dass einen Baum erzeugt, deren Knotengrad einer Potenzverteilung folgt. Ein Beispiel ist in Abbildung 5 zu sehen. Es wurden mit diesem Model 30 Graphen generiert mit einer Knotenanzahl zwischen 10 und 160.

3.1.6 UCINetworkDataRepository

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 [28].

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 [14].

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 [22].

Der Graph `lesmiserables` ist ein Netwerke von Figuren, die in dem Roman Les Misérables"von Victor Hugo, zur gleichen Zeit auftreten [16].

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 [17].

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

3.1.7 bio1

Anzahl: 147 Was ist die Quelle für diese Daten???

3.1.8 bio2

Der Datensatz `bio2` sind COG protein similarity data [24] [7] Es sind 360 Graphen mit einer Knotenanzahl zwischen 3 und 80.

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.

4 Implementation

4.1 Repräsentation vom Graphen

Die Graphen werden in einer Adjazenzmatrix gespeichert.

4.2 Das Finden von induzierten Subgraphen

[26] Wie verwenden einen VF Algorithmus für FINDFS(`graph`,`forbidden`). Dieser gibt gibt eine Menge von Subgraphen zurück.

4.2.1 Vergleich VFLib, Boost und eigene Implemtation

•	find all p3s	count all p3s	has a p3
Spezial	0.73s	0.04s	0.00016s
VFLib	1.73s	0.87s	0.01236s
Boost	3.04s	1.68	0.00102s

Bei VFLib ist der Graph immutable und bei der Suche nach einem Subgraphen müssen wir jedes Mal den Graphen neu erstellen.

5 Auswertung

6 Vergleich mit anderen Heuristiken

6.1 Cluster-Editing

6.1.1 2K-Heuristik

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 1: Vergleich der durchschnittlichen Lösungsgröße

Programm / Datensatz	Aprox2k		ExploredGrowReduce[3]	
	mean	std	mean	std
bio1 [4]	**43.44**	79.81	133.59	249,62
bio2 [5]	**34.11**	19.79	113.81	166,01
duplication-divergence [6]	155.17	217.38	**92.30**	115,22
newman-watts-strogatz [7]	165.22	149,86	**152.34**	127,51
albert-barabasi [8]	324.30	316,32	**248.84**	226,87
binomial [9]	**493.17**	545,19	554.61	670,91

6.2 Quasi-Threshold Mover

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 2 wird die Lösungsqualität von dem Quasi-Thresold-Mover und unserem Algorithmus ExploredGrowReduce verglichen. In der Tabelle 3 wird die durschnittliche Größe der Lösungen verglichen. Der Quasi-Threschold-Mover ist unserem Ansatz weit überlegen.

Tabelle 2: Vergleich der durchschnittlichen Lösungsqualität

Programm / Datensatz	ExploredGrowReduce[10]		Quasi-Threshold-Mover	
	mean	std	mean	std
bio1 [11]	1.61x	1.30	**1.05x**	0.14
bio2 [12]	1.69x	1.03	**1.05x**	0.10
duplication-divergence [13]	1.42x	0.33	**1.04x**	0.05
newman-watts-strogatz [14]	1.38x	0.22	**1.06x**	0.05

Tabelle 3: Vergleich der durchschnittlichen Lösungsgröße

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

Programm / Datensatz	ExploredGrowReduce mean	std	Quasi-Threshold-Mover 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

7 Zukünftige Forschungsmöglichkeiten

7.1 Besser Sortierung von Grow-Reduce

7.2 Besser Konvergenzkriterium

7.3 Obere Schranken

7.4 Obere Schranken

8 Zusammenfassung

9 Anhang

9.1 Kleine Graphen

Literatur

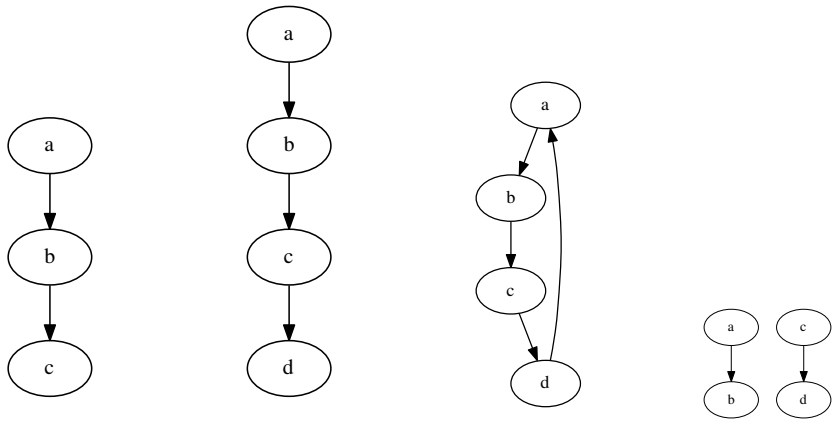
[1] Books about us politics. `http://networkdata.ics.uci.edu/data.php?d=polbooks`.

[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



(a) P_3 (b) P_4 (c) C_4 (d) $2K_2$

Abbildung 6: Einige Graphen

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] 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.

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

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

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

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

[17] 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.

[18] Sara C. Madeira and Arlindo L. Oliveira. Biclustering algorithms for biological data analysis: A survey. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 1(1):24–45, January 2004.

[19] 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.

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

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

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

[23] 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.

[24] 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.

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

[26] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the Association for Computing Machinery*, 23(1):31–42, 1976.

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

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