Heuristiken für das Entfernen von verbotenen Teilgraphen

Paul Walger

22. März 2016

Inhaltsverzeichnis

1 Abstract

2 Einleitung

2.1 Motivation

2.2 Anwendungsbeispiele

2.2.1 Soziale Netzwerke

(P_4, C_4)-freie Graphen modellieren eine soziale Struktur.[8]

Ähnlich dazu sind (P_5, C_5)-freie Graphen die auch soziale Strukturen modellieren und afür geeignet sind Gemeinschaften zu identifizieren. [9]

2.2.2 Protein interaction networks

$(2K_2, C_4, C_5)$-freie Graphen haben gewisse Vorteile für die Untersuchung von Interaktionsnetzwerken von Proteinen [5].

2.3 Definitionen

2.3.1 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.

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 $G[S] = (S, E \cap \{\{u, v\} \mid u \in S \wedge w \in S\})$

$N(u)$ ist die Nachbarschaft vom Knoten u. $N^*(u)$ ist die Nachbarschaft von u mit u inklusive.

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 $(f(u), f(v)) \notin E$.

2.3.2 Problemstellung

2.4 Ähnliche Arbeiten

Implicit Hitting Set hilft hier leider nicht viel.[7]

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

3 Implementation

3.1 Repräsentation vom Graphen

Die Graphen werden in einer Adjazenzmatrix gespeichert.

3.2 Das Finden von induzierten Subgraphen

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

3.2.1 Vergleich VFLib, Boost und eigne Implemtation

•	find all p3s	count all p3s	has a p3
Spezial	0.73s	0.04s	0.00016s
VFLib	1.73s	0.87s	0.0253s
Boost	3.04s	1.68	0.0021s

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

4 Algorithmen

Die nachfolgenden beschriebenen Algorithmen basieren alle auf dem folgenden Prinzip: Suche einen validen Graphen, welcher die verbotenen Subgraphen nicht enthält, der minimal unterschiedlich ist zu dem Eingabegraphen. Wiederhole dies, wenn notwendig. Dann gebe, die Differenz zwischen dem erstellen validen Graphen und dem Eingabegraphen. Dies wird in dem Algorithmus 1 noch einmal beschrieben. Dabei ist SOLVEALGO einer der Alorithmen, die wir in den folgenden Abschnitten betrachten werden.

Algorithm 1 F-Free BLP

```
1: function SOLVE(graph, forbidden, iterations)
2:     bestGraph = (∅, ∅)
3:     for i = 1 to iterations do
4:         validGraph = SOLVEALGO(graph, forbidden)
5:         if DIFFERENCE(bestGraph, graph) < DIFFERENCE(validGraph,
   graph) then
6:             bestGraph = validGraph
7:         end if
8:     end for
```

```
    print DIFFERENCE(graph, bestGraph)
end function
```

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...

4.1 Top-Bottom

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

4.1.1 Random2

Das ist der einfachste Algorithmus. Das Problem bei diesen ist, dass er nicht in absehbarer Zeit terminieren muss.

Algorithm 2 F-Free BLP

```
function StateRandom2Solve(graph, forbidden)
    for Graph f ∈ forbidden do
        while findForbiddenSubgraph(graph, f) != ∅ do
            change a random node in the forbidden subgraph
        end while
    end for
    return (graph)
end function
```

4.1.2 Random

Es ist wie random2 aber mit einer kleineren Wahrscheinheitlich, dass bereits editierte Kanten geändert werden und mit einem optionalen Konvergenzkriterum.

4.1.3 Random Half

4.2 Bottom-Top

4.2.1 Extend

Algorithm 3 F-Free Extend

```
1: function StateExtendSolve(input, forbidden)
2:     graph = (∅, ∅)
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
```

4.3 Grow-Reduce

Ist der Grow-Reduce Ansatz ein Greedy Randomized Adaptive Search Procedure? Siehe [3]

4.4 FPT

4.5 Lineare Programmierung

4.5.1 Lineare Optimierung

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

4.5.2 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

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

.

Da alle möglichen Bedingungen hinzuzufügen, welche bei alle verbotenen Subgraphen ausschließen wpürden, viel zum umfangreich wäre, werden die Bedingungen iterative dort hinzu gefügt, wo es einen verbotenen Teilgraphen gibt. Dann wird der Problem gelöst und und die Änderungen auf den Graphen übertragen. Dann wird wieder nach alle verboteten Subgraphen gesucht. Dies wird solange wiederholt bis es keine mehr gibt. Nun ist die minimale Anzahl von Änderungen gefunden.

Algorithm 4 F-Free BLP

```
1: function solveBLP(graph, algo:blp)
2:     for graph f ∈ forbidden do
3:         while findForbiddenSubgraph(graph, f) != ∅ do
4:             for each graph M ∈ findeVerboteneSubgraphen(graph,
   f) do
5:                 contstraint = 0
6:                 for each {u, v} ∈ kanten(M) do
7:                     if {u, v} ∈ kanten(graph) then
8:                         contstraint += 1 - e_uv
9:                     else
10:                        contstraint += e_uv
11:                    end if
12:                end for
13:                addConstraint(contstraint)
14:            end for
15:            graph = lpSolve(graph)
16:        end while
17:    end for
18:    return (graph)
19: end function
```

4.6 Relaxierte Lineare Programmierung

5 Aufbau der Test

5.1 Datensätze

Folgende Datensätze wurden verwendet.

5.1.1 albert barabasi

Für den Datensatz albert barabasi wurde das Barabasi–Albert Modell, welches ein zufälliges skalenfreies Netz erzeugt.[1]
Anzahl: 56

5.1.2 newman watts strogatz

Anzahl: 144

5.1.3 UCINetworkDataRepository

Anzahl: 9

5.1.4 bio1

Anzahl: 147 Was ist die Quelle für diese Daten

5.1.5 bio2

Anzahl: 360 Was ist die Quelle für diese Daten

5.2 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.

6 Auswertung

7 Vergleich mit anderen Heuristiken

7.1 Cluster-Editing

7.1.1 2K-Heuristik

Die 2K-Heuristik, basiert auf einem Kernel für das Cluster-Editing-Problem, welches maximal 2K Knoten liefert [6]. Wenn man dort eine Bedingung für die ?. Reduktionsregel abschwächt abschwächt, kommt eine sehr gute Heuristik für das Cluster-Editing-Problem heraus. Dabei wird die Bedingung mit jedem Durchlauf abgeschwächt.

Algorithm 5 2K Heuristik

```
1: function SOLVE2K(g :: Gewichteter Graph)
2:     a = 1,0
3:     while graph hat einen P3 do
4:         for each knoten u ∈ g do
5:             if 2 · a · costClique(g, u) + a · 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:        a = 0, 99 · a − 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
```

7.1.2 Andere Heuristiken

[3] Effiziente Algorithmen
GRASP Heuritik ILS Heuristik

7.2 Quasi-Threshold Mover

In [4] 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.

Vergleich mit mit meinem Algorithmus.

8 Zukünftige Forschungsmöglichkeiten

9 Zusammenfassung

Literatur

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

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

[3] 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.

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

[5] 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.

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

[7] 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.

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

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

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