Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Graphentheoretische Konzepte und Algorithmen

Referat - Ausarbeitung

Aufgabe 3: Flußprobleme

Autor: Adrian Helberg

Vortrag: 16.12.2019

Referat eingereicht im Rahmen der Vorlesung
Graphentheoretische Konzepte und Algorithmen

im Studiengang Angewandte Informatik (AI)
am Department Informatik der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. C. Klauck
Abgegeben am 15. Dezember 2019

Inhaltsverzeichnis

1 Einleitung **3**
1.1 Flußprobleme 3
1.2 Laufzeitmessung 3

2 Kontext **3**
2.1 Aufgabenstellung 3
2.2 Recherche 5
2.3 Komplexität 6

3 Entwurf **7**
3.1 Ford-Fulkerson 8
3.1.1 Algorithmus 8
3.1.2 Schnittstelle 10
3.1.3 Datenstrukturen 11
3.1.4 Ablauf 14
3.1.5 Pseudocode 15
3.2 Edmonds-Karp 16
3.2.1 Algorithmus 16
3.2.2 Schnittstelle 16
3.2.3 Datenstrukturen 18

4 Laufzeitmessung **18**
4.1 Versuchsaufbau 18
4.2 Parameter 19

5 Auswertung **19**

6 Quellen **20**

7 Erklärung zur schriftlichen Ausarbeitung **21**

1 Einleitung

1.1 Flußprobleme

Ähnlich wie beim Problem des kürzesten Weges oder bei den elektrischen Netzwerken handelt es sich hier um eine der ursprünglichsten Aufgabenstellungen für Graphen, wo eben die Kanten als Verbindungen mit gewissen festen Eigenschaften wie Längen, Widerstand, etc. interpretiert werden. [2] (Seite 79, Kapitel 10)

Offensichtlich steht bei der genannten Problematik der Transport im Vordergrund, wobei die Durchflussmenge durch einen konstanten Wert begrenzt wird. Klassische Beispiele sind Verkehrs-Netze, Gleichstrom-Netzwerke und Abwassersysteme.

1.2 Laufzeitmessung

Ein Teil der Aufgabenstellung beinhaltet das Messen von Laufzeiten zur Analyse implementierter Algorithmen. Hierzu sind Testszenarien mit einem erwarteten Ergebnis zu erstellen und diese nachzuweisen. Weiter werden die sich ergebenen Laufzeiten nicht in Zeiteinheiten, sondern in der *Landau-Notation* angegeben.

2 Kontext

Hier ist die wissenschaftliche Vorarbeit zu einer gegebenen Aufgabenstellung gemeint. Hierzu gehört die Recherche und Erarbeitung benötigter Algorithmen und Thematiken.

2.1 Aufgabenstellung

Ziel der Aufgaben [1] ist sowohl eine Implementierung zweier Algorithmen zum Finden des maximalen Durchsatzes (Flusses), als auch deren Vergleich und Analyse.

Folgende Algorithmen werden bearbeitet:

I. Der Algorithmus von **Ford und Fulkerson** [1]

II. Der Algorithmus von **Edmonds und Karp** [2]

Zusatz: Es soll nicht mittels Residualnetzwerks[1] gearbeitet werden

[1] „Ford-Fulkerson“, 1956, L.R.Ford & D.R.Fulkerson

[2] „Edmonds-Karp“, 1970, Yefim Dinitz, 1972, J.Edmonds & R.Karp

[1] Da es in der Aufgabenstellung um das Verstehen und Herausarbeiten verschiedener Strategien geht, wird ein weniger effizienter Algorithmus implementiert, als der mittels Residualnetzwerk, welcher intelligent die Markierung über Vorwärtskanten vornimmt

Weitere Vorgaben:

- Ergebnisse sind nachvollziehbar:
  - Ausgaben in Dateien
  - Ausgabe auf dem Bildschirm
  - Generierung von Graphen als Dot-Dateien und Bildern
- Berechneter Fluss ist als Attribut an Kanten zu speichern (mittels interner Datenstruktur) zur Generierung von Flussgraphen als Bild
- Schnittstellen:
  - fordfulkerson:fordfulkerson($< Filename >$,$< Quelle >$,$< Senke >$):
    [<Liste der im letzten Lauf inspizierten Ecken>]
  - fordfulkerson:fordfulkersonT($< Graph >$,$< Quelle >$,$< Senke >$):
    [<Liste der im letzten Lauf inspizierten Ecken>]
  - edmondskarp:edmondskarp($< Filename >$,$< Quelle >$,$< Senke >$):
    [<Liste der im letzten Lauf inspizierten Ecken>]
  - edmondskarp:edmondskarpT($< Graph >$,$< Quelle >$,$< Senke >$):
    [<Liste der im letzten Lauf inspizierten Ecken>]
- Erweiterung des abstrakten Datentyps „adtgraph“ um eine Funktion $printGFF(< Graph >, < Filename >)$ zur Erstellung von $*.dot$-Dateien aus gegebenen Graphen

  ! **Entfällt hier, da der mitgelieferte abstrakte Datentyp „adtgraph“ nur als Kompilat vorliegt und somit unveränderlich ist**
- Nachweis der erwarteten Komplexität durch Laufzeitmessung mittels bereitgestellter Module
- Gegebene Graphen sind zum Test der Korrektheit anzuwenden
- Logdateien zur Zeitmessung der Algorithmen sind anzulegen
- Bildschirmausgabe des Tests der Datei aufg3test.beam ist zu protokollieren

2.2 Recherche

Graphen werden hier durch $G(V, E)$ mit $|V|$ Ecken und $|E|$ Kanten beschrieben.

Im Folgenden wird auf das Kapitel „Flussprobleme“ aus dem Buch zur Vorlesung eingegangen [3] (ab Seite 95).

[...] *grundsätzlich mit schwach zusammenhängenden, schlichten gerichteten Graphen* [...]

Ein Graph heißt

- **zusammenhängend**, wenn die Knoten paarweise durch eine Kantenfolge verbunden sind
- **schwach zusammenhängend**, wenn der Graph, der entsteht, wenn man jede gerichtete Kante duch eine ungerichtete Kante ersetzt, zusammenhängend ist
- **schlicht**, wenn er ungerichtet ist und weder Mehrfachkanten, noch Schleifen besitzt

[...] *jede Kante gibt die Kapazität* $c(e_{ij}) = c_{ij}$ *der Kante an* [...]. *Aus praktischen Gründen nehmen wir dabei an, dass alle* c_{ij} *rationale Zahlen sind.*

In dieser Arbeit wird im Weiteren der Begriff **Netzwerk** verwendet, sollten Graphen die genannten Eigenschaften aufweisen.

Definition 4.1 [...] *Eine Kapazität ist eine Funktion c, die jeder Kante* $e_{ij} \in E$ *eine positive rationale Zahl* (> 0) *als Kapaziät zuordnet. Ein Fluss in G von der Quelle* $q = v_1$ *zu der Senke* $s = v_n$ *ist eine Funktion f, die jeder Kante* $e_{ij} \in E$ *eine nicht negative rationale Zahl zuordnet* [...]

Notiz: In einer Implementierung der Algorithmen kann c als Kapazität und f als Fluss für Namen des jeweile Kantenattributs gesetzt werden.

Weiter gilt die

- **Kapazitätsbeschränkung** $(e_{\mathrm{ij}} : f(e_{\mathrm{ij}} \leq c(e_{\mathrm{ij}}))$ und
- die **Flusserhaltung**
  $(\forall_{\mathrm{j}} \in \{1, \dots n\} : \sum_{e_{\mathrm{ij}} \in O(v_{\mathrm{i}}, e_{\mathrm{ji}} \in I(v_{\mathrm{i}}))} f(e_{1\mathrm{j}}) = \sum (f(e_{\mathrm{ij}}) - f(e_{\mathrm{ji}})) = 0)$

In dieser Arbeit wird im Weiteren der Begriff **Flussnetzwerk** verwendet, sollten Graphen diese Eigenschaften aufweisen (V, E, f, c).
Der Wert des **Flusses** f mit $d = \sum_{e_{1\mathrm{j}} \in O(q)} f(e_{1\mathrm{j}}) = \sum_{e_{\mathrm{in}} \in I(s)} f(e_{\mathrm{in}})$ beträgt in den Ecken

q (Quelle) und s (Senke) 0 [Null] (**Flusserhaltung**). Die maximale Menge, die von der Quelle zur Senke transportiert werden kann ist ein Fluss maximaler Stärke.

Definition 4.2 *Ein Schnitt ist die Menge von Kanten $A(X, \bar{X})$, wobei $q \in C$ und $s \in \bar{X}$.*

Definition 4.3 *Ein Fluss, dessen Wert $min\{c(X, \bar{X}) \| A(X, \bar{X})$ ist ein beliebiger Schnitt$\}$ entspricht heißt ein **maximaler Fluss**.*

Definition 4.4 *Ein ungerichteter Weg von der Quelle q zur Senke s heißt ein vergrüßernder Weg, wenn gilt:*

- Für jede Kante e_{ij} , die auf dem Weg entsprechend ihrer Richtung durchlaufen wird (sie wird als **Vorwärtskante** bezeichnet), ist $f(e_{\mathrm{ij}}) < c(e_{\mathrm{ij}})$.
- Für jede Kante e_{ij} , die auf dem Weg entgegen ihrer Richtung durchlaufen wird (sie wird als **Rückwärtskante** bezeichnet), ist $f(e_{\mathrm{ij}}) > 0$).

Satz 4.2 *Wenn in einem Graphen G ein Fluss der Stärke d von der Quelle q zur Senke s fließt, gilt genau eine der beiden Aussagen:*

1. *Es gibt einen vergrößernden Weg.*
2. *Es gibt einen Schnitt $A(X, \bar{X})$ mit $c(X, \bar{X}) = d$.*

Notiz: *1.* kann bei einer Implementierung eine Rekursion auslsen („Ein maximaler Fluss ist noch nicht gefunden“ - Erweiterbarkeit ist gegeben), wobei *2.* die Abbruchbedingung beschreibt („Ein maximaler Fluss ist gefunden“ - Erweiterbarkeit nicht gegeben).

Satz 4.3 (Max-flow-min-cut Theorem von Ford und Fulkerson) *In einem schwach zusammenhängendem schlichten Digraphen G mit genau einer Quelle q und genau einer Senke s sowie der Kapazitätsfunktion c und dem Fluss f ist das Minimum der Kapazität eines q und s trennenden Schnitts gleich der Stärke eines maximalen Flusses von q nach s.*

2.3 Komplexität

Die Komplexität des **Ford-Fulkerson** Algorithmus wird mit $O(|E| * d_{\max})$, mit $d_{\max}$ als Maximalwert von d, angegeben. Jede Kante muss maximal zweimal inspiziert[2] werden (in jede Richtung einmal), um einen vergr. Weg zu finden (Worst-Case-Betrachtung). Die Inspektion benötigt hier eine konstante Anzahl von Arbeitsschritten ($O(1)$). Ein neuer verg.

[2] siehe Kapitel 3.1

Weg ist also nach jeweils $O(|E|)$ Schritten gefunden (wenn es einen gibt). Da maximal $d_{\max}$ vergr. Wege gefunden werden müssen, ergibt sich eine Komplexität von $O(|E| * d_{\max})$ [3] (Seite 105).

Weiter lässt sich der Speicherplatzbedarf der Eingabe durch $O(|V|^2 * log(c_{\max}))$, mit $c_{\max}$ als Maximalwert der Kapazitäten aller Eckenpaare, abschätzen [3] (Seite 105).

Der Algorithmus ist nicht polynomial, da der Arbeitsaufwand mit zunehmender Eingabe exponentiell ansteigt, weil $c_{\max} \leq d_{\max}$ verausgesetzt werden kann[3] ($O(c_{\max}) = O(e^{\log(c_{\max})})$) [3] (Seite 106).

Satz 4.4 *Wenn jede Vergrößerung der Flussstärke d durch einen vergr. Weg minimaler Kantenanzahl erfolgt, dann sind höchstens $O(|E| * |V|)$ vergr. Wege zu berechnen, bis d seinen Maximalwert erreicht hat.* [3] (Seite 106).

Satz 4.4 Ändert den Algorithmus ab, damit die Problemgröße in Polynomialzeit anwächst.

Der **Edmonds-Karp** Algorithmus verbessert die Laufzeit des **Ford-Fulkerson**, welche $O(|E| * d_{\max})$ beträgt, zu einer Laufzeit von $O(|V| * |E|^2)$, macht sie also unabhängig von $d_{\max}$. Da die Suchreihenfolge nach einem verg. Weg wohl definiert ist, resultiert dieser Algorithmus in einer geringeren Laufzeitkomplexität.

3 Entwurf

Der Entwurf dient der allgemeinen Beschreibung der Vorgänge und der technischen Umsetzung als alleinige Vorlage, sodass eine Implementierung in beliebigen Programmiersparachen möglich ist, ohne weitere Dokumente zu benötigen. Sowohl die zu implementierenden Algorithmen als auch gewisse Datenstrukturen sind hier zu finden. So ist es möglich auf bestimmte Stärken und Schwächen von Programmiersprachen in diesem Kontext einzugehen, um so effiziente Softwarelösungen zu erstellen.

Beschreibungen von Schnittstellen sind hier jeweils für den fettgedruckten Teil gegeben und Beispiele für eine mögliche Implementierung und Datenstrukturen sind für die funktionale und prädikative Programmiersprache **Erlang** [5] verfasst, da sich die Aufgabenstellung auf eine Implementierung in **Erlang** bezieht.

[3] Der Fluss steigt maximal bis d an; deshalb sind Kapazitäten, die größer als d sind, vernachlässigbar

3.1 Ford-Fulkerson

Der Algorithmus beruht auf der Idee, einen Weg von der Quelle zur Senke zu finden, entlang dessen der Fluss weiter vergrößert werden kann, ohne die Kapazitätsbeschränkungen der Kanten zu verletzen. [4] (Wirkungsprinzip)

Dieser Algorithmus baut im wesentlichen auf dem Beweis des Satzes 4.3 auf.

3.1.1 Algorithmus

Ecken des Graphen werden während der Suche nach einem vergrößernden (Abkïrzung: vergr.) Weg mit $(+/-, Vorg_\mathrm{i}, \delta_\mathrm{i})$ markiert.

- $Vorg_\mathrm{i}$ beschreibt den Vorgänger von v_i auf einem vergr. Weg
- δ_i gibt die bisher auf dem vergr. Weg maximal mögliche Änderung der Flussstärke an
- $Vorg_\mathrm{i}$ wird mit einem Vorzeichen versehen, das angibt, ob eine Kante entsprechend $(+)$ oder entgegen $(-)$ ihrer Richtung durchlaufen wird

Algorithmus:

1. *(Initialisierung)*
   Weise allen Kanten $f(e_{ij})$ als einen (initialen) Wert zu, der die Nebenbedingungen erfüllt. Markiere q mit $(undefiniert, \infty)$.

2. *(Inpektion und Markierung)*

   (a) *Falls alle markierten Ecken inspiziert wurden, gehe nach 4.*

   (b) *Wähle eine beliebige markierte, aber noch nicht inspizierte Ecke v_i und inspiziere sie wie folg (Berechnung des Inkrements)*

   - *(Vorwärtskante) Für jede Kante $e_{ij} \in O(v_i)$ mit unmarkierter Ecke v_j und $f(e_{ij}) < c(e_{ij})$ markiere v_j mit $(+v_i, \delta_j)$, wobei δ_j die kleinere der beiden Zahlen $c(e_{ij}) - f(e_{ij})$ und δ_i ist.*
   - *(Rückwärtskante) Für jede Kante $e_{ji} \in I(v_i)$ mit unmarkierter Ecke v_j und $f(e_{ji}) > 0$ markiere v_j mit (v_i, δ_j), wobei δ_j die kleinere der beiden Zahlen $f(e_{ji})$ und δ_i ist.*

   (c) *Falls s markiert ist, gehe zu 3., sonst zu 2.(a).*

3. *(Vergrößerung der Flussstärke)*
   Bei s beginnend lässt sich anhand der Markierungen der gefundene verg. Weg bis zur Ecke q rückwärts durchlaufen. Für jede Verwärtskante wird $f(e_{ij})$ um δ_s erhöht, und für jede Rückwärtskante wird $f(e_{ji})$ um δ_s vermindert. Anschließend werden bei allen Ecken mit Ausnahme von q die Markierungen entfernt. Gehe zu 2.

4. *Es gibt keinen verg. Weg. Der jetzige Wert von d ist optimal. Ein Schnitt $A(X, \bar{X})$ mit $c(X, \bar{X}) = d$ wird gebildet von genau denjenigen Kanten, bei denen entweder die Anfangsecke oder die Endecke inspiziert ist.*

Notiz: In Schritt *1.* wird i.allg. $f(e_{\mathrm{ij}}) := 0$ für alle i und j gewählt.

Das Buch zur Vorlesung [3], aus dem die Arbeitsweise der Algorithmen entnommen wird, zeigt, wie der **Ford-Fulkerson** Algorithmus mithilfe einer Tabelle verarbeitet wird (ab Seite 103). Diese beschreibt zwei Zellen „gekennzeichnete Ecke“ und „Kennzeichnung“. Erstere beinhaltet markierte Ecken, die mit „*“ versehen werden sollten diese inspiziert werden, und die zweite Zelle zeigt die Markierung der Ecke.

Notiz: Es bietet sich an für eine Implementierung eine ähnliche Datenstruktur beim Markieren zu wählen (z.B. Tupel). Das Inspizieren kann über Attribute an Kanten gesetzt werden z.B. $\{+, 18119, 5\}$.

3.1.2 Schnittstelle

fordfulkerson:fordfulkerson(< *Filename* >,< *Quelle* >,< *Senke* >):
[<Liste der im letzten Lauf inspizierten Ecken>]

- Beschreibung: Der **Ford-Fulkerson** Algorithmus ist in einem Arbeitspaket „fordfulkerson“ zu implementieren
- Beispiel: Erlang-Modul

```
% The erlang file containing this code is named fordfulkerson.erl
-module(fordfulkerson).
% ...
```

fordfulkerson:**fordfulkerson**(< *Filename* >,< *Quelle* >,< *Senke* >):
[<Liste der im letzten Lauf inspizierten Ecken>]

- Beschreibung: Der Name in der Funktionssignatur ist fordfulkerson
- Beispiel: Erlang-Funktion

```
% ...
fordfulkerson(...) -> ... .
% ...
```

fordfulkerson:fordfulkerson(<**Filename**>,< *Quelle* >,< *Senke* >):
[<Liste der im letzten Lauf inspizierten Ecken>]

- Beschreibung: Filename steht hier für den Namen der Datei, in der ein Graph gespeichert ist. Die Datei liegt im selben Verzeichnis, wie das Modul **fordfulkerson**
- Beispiel: Dateiname als Erlang-Atom

```
% ...
fordfulkerson('graph01.graph', ...) -> ... .
% ...
```

fordfulkerson:fordfulkerson(< *Filename* >,<**Quelle**>,<**Senke**>):
[<Liste der im letzten Lauf inspizierten Ecken>]

- Beschreibung: Quelle und Senke geben an, wo die Flüsse berechnet werden. Flüsse fließen von der Quelle bis zur Senke

- Beispiel: Ganzzahlige Quelle und Senke

```
% ...
fordfulkerson(..., 1, 10) -> ... .
% ...
```

fordfulkerson:fordfulkerson(< *Filename* >,< *Quelle* >,< *Senke* >):
[<**Liste der im letzten Lauf inspizierten Ecken**>]

- Beschreibung: Rückgabe sind die letzten inspizierten Ecken des Algorithmus
- Beispiel: Liste mit ganzzahligen Ecken

```
% ...
fordfulkerson(...) ->
Ecken = [1, 2, ...],
Ecken.
% ...
```

3.1.3 Datenstrukturen

Um Überlegungen für die Datenstrukturen anzustellen, muss klar sein, wie auf Daten zugegriffen wird. Dazu wird der Algorithmus aus **Kapitel 3.1.1** untersucht.

Begriffserklärungen

- < *Graph* > entspricht einer internen Datenstruktur aus **adtgraph**, welche beispielsweise mit *adtgrap* : *createG*, *adtgraph* : *importG*, etc. erstellt werden kann
- < *Node* > bezeichnet eine Ecke eines Graphen, die auch mit einem Index versehen werden kann, sollten mehrere Ecken angesprochen werden (z.B. < *Node*1 >)
- < *Name* > meint hier den Namen eines Attributs, das an einer Ecke oder einer Kante gesetzt werden kann
- < *Value* > wird als Wert des Attributs gesetzt
- < *Vorzeichen* > bezeichnet Vorwärtskanten (+), Rückwärtskanten (−), oder die Quell-Ecke (/)
- < *Vorg* > ist der vorangegangene Knoten, der in der Markierung angegeben wird
- < *Delta* > bezeichnet das im Algorithmus verwendete δ

Überlegungen zu Datenstrukturen zur Umsetzung des Algorithmus

aus 1. *(Initialisierung)*: Setzen von Werten aller Kanten

- Werte können als Kantenattribut im ADT[4] **adtgraph** gesetzt werden. Die Funktionalität ist bereits gegeben und im Kompilat „adtgraph.beam“ definiert. ($O(1)$)

```
% Usage of adtgraph:setAtE function
adtgraph:setAtE(<Graph>,<Node1>,<Node2>,<Name>,<Value>).
% Example: adtgraph:setAtE({...},1,2,'flow',5).
```

  $< Node1 >$ und $< Node2 >$ formen hier eine Kante des Graphen $< Graph >$

- Um Werte *aller* Kanten setzen zu können, wird eine Funktionalität gebraucht, die Kanten iteriert. ($O(N)$)
  Hier bietet es sich an eine Liste von Kanten mittels Rekursion zu iterieren

aus 2. *(Inspektion und Markierung)*:

(a) Überprüfung auf Markierung und Inspektion

- Markiert wird mit Tuple $\{< Vorzeichen >, < Vorg >, < Delta >\}$ oder *nil* als „fehlende“ Markierung
- Inspiziert wird mit Atom: $'*'$ oder *nil* als „fehlende“ Inspektion
- Da eine Fallunterscheidung $A\&\&B$, mit A als vorhandene Markierung und B als vorhandene Inspektion, benötigt wird, können hier alle Knoten des Graphen rekursiv durchlaufen werden, um diese so auf Markierung und Inspektion zu überprüfen mittles Funktion „getValV“ aus **adtgraph** ($O(N)$)

```
% Usage of adtgraph:getValV function
adtgraph:getValV(<Graph>,<Node>,<Name>).
% Example: adtgraph:getValV({...},5,'Marke'). -> {'+',1,20}
```

  Attribute von Kanten können über die Funktion „getValE“ aus **adtgraph** abgefragt werden ($O(1)$)

```
% Usage of adtgraph:getValE function
adtgraph:getValE(<Graph>,<Node1>,<Node2>,<Name>).
% Example: adtgraph:getValE({...},1,2,'flow'). -> 20
```

[4] abstrakter Datentyp

(b) Wählen einer beliebigen Ecke ($O(N)$), Vorwärts- und Rückwärtskanten verarbeiten

- Um unabhängig von der gegebenen Datenstruktur aus **adtgraph** arbeiten zu können, wird eine Liste über alle Ecken des Graphen gemischt, um so eine **beliebige**, markierte, nicht inspizierte Ecke zu finden ($O(N)$)

```
% Usage of util:shuffle function
util:shuffle(<List>).
% Example: util:shuffle([1,2,3,4]). -> [3,1,4,2]
```

- Inzidente Kanten der beliebig gewählten Ecke können mithilfe der Funktion „getIncident“ aus **adtgraph** abgefragt werden ($O(1)$)

```
% Usage of adtgraph:getIncident function
adtgraph:getIncident(<Graph>,<Node>).
% Example: adtgraph:getIncident({...},1). -> [1,2,1,3,4,1]
```

- Das Markieren der Kanten erfolgt ähnlich wie das Setzen von Attributen an Kanten

```
% Usage of adtgraph:setAtE function
adtgraph:setAtE(<Graph>,<Node1>,<Node2>,<Name>,<Value>).
% Example: adtgraph:setAtE({...},1,2,'flow',5).
```

aus 3. *Verößerung der Flussstärke*

- Die Aufgebenstellung gibt den Wert

  [<*Liste der im letzten Lauf inspizierten Ecken*>]

  als Rückgabewert an. Darum ist es nötig die durchlaufenden Knoten zu halten. Dies kann über eine Liste (< *Liste* >) geschehen

aus 4.

- Beim vorgegebenen Rückgabewert des Algorithmus ist es nicht nötig einen Schnitt (und damit d) zu berechnen. Hier müssen keine Überlegungen zu Datenstrukturen angestellt werden. Eine zu dieser Arbeit angefertige Inplementation kann schon nach Schritt 3 des Algorithmus terminieren und das Ergebnis zurückliefern

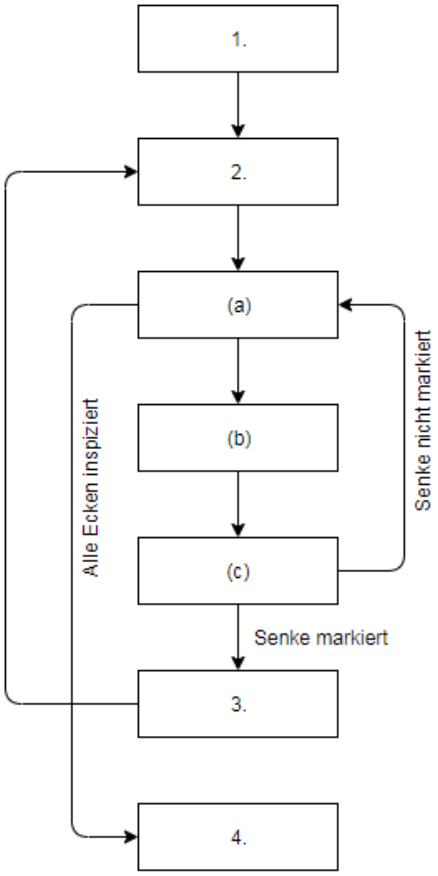
Zusatz:

- Das Einlesen von Graphen aus Dateien, kann über „importG“ aus **adtgraph** erfolgen

```
% Usage of adtgraph:importG function
adtgraph:importG(<Filename>, <d/ud>).
% Example: adtgraph:importG('test', d). d for directed
```

3.1.4 Ablauf

In der folgenden Abbildung ist der Ablauf der einzelnen Schritte im Gesamtkonzept des Algorithmus zu sehen.



Dem Ablauf ist zu entnehmen, dass es sich anbietet den Algorithmus mittels zweier Schleifen zu implementieren:

- Äußere Schleife ($2. \rightarrow 2a. \rightarrow 2b. \rightarrow 2c \rightarrow 3. \rightarrow 2 \rightarrow ...$) und
- Innere Schleife ($2a. \rightarrow 2b. \rightarrow 2c. \rightarrow 2a. \rightarrow ...$)

3.1.5 Pseudocode

In diesem Abschnitt wird der Pseudocode für eine mögliche Implementierung vorgestellt. Die Implementierung die der Aufgabenstellung zugrundeliegt, ist anhand dieses Codes erstellt worden. Wenn nicht anders angegeben, läuft der Algorithmus von Oben nach Unten durch

I. Lese gerichteten Graphen G aus Datei (Dateiname ist Inputparameter) ein. Setze den Fluss jeder Kante (rekursiv) des Graphen auf 0. (Attributname „flow“). Markiere Quell-Ecke (Inputparameter) mit $\{'/', undefined, 'Infinity'\}$.

   Anfang **Äußere Schleife**:

II. Prüfe, ob alle markierten Ecken inspiziert sind. Wenn Ja gehe zu VI.. Frage Knotenliste des Graphen ab und mische diese, hole ersten Knoten dieser Liste. Inspiziere Knoten mit $'*'$. Hole alle inzidenten Kanten des Knotens und gehe diese wie folgt (rekursiv) durch.

   Anfang **Innere Schleife**:

III. Prüfe auf vorhandene Markierung und fehlende Inspektion. Trifft eines der beiden nicht zu, wiederhole III. mit der nächsten Kante. Prüfe Richtung der Kante. Markiere Knoten mit $\{< Richtung >, < Vorg >, < Delta >\}$ mit $< Vorg >$ als Knoten, der mit dem inspizierten Knoten die aktuelle Kante bildet und mit $< Delta >$ als Minimum aus $< Kapazität >$ (Attributname „weight“) $- < Fluss >$ (Attributname „flow“) und dem $< Delta >$ des Knotens. Sind noch nicht alle Kanten bearbeitet, gehe zu VIII mit der nächsten Kante.

   Ende **Innere Schleife**

IV. Prüfe auf vorhandene Markierung des Senke-Knotens (Inputparameter). Fehlt die Markierung, gehe zu II.

V. Gehe Markierungen (rekursiv) durch, beginnent mit dem Senke-Knoten, und setze den Fluss („flow“) der Kanten, die aus aktuellen Knoten und $< Vorg >$ der Markierung entsprechen, auf $Flow + Delta$ der Markierung des Senke-Knotens. Füge bearbeitete Knoten der Knotenliste hinzu. Gehe zu II.

   Ende **Äußere Schleife**

VI. Gib die Knotenliste zurück.

3.2 Edmonds-Karp

Der Algorithmus ist eine Modifikation des **Ford-Fulkerson** Algorithmus und bedarf deshelb nur wenige Änderungen im Entwurf und in einer Implementierung.

3.2.1 Algorithmus

Markierungen und Inspektionen erfolgen hier wie beim **Ford-Fulkerson** (Siehe Kapitel 3.1.1). Zur vereinfachung sind identische Teile beider Algorithmen weggelassen.

Algorithmus

1. *(Initialisierung)*
   ... Markiere q mit $(undefiniert, \infty)$ *und füge q einer Queue hinzu.*
2. *(Inspektion und Markierung)*
   Wähle die nächste Ecke aus der Queue aus und inspiziere sie ...
   - *Knoten, die markiert werden, werden der Queue hinzugefügt; Knoten die inspiziert werden, werden der Queue entnommen*
3. *... Anschließend werden bei allen Ecken mit Ausnahme von q die Markierungen entfernt und die Queue wird geleert. Gehe zu 2.*

3.2.2 Schnittstelle

edmondskarp:edmondskarp(< *Filename* >,< *Quelle* >,< *Senke* >):
[<Liste der im letzten Lauf inspizierten Ecken>]

- Beschreibung: Der **Edmonds-Karp** Algorithmus ist in einem Arbeitspaket „edmondskarp“ zu implementieren
- Beispiel: Erlang-Modul

```
% The erlang file containing this code is named edmondskarp.erl
-module(edmondskarp).
% ...
```

edmondskarp:**edmondskarp**(< *Filename* >,< *Quelle* >,< *Senke* >):
[<Liste der im letzten Lauf inspizierten Ecken>]

- Beschreibung: Der Name in der Funktionssignatur ist edmondskarp
- Beispiel: Erlang-Funktion

```
% ...
edmondskarp(...) -> ... .
% ...
```

edmondskarp:edmondskarp(<**Filename**>,< *Quelle* >,< *Senke* >):
[<Liste der im letzten Lauf inspizierten Ecken>]

- Beschreibung: Filename steht hier für den Namen der Datei, in der ein Graph gespeichert ist. Die Datei liegt im selben Verzeichnis, wie das Modul **edmondskarp**
- Beispiel: Dateiname als Erlang-Atom

```
% ...
edmondskarp('graph01.graph', ...) -> ... .
% ...
```

edmondskarp:edmondskarp(< *Filename* >,<**Quelle**>,<**Senke**>):
[<Liste der im letzten Lauf inspizierten Ecken>]

- Beschreibung: Quelle und Senke geben an, wo die Flüsse berechnet werden. Flüsse fließen von der Quelle bis zur Senke
- Beispiel: Ganzzahlige Quelle und Senke

```
% ...
edmondskarp(..., 1, 10) -> ... .
% ...
```

edmondskarp:edmondskarp(< *Filename* >,< *Quelle* >,< *Senke* >):
[<**Liste der im letzten Lauf inspizierten Ecken**>]

- Beschreibung: Rückbage sind die letzten inspizierten Ecken des Algorithmus
- Beispiel: Liste mit ganzzahligen Ecken

```
% ...
edmondskarp(...) ->
Ecken = [1, 2, ...],
Ecken.
% ...
```

3.2.3 Datenstrukturen

Um Überlegungen für die Datenstrukturen anzustellen, muss klar sein, wie auf Daten zugegriffen wird. Dazu wird der Algorithmus aus **Kapitel 3.2.1** untersucht. Da sich dieser Algorithmus sehr ähnlich zum **Ford-Fulkerson** verhält, wird dieses Kapitel stark abgekürzt. Siehe hierzu Kapitel 3.1.3

Da in der Aufgabenstellung spezifiziert ist, dass die Implementierung beider Algorithmen **Ford-Fulkerson** und **Edmonds-Karp** so ähnlich wie möglich sein sollen, wird hier nur darauf eingegangen, dass eine Queue in Form einer Liste ($< List >$) beim **Edmonds-Karp** verwendet wird.

4 Laufzeitmessung

Dieses Kapitel stellt den Teil der Aufgabestellung dar, in dem Laufzeitmessungen für beide vorgestellten Algorithmen durchgeführt werden sollen, um somit die erwarteten Laufzeitkomplexitäten zu beweisen.

4.1 Versuchsaufbau

Um Laufzeitkomplexitäten erfassen zu können, werden die Algorithmen mehrfach mit verschiedenen Eingabedaten ausgeführt. Somit kann untersucht werden, wie sich die Algorithmen verhalten (z.b. bei linear wachsenden Eingabedaten). Hierbei ist zu beachten, dass die zugrunde liegende Implementation in Erlang erstellt wurde und auf einem Windows-System ausgeführt wird. Je nach Auslastungen des Systems, auf dem die Algorithmen ausgeführt werden, können ungenaue Messungen o.ä. entstehen.

Zur Generieung von Inputdaten wird das Tool „gengraph“ (**gengraph.beam**) genutzt:

```
gengraph:gengraph(<Anzahl Ecken>,<minimaler Grad>,<Maximaler Grad>,<Min.
    Gewicht>,<Max. Gewicht,<Dateiname>).
% Example: gengraph:gengraph(100,3,5,1,100,'bsp').
```

Das Beispiel zeigt, wie eine Generierung eines randomisierten Graphen mit 100 Ecken, einem minimalen Grad von 3, einem maximalen Grad von 5, einem Mindestgewicht von 1 und einem Maximalgewicht von 100 funktioniert. Mit derartigen Graphen werden beide Algorithmen ausgeführt und dann verglichen. Um eine möglichst gute Wahl von Quell- und Senken-Knoten zu erlangen, wird die Knotenliste des generierten Graphen zuerst sortiert und dann das erste Element der sortierten Liste als Quelle und das letzte Elmenet als Senke genutzt. Der minimale Grad aller Knoten beläuft sich stets auf $|V| - 1$, um so eine Vollvermaschung des Flussgraphen zu erreichen.

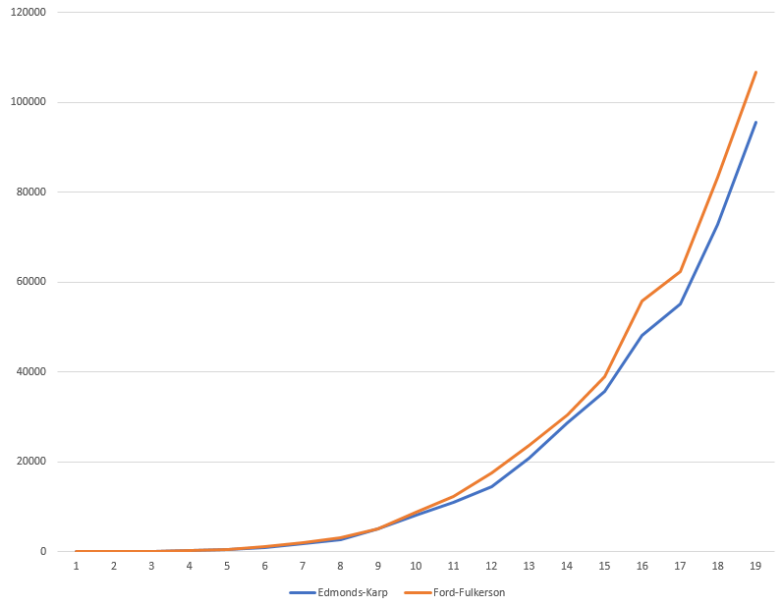
4.2 Parameter

Da in der Aufgabenstellung spezifiziert ist, dass das Tool **gengraph** zu verwenden ist, sind die Parameter für die Laufzeitmessung auf die Inputparameter des Tools beschränkt.

- Anzahl Ecken
- minimaler Grad
- maximaler Grad
- Mindestgewicht
- Maximalgewicht

5 Auswertung

Edmonds-Karp	**Ford-Fulkerson**	Anzahl Ecken	Anzahl Kanten
5	7	10	45
22	25	20	190
79	93	30	434
217	232	40	780
443	503	50	1225
886	1001	60	1770
1662	1896	70	2415
2729	3072	80	3160
5003	5144	90	4005
8111	8785	100	4950



Die Tabelle und der Graph zeigen, dass sich beide Algorithmen ähnlich verhalten bei linear steigender Anzahl Ecken, aber entgegen der Erwarteten Laufzeitkomplexität exponentiell in der Laufzeit steigen.

6 Quellen

Literatur

[1] Prof. Dr. C. Klauck. Aufgabe 3: Flußprobleme, 2019. `https://users.informatik.haw-hamburg.de/~klauck/GKA/aufg3.html`.

[2] Peter Läuchli. *Algorithmische Graphentheorie*. Birkhäuser, Basel, 9. edition, 1991. ISBN: 978-3-0348-5635-5.

[3] Christoph Klauck & Christoph Maas. *Graphentheorie für Studierende der Informatik*. HAW Hamburg, 6. edition, 2015.

[4] Carsten Milkau. Algorithmus von ford und fulkerson, Juni 2019. Revision 21.06.2019 `https://de.wikipedia.org/wiki/Algorithmus_von_Ford_und_Fulkerson`.

[5] Marc van Woerkom. Erlang (programmiersprache), November 2019. `https://de.wikipedia.org/wiki/Erlang_(Programmiersprache)`.

7 Erklärung zur schriftlichen Ausarbeitung

Hiermit erkläre ich, dass ich diese schriftliche Ausarbeitung meines Referates selbstständig und ohne fremde Hilfe verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe sowie die aus fremden Quellen (dazu zählen auch Internetquellen) direkt oder indirekt übernommenen Gedanken oder Wortlaute als solche kenntlich gemacht habe. Zudem erkläre ich, dass der zugehörige Programmcode von mir selbständig implementiert wurde ohne diesen oder Teile davon von Dritten im Wortlaut oder dem Sinn nach übernommen zu haben. Die Arbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher nicht veröffentlicht.

Hamburg, den 15. Dezember 2019 _____