Abschlussbericht zur Projektgruppe Max-Flow-Algorithmen: Edmonds-Karp, Push-Relabel

Florian Nelles, Moritz Rehbach

Institut für Informatik I, Universität Bonn

1 Einleitung

Unsere Aufgabe im Projekt war es, ein Programm in Java zu schreiben, mit dem man Max-Flow-Instanzen erstellen, betrachten und schrittweise lösen kann.
Wir hatten die Algorithmen Edmonds-Karp [**?**] und Push-Relabel [**?**] zu betrachten und zunächst in einem Kurzvortrag vozustellen. Das Programm sollte das Visualisieren, sowie das Laden, Speichern und Bearbeiten von Graphen ermöglichen. Hauptzweck des Programms ist es, die Algorithmen durch schrittweise Durchführung anschaulich zu machen. Die Algorithmen sollten auch unabhängig von der Visualisierung benutzbar sein.

2 Problemstellung

Das "Max-Flow-Problem" und seine Varianten treten in vielen Anwendungsfällen auf. Ein einfaches Beispiel ist die Frage, wie man möglichst viel von einem Gut über verschiedene Wege mit begrenzten Kapazitäten von A nach B transportiert. Formal beschreibt man ein Max-Flow-Problem durch ein *Netzwerk* $N = (G, c, s, t)$, bestehend aus einem gerichteten Graphen $G = (V, E)$ mit Kapazitätsfunktion $c : E \to \mathbb{R}_+$ sowie den zwei ausgezeichneten Knoten s, der Quelle, und t, der Senke. Ein *s-t-Fluss* ist eine Zuordnung $f : E \to \mathbb{R}_+$, wobei gelten muss:

- $\forall e \in E : f(e) \leq c(e)$
- $\forall v \in V \backslash \{s, t\} : ex_f(v) := \sum_{e \in \delta^-(v)} f(e) - \sum_{e \in \delta^+(v)} f(e) \quad = 0$

Der Wert eines Flusses bezeichnet dann den Überschuss $ex(t)$ in der Senke. Das Ziel ist es nun, einen maximalen Fluss in N zu bestimmen.

3 Algorithmen

Um den maximalen Fluss zu finden, benutzen wir das sogenannte *Restnetzwerk*. Sei $N = (G, c, s, t)$ ein Flussnetzwerk und f ein Fluss in N. Definiere

$$rest_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{, falls } (u,v) \in E \\ f(v,u) & \text{, falls } (v,u) \in E \\ 0 & \text{, sonst} \end{cases}$$

Dann heißt $G_f = (V, E_f)$ mit $E_f = \{(u,v) \in VxV | rest_f(u,v) > 0\}$ das Restnetzwerk zu f und N. Einen Weg P von s nach t im Restnetzwerk nennen wir einen *f-augmentierenden Pfad*. Man kann den Wert von $|f|$ erhöhen, indem man auf allen Kanten des Pfads den Fluss um $\delta := \min_{e \in P}(c(e))$ erhöht.

Für einen maximalen Fluss müssen zwei Bedingungen erfüllt sein:

1. $\forall v \in V \backslash \{s,t\} : ex_f(v) = 0$
2. Es gibt keinen f-augmentierenden Pfad

Die erste Bedingung stellt sicher, dass f ein gültiger Fluss ist, die zweite, dass der Wert maximal ist.
Edmonds-Karp erfüllt die erste Bedingung in allen Zwischenschritten und arbeitet auf die Erfüllung der zweiten Bedingung hin. *Push-Relabel* erfüllt umgekehrt stets die zweite Bedingung und nähert sich der Erfüllung der ersten Bedingung.

3.1 Edmonds-Karp

Eine naheliegender Ansatz zum Lösen des Max-Flow-Problems ist das wiederholte Erhöhen entlang f-augmentierender Pfade, solange ein solcher existiert. Im Edmonds-Karp-Algorithmus wird diese Idee realisiert, mit der zusätzlichen Bedingung, dass immer ein augmentierender Pfad minimaler Länge ausgewählt wird. Für das Laufzeitverhalten ist diese Wahl günstiger als die eines beliebigen Pfades von s nach t im Restnetzwerk.

Ablauf des Algorithmus: Zunächst initialisieren wir alle Flusswerte mit 0. Dann wiederholen wir folgende Schritte:

1. Erzeuge das Restnetzwerk
2. Suche darin einen kürzesten Pfad P von s nach t mittels Breitensuche
   - Wenn keiner existiert, ist der Fluss maximal
3. Bestimme $\delta := \min_{e \in P}(c(e))$
4. Augmentierung der Flusswerte entlang P:

$$f(u,v) = \begin{cases} f(u,v) + \delta & \text{, falls } (u,v) \in E \text{ und } (u,v) \in P \\ f(u,v) - \delta & \text{, falls } (u,v) \in E \text{ und } (v,u) \in P \\ f(u,v) & \text{, sonst} \end{cases}$$

3.2 Push-Relabel

Bei *Edmonds-Karp* entsteht in jedem Schritt ein Fluss, der die erste der oben genannten Bedingungen erfüllt. Bei *Push-Relabel* gibt es dagegen nie einen Pfad von s nach t im Restnetzwerk, dafür ist der Fluss i.A. nicht gültig.
Die Grundidee ist, zunächst alle von s ausgehenden Kanten zu fluten, man setzt also den Flusswert gleich der Kantenkapazität.
Anschließend ist das Ziel, den so entstanden Überschuss $ex_f(v)$ in den erreichten Knoten abzutragen, bis am Schluss gilt: $\forall v \in V \backslash \{s,t\} : ex_f(v) = 0$.
Alle Knoten $v \in V \backslash \{s,t\}$ mit Überschuss $ex_f(v) > 0$ heißen *aktive Knoten*.

Zur Hilfe nehmen wir eine *Höhenfunktion* $\psi : V \to \mathbb{N}$.
Gilt für eine Kante $(u,v) \in E_{rest}$: $\psi(u) = \psi(v) + 1$, so bezeichnen wir sie als *erlaubte Kante*. Das Ändern des Flusses an einer Kante nennen wir *Push*. Gepusht wird nur über erlaubte Kanten im Restnetzwerk.
Wenn es keine erlaubte Kante von einem aktiven Knoten aus gibt, wird die Operation *Relabel* durchgeführt, d.h. das Ändern der Höhe. Hier wird der Wert $\psi(v)$ des aktiven Knotens v um den minimal nötigen Wert erhöht, damit eine erlaubte Kante entsteht.

Ablauf des Algorithmus: Zu Beginn setzen wir den Fluss an allen Kanten auf 0, die Höhe $\psi(s) = |V|$ und $\psi(v) = 0$ für alle anderen Knoten v. Dann wiederholen wir folgende Schritte, solange noch ein aktiver Knoten existiert:

- Wähle einen aktiven Knoten v
  - Erlaubte Kante (v,w) existiert $\rightarrow$ PUSH((v,w))
  - Keine erlaubte Kante existiert $\rightarrow$ RELABEL(v)

4 Implementierung

4.1 Datenstrukturen

Wir verwenden zur Beschreibung von Netzwerk und Fluss folgende Klassen:

- **Flow.java**
  Enthält ein Netzwerk $N = (G, c, s, t)$ und darauf definierte Flusswerte $f : E \to \mathbb{R}_+$. Die Flusswerte sind in Form einer Adjazenzmatrix gespeichert, wie in der Klasse `Graph`. Der Fluss muss nicht gültig sein. Das zugehörige Restnetzwerk, der Flusswert und andere Eigenschaften können zurückgegeben werden. Auch Methoden zum Ändern von Flusswerten stehen zur Verfügung.
  - **Graph.java**
    Definiert einen Graphen $G = (V, E)$. Bei uns sind die Kapazitäten (Double) Teil der Kanten selbst. Die Kanten sind in Form einer Adjazenzmatrix gespeichert. Beim Hinzufügen eines Knotens mit Label "s" oder "t" wird die entsprechende ID markiert.
    Die Adjazenzmatrix wird in der HashMap `AdjacencyMap` gespeichert,

in der Form `Map⟨u, Map⟨v, Edge⟩⟩`, wobei u und v Knoten-IDs sind. Die Implementierung als HashMap (statt Array o.ä.) lag nahe, so wird nicht immer eine voll besetzte Matrix gespeichert wird. Allerdings war zu beachten, dass Modifikationen an der HashMap nicht in Schleifen erfolgen können, die über alle Elemente iterieren. Die Knoten sind in der HashMap `VertexList` gespeichert.

  - **Vertex.java**
    Ein Knoten erhält eine eindeutige ID (Integer) und ein Label (String). Zum Referenzieren von Knoten wird immer deren ID benutzt.
  - **Edge.java**
    Eine Kante ist definiert durch Startknoten (ID, Integer), Endknoten (ID, Integer) und Kapazität.

- **Path.java**
  Eine verkettete Liste von Kanten.
- **PathInt.java**
  Eine verkettete Liste von Integer-Tupeln (u, v).

4.2 Interface "Algo"

Die beiden Algorithmen implementieren beide das Interface `Algo.java`. Es erweitert das generische Interface `Iterable⟨T⟩` für den Typ `Flow` und legt einige Methoden fest, die in beiden Algorithmen mit identischer Signatur implementiert sind:

- **Flow next()**
  Führt einen Schritt im Algorithmus durch und gibt entsprechendes `Flow`-Objekt (s.o.) zurück.
- **boolean hasNext()**
  Gibt zurück, ob ein weiterer Schritt durchgeführt werden kann ($false \rightarrow$ Fluss ist maximal).
- **int getStepCount()**
  Gib zurück, wieviele Schritte bisher durchgeführt wurden.
- **void reset()**
  Setzt den Algorithmus und alle Datenstrukturen zurück.
- **double getFlowValue()**
  Gibt die Summe aller Flusswerte an Kanten nach t zurück. Dabei spielt erstmal keine Rolle, ob der Fluss gültig ist.

4.3 Edmonds-Karp

Auf einem Flussnetzwerk (Typ `Graph`) wird zunächst ein `Flow`-Objekt erzeugt, wobei alle Kanten mit Fluss 0 initialisiert werden. Dann werden bis zum Terminieren (siehe `hasNext()`) folgende Schritte beim Aufruf von `next()` wiederholt:

- **Erzeugung des Restnetzwerks**
  Aufruf von `Flow.getResidualNetwork()`.

- **Suche eines kürzesten Pfades P von s nach t**
  Aufruf von `EdmondsKarp.augmentingPath()`:
  - Aufruf von `GraphSearch.BFS(Graph, startNode, goalNode)` (Breitensuche)
  - Setzen der Gewichte auf $\delta := \min_{e \in P}(c(e))$
- **Augmentierung der Flusswerte entlang P**
  Wie im Kapitel Algorithmen beschrieben

Der Pfad wird in Form eines `Path`-Objekts verwaltet. Zur Reduzierung unnötigen Speicherbedarfs wird intern ein `PathInt`-Objekt verwendet, das nur noch die IDs von Start- und Endknoten jeder Kante enthält.

4.4 Push-Relabel

Der Algorithmus benötigt einige Datenstrukturen zur Verwaltung von Höhe, Überschuss und aktiven Knoten. Die Höhen und Überschüsse der Knoten werden jeweils in einer HashMap gespeichert (`vertexHeight`, `vertexExcess`), die aktiven Knoten auf dem Stack `vertexActive`. Zusätzlich verwendet der Algorithmus, statt in jedem Schritt das komplette Restnetzwerk zu erzeugen, eine weitere Datenstruktur `reverseEdge`. Darin wird zu jedem Knoten v die Menge der Knoten mit einer Kante nach v gespeichert. Die Methode `reset()` initialisiert Flusswerte und die eben beschriebenen Datenstrukturen.
Dann werden bis zum Terminieren (siehe `hasNext()`) folgende Schritte beim Aufruf von `next()` wiederholt:

- **Erster Schritt**
  Falls `getStepCount()`=0, wird das Netzwerk durch Aufruf von `firstStep()` geflutet.
- **Wahl eines aktiven Knotens**
  ID v wird vom Stack `vertexActive` genommen.
- **Suche nach erlaubter Kante**
  Die Hilfsfunktion `validEdge(v)` findet zu einem aktiven Knoten v einen Nachbarn w, so dass die Kante (v, w) eine erlaubte Kante ist. Sie gibt dann die ID von w sowie die Restkapazität r der Kante (v, w) zurück, oder $(-1, -1)$, wenn es keine erlaubte Kante gibt.
- **PUSH:** Wenn eine erlaubte Kante existiert, ändern wir den Fluss um $\delta := \min(r, ex_f(v))$. Die Methode `PushRelabel.push(u,v,δ)` prüft, ob eine Kante (v, w) im ursprünglichen Graph vorhanden und deren Restkapazität $\geq \delta$ ist. Wenn ja, wird der Fluss auf (v, w) um δ erhöht, sonst auf (w, v) um δ gesenkt. Danach werden `vertexActive` und `vertexExcess` entsprechend aktualisiert.
- **RELABEL:** Wenn keine erlaubte Kante gefunden wird, muss die Höhe des aktiven Knotens angepasst werden. Dies erledigt die Funkion `relabel(v)`.

5 GUI

5.1 Bedienung

Der Algorithmus kann schrittweise nachvollzogen oder direkt komplett durchgeführt werden. Man kann mit Hilfe der Bedienelemente Graphen erzeugen und bearbeiten sowie in Dateien laden und speichern. Die Dateien enthalten dann das Flussnetzwerk, die Positionen der Knoten und den Zustand des Algorihmus. Außerdem gibt es einen Generator für Beispielgraphen, einstellbar sind hier Größe, maximale Kantenkapazität und die Anzahl direkt mit t verbundener Knoten.

Edmonds-Karp In der Ansicht für den Edmonds-Karp-Algorithmus wird das Fenster zweigeteilt, links wird immer das unveränderte Netzwerk anzeigt, rechts die gesetzten Flusswerte (ohne die ursprünglichen Kanten) oder das Restnetzwerk. Der letzte augmentierende Pfad wird in der "Fluss"-Ansicht hell hervorgehoben.

Push-Relabel Der Push-Relabel-Algorithmus verwendet rote Farbe für die Darstellung der aktiven Knoten, der gerade ausgewählte wird pink gezeichnet. In den Knoten, (oder beim Klick darauf unten rechts im Fenster), werden Überschuss (ex) und Höhe (h) angezeigt.

5.2 Visualisierung der Graphen

Zum Zeichnen der Graphen haben wir die Bibliothek *JGraphX* [**?**] verwendet. Die Klasse `GraphPanel.java` erweitert das normale `JPanel` und enthält jeweils die nötigen Objekte vom Typ `mxGraphComponent, mxGraph, mxGraphLayout`. Dort werden die Grafikobjekte gezeichnet und gespeichert, um später Interaktionen mit dem Graph zu ermöglichen. Gespeichert werden die erzeugten Objekte in den HashMaps `graphVertices` und `graphEdges`. Diese sind analog zu `VertexList` und `AdjacencyMap` in der Klasse `Graph.java` aufgebaut, enthalten aber hier Objekte vom Typ `mxCell`.
Beim Klick auf den Hilfe-Button steht auch eine Textausgabe der Graphen zur Verfügung.

Wichtige Methoden sind:

- **drawGraph(Graph g)**
  Linke Ansicht Edmonds-Karp
- **drawRightEK_Flow(EdmondsKarp a)**
  Rechte Ansicht Edmonds-Karp: Fluss
- **drawRightEK_Residual(EdmondsKarp a)**
  Rechte Ansicht Edmonds-Karp: Restnetzwerk.
- **drawFlowPushRelabel(PushRelabel a)**
  Ansicht PushRelabel

6 Schlussbemerkungen

Nachdem wir ein Grundgerüst aufgebaut hatten, haben wir zunächst getrennt voneinander die Algorithmen implementiert. Dabei entstanden unterschiedliche Versionen einiger Graph-Klassen, so wurden z.B. Flusswerte oder algorithmusspezifische Daten wie die Höhe in Knoten und Kanten gespeichert. Später haben wir diese Felder in die Algorithmen selbst ausgelagert, so dass die Klassen `Graph.java` und `Flow.java` einheitlich bleiben konnten.
Als wir mit der Implementierung fertig waren, mussten wir feststellen, dass noch lange Wartezeiten und teilweise sogar Berechnungsfehler in einigen Situationen auftraten. Es gab auch übersehene Fehler in den Testdaten, wie Kanten mit Gewicht 0, negative Gewichten oder Kanten eines Knotens zu sich selbst (nicht alle davon waren in der Darstellung sichtbar).
Die meiste Zeit haben wir auf die Visualisierung, das Testen und die anschließende Fehlersuche verwendet. Ein optimales Layout für die Graphen konnten wir nicht automatisch erzeugen. Umso wichtiger war deshalb, dass der Benutzer durch Verschieben von Knoten bzw. Kanten selbst das Layout gestalten kann und diese Änderungen auch beibehalten werden.

References

1. Röglin: Skript zur Vorlesung "Algorithmen und Berechnungskomplexität I", WiSe 10/11, http://roeglin.org/teaching/ws2010/AuBI.html
2. Korte, Vygen: Kombinatorische Optimierung, Springer-Verlag
3. JGraphX: GitHub repository, http://github.com/jgraph/jgraphx