Algorithmen für Graphen

Marco Romanutti[1,2]

[1] *Fachhochschule Nordwestschweiz FHNW, Brugg*
[2] `Effiziente Algorithmen, Klasse 5ld`

Graphen stellen grundlegende Datenstrukturen dar, mit welchen Sachverhalte aus dem realen Leben oft einfach abgebildet werden können. Nachfolgend werden verschiedene Algorithmen für solche Graphen vorgestellt. Die Implementationen werden dabei nach Korrektheit und Skalierbarkeit beurteilt.

1 Cycle detection

Ein Zyklus in einem gerichteten, kantengewichteten Graphen $G = (V, E)$, bestehend aus Knoten (V, engl. Vertices) und Kanten (E, engl. Edges) liegt vor, falls Start- und Endknoten eines Weges $(v_1, ..., v_n)$ mit $v_i \in V$ übereinstimmen - also $v_1 = v_n$ gilt.

1.1 Lösungsansatz

Zyklen können mit einer modifizierten Depth-First-Search (DFS) entdeckt werden: Von jedem Knoten aus wird mit der Funktion `CycleDetector.hasCycle` geprüft, ob die DFS zu einem bereits besuchten Knoten führt. Im implementierten Algorithmus werden triviale Kreise[1] und Schleifen ebenfalls als Kreise gewertet.

1.2 Korrektheit

Als Vorbedingung (ohne allfällige Laufzeit und Memory-Einschränkungen, vgl. Kapitel 1.3) für den Input-Graphen gilt

$$|V| <= Integer.MAX\_VALUE \wedge \tag{1}$$
$$\forall e \in E : e.w <= Integer.MAX\_VALUE \tag{2}$$

wobei das Attribut `e.w` in (2) das Kantengewicht beschreibt.

Die Korrektheit wurde mithilfe verschiedener Tests geprüft. Die Input-Files im Verzeichnis `src/test/java/resources` beginnen mit einer Nummer, welche die Anzahl Knoten beschreibt, gefolgt vom eigentlichen Graphen in Matrixform. Daraus ergibt sich folgendes Format:

```
4
0 1 0 0
0 0 1 0
0 0 0 1
0 0 1 0
```

In der Tabelle 1 im Anhang sind die verschiedenen Edge-Cases für Cycle Detection aufgelistet. Die Tabelle zeigt Testfall-Nummer, eine Beschreibung des Tests und den erwarteten Output. In der Funktion `hasCycle` wird das Kantengewicht nicht verwendet. Auf Tests für ungewichtete Graphen oder Kanten mit negativen Gewichten kann daher verzichtet werden.

1.3 Laufzeit- und Speicheranalyse

Der Graph wird als Adjazenzliste gespeichert. Die Laufzeit beträgt deshalb $O(|V| + |E|)$ analog einer normalen DFS, falls alle mögliche Pfade zu allen möglichen Kanten einmal betrachtet werden.

Mit der aktuellen rekursiven Implementierung kann die Laufzeit tief gehalten werden, allerdings ist die Skalierbarkeit durch den Stack limitiert. Mit der Funktion `CycleDetectorTest.scalability_1` kann die Skalierbarkeit getestet werden. Dazu wird ein Worst-Case-Graph generiert, der aus einem grossen Spannbaum besteht - jedoch keinen Kreis enthält. Ein solcher Graph führt zu unzähligen aufeinanderfolgenden iterativen Aufrufen der Funktion `hasCycle`. Das Limit für die Anzahl Knoten eines solchen Graphen liegt bei ~20700. Bis zu diesem Limit liegen alle Ausführungen von `hasCycle` im Millisekunden-Bereich. Grössere Graphen führen zu `StackOverflow`-Errors.

[1] Zyklus mit weniger als drei Knoten

2 Floyd-Warshall

Beim Floyd-Warshall-Algoritmus wird die Länge des kürzesten Pfades von Knoten v_i zu Knoten v_j berechnet, wobei der Pfad direkt oder über die ersten k Knoten $v_0, v_1, ..., v_k - 1$ führt.

2.1 Lösungsansatz

Die jeweiligen Kantengewichte werden in einer Gewichtungsmatrix nachgetragen. Für die ersten k Knoten $v_0, v_1, ..., v_k - 1$ wird in der Funktion `shortestPath(i, j, k)` geprüft, ob ein kürzerer Weg via k existiert. Die Funktion gibt das Gewicht des kürzesten Pfad von Knoten v_i zu Knoten v_j aus. Falls ein negativer Zyklus besteht, wird `neg.Cycle` ausgegeben. Ein negativer Zyklus ist daran erkennbar, dass ein Diagonalelement der Matrix einen negativen Wert enthält. Falls ein Knoten `j` nicht erreicht werden kann, wird `Infinity` ausgegeben.

2.2 Korrektheit

Die Input-Files für diesen Algorithmus unterscheiden sich leicht vom vorigen, weil zusätzlich die Parameter i, j und k eingelesen werden müssen. Diese folgen auf der ersten Zeile in obiger Reighenfolge gleich nach der Anzahl Knoten. Anschliessend ist wieder der Graph in Matrixform angegeben. Daraus ergibt sich folgendes Format für $|V| = 4$, $i = 0$, $j = 3$, $k = 4$ in der ersten Zeile:

```
4 0 3 4
0 1 0 0
0 0 1 0
0 0 0 1
0 0 1 0
```

Als Vorbedingung für den Input-Graphen gilt

$$|V| <= Integer.MAX\_VALUE \wedge \tag{3}$$
$$\forall e \in E : e.w <= Double.MAX\_VALUE \wedge \tag{4}$$
$$\forall i, j : i, j \in V \wedge \tag{5}$$
$$\forall k : k >= 0 \ \wedge \ k <= |V| \tag{6}$$

wobei das Attribut `e.w` in (4) das Kantengewicht beschreibt.

In der Tabelle 2 im Anhang sind die verschiedenen Edge-Cases für Floyd-Warshall aufgelistet.

2.3 Laufzeit- und Speicheranalyse

Die Implementation von Floyd-Warshall besitzt die Komplexität $O(|V|^3)$. Mit der Funktion `FloydWarshallTest.scalability_1` kann die Skalierbarkeit getestet werden. Im verwendeten Worst-Case-Graphen ist jeder Knoten mit allen anderen Knoten verbunden - jedoch ohne negative Kantengewichte, um negative Zyklen zu verhindern. Es wird ausserdem der Shortest Path mit $k = |V|$ gesucht, sodass immer die ganze Matrix berechnet werden muss. Abbildung 1 zeigt die Entwicklung der Laufzeiten. Die $O(|V|^3)$-Charakteristik ist darin gut ersichtlich. Für das Speichern des Graphen als Matrix wird viel Memory benötigt. Die Tests zeigen, dass ab etwa ~17000 ein `OutOfMemoryError` erwartet werden muss.

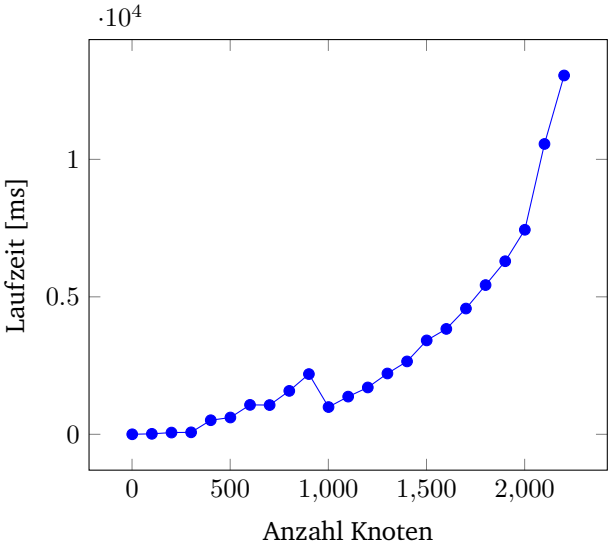

Figure 1: *Laufzeit von Floyd-Warshall*

Falls es sich beim Input-Graphen um einen ungerichteten Graphen handelt, könnte die Implementation optimiert und das Prozessieren des Graphen auf die halbe Adjazenzmatrix reduziert werden. Das folgende Listing zeigt die notwendigen Anpassungen im Body der Funktion `floydWarshall`.

```
for (int a = 0; a < k; a++) {
  for (int b = 0; b < nodeCount; b++) {
    for (int c = 0; c <= b; c++) {
      if (matrix[b][c] > matrix[b][a] + matrix[a][c]) {
        matrix[b][c] = matrix[b][a] + matrix[a][c];
        matrix[c][b] = matrix[b][c];

        if(c == b && matrix[c][b] < 0){
          return false;
        }
      }
    }
  }
}
```

3 Kruskal

Der Algorithmus von Kruskal findet Minimale Spannbäume (MST). Im Gegensatz zu den restlichen Algorithmen wird hier von ungerichteten Graphen ausgegangen.

3.1 Lösungsansatz

Beim Kruskal-Algorithmus wird jeweils die kleinstgewichtete, noch nicht mit dem Spannbaum verbundene Kante e zum Spannbaum hinzugefügt. Dies wird mit einer PriorityQueue umgesetzt - wobei vor dem Hinzügen jedes mal geprüft werden muss, ob die *beiden* Knoten der Kante schon vorher über den Teilbaum erreicht werden konnten. Dazu wird *weighted quick-union* eingesetzt, wie von Robert Sedgewick und Kevin Wayne beschrieben (vgl. [3], Seite 227).

Die aktuelle Implementation liefert die zu verbindenden Kanten, um alle minimalen Spannbäume in einem Graphen zu finden. Falls der Graph nicht zusammenhängend ist, kann die Resultatmenge Kanten aus mehreren Spannbäumen enthalten (vgl. Tests `[6]` - `[8]`).

3.2 Korrektheit

Die Korrektheit wurde mithilfe verschiedener Tests geprüft. Die Input-Files im Verzeichnis `src/test/java/resources` beginnen mit einer Nummer, welche die Anzahl Knoten beschreibt, gefolgt vom eigentlichen Graphen in Matrixform. Daraus ergibt sich wiederum folgendes Format:

```
4
0 1 0 0
0 0 1 0
0 0 0 1
0 0 1 0
```

In der Tabelle 3 im Anhang sind die verschiedenen Edge-Cases für Cycle Detection aufgelistet. Test `[3]` liefert keinen Spannbaum, weil die einzige Kante (Schleife) einen Zyklus enthält und deshalb kein Spannbaum sein kann.

In der Schleife in `mst` wird nach neuen Kanten gesucht, solange die Grösse des Spannbaums kleiner als $|V| - 1$ ist. Die Korrektheit dieser Invariante kann folgendermassen bewiesen werden.

Behauptung:

$$T = (V, E),\ |E| = |V| - 1 \tag{7}$$

Beweis:

[*] Falls |V| = 1 haben wir einen Baum mit einem Knoten

$$|E| = |V| - 1 = 0 \tag{8}$$

[**] Falls wir von einem Baum $T_n = (E_n, V_n)$ mit $1 <= |V| <= n$ einen Blattknoten inklusive der dazugehörigen Kante entfernen, erhalten wir einen kleineren Baum $T_{n-1} = (E_{n-1}, V_{n-1})$. Mit

$$|E_{n-1}| = |V_{n-1}| - 1 \tag{9}$$
$$|V_{n-1}| = |V_n| - 1 \tag{10}$$
$$|E_{n-1}| = |V_{n-1}| - 1 \tag{11}$$
$$= |V_n| - 2 \tag{12}$$
$$= |E_n| - 1 \tag{13}$$
$$\tag{14}$$

kann obige Behauptung gezeigt werden.

3.3 Laufzeit- und Speicheranalyse

In der Funktion `mst` wird für jede Kante geprüft, ob beide Enden bereits über einen Spannbaum verbunden sind. Die Komplexität des Algorithmus errechnet sich also aus $|E|$ multipliziert mit der Komplexität, welche für das überprüfen mit der Funktion `connected` benötigt wird.

Die Funktion `connected` benötigt also maximal $\log|E|$ für das Überprüfen, ob die Kante zum Spannbaum hinzugefügt werden kann. Daraus resultiert gesamthaft die Komplexität von $O(|E| \log |E|)$. Abbildung 2 zeigt die deutlich flachere Kurve für die Entwicklung der Laufzeiten (andere Skala ggü. Floyd-Warshall).

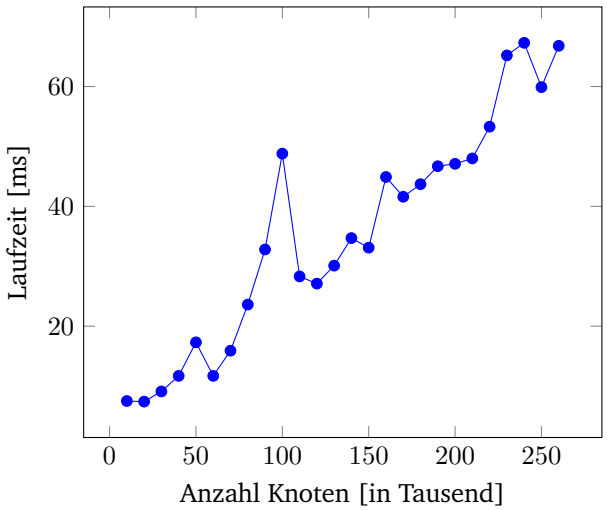

Figure 2: *Laufzeit von Kruskal*

In der PriorityQueue können maximal $|E|$ Kanten gespeichert sein, und auch die einzelnen Komponenten (in Klasse `Kruskal.Components`) können in einem Array gespeichert werden und benötigten maximal $|V|$ Einträge.

Die verschiedenen Komponenten des Spannbaums werden in der Klasse `Kruskal.Components` verwaltet. Beim Zusammenhängen zweier nicht verbundener Komponenten besteht die Gefahr, dass der daraus resulierende Komponentenbaum unausbalanciert wird. In einem solchen Fall würde die Suche nach dem Parent in `Components.findRoot` im schlechtesten Fall der

Höhe des Baumes entsprechen. Abbildung 3 zeigt ein solches Worst-Case Szenario.

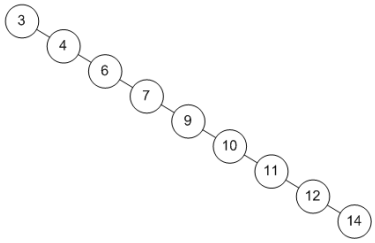

Figure 3: *Worst-Case (Ref: quora.com/What-is-an-AVL-tree)*

Um dies zu verindern, wird für jede Komponentenstruktur auch deren Grösse gespeichert und jeweils die kleinere an die grösseren Komponente gehängt. Damit wird sichergestellt, dass der Baum balanciert bleibt.

4 Ford-Fulkerson

Mit dem Ford-Fulkerson Algorithmus können maximale Flüsse zwischen zwei Knoten s und t in einem Graphen berechnet werden.

4.1 Lösungsansatz

Beim eingesetzten Lösungsansatz werden *augmenting paths* mittels einer Breadth-First Seach (BFS) gesucht. Solange es solche Pfade gibt, werden die Flüsse entlang des *augmenting paths* angepasst. Bei der BFS in der Funktion `augmentingPathExists` wird jeweils der direkteste gefundene Pfad gespeichert und anschliessend die Flüsse dafür innerhalb der Funktion `maxFlow` angepasst. Der Algorithmus entspricht durch den Einsatz der BFS, dem Verhalten wie von Edmonds-Karp beschrieben. Mit dem Test `[8]` wird das Verhalten anhand des Beispielgraphen aus dem Skript geprüft. Abbildung 4 zeigt den Graphen, der für diesen Test eingesetzt wurde. Durch die BFS wird der rot eingezeichnete Pfad vermieden.

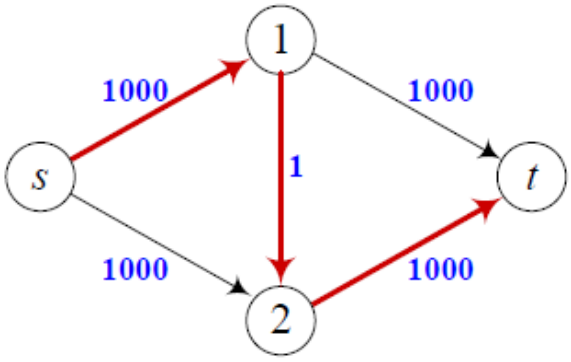

Figure 4: *Testgraph für Edmond-Karp*

4.2 Korrektheit

Die Input-Files benötigen die Parameter s und t, die eingelesen werden müssen. Die beiden Parameter folgen auf der ersten Zeile gleich nach der Anzahl Knoten. Anschliessend ist wieder der Graph in Matrixform angegeben. Daraus ergibt sich folgendes Format für $|V| = 4$, $s = 0$, $t = 3$ in der ersten Zeile:

```
4 0 3
0 1 0 0
0 0 1 0
0 0 0 1
0 0 1 0
```

Als Vorbedingung für den Input-Graphen gilt

$$|V| <= Integer.MAX\_VALUE \wedge \tag{15}$$
$$\forall e \in E : e.w <= Double.MAX\_VALUE \wedge \tag{16}$$
$$\forall s,t : s,t \in V \ \wedge \ s \neq t \tag{17}$$

In der Tabelle 4 im Anhang sind die verschiedenen Edge-Cases für Ford-Fulkerson aufgelistet.

4.3 Laufzeit- und Speicheranalyse

Die Breitensuche mittels `augmentingPathExists` besucht maximal $|E|$ Kanten. Mit jedem gefundenen Augmenting-Path wird die Restkapazität von mindestens einer der Kanten aus E auf 0 gesetzt, wobei die Diestanz dieser Kante zu s maximal $|V|$ beträgt. Dies führt zu einer Komplexität von $O(|E|^2|V|)$. Die schnell anwachsenden Laufzeiten sind in Abbildung 5 gut ersichtlich.

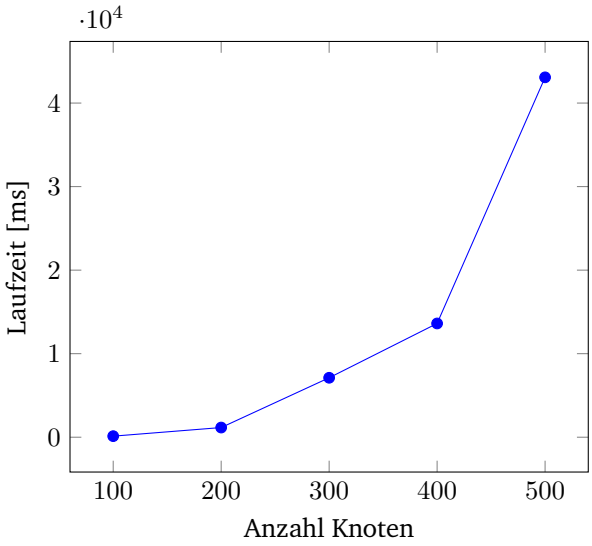

Figure 5: *Laufzeit von Ford-Fulkerson*

Der Graph wird als Adjazenzliste gespeichert und für das Prozessieren von `maxflow` werden Datenstrukturen mit maximal $|V|$ Elementen benötigt.

5 Appendix

Table 1: *Edge-Cases Cycle Detection*

Beschreibung	Output
[1] Graph mit 0 Knoten	`false`
[2] Graph mit 1 Knoten ohne Schleife	`false`
[3] Graph mit 1 Knoten und Schleife	`true`
[4] Gewichteter, gerichteter Graph ohne Zyklus	`false`
[5] Gewichteter, gerichteter Graph mit Zyklus	`true`
[6] Gewichteter, gerichteter Graph ohne Zyklus und mit Schleife	`true`
[7] Gewichteter, gerichteter Graph mit Zyklus und Schleife	`true`
[8] Gewichteter, gerichteter Graph mit trivialem Zyklus	`true`
[9] Negativ gewichteter, gerichteter Graph ohne Zyklus	`false`
[10] Negativ gewichteter, gerichteter Graph mit Zyklus	`true`
[11] Graph mit gemeinsamen Target aber ohne Zyklus	`false`
[12] Nicht zusammenhängender Graph mit Zyklus	`true`

Table 2: *Edge-Cases Floyd-Warshall*

Beschreibung	Output
[1] Graph mit 0 Knoten	`0.0`
[2] Graph mit 1 Knoten ohne Schleife	`0.0`
[3] Graph mit 1 Knoten und Schleife	`0.0`
[4] Gerichteter, gewichteter Graph i = 2, j = 1, k = 0	`Infinity`
[5] Gerichteter, gewichteter Graph i = 0, j = 2, k = 0	`Infinity`
[6] Gerichteter, gewichteter Graph i = 2, j = 1, k = 1	`12.0`
[7] Gerichteter, gewichteter Graph i = 0, j = 2, k = 2	`9.0`
[8] Gerichteter, gewichteter Graph i = 2, j = 1, k = 4	`7.0`
[9] Gerichteter, gewichteter Graph i = 0, j = 2, k = 4	`4.0`
[10] Gerichteter, gewichteter Graph mit negativem Zyklus	`neg.Cycle`
[11] Gerichteter, gewichteter Graph mit negativen Kanten	`4.0`
[12] Ungerichteter, gewichteter Graph	`8.0`
[13] Nicht zusammenhängender Graph	`Infinity`

Table 3: *Edge-Cases Kruskal*

Beschreibung	Output
[1] Graph mit 0 Knoten	
[2] Graph mit 1 Knoten ohne Schleife	
[3] Graph mit 1 Knoten mit Schleife	

[4] Ungerichteter, gewichteter Graph	`3/2:1`
	`2/1:2`
	`1/0:3`

[5] Ungerichteter, gewichteter Graph mit negativen Kanten	`2/0:-4`
	`3/2:1`
	`2/1:2`

[6] Nicht zusammenhängender Graph ein Baum	`3/2:1`
	`2/1:2`
	`1/0:3`

[7] Nicht zusammenhängender Graph zwei Bäume und neues Gewicht	`3/2:1`
	`2/1:2`
	`1/0:3`
	`5/4:5`

[8] Nicht zusammenhängender Graph zwei Bäume und vorhandenes Gewicht	`3/2:1`
	`5/4:2`
	`2/1:2`
	`1/0:3`

Table 4: *Edge-Cases Ford-Fulkerson*

Beschreibung	Output
[1] Graph mit 0 Knoten	`0.0`
[2] Graph mit 1 Knoten ohne Schleife	`0.0`
[3] Gerichteter, gewichteter Graph s = 0, t = 5	`8.0`
[4] Gerichteter, gewichteter Graph s = 0, t = 1	`4.0`
[5] Gerichteter, gewichteter Graph s = 0, t = 0	`0.0`
[6] Nicht zusammenhängender Graph unverbundene Knoten	`0.0`
[7] Nicht zusammenhängender Graph verbundene Knoten	`2.0`
[8] Graph zur Verifikation von Edmonds-Karp	`20000.0`

References

[1] Wikipedia: Zyklus, https://de.wikipedia.org/wiki/Zyklus_(Graphentheorie)

[2] Princeton University: Graph generator, https://algs4.cs.princeton.edu/41graph/GraphGenerator.java.html

[3] Robert Sedgewick, 2011: Algorithms, 4th edition, Pearson Education, Boston