Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Dr. Christian Schulz, Dr. Simon Gog
Michael Axtmann

3. Übungsblatt zu Algorithmen II im WS 2016/2017

http://algo2.iti.kit.edu/AlgorithmenII_WS16.php
{christian.schulz,michael.axtmann,sanders,simon.gog}@kit.edu

Aufgabe 1 (*Analyse: Äquivalenzrelation*)

Gegeben sei ein gerichteter Graph $G = (V, E)$ und folgende Relation

$$v \overset{*}{\longleftrightarrow} w \qquad \text{gdw.} \qquad \text{es gibt Pfade } \langle v, \ldots, w\rangle \text{ und } \langle w, \ldots, v\rangle \text{ in } G$$

für $v, w \in V$.
Zeigen Sie, dass $\overset{*}{\longleftrightarrow}$ eine Äquivalenzrelation ist.

Aufgabe 2 (*Analyse+Entwurf: Artikulationspunkte (*)*)

Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph. Ein Knoten v des Graphen G wird als *Gelenkpunkt* bezeichnet, wenn dessen Entfernen die Zahl der Zusammenhangskomponenten erhöht.

a) Zeigen Sie, dass es in jedem Graphen G *ohne Gelenkpunkte* und mit $|V| \geq 3$ immer mindestens ein Knotenpaar (i, j), $i, j \in V$, $i \neq j$ gibt, so dass zwei Pfade $P_1 = \langle i, \ldots, j\rangle$ und $P_2 = \langle i, \ldots, j\rangle$ existieren, die bis auf die Endpunkte knotendisjunkt sind, d.h.: $P_1 \cap P_2 = \{i, j\}$.

b) Beweisen Sie in jedem Graphen G *mit Gelenkpunkten* die Existenz eines Knotens v, für den gilt: Man kann einen Knoten w entfernen, so dass es von v aus keine Pfade mehr zu mindestens der Hälfte der verbleibenden Knoten gibt.

c) Zeigen Sie, dass in einem zusammenhängenden Graphen $G = (V, E)$ stets ein Knoten v existiert, so dass G nach Entfernen von v weiterhin zusammenhängend ist.

d) Vervollständigen Sie den angegebenen allgemeinen DFS-Algorithmus, so dass er in $O(|V| + |E|)$ alle Gelenkpunkte eines ungerichteten Graphen berechnet. Geben Sie an, was die Funktionen `init`, `root(s)`, `traverseTreeEdge(v,w)`, `traverseNonTreeEdge(v,w)` und `backTrack(u,v)` machen.
Überlegen Sie sich zunächst, wie Sie mit Hilfe der DFS-Nummerierung Gelenkpunkte erkennen können.

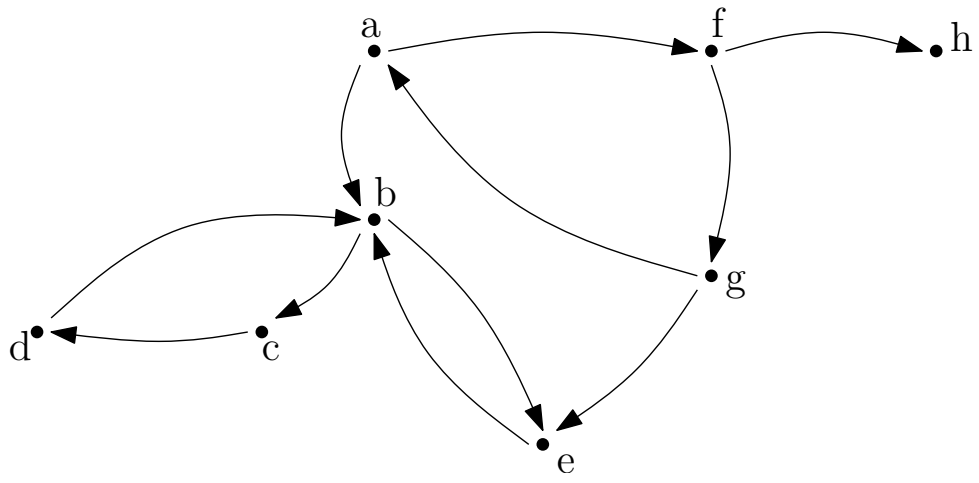
Depth-first search of graph $G = (V, E)$
unmark all nodes
init
for all $s \in V$ **do**
  if s is not marked **then**
    mark s
    root(s)
    DFS(s,s)
  end if
end for

```
procedure DFS(u,v : NodeID)
    for all (v, w) ∈ E do
        if w is marked then
            traverseNonTreeEdge(v,w)
        else
            traverseTreeEdge(v,w)
            mark w
            DFS(v,w)
        end if
    end for
    backtrack(u,v)
end procedure
```

Hinweis: Diese Aufgabe war Teil der Nachklausur Algorithmen II im letzten Semester!

Aufgabe 3 (*Rechnen: SCC mit Tiefensuche*)

Gegeben sei folgender Graph $G = (V, E)$:



Führen Sie den Algorithmus zur Bestimmung aller starken Zusammenhangskomponenten aus der Vorlesung auf dem Graph G aus. Geben Sie nach jedem Schritt den Zustand von `oReps`, `oNodes` und `component` an.

Aufgabe 4 (*Kleinaufgaben: Eigenschaften von Flüssen*)

a) Nach Vorlesung ist eine gültige Distanzfunktion $d(\cdot)$ für *Dinics Algorithmus* gegeben durch:

- $d(t) = 0$
- $d(u) \leq d(v) + 1 \quad \forall\ (u, v) \in G_f$

Zeigen Sie, falls $d(s) \geq n$, existiert kein *augmentierender Pfad*.

b) In der Vorlesung wurde gezeigt, dass die Laufzeit von *Dinics Algorithmus* für Graphen mit Kantengewichten gleich 1 (*unit edgeweights*) in $O((n+m)\sqrt{m})$ liegt. Vergleichen Sie diese Laufzeit zum *Ford Fulkerson Algorithmus*. Für welche Graphen mit *unit edgeweights* ist welcher der beiden Algorithmen schneller?

c) Sei $G = (V, E)$ ein gerichteter Graph, in dem maximale Flüsse berechnet werden sollen. Sei $e = (i, j) \in E$ ebenso wie $e' = (j, i) \in E$, d. h. G besitzt ein Paar entgegengesetzter Kanten. Außerdem sei $c(e) \geq c(e')$. Widerlegen Sie durch ein Gegenbeispiel:

Entfernt man e' aus E und reduziert $c(e) := c(e) - c(e')$, ändert sich der maximale Fluss nicht, d. h. man kann entgegengesetzte Kanten a-priori (für beliebige s und t) gegeneinander aufrechnen.

Aufgabe 5 (*Rechnen: Segmentierung mit Flüßen (*)s*)

Wir betrachten einen einfachen Fall für Bildbearbeitung. Die Vorder-/Hintergrundsegmentierung. Das Ziel dieses Prozesses ist es, ein Bild in Vorder und Hintergrund zu zerlegen. Die Transformation dafür weißt jedem Pixel $p_{i,j}$ des Bildes einen Knoten $v_{i,j}$ im Graphen zu. Für jedes Paar von benachbarten Pixeln $p_{i,j}$ und $p_{k,l}$ ($|i-k|+|j-l|=1$) fügen wir eine ungerichtete Kante $(v_{i,j}, v_{k,l})$ ein. Zusätzlich fügen wir je einen Knoten s für Vordergrund (Quelle) und einen Knoten t für Hintergrund (Senke) ein. Von Knoten s existiert eine gerichtete Kante zu jedem Knoten $p_{i,j}$ und von jedem Knoten $p_{i,j}$ existiert eine gerichtete Kante zu Knoten p. Wir definieren darüber hinaus folgende Kantengewichte:

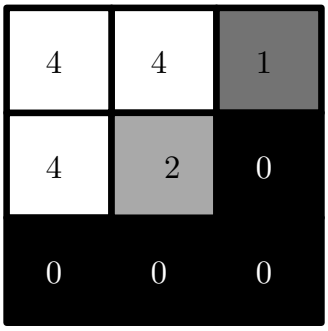
$$c(e=(u,v)) = \begin{cases} p_v(v) & u=s \\ p_h(u) & v=t \\ f(u,v) & sonst \end{cases}$$

Wobei mit $p_v(x)$ die Wahrscheinlichkeit gegeben ist, dass x Vordergrundknoten ist, mit $p_h(x)$ die Wahrscheinlichkeit für einen Hintergrundknoten und mit $f(x,y)$ eine Penaltyfunktion für das Trennen der beiden Knoten x und y. Für ein Graustufenbild B definieren wir
$p_v(x,y) = B[x,y]^2$, $p_h(x,y) = (4-B[x,y])^2$ sowie $f((x_1,y_1),(x_2,y_2)) = (4-|B[x_1,y_1]-B[x_2,y_2]|)^2$.
Hinweis: Diese Modellierung ist nur ein Beispiel und keine allgemeingültige Modellierung. Sie soll nur verdeutlichen wie Flow Alglorithmen für andere Probleme eingesetzt werden können.

a) Geben Sie den Flussgraphen für das unten angegebene Graustufenbild an.

b) Führen Sie einen augmenting Path Algorithmus auf dem entstandenen Graphen aus.

c) Wie würde die Segmentierung in Vorder- und Hintergrund im Bild als Ergebnis aussehen?



Aufgabe 6 (*Analyse: Königs Theorem*)

Das *Theorem von König* besagt, dass in jedem bipartiten Graphen $G=(V,\ E)$ die Größe eines Matchings größter Wertigkeit (*maximum-cardinality matching*) gleich der Größe einer minimalen Knotenüberdeckung (*minimal vertex cover*) ist.

Vertex Cover:
Ein *Vertex Cover* ist definiert als eine Teilmenge der Knoten $S \subseteq V$, so dass für alle Kanten $e=(u,v) \in E$ gilt $u \in S \vee v \in S$. Ein *minimales Vertex Cover* besitzt unter allen korrekten die kleinste Teilmenge an Knoten S.

Matching:
Ein *Matching* ist definiert als eine Teilmenge von Kanten $S \subseteq E$, so dass jeder Knoten $v \in V$ Endpunkt von höchstens einer Kante in S ist. Eine *maximales Matching* besitzt unter allen korrekten die größte Teilmenge an Kanten S.

Bipartiter Graph:
Ein bipartiter Graph enthält ausschließlich Kanten zwischen disjunkten Teilmengen der Knotenmenge:
$e \in E \leftrightarrow (u,v) \in S \times T,\ V = S \cup T,\ S \cap T = \varnothing$.

Beweisen Sie das *Theorem von König*.

Aufgabe 7 (*Analyse+Entwurf+Rechnen: Grenzüberwachung*)

Eine (eindimensionale) Grenzlinie soll durch ein Sensornetz überwacht werden. Zu diesem Zweck wurde eine große Anzahl an Sensorknoten unregelmäßig an der Grenze ausgebracht. Jeder Knoten kann einen Bereich der Grenze für eine gewisse Zeit proportional zu seiner Batteriekapazität überwachen. Die Grenze gilt als vollständig gesichert, wenn jeder Abschnitt der Grenzlinie von mindestens einem Sensorknoten abgedeckt ist. Aufgrund der großen Menge an Knoten sind ihre Überwachungsbereiche stark überlappend. Daher müssen nicht immer alle Knoten aktiv sein, um eine vollständige Sicherung der Grenze zu gewährleisten. So kann Energie gespart werden und die maximale Dauer der Grenzsicherung erhöht werden.
Durch die unregelmäßige Anbringung der Knoten und durch große Fertigungstoleranzen in der Batteriekapazität und dem Überwachungsbereich (*man hat unbedingt beim billigsten Hersteller einkaufen müssen...*) ist zunächst nicht klar, wie lange die Grenze maximal vollständig gesichert werden kann. Glücklicherweise wurden die Positionen der Knoten und ihre jeweiligen Kapazitäten und Detektionsbereiche protokolliert und können verwendet werden, um diese Frage zu beantworten.

a) In der Vorlesung haben Sie Flussprobleme mit beschränkten Kantenkapazitäten $c(e)$ kennengelernt. Ebenso können Flussprobleme mit beschränkten Knotenkapazitäten $c(v)$ sinnvoll sein. In diesem Fall darf für einen gültigen Fluss die Summe der in den Knoten ankommenden bzw. ausgehenden Flüsse die Kapazität des Knotens nicht überschreiten. Außerdem muss wie bisher für jeden Knoten (außer der Quelle und Senke) die Summe der ankommenden Flüsse gleich der Summe der ausgehenden Flüsse sein.

   Erklären Sie, wie maximale Flüsse mit Knotenkapazitäten berechnet werden können. Begründen Sie kurz, warum Ihr Ansatz einen zulässigen und optimalen Fluss berechnet.

b) Konstruieren Sie ein Flussnetzwerk, das das oben beschriebene Problem der Bestimmung einer maximalen Dauer für die vollständige Grenzüberwachung lösen kann.

   Hinweis: Jeder Knoten entspricht einem Sensorknoten. Batteriekapazität kann als äquivalent zur Flussmenge betrachtet werden.

c) Erstellen Sie ein Flussnetz, das dem folgenden Sensornetz entspricht. Wie lange kann dieses Netz die Grenze im Bereich $[0, 13]$ überwachen? Welche Sensorknoten müssen wann aktiv sein?

   Format der Angaben: $x_{nodeID} = \{[begin_range, end_range], capacity\}$

   $x_1 = \{[0, 5], 4\}$
   $x_2 = \{[0, 7], 3\}$
   $x_3 = \{[4, 9], 2\}$
   $x_4 = \{[3, 8], 5\}$
   $x_5 = \{[8, 13], 5\}$
   $x_6 = \{[7, 11], 3\}$
   $x_7 = \{[11, 15], 2\}$

   Hinweis: Bevor Sie langwierig einen maximalen Fluss berechnen, versuchen Sie ihn durch *scharfes Hinschauen* zu bestimmen.

Aufgabe 8 (*Implementierung: Ford Fulkerson*)

Implementieren Sie den Algorithmus von Ford Fulkerson in `C++` mit dem KaHIP Framework. Laden Sie sich den KaHIP Fork[1] herunter und wechseln Sie in den Branch FordFulkerson. Beachten Sie, dass die Lösung dieser Aufgabe bereits als neuer Commit im Branch FordFulkerson veröffentlicht wurde. Die leeren Methodenrümpfe sind im Branch FordFulkerson unter `sha=1dddb72` eingecheckt, die Lösung ist im Branch FordFulkerson unter `sha=3a113eb` bereitgestellt. Das KaHIP Framework stellt bereits einen Flussgraph (`lib/data_structure/flow_graph.h`) bereit. Diese Datenstruktur implementiert die Basisfunktionen von Residualgraphen. Erweitern Sie die Funktionalität des Residualgraphen um die Funktionen

```
FlowType getEdgeResFlow(NodeID source, EdgeID e);
bool isSaturated(NodeID source, EdgeID e);
void increaseEdgeFlow(NodeID source, EdgeID e, FlowType flow);
EdgeID get_first_flow_edge(NodeID node);
EdgeID get_next_flow_edge(NodeID node, EdgeID e);
```

Implementieren Sie die Methode

```
Die Funktion FlowType res_graph_dfs(graph& G,
                std::vector<NodeID>& nodes, std::vector<EdgeID>& edges,
                const NodeID s, const NodeID t);
```

in der Datei `res_graph_dfs.hpp`. Diese Methode soll einen augmentierenden Pfad von s nach t mittels Tiefensuche suchen. Findet die DFS Suche einen augmentierenden Pfad, so gibt die Methode den maximalen Fluss des gefundenen Pfades zurück. In diesem Fall werden im Vector `nodes` die Knoten auf dem Pfad und im Vector `edges` die dazugehörenden Kanten gespeichert. Die Methode gibt eine Kapazität von 0 zurück, falls kein augmentierender Pfad im Residualgraph gefunden wurde.
Nutzen Sie die DFS Suche um den Ford Fulkerson Algorithmus in der Datei `app/ford_fulkerson/ford_fulkerson.cpp` zu implementieren. Compilieren Sie Ihre Implementierung danach mithilfe des Skripts `compile.sh` und führen Sie den Algorithmus auf der Kommandozeile aus:

```
> ./optimized/maxflow ./examples/flow_graphs/af_shell9.graph.hierarchy.5.h.inp
```

Im Verzeichnis `./examples/flow_graphs/` finden Sie einige Flussgraphen. Testen Sie Ihren Algorithmus zu Beginn mit kleinen Eingabeinstanzen!

Ausgabe: 22.11.2016
Abgabe: keine Abgabe, keine Korrektur

[1] `https://github.com/MichaelAxtmann/KaHIP`