

2. Übungsblatt zu Algorithmen II im WS 2018/2019

http://algo2.iti.kit.edu/AlgorithmenII_WS18.php
{sanders, lamm, hespe}@kit.edu

Aufgabe 1 (Rechnen: Monotone ganzzahlige Priority Queues)

Bei einer Ausführung von *Dijkstra's Algorithmus* wird folgender Ausschnitt an *Priority Queue* Operationen protokolliert:

- ...
- insert(a, 06 [00110]) (Parameter: Knotenbezeichnung, Distanz [Distanz binär])
 - insert(b, 10 [01010])
 - insert(c, 07 [00111])
 - deleteMin()
 - deleteMin()
 - insert(d, 12 [01100])
 - deleteMin()
 - insert(e, 16 [10000])
- ...

Zusätzlich wissen Sie, dass das maximale Kantengewicht im Graphen $C = 6$ beträgt und dass vor der ersten protokollierten Operation das letzte enthaltene Element aus der *Priority Queue* entfernt wurde. Dieses hatte den Wert $min = 5$.

- a) Führen Sie die Operationen auf einer *Bucket Queue* aus. Geben Sie den Zustand der Datenstruktur nach jeder Operation an.
- b) Wieviele *Buckets* werden für eine Ausführung auf einem *Radix Heap* benötigt? Führen Sie die Operationen auf einem *Radix Heap* aus. Geben Sie den Zustand der Datenstruktur und den Wertebereich der *Buckets* nach jeder Operation an.

Aufgabe 2 (Analyse: Laufzeit von Dijkstra's Algorithmus)

Gegeben sei ein gerichteter Graph $G = (V, E)$ mit $|V| = n$ und $|E| = m$, sowie eine Kantengewichtungsfunktion $c : E \rightarrow \mathbb{R}_0^+$.

- a) Beweisen Sie die Behauptung aus der Vorlesung, dass für $m = \Omega(n \log n \log \log n)$ Dijkstra's Algorithmus mit einem *binary heap* eine durchschnittliche Laufzeit von $O(m)$ besitzt.
- b) Eine spezielle *Priority Queue* habe folgende Laufzeiteigenschaften:
- **insert:** $O(\log n)$
 - **decreaseKey:** $O(1)$
 - **deleteMin:** $O(\sqrt{m})$

(ob eine Datenstruktur mit diesen Eigenschaften existiert und Dijkstra's Algorithmus mit ihr korrekt arbeitet, ist eine andere Frage, aber wir nehmen für diese Aufgabe an es ginge :-)

Geben Sie eine kleinste obere Schranke für die Laufzeit von Dijkstra's Algorithmus unter Verwendung dieser *Priority Queue* an. Unter welcher Bedingung an das Verhältnis der Anzahl Knoten n zu Kanten m wird die Laufzeit linear in der Eingabegröße? Die Eingabe erfolgt in Form einer Adjazenzliste.

Aufgabe 3 (Einführung+Analyse: Bidirektionaler Dijkstra)

In Vorlesung und Saalübung wurde eine bidirektionale Variante von Dijkstra's Algorithmus angesprochen, die in dieser Aufgabe näher untersucht werden soll.

Zur Wiederholung:

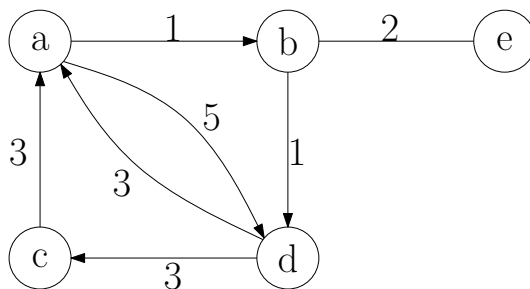
Gegeben sei –wie üblich– ein gerichteter Graph $G = (V, E)$ mit $|V| = n$ und $|E| = m$, sowie eine Kantengewichtungsfunktion $c : E \rightarrow \mathbb{R}_0^+$. Gesucht ist der kürzeste Pfad $p = \langle s, \dots, t \rangle$ zwischen zwei Punkten $s, t \in V$.

Eine bidirektionale Suche löst dieses Problem wie folgt: Es werden zwei unidirektionale Suchen mit Dijkstra's Algorithmus gestartet. Die *Vorwärtssuche* beginnt bei Knoten s und operiert auf dem normalen Graphen G , auch *Vorwärtsgraph* genannt. Die *Rückwärtssuche* beginnt bei Knoten t und operiert auf dem *Rückwärtsgraph* $G^r = (V, E^r)$ mit Kantengewichtungsfunktion c^r . Dieser Graph entsteht aus G durch Umkehrung aller Kanten. Der Algorithmus scannt abwechselnd einen Knoten in der Vorwärtssuche und in der Rückwärtssuche, beginnend mit der Vorwärtssuche.

Wird während des Scans von Knoten u Kante (u, v) relaxiert, so wird überprüft, ob die Distanz $d_{\text{forward}}[v] + d_{\text{backward}}[v]$ kleiner ist als die momentan minimale gefundene Distanz von s nach t und diese gegebenenfalls angepasst ($d_{\text{forward}}[v]$ gibt die bisher kürzeste gefundene Distanz von s nach v in der Vorwärtssuche und $d_{\text{backward}}[v]$ die bisher kürzeste gefundene Distanz von v nach t in der Rückwärtssuche an).

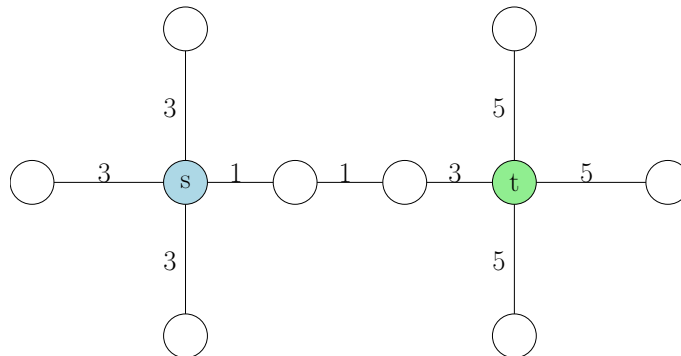
Sobald ein Knoten in einer Richtung gescannt werden soll, der bereits in der anderen Richtung gescannt worden ist, kann die Suche beendet werden (*Abbruchbedingung*). Die aktuelle minimale gefundene Distanz ist dann die tatsächliche minimale Distanz zwischen s und t .

- a) Zeichnen Sie den Rückwärtsgraph G^r zum angegebenen Graphen. Geben Sie die Kantengewichte $c(a, d)$, $c^r(a, d)$ sowie $c(b, e)$, $c^r(b, e)$ an.



(Kante (b, e) ist eine bidirektionale [bzw. ungerichtete] Kante)

- b) Geben Sie an, in welcher Reihenfolge der unten angegebene Graph durchlaufen wird.



- c) Zeigen Sie, dass die Abbruchbedingung korrekt ist.
 d) Wann kann es passieren, dass die Suche nach dem Scan von Knoten u beendet wird, dieser aber nicht Teil des kürzesten Weges ist. Geben Sie ein Beispiel an.

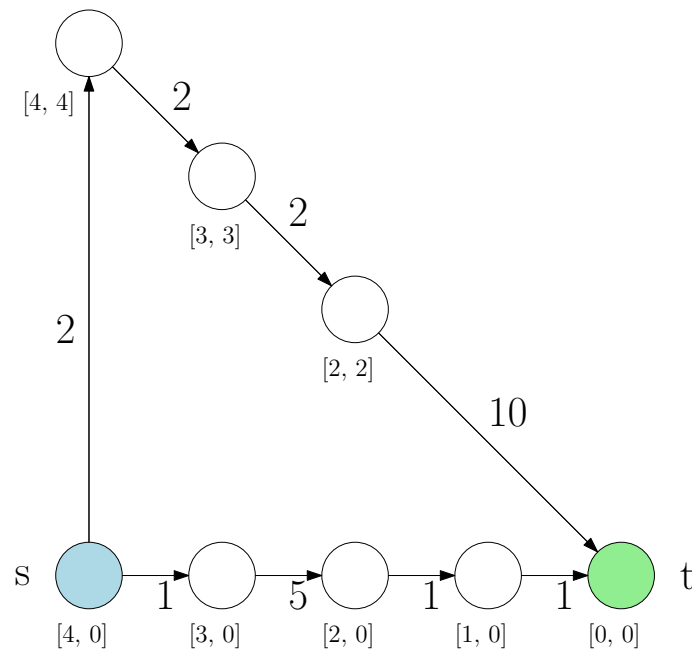
Aufgabe 4 (Rechnen: A* Suche)

Gegeben sei der unten abgebildete Graph. An den Kanten sind Kosten für die Nutzung der Verbindung eingetragen und die Knoten tragen Ortskoordinaten.

- a) Ergänzen Sie den gegebenen Graphen um Knotenpotentiale für eine A* Suche von s nach t . Verwenden Sie Knoten t als Landmarke und die Manhattan-Distanz ($\hat{=}$ Einsnorm $\|\cdot\|_1$) als Abschätzung für die Entfernung zum Ziel.

Hinweis: $\|\cdot\|_1 : \|(x_1, y_1), (x_2, y_2)\|_1 = y_2 - y_1 + x_2 - x_1$.

- b) Tragen Sie die reduzierten Kantengewichte in den Graphen ein.
- c) Wieviele `deleteMin` Operationen führt die A* Suche auf dem Graphen aus? Wieviele eine normale Suche mit Dijkstra's Algorithmus?



Aufgabe 5 (Kleinaufgaben: A* Suche)

- a) Sei $\text{pot}(\cdot)$ eine gültige Potentialfunktion für die A* Suche nach Knoten t in Graph $G(V, E)$.
Überprüfen Sie, ob

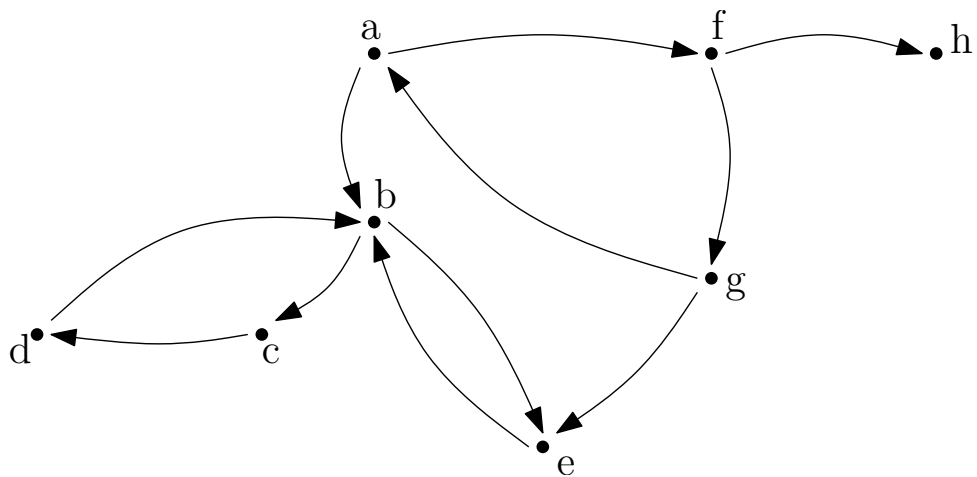
$$\text{pot}^c = \text{pot} + c, \quad c = \text{const.}$$

ebenfalls eine gültige Potentialfunktion darstellt.

- b) Kann es vorkommen, dass eine A* Suche mehr Knoten absucht als eine Suche mit Dijkstra's Algorithmus für die gleiche Anfrage? Begründen Sie warum nicht oder geben Sie ein Beispiel an.

Aufgabe 6 (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 7 (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.

- 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 $P_1 = \langle i, \dots, j \rangle$ und $P_2 = \langle i, \dots, j \rangle$ existieren, die bis auf die Endpunkte knotendisjunkt sind, d.h.: $P_1 \cap P_2 = \{i, j\}$.
- 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.
- 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.
- 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 $(v, w) \in E$ **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

Ausgabe: 06.11.2018

Abgabe: keine Abgabe, keine Korrektur