Albert-Ludwigs-Universität
Institut für Informatik
Prof. Dr. Fabian Kuhn
P. Bamberger, Y. Maus, P. Schneider

Informatik 2 - Sommersemester 2018
Korrekturanweisung Übungsblatt 8

Abgabe: Mittwoch, 27. Juni, 14:00 Uhr

Aufgabe 1: Prims Algorithmus ohne `Decrease-Key` *(5 Punkte)*

Gegeben sei die Implementierung einer Prioritätswarteschlange als *Min-Heap*. Geben Sie eine Variante von Prims Algorithmus in Pseudo-Code an, welche nur die Operationen `insert` und `delete-min` benutzt und dabei die gleiche asymptotische Laufzeit $\mathcal{O}(m \log n)$ wie Prims Algorithmus in der Variante der Vorlesung mit Min-Heap aufweist. Begründen Sie kurz die Laufzeit.

Musterlösung

Statt eines `Decrease-Key` fügen wir die entsprechenden Knoten ein weiteres mal mit der aktualisierten Distanz in H ein. Gleichzeitig markieren wir Knoten die das erste mal aus H mittels `deleteMin` extrahiert wurden und verwerfen bereits markierte Knoten direkt wieder. Insgesamt haben wir $\mathcal{O}(n+m) = \mathcal{O}(m)$ `insert` und `delete-min` Operationen. Wir fügen $\mathcal{O}(n+m)$ Knoten in H ein. Das heißt jede Operation dauert $\mathcal{O}\big(\log(n+m)\big) = \mathcal{O}\big(\log(n)\big)$ Zeitschritte. Insgesamt haben wir also $\mathcal{O}(m \log n)$.

Algorithm 1: `Prim'`(G, w, s)

```
H ← new Min-Heap; A ← ∅
for all u ∈ V\{s} do
    H.insert(u, ∞); u.marked ← false; α(u) ← NULL
H.insert(s, 0); s.marked← false; α(s) ← NULL
while H is not empty do
    u ← H.deleteMin()
    if not u.marked then
        u.marked ← true
        for all neighbors v of u do
            if w(u, v) < d(v) then
                H.insert(v, w(u, v)); α(v) ← u
    if u ≠ s then
        A ← A ∪ {{u, α(u)}}
```

Aufgabe 2: Dijkstras Algorithmus und Kantengewichte *(6 Punkte)*

Sei G ein gewichteter Graph mit Gewichtsfunktion $w : E \to \mathbb{R}$ (d.h. negative Gewichte sind möglich).

(a) Ein Mitarbeiter der Professur hat die folgende Idee um in G kürzeste Pfade mittels Dijkstras Algorithmus zu finden. Sei G' der Graph der entsteht wenn man den Betrag der Länge der kürzesten Kante auf alle Kanten addiert. Da G' nun keine Kanten negativer Länge mehr hat, findet Dijkstra die korrekten kürzesten Pfade in G', wodurch wir auch die kürzesten Pfade in G erhalten. Beweisen oder widerlegen Sie die Korrektheit dieses Verfahrens. *(2 Punkte)*

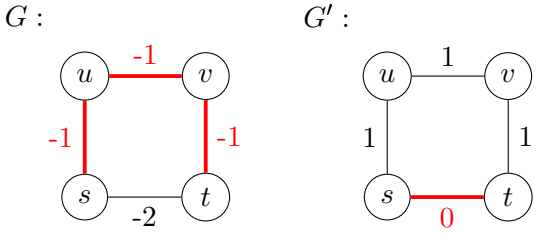
Sei nun G ein gewichteter Graph mit Gewichtsfunktion $w : E \to \{0, \ldots, W\}$ mit $W \in \mathbb{N}$ konstant.

(b) Modifizieren Sie Dijkstras Algorithmus so, dass er kürzeste Pfade in $\mathcal{O}(W \cdot |V| + |E|)$ berechnet. Begründen Sie die Laufzeit. *(4 Punkte)*

Hinweise: Es reicht die Prioritätswarteschlange anzupassen. Sie erhalten Teilpunkte wenn Sie die Aufgabe mit der Annahme lösen, dass W die maximale Länge aller kürzesten Pfades in G ist.

Musterlösung

(a) Wir geben ein einfaches Gegenbeispiel mit negativen Kantengewichten, in welchem ein kürzester Pfad in G' keinem kürzesten Pfad in G entspricht (kürzeste Pfade in rot):



(b) Wir definieren eine Prioriätswarteschlange H als Array der Länge $|V| \cdot W$ welches doppelt verkettete Listen enthält. Die i-te Liste $H[i]$ soll dabei lediglich Knoten mit Priorität i enthalten. *(1 Punkt)*

Wird H.`insert`$(v, \delta(v))$ ausgeführt fügen wir v in die Liste an der Stelle $H[\delta(v)]$ in $\mathcal{O}(1)$ ein. Alle H.`insert` Operationen zusammen dauern $\mathcal{O}(|V|)$ Zeitschritte. *(0,5 Punkte)*

Für H.`decreasyKey`$(pointer(v), \delta(v))$ bekommen wir einen Pointer mit übergeben, der auf das Listenelement in der Datenstruktur zeigt dessen Priorität wir verringern wollen (ein Dictionary das jedem Knoten sein Listenelement in H in $\mathcal{O}(1)$ zuordnet können wir separat verwalten). Dieses Listenelement entfernen wir in $\mathcal{O}(1)$ (wir müssen nur Pointer umsetzen) und fügen es mit neuer Priorität in $\mathcal{O}(1)$ wieder ein (s.o.). Alle H.`decreasyKey` Operationen zusammen dauern $\mathcal{O}(|E|)$ Zeitschritte. *(1 Punkt)*

Die scheinbar aufwändigste Operation ist H.`deleteMin`(). Hier müssen wir das Array H iterieren um die erste Liste zu finden die nicht leer ist. Aus der ersten nicht leeren Liste entfernen wir einen Knoten in $\mathcal{O}(1)$ und geben ihn zurück. Iterieren wir die Liste jedes mal vom Anfang an kann das jedes mal $\mathcal{O}(|V| \cdot W)$ Zeitschritte kosten.

Der Trick ist, sich zu merken bei welcher Liste (d.h. Distanz von s) man zuletzt war und die Iteration ab dort zu starten. Da Dijkstra alle Knoten mit Distanzen *echt kleiner* derjenigen des zuletzt betrachteten Knotens bereits abgearbeitet haben muss, gibt es keine Knoten mehr mit niedrigerer Priorität (Distanz zu s). Mit dieser Vorgehensweise iterieren wir für alle H.`deleteMin`() Operationen zusammengenommen nur einmal über H. Die Gesamtdauer ist also $\mathcal{O}(W \cdot |V|)$. *(1,5 Punkte)*

Aufgabe 3: Dijkstra Implementieren *(9 Punkte)*

Dijkstras Algorithmus findet alle kürzesten Wege von einem Startknoten $s \in V$ zu allen anderen Knoten in einem Graphen $G = (V, E)$ (mit positiven Kantengewichten). Nun wählen wir zusätzlich zu einem Startknoten s auch einen Zielknoten $t \in V$. Wenn man Dijkstras Algorithmus von s aus startet und abbricht, sobald der Knoten t aus der Prioritätswarteschlange entfernt wird, so erhält man einen kürzesten Weg zwischen s und t. Wir nennen dies Dijkstras s-t-Algorithmus.

(a) Lesen Sie die Datei `Freiburg.txt`[1] ein (eine Abstraktion des Freiburger Straßenverkehrsnetzes) und erstellen Sie einen Graphen mit der vorgegebenen Klasse `Graph.py` welche eine Adjazenliste implementiert. *(3 Punkte)*

[1] Alle benutzten Dateiformate sind am Ende des Übungsblatts erklärt.

(b) In der Datei `points.txt`[1] sind zwei Knoten IDs gespeichert, welche die Knoten s und t darstellen sollen. Implementieren Sie Dijkstras s-t-Algorithmus wobei Sie die Datenstruktur *Priority Queue* aus der Python Bibliothek benutzen dürfen. Bestimmen Sie mit Ihrem Algorithmus den Pfad von s nach t und speichern Sie das Resultat in der csv-Datei `dijkstra.csv`[1] im SVN. Laden Sie die Datei außerdem auf die Web-Seite

`http://www.gpsvisualizer.com`

hoch um den Pfad zu visualisieren. Fügen Sie die Visualisierung Ihrer Lösung hinzu (ein Screenshot ist ausreichend). Um ein gutes Bild zu bekommen, wählen Sie auf der Webseite die Option "Plot data points". *(6 Punkte)*

Dateiformate

`Freiburg.txt`

Zeile 0: *int* N - Anzahl der Knoten im Graph
Zeilen $1 \dots N$: Knoten. Erste Zahl ist *int* ID, zweite Zahl *float* Breitengrad des Knotens, dritte Zahl *float* Längengrad des Knotens. Die Zahlen sind durch Leerzeichen getrennt.
Zeile $N + 1$: *int* M - Anzahl der Kanten im Graph
Zeilen $N + 2 \dots N + M + 1$ - Kanten. Die ersten zwei Zahlen sind die *int* IDs von inzidenten Knoten. Die dritte Zahl ist das *float* Gewicht der Kante. Die Zahlen sind durch Leerzeichen getrennt.

`points.txt`

Zeile 0: ID des Startknotens
Zeile 1: ID des Zielknotens

`dijkstra.csv`

Jede Zeile muss den Breitengrad und Längengrad als *float* und durch ein Komma getrennt enthalten.