Albert-Ludwigs-Universität
Institut für Informatik
Prof. Dr. F. Kuhn
M. Fuchs, G. Schmid

Algorithmen und Datenstrukturen
Sommersemester 2022
Musterlösung Übungsblatt 10

Abgabe: Dienstag, 2. Juli, 2024, 10:00 Uhr

Aufgabe 1: Dijkstra Lowerbound *(10 Punkte)*

Wir haben in der Vorlesung gesehen, dass die Laufzeit von Dijkstra mit Fibonacci heaps $O(m+n \log n)$ ist. Doch ist dies die echte Laufzeit von Dijkstra, oder ist unsere Analyse nur nicht gut genug, oder vielleicht unser Heap nicht gut genug? Um diese Frage zu beantworten zeigen wir das die Abhängigkeit in m und die Abhängigkeit in n beide korrekt sind.

(a) Argumentieren Sie, dass jeder Algorithmus für SSSP (Single Source Shortest Paths) mindestens $\Omega(m)$ Zeit braucht. Hier reicht eine Intuitive Beschreibung. *(1 Punkt)*

(b) Beweisen Sie, dass Dijkstra die kürzesten Wege in sortierter Reihenfolge ausgibt. Für einen Startknoten v wird also für alle anderen Knoten $u \neq w$ die Distanz $d(v,u)$ vor $d(v,w)$ ausgegeben, wenn $d(v,u) < d(v,w)$. Wir erwarten eine genaue Beschreibung und saubere formale Argumente. *(4 Punkte)*

(c) Beweisen Sie, dass Dijkstra $\Omega(n \log n)$ Zeit benötigt, wenn der Algorithmus mit einem Vergleichsbasierten Heap implementiert wird. Die Idee ist folgende: Reduzieren Sie das Sortieren von n Zahlen auf das Problem kürzeste Wege auszurechnen. (Also gegeben n Zahlen in einem Array kreieren Sie eine Instanz von SSSP.) Wir erwarten eine genaue Beschreibung und saubere formale Argumente. *(5 Punkte)*

Musterlösung

(a) Angenommen der Algorithmus braucht nur $T(m) \in o(m)$ Zeit, dann existiert ein m_0 so dass $T(m_0) \leq \frac{1}{2}m_0$[1]. Also sieht der Algorithmus nicht mal die Hälfte aller Kanten. Wir stellen uns eine Instanz vor, in der diese ungesehenen Kanten die kleinsten Gewichte haben. In dieser Instanz kann der Algorithmus nicht korrekt sein,weil er nicht wissen kann ob diese ungesehenen Kanten nicht vielleicht Teil der kürzesten Wege sind.

(b) Dijkstra markiert einen Knoten u wenn der kürzeste Pfad korrekt ausgerechnet wurde. Dies passiert wenn u aus der Priority Queue entfernt wird. Sei also v der Startknoten und $u \neq w$ zwei weitere Knoten mit $d(v,u) < d(v,w)$. Aus dieser Ungleichung folgt auch $d(v,p_1) \leq \ldots \leq d(v,p_k) < d(v,w)$ für den kürzesten Pfad $(v = p_0, p_1, \ldots, p_k = u)$[2]. Da p_1 ein direkter Nachbar von v ist, wird p_1 im ersten Schritt in den Heap mit eingefügt. Wenn ein p_i markiert wird, wird im selben Schritt p_{i+1} eingefügt. Also gilt: Solange $p_k = u$ noch nicht als Minimum aus dem Heap entfernt wurde, befindet sich noch ein Element aus $\{p_1, \ldots, p_k = v\}$ im Heap. Da $d(v,p_1) \leq \ldots \leq d(v,p_k) < d(v,w)$ würde jedes dieser Elemente vor w aus dem Heap entfernt werden und somit muss zuerst der gesamte Pfad von v nach u abgearbeitet werden, bevor $d(v,w)$ ausgegeben wird.

[1] Das ist eine Direkte Konsequenz von $T(m) \in o(m)$ für $c = \frac{1}{2}$

[2] vgl. Optimalität von Teilpfaden aus der Vorlesung

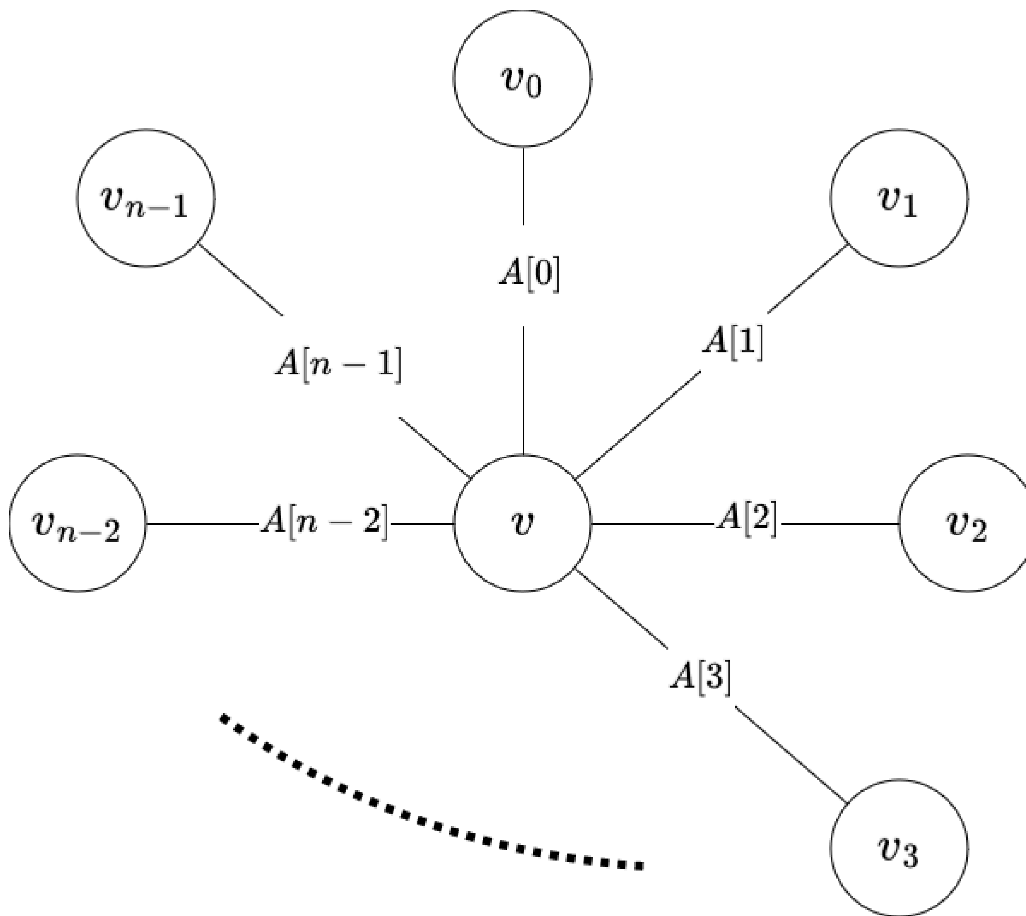

Figure 1: Die Konstruktion um das Sortieren von n Zahlen als SSSP Instanz zu lösen.

(c) Angenommen Dijkstra läuft in Zeit $T_{Dij}(n)$ wir zeigen, dass wir n Zahlen in $O(n) + T_{Dij}(n+1)$ sortieren können. Gegeben n Zahlen in einem Array A, wir generieren einen Stern-Graph mit Mittelknoten v und n Nachbarn $v_0, \ldots, v_{n-1}$ jede Kante $\{v, v_i\}$ hat Gewicht $w(\{v, v_i\}) = A[i]$. Vergleiche Figure 1 für eine Illustration der Konstruktion. Trivialerweise sind die direkten Kanten $\{v, v_i\}$ gleichzeitig die kürzesten Wege von v zu den äußeren Knoten.
Wir rufen nun Dijkstra auf dieser neuen Instanz mit Startknoten v auf. Nach b) werden die kürzesten Wege und damit auch die Kantengewichte in sortierter Reihenfolge ausgegeben. Diese Reihenfolge ist also auch eine sortierte Reihenfolge der Zahlen des Arrays.
Die SSSP Instanz hat $n+1$ Knoten und n Kanten und kann mit $O(n)$ Zeit konstruiert werden. Die sortierte Reihenfolge am Ende zurück in A zu schreiben braucht auch nur $O(n)$ Zeit. Damit haben wir die Zahlen in $O(n) + T_{Dij}(n+1)$ sortiert, wie am Anfang behauptet.
Angenommen $T_{Dij}(n) \in o(n \log n)$ dann könnten wir also auch in $o(n \log n)$ sortieren, was ein Wiederspruch zu dem Vergleichsbasierten Sortier-Lowerbound wäre.

Aufgabe 2: Währungsarbitrage *(10 Punkte)*

Gegeben seien n Währungen $w_1, \ldots, w_n$. Die Umrechungskurse der Währungen seien in einer $n \times n$-Matrix A mit Einträgen a_{ij} $(i, j \in \{1, \ldots, n\})$ gegeben. Eintrag a_{ij} ist der Umrechnungskurs von w_i nach w_j, d.h. für eine Einheit von w_i bekommt man a_{ij} Einheiten von w_j.
Gegeben eine Währung w_{i_0} möchte man herausfinden, ob es eine Folge $i_0, i_1, \ldots, i_k$ gibt, sodass man Gewinn macht, wenn man eine Einheit von w_{i_0} zu w_{i_1} tauscht, danach zu w_{i_2} etc. bis zu w_{i_k} und schließlich wieder zurück zu w_{i_0}.

(a) Formulieren Sie die Fragestellung als Graphproblem. Definieren Sie dazu einen geeigneten Graphen

sowie eine Bedingung, welche der Graph genau dann erfüllt, wenn es eine Folge von Währungen wie oben beschrieben gibt. *(4 Punkte)*

(b) Geben Sie einen Algorithmus an, welcher in $\mathcal{O}(n^3)$ Zeitschritten entscheidet, ob es eine Folge von Währungen wie oben beschrieben gibt. Begründen Sie Laufzeit und Korrektheit. *(6 Punkte)*

Hinweis: Es gilt $a \cdot b > 1 \iff (-\log a) + (-\log b) < 0$.

Musterlösung

(a) Wir definieren einen gewichteten Graphen $G = (V, E, w)$ mit $V = \{1, \ldots, n\}$, $E = V^2$ (d.h. der Graph ist gerichtet und vollständig) und $w(i, j) = a_{ij}$ (d.h. A entspricht der Adjazenzmatrix). Eine Folge von Währungen wie beschrieben gibt es genau dann, wenn es einen Kreis $(i_0, i_1, \ldots, i_k, i_0)$ gibt, so dass

$$\prod_{j=0}^{k-1} w(i_j, i_{j+1}) \cdot w(i_k, i_0) > 1 \ . \tag{1}$$

(b) Wie ersetzen in der Adjazenzmatrix a_{ij} durch $-\log a_{ij}$, d.h. wir definieren einen Graphen $G = (V, E, w')$ mit V und E wie oben und $w'(i, j) = -\log w(i, j)$. Wir wenden Bellman-Ford mit Startpunkt i_0 an. Dieser prüft, ob es einen negativen Kreis gibt, d.h. Knoten $i_0, \ldots, i_k$ mit

$$\begin{aligned}
& \sum_{j=0}^{k-1} w'(i_j, i_{j+1}) + w'(i_k, i_0) < 0 \\
\iff \quad & \sum_{j=0}^{k-1} -\log w(i_j, i_{j+1}) - \log w(i_k, i_0) < 0 \\
\iff \quad & \sum_{j=0}^{k-1} \log w(i_j, i_{j+1}) + \log w(i_k, i_0) > 0 \\
\iff \quad & \log \left(\prod_{j=0}^{k-1} w(i_j, i_{j+1}) \cdot w(i_k, i_0) \right) > 0 \\
\iff \quad & \prod_{j=0}^{k-1} w(i_j, i_{j+1}) \cdot w(i_k, i_0) > 1 \ .
\end{aligned}$$

Das heißt der Algorithmus prüft, ob Bedingung (1) aus Teil (a) erfüllt ist. Die Laufzeit von Bellman-Ford ist $\mathcal{O}(|V| \cdot |E|)$. Mit $|V| = n$ und $|E| = n^2$ erhalten wir also die Laufzeit $\mathcal{O}(n^3)$.