Karlsruher Institut für Technologie

Institut für Theoretische Informatik
T.T.-Prof. Dr. Thomas Bläsius
https://scale.iti.kit.edu

Übungsblatt 14

Algorithmen I – Sommersemester 2022

Abgabe im ILIAS bis 03.08.2022, 14:00 Uhr

Bitte beschrifte Deine Abgabe gut sichtbar mit Deinem Namen und Deiner Matrikelnummer. Achte insbesondere bei handschriftlichen Abgaben auf Lesbarkeit und genügend Platz für Korrektur-Anmerkungen. Die Abgabe erfolgt über das Übungsmodul in der Gruppe Deines Tutoriums im ILIAS. Gib Deine Ausarbeitungen in *einer* PDF-Datei ab. Achte darauf, effiziente Algorithmen zu formulieren, also solche mit möglichst geringer asymptotischer Laufzeit!
Wenn du die Korrektheit eines Algorithmus begründen oder dessen Laufzeit analysieren sollst, tue dies getrennt von der Beschreibung des Algorithmus.
Wenn nicht anders spezifiziert oder aus dem Kontext ersichtlich, bezeichnen wir mit Graph einen einfachen ungerichteten Graphen.

Aufgabe 1 - Baobabs (0+5 Punkte)

Ein *Binärbaum* ist ein Baum, bei dem jeder innere Knoten genau zwei Kinder hat. Gegeben sei ein beliebiger Baum $T = (V, E)$. Beschreibe einen Algorithmus, der einen Teilbaum $T' = (V', E')$ von T ausgibt, sodass T' ein Binärbaum und $|V'|$ maximal ist. Begründe die Korrektheit deines Algorithmus. Nenne und begründe außerdem seine asymptotische Laufzeit. (5 Punkte)

Lösung 1

Im folgenden nehmen wir an, dass T gewurzelt ist (der andere Fall wird später betrachtet). Zunächst bestimmen wir für jeden Knoten die Lage, auf der er sich in G befindet. Dabei liegt $t \in V$ auf Lage l, wenn der kürzeste s-t-Pfad in T Länge l hat. Sei zudem h die Höhe von T.
Wir können das Problem, den größten Binärbaum in T zu finden, knotenweise betrachten, indem wir für jeden Knoten $v \in V$ den größten Binärbaum suchen, der im Teilbaum unter v enthalten ist. Die Frage nach dem größten Binärbaum in T ist dann die Frage nach dem größten Binärbaum unter s.

Zuerst arbeiten wir T lagenweise von der untersten zur obersten Lage ab. Für jeden Knoten $v \in V$ speichern wir dabei die Anzahl Knoten $\texttt{maxRooted}(v)$ im größten an v gewurzelten Binärbaum und die Anzahl Knoten $\texttt{maxTotal}(v)$ im größten Binärbaum, der im unter v liegt.
Ist v ein Blatt, so ist $\texttt{maxRooted}(v) = \texttt{maxTotal}(v) = 1$. Ansonsten greift einer der beiden folgenden Fälle:

1. v hat nur ein Kind w in G. Dann ist $\texttt{maxRooted}(v) = 1$ und $\texttt{maxTotal}(v) = \texttt{maxTotal}(w)$.

2. v hat mindestens zwei Kinder. Seien r_1, r_1 die Kinder von v mit den größten maxRooted-Werten und sei t dasjenige Kind mit dem größten maxTotal-Wert.
   Dann ist $\texttt{maxRooted}(v) = \texttt{maxRooted}(r_1) + \texttt{maxRooted}(r_2) + 1$ und $\texttt{maxTotal}(v) = \max\{\texttt{maxRooted}(v), \texttt{maxTotal}(t)\}$

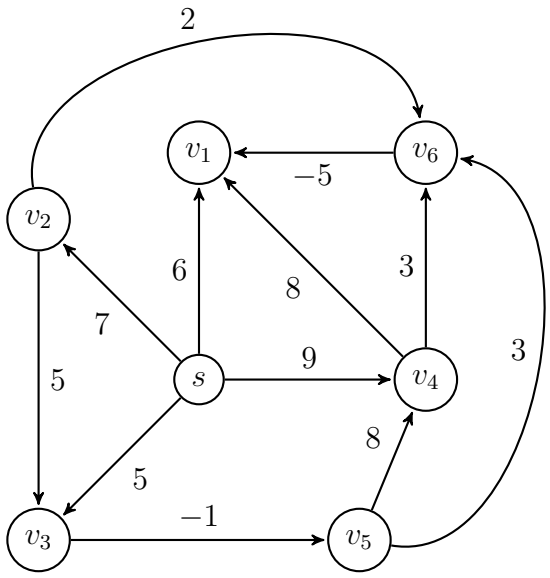
Wir können nun eine Lösung $T' = (V', E')$ konstruieren, indem wir T erneut lagenweise abarbeiten, nun jedoch von der Wurzel zur untersten Lage hin.
Gilt $\texttt{maxTotal}(s) = 1$, so ist $T' = (\{s\}, \emptyset)$. Ansonsten suchen wir bei s beginnend den ersten Knoten v mit $\texttt{maxRooted}(v) = \texttt{maxTotal}(v)$. Solange wir v noch nicht gefunden haben, bewegen wir uns stets zu dem Kind-Knoten mit dem größten $\texttt{maxTotal} - Wert$. Haben wir v gefunden, so fügen wir v zu V' und die beiden Kanten $(v, u), (v, w)$ zu E' hinzu, wobei u, w die beiden Kinder von v mit den höchsten maxRooted-Werten sind. Gilt $\texttt{maxRooted}(u) = \texttt{maxRooted}(w) = 1$, so beenden wir die Konstruktion an dieser Stelle und fügen u, w zu V' hinzu. Ansonsten führen wir das Verfahren rekursiv für u und w durch.
Sei für jeden Knoten T_v der größte an einem Knoten v gewurzelte Binärbaum. Es ist $\texttt{maxRooted}(v)$ gleich der Anzahl Knoten in T_v, denn: Entweder, v hat weniger als 2 Kinder und kann in einem Binärbaum nur ein Blatt sein, oder T_v ergibt sich aus den beiden größten Binärbäumen, die an Kindern von v gewurzelt sind. Außerdem ist $\texttt{maxTotal}(v)$ gleich der Anzahl Knoten im größten Binärbaum unter v, da $\texttt{maxTotal}(v)$ als das Maximum der maxRooted-Werte aller Nachfahren von v bestimmt wird. Der Algorithmus konstruiert einen größten Binärbaum in T, weil er bei dem Knoten beginnt, an dem der größte Binärbaum gewurzelt ist und anhand der maxRooted-Werte diesen Binärbaum rekonstruiert.
Um für einen Knoten die Lage zu bestimmen, auf der er in T liegt, können wir eine Breitensuche auf T ausführen. Da T ein Baum ist und also $n - 1$ Kanten hat, benötigt dies Zeit in $\Theta(n + n - 1) = \Theta(n)$. In den beiden darauf folgenden lagenweisen Betrachtungen aller Knoten wird jeweils ein mal über die Nachbarschaft jedes Knoten iteriert. Dies benötigt insgesamt Zeit in $\Theta(m) = \Theta(n)$. Damit liegt die Gesamtlaufzeit unseres Algorithmus in $\Theta(n)$.

Sollte T nicht gewurzelt sein, muss der zuvor beschriebene Algorithmus einmal für jede potenzielle Wurzel ausgeführt werden. Dadurch erhöht sich die Laufzeit um einen Faktor n.

Aufgabe 2 - Sind wir schon da? (0+4 Punkte)

Sei $G = (V, E)$ ein DAG und $s \in V$ die einzige Quelle. Wir wollen nun mittels Bellman-Ford Algorithmus die Distanzen von s zu allen anderen Knoten bestimmen.

1*. Gib für folgenden DAG an, in welcher Reihenfolge die Kanten relaxiert werden müssten, wenn man alle gesuchten Distanzen in der ersten Iteration des Algorithmus bestimmt haben möchte. (1 Punkt)



2*. Gib an und begründe, ob deine in Teilaufgabe 1 angegebene Reihenfolge, die einzig mögliche Antwort ist. (1 Punkt)

3*. Gib für einen beliebigen DAG G an, in welcher Reihenfolge die Kanten relaxiert werden müssten, wenn man alle gesuchten Distanzen in der ersten Iteration des Algorithmus bestimmt haben möchte. (2 Punkte)

Lösung 2

1*.

$$\{s, v_1\}, \{s, v_2\}, \{v_2, v_3\}, \{s, v_3\}, \{v_3, v_5\}, \{v_5, v_4\},$$
$$\{s, v_4\}, \{v_4, v_1\}, \{v_4, v_6\}, \{v_5, v_6\}, \{v_2, v_6\}, \{v_6, v_1\}$$

2*. Die Reihenfolge aus Teilaufgabe 1 ist nicht eindeutig. Es kann z.B. $\{s, v_3\}$ vor $\{v_2, v_3\}$ exploriert werden, ohne, dass sich das Ergebnis ändert.

3*. Da G ein DAG ist, gibt es eine topologische Sortierung von G. Eine Kante (u_1, v_1) muss vor einer Kante (u_2, v_2) relaxiert werden, wenn u_1 in der topologischen Sortierung vor u_2 kommt. Die Reihenfolge der Kanten, die vom selben Knoten ausgehen, ist dabei beliebig.

Aufgabe 3 - Let's play a game! (0+7 Punkte)

Wir wollen eine Partie Patience mit n Karten spielen. Der Wert einer Karte ist durch eine natürliche Zahl definiert. Ein Spiel läuft wie folgt ab:
Wir beginnen mit einem unsortierten Deck von n Karten. Diese sollen in sortierte Stapel eingeordnet werden, welche von links nach rechts angeordnet auf dem Tisch liegen. Wir nehmen die Karten nacheinander vom Deck und führen eine der folgenden Aktionen aus: Sei dabei w der Wert der aktuellen Karte.

- Wurden noch keine Stapel angelegt oder die jeweils oberste Karte jedes Stapels hat einen Wert kleiner w, wird rechts von den bereits bestehenden Stapeln ein neuer Stapel angelegt und die aktuelle Karte dort eingefügt.
- Gibt es mindestens einen Stapel, dessen oberste Karte einen Wert von mindestens w hat, legen wir die aktuelle Karte auf den linkesten dieser Stapel.

1*. Das folgende Array stellt ein Deck dar, wobei jeder Eintrag eine Karte repräsentiert. Gib die Stapel an, die sich durch das Einsortieren der Karten wie oben beschrieben ergeben. (1 Punkt)

$$\langle 5, 4, 1, 8, 8, 7, 3, 4, 3, 7, 5 \rangle$$

2*. Beschreibe einen Algorithmus, der die Karten, nachdem sie wie beschrieben in Stapel eingeordnet wurden, zu einem einzigen sortierten Stapel zusammengefügt. (1 Punkt)

3*. Gegeben sei ein Array `cards` : $[\mathbb{N};\ n]$, welches eine unsortierte Menge Spielkarten repräsentiert. Zudem ist eine Methode RECONSTRUCT gegeben, die deinen Algorithmus aus Teilaufgabe 2 (Stapel zusammenführen) umsetzt. Gib einen Algorithmus in Pseudocode an, der als Eingabe `cards` erhält, Patience spielt und schließlich das die sortierte Kartenmenge zurückgibt. Verwende die folgende Signatur: (3 Punkte)

PATIENCE(`cards`: $[\mathbb{N};\ n]$): $[\mathbb{N};\ n]$

4*. Nenne und begründe die Laufzeit deines Algorithmus und ob er stabil ist. (2 Punkte)

Lösung 3

1*.

$\langle 5, 4, 1 \rangle$
$\langle 8, 8, 7, 3, 3 \rangle$
$\langle 4 \rangle$
$\langle 7, 5 \rangle$

2*. Wir gehen aus von k sortierten Stapeln, wobei $k \leq n$ und die Karten mit dem kleinsten Wert oben liegen. Um die Stapel nun zu einem sortierten Stapel S zusammenzufügen, nutzen wir MERGE. Wir mergen jeweils zwei benachbarte Stapel. Sollte k ungerade sein, bleibt ein also Stapel unverändert. Damit erhalten wir insgesamt $\lceil k/2 \rceil$ sortierte Stapel. Dies führen wir so oft durch, bis wir einen sortierten Stapel erhalten.

3*. Wir nutzen neben RECONSTRUCT die Hilfsmethode FINDSTACK, die für eine gegebene Karte den Stapel bestimmt, auf den sie eingeordnet werden muss. Da die obersten Karten der Stapel von links nach rechts aufsteigend sortiert sind, orientieren wir uns dabei an der binären Suche.

```
PATIENCE(cards: [ℕ; n]): [ℕ; n]
    stacks: [List⟨ℕ⟩; n] = ⟨⟨⟩, ⟨⟩, ..., ⟨⟩⟩
    rightmost: ℕ = 0
    for i ∈ {0, ..., n − 1} do
        left, right := 0, rightmost
        stack_num = FINDSTACK(stacks, 0, rightmost, cards[i])
        if stack_num = ⊥ then          // neuer Stapel muss angelegt werden
            rightmost := rightmost + 1
            stacks[rightmost].pushFront(cards[i])
        else
            stacks[stack_num].pushFront(cards[i])
        end
    end
    return RECONSTRUCT(stacks[0 ... rightmost])
```

```
FINDSTACK(stacks: [ℕ; n], left, right, card: ℕ): ℕ
    if left = right then
        if stacks[left] ≥ card then
            return left
        end
        return ⊥
    end
    mid: ℕ = ⌊left + (right − left)/2⌋
    if stacks[mid].first < card then
        return FINDSTACK(stacks, mid + 1, right, card)
    else
        return FINDSTACK(stacks, left, mid, card)
    end
```

4*. Das Einordnen der Karten, also das Anlegen der Stapel, benötigt Zeit in $\Theta(n \log(n))$, denn jede Karte muss in genau einen Stapel einsortiert werden. Der korrekte Stapel kann mittels FINDSTACK in logarithmischer Laufzeit bestimmt werden. Die

Laufzeit von FINDSTACK liegt in $\Theta(\log(n))$, denn die Anzahl Stapel, unter denen gesucht wird, halbiert sich in jedem Schritt und es gibt maximal n Stapel. RECONSTRUCT benötigt Zeit in $\Theta(n \log(n))$, denn es wird bei bis zu n Stapeln begonnen und in jedem Schritt halbiert sich die Anzahl Stapel etwa. Die Stapel werden in Linearzeit gemerged. Insgesamt benötigt PATIENCE also Zeit in $\Theta(n \log(n))$.
Zudem ist PATIENCE nicht stabil, denn: Betrachte das folgende Array:

$$\texttt{cards} = \langle 1_1, 2, 1_2, 1_3 \rangle$$

Die Einträge von `cards` werden in Stapel

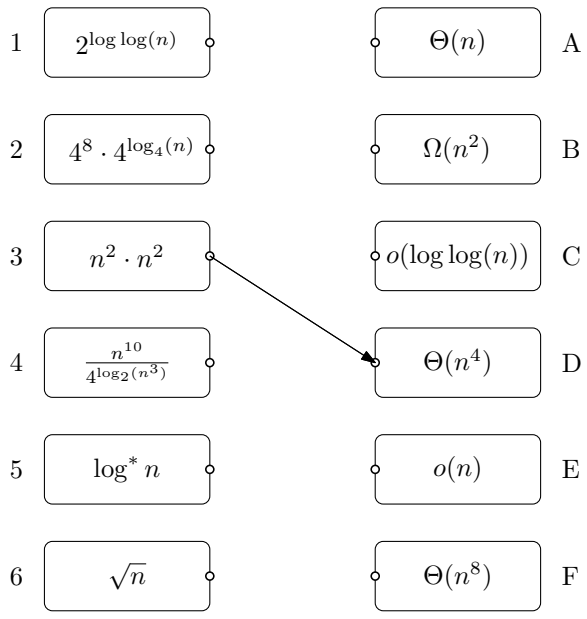
$$\langle 1_1 \rangle \qquad \langle 2, 1_2, 1_3 \rangle$$

einsortiert. Das Ergebnis von RECONSTRUCT ist dann

$$\langle 1_1, 1_3, 1_2, 2 \rangle$$

Aufgabe 4 - "Oh no"-Tation (0+4 Punkte)

Im folgenden Graphen soll es von jedem Knoten auf der linken Seite $(1, 2, 3, 4, 5, 6)$ genau dann eine gerichtete Kante zu einem Knoten auf der rechten Seite (A, B, C, D, E, F) geben, wenn die zugehörige Funktion in der jeweiligen Menge enthalten ist. So beschreibt die eingezeichnete Kante (3, D), dass $n^2 \cdot n^2 \in \Theta(n^4)$ gilt. Zeichne die fehlenden Kanten ein, oder gib die zugehörigen Tupel aus Zahlen und Buchstaben an.
Hinweis: Jede fehlende und jede falsche Kante geben einen halben Punkt Abzug. (4 Punkte)



Lösung 4

Die gesuchten Kanten sind:

- (1, E)
- (2, A)
- (3, B), (3, D)
- (4, B), (4, D)
- (5, C), (5, E)
- (6, E)

Aufgabe 5 - Daten, Daten (0+5 Punkte)

Bestimme für folgende Situationen, welche möglichst einfache Datenstruktur geeignet ist und begründe deine Wahl kurz.
Hinweis: In der Vorlesung haben wir folgende Datenstrukturen kennengelernt: Listen, dynamische Arrays, Hashtabellen, binäre Heaps, $(2, 3)$-Bäume, Union-Find, Adjazenzliste.

1*. Den durchnummerierten Knoten in einem Graphen sollen Farben zugeordnet werden, welche anschließend anhand der Knoten-ID ausgelesen werden können. (1 Punkt)

2*. Bei einer Volkszählung geben Leute ihren Wohnort an. Dabei soll zu jedem Zeitpunkt bestimmt werden können, wie viele Leute aus einem gegebenen Ort sich bereits gemeldet haben. (1 Punkt)

3*. Ein Pizza-Lieferant bekommt im Laufe des Abends Bestellungen, die in der Reihenfolge abgearbeitet werden, in der sie ankommen. (1 Punkt)

4*. Für jede Stadt soll gespeichert werden, welche Krankenhäuser sich im Umkreis von 25km befinden. (1 Punkt)

5*. Bei einer Verlosung haben sich viele Glückspilze registriert. Jede Woche wird ein $k \in \mathbb{N}$ zufällig bestimmt und Preise an jede k-te registrierte Person verschenkt, wobei die Reihenfolge betrachtet wird, in der sich die Leute registriert haben. (1 Punkt)

Lösung 5

1*. Wähle ein Array mit einem Eintrag pro Knoten, in welchem die Farbe gespeichert wird. Dies ermöglicht Zugriff auf die Farbe eines beliebigen Knoten in $\Theta(1)$.

2*. Wähle eine Hashmap, welche pro Stadt einen Eintrag mit der Anzahl gemeldeter Einwohner hält und die Postleitzahl einer Stadt als Schlüssel verwendet. damit können wir in erwartet konstanter Zeit auf die Einwohnerzahl eines Ortes zugreifen und es können während der Volkszählung Städte bei ihrer ersten Nennung ohne Zusatzaufwand angelegt werden.

3*. Wähle eine Liste. Eine neue Bestellung kann in $\Theta(1)$ mittels `pushBack` eingefügt werden, die nächste zu bearbeitenden Bestellung kann mittels `popFront` in $\Theta(1)$ entfernt werden.

4*. Wähle eine Adjazenzliste mit einem Eintrag pro Stadt. In der Liste zu einer Stadt c werden genau die Krankenhäuser gespeichert, die sich im Umkreis von 25km um c befinden. Damit lässt sich in konstanter Zeit auf die Menge der Krankenhäuser in der Nähe einer beliebigen Stadt zugreifen.

5*. Wähle ein (dynamsiches) Array mit einem Eintrag pro registrierter Person. Damit kann eine neue Person in amortisiert $\Theta(1)$ registriert werden und wir können unabhängig von k die Gewinner einer Woche in Linearzeit bestimmen.