Karlsruher Institut für Technologie

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

Übungsblatt 08

Algorithmen I – Sommersemester 2022

Abgabe im ILIAS bis 22.06.2022, 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 - Kleiner Haufen ganz groß (4 Punkte)

Ein binary Min-Heap kann mithilfe eines Arrays umgesetzt werden, um Elemente effizient so zu verwalten, dass man schnell das Minimum extrahieren kann. Natürlich lässt sich auch leicht eine Variante definieren, die einem schnellen Zugriff auf das Maximum gibt.

1. Zeichne den Max-Heap der entsteht, wenn man die folgende Reihe von Zahlen einfügt:

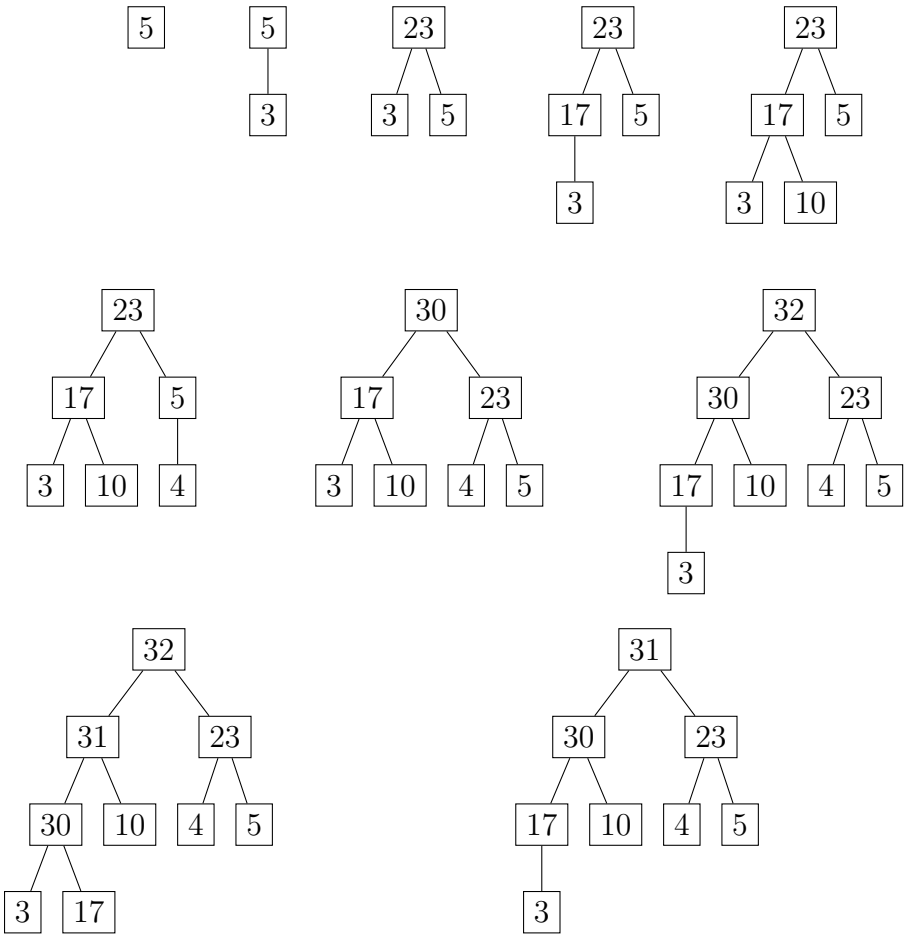
   $$(5, 3, 23, 17, 10, 4, 30, 32, 31)$$

   Zeichne außerdem den Max-Heap, nachdem ein Mal `popMax`() aufgerufen wurde.
   (1 Punkt)

2. Beschreibe einen Algorithmus, der als Eingabe ein Array $A\colon [\mathbb{N}; n]$ erhält und in Zeit $O(n)$ entscheidet, ob A einen Max-Heap repräsentiert. Begründe die Korrektheit deines Algorithmus und dass er das geforderte Laufzeitverhalten hat.
   (3 Punkte)

Lösung 1

1. Einfügeoperationen (Zwischenschritte mussten nicht angegeben werden):



Ergebnis nach Einfügen

Ergebnis nach `popMax`()

2. Wir können analog zu Min-Heaps eine Invariante für Max-Heaps definieren. Repräsentiert $A : [\mathbb{N};\ n]$ einen Max-Heap, so gilt:

$$\forall i \in \{0, \ldots, \left\lfloor \frac{n-2}{2} \right\rfloor\} : A[i] \geq A[2i+1] \wedge (2i+2 < n \Rightarrow A[i] \geq A[2i+2])$$

Um zu überprüfen, ob ein gegebenes Array einen Max-Heap repräsentiert, müssen wir also über die ersten $\lfloor \frac{n-2}{2} \rfloor + 1$ Elemente iterieren und prüfen, ob sie größer / gleich ihren Kind-Elementen sind.
Hält die Invariante für jedes überprüfte Element, so haben wir sichergestellt, dass jeder innere Knoten des repräsentierten Heaps größer / gleich seiner Kind-Knoten ist, damit liegt ein Max-Heap vor. Außerdem muss die Invariante in einem gültigen Max-Heap für jeden inneren Knoten erfüllt sein, sonst ist nicht mehr gegeben, dass

das Maximum tatsächlich in der Wurzel steht.
Da pro Element zwei Vergleiche durchgeführt werden müssen, welche jeweils konstanten Zeitbedarf haben, liegt die Laufzeit dieses Algorithmus in insgesamt $\Theta(n)$.

Aufgabe 2 - Déjà vu (3+4 Punkte)

Schon wieder haben wir ein unsortiertes Array $A\colon [\mathbb{N}; n]$ und wollen das k-kleinste Element in A finden.

1. Beschreibe einen Algorithmus, der unter Benutzung eines Heaps das k-kleinste Element in A mit einem Zeitbedarf in $O(n \log(k))$ ausgibt. Begründe die Korrektheit deines Algorithmus und dass er das geforderte Laufzeitverhalten hat.
(2 Punkte)

*. Beschreibe einen Algorithmus, der unter Benutzung von Heaps das k-kleinste Element in A mit einem Zeitbedarf in $O(n + k \log(k))$ ausgibt. Begründe die Korrektheit deines Algorithmus und dass er das geforderte Laufzeitverhalten hat.
(4 Punkte)

2. Wir wollen nun die Laufzeit des Algorithmus aus 2.* mit der des Algorithmus aus der Bonusaufgabe von Blatt 04 vergleichen. Beschreibe unter welchen Umständen man welchen Algorithmus eher verwenden sollte. (1 Punkt)

Lösung 2

1. Wir verwenden einen Max-Heap H der Größe $k + 1$. Um das k-kleinste Element in A zu bestimmen, gehen wie folgt vor: Zunächst fügen wir die ersten k Elemente von A in H ein. Nun iterieren wir über die übrigen Elemente in A. Dabei fügen wir das jeweils aktuelle Element in H ein rufen dann einmal `popMax` auf H auf. Das Element, das anschließend in H an der Wurzel steht, ist das von uns gesuchte. Dieser Algorithmus bestimmt das k-kleinste Element in A, denn aus H werden nacheinander die $n - k$ größten Elemente aus A entfernt. Anschließend befinden sich die k kleinsten Elemente von A in H, deren Maximum das k-kleinste Element ist. Da H ein Max-Heap ist, steht dieses an der Wurzel.
Insgesamt werden n Mal `push` und $n-k$ Mal `popMax` aufgerufen. Da beide Operationen einen Zeitbedarf in $\Theta(\log(k))$ haben und der übrige Zeitaufwand pro Element in A konstant ist, ergibt sich eine Gesamtlaufzeit in $\Theta(n \log(k))$.

*. Zunächst führen wir auf A einmal `buildHeap` aus. Nun legen wir uns einen Heap H an, in welchen wir Elemente aus A einfügen werden. Zu jedem Element merken wir uns dabei den Index, an dem es in A steht.
Vorbereitend fügen wir $A[0]$ in H ein. Dann wiederholen wir $k-1$ Mal die folgenden Schritte: wir rufen `popMin` auf H auf und fügen die beiden Kind-Elemente des entsprechenden Elements in A in H ein. Anschließend rufen wir ein Mal `popMin` auf H auf, um das k-kleinste Element zu erhalten.

Zu Anfang enthält H nur das minimale Element aus A, d.h. der erste `popMin`-Aufruf liefert das kleinste Element in A. In der i-ten Iteration wird das i-kleinste Element von A entfernt und dessen Kind-Elemente in H eingefügt. Dann befindet sich in H auch das $(i+1)$-kleinste Element von A in H, denn es wurden bereits die Kinder der i kleinsten Elemente und damit auch alle Elemente kleiner gleich des $(i+1)$-kleinsten Elementes eingefügt. Darüber hinaus musste das i-kleinste Element in H an der Wurzel stehen, denn es wurde bereits $i-1$ Mal `popMin` aufgerufen.
Die Methode `buildHeap` auf einem bereits bestehenden Array der Kapazität n hat eine Laufzeit in $\Theta(n)$. Sowohl `popMin` als auch `push` haben auf H eine Laufzeit in $\Theta(\log(k))$. Wir führen insgesamt k-Mal `popMin` und $(2k-2)$-mal `push` auf H aus. Damit ergibt sich eine Gesamtlaufzeit in $O(n + k\log(k))$.

2. Die Laufzeit des auf Blatt 04 entwickelten Algorithmus liegt in *erwartet* $O(n)$. Allerdings ergab sich eine worst case Laufzeit in $O(n^2)$. Wünschen wir uns also einen Algorithmus, dessen Laufzeit auch im worst case besser als $O(n^2)$ ist und ist i.d.R. $k \ll n$, so wählen wir eher die Heap-Variante, ansonsten eher den Algorithmus von Blatt 04.

Aufgabe 3 - Dairy Min-Heap (7 Punkte)

Das Prinzip von binary Min-Heaps lässt sich leicht auf Min-Heaps höheren Grades erweitern. Während ein Knoten im binary Min-Heap maximal zwei Kinder hat, sind es beim d-ary Min-Heap bis zu d Kinder.

1. Beschreibe einen Algorithmus der die Operationen `push` in einem d-ary Min-Heap umsetzt und analysiere dessen Laufzeit in Abhängigkeit von n und d. (2 Punkte)

2. Beschreibe einen Algorithmus der die Operationen `decPrio` in einem d-ary Min-Heap umsetzt und analysiere dessen Laufzeit in Abhängigkeit von n und d. (2 Punkte)

3. Beschreibe einen Algorithmus der die Operationen `popMin` in einem d-ary Min-Heap umsetzt und analysiere dessen Laufzeit in Abhängigkeit von n und d. (2 Punkte)

4. Beschreibe unter welchen Umständen man einen d-ary Min-Heap für $d > 2$ einem binary Min-Heap vorziehen sollte. (1 Punkt)

Lösung 3

1. Wir wollen ein Element e in einen d-ary Min-Heap H einfügen. Wie bei `push` auf einem Binary Min-Heap wird e als letztes Element an H angefügt und anschließend eventuelle Verletzungen der Heap-Invariante mit `bubbleUp` behoben. In einem d-ary Min-Heap in Array-Repräsentation befinden sich die Kind-Elemente

eines Knotens v an den Stellen $dv+1, \ldots, dv+d$ und der Elter-Knoten an der Stelle $\lfloor \frac{v-1}{d} \rfloor$. Abseits davon ändert sich nichts im Vergleich zu einem Binary Min-Heap. In einem d-ary Min-Heap sind die Elemente auf $\Theta(\log_d(n))$ Layers verteilt. Wenn wir ein Element einfügen, müssen wir also $\log_d(n)$ Vergleiche und `swap`-Operationen durchführen. Damit liegt die benötigte Laufzeit in $\Theta(\log_d(n))$.

2. Wir übernehmen den Algorithmus für `decPrio` unverändert von Binary Min-Heaps, dekrementieren also die Priorität des entsprechenden Elements und beheben anschließend eventuelle Verletzungen der Heap-Invariante mit `bubbleUp`. Dabei wird im worst case ein Element aus der unsortierten Layer an die Wurzel getauscht. Analog zu `push` ergibt sich damit eine Laufzeit in $\Theta(\log_d(n))$.

3. Auch in einem d-ary Min-Heap steht das Minimum an der Wurzel. Also gehen wir bei der Umsetzung von `popMin` genauso vor wie bei Binary Min-Heaps. Während eines Aufrufs von `sinkDown` werden in d-ary Min-Heaps pro Knoten bis zu d Vergleiche und eine `swap`-Operation durchgeführt. Da der Heap $\Theta(\log_d(n))$ Layers hat, liegt die Gesamtlaufzeit für einen `popMin`-Aufruf in $\Theta(d \cdot \log_d(n))$.

4. Im Allgemeinen werden `decPrion-` und `push`-Aufrufe im Vergleich zu einem Binary Heap günstiger, `popMin`-Aufrufe eher teurer. Wissen wir also, dass auf dem Heap häufiger die ersten beiden Operationen aufgerufen werden, lohnt sich eher der Einsatz eines d-ary Heaps.
   Bei der Anwendung von Dijkstra's Algorithmus auf dichten Graphen kann es lohnen, eine Priority-Queue zu verwenden, welche auf einem d-ären Heap mit $d \approx \frac{m}{n}$ einzusetzen, denn die sich ergebende Laufzeit in $\Theta(m \log_{m/n}(n))$ stellt eine Verbesserung zur Laufzeit in $\Theta(m \log(n))$ dar.

Aufgabe 4 - Kontaktbeschränkungen (4 Punkte)

Bevor Dr. Meta ihn zur Rechenschaft hat ziehen können, ist der Verräter getürmt. Doch das beruhigt Dr. Meta nicht, im Gegenteil: er befürchtet, dass sich noch weitere seiner Handlanger haben korrumpieren lassen. Deswegen greift er zu drastischeren Maßnahmen. Er hat einen Graph erstellt, der abbildet welche Mitarbeiter miteinander in Kontakt stehen. Um zu verhindern, dass die Verräter in Zukunft noch viele weitere Mitarbeiter gegen ihn aufhetzen, soll dieser Graph nun ausgedünnt werden. Der Plan ist, nach und nach den Mitarbeiter mit den meisten Kontakten zu „entfernen", bis die Anzahl der verbleibenden Kontakte ausreichend klein ist. Dabei sollen jedoch nicht unnötig viele Arbeitskräfte verloren gehen.

Im folgenden soll nun ein Algorithmus entworfen werden, der die zu entfernenden Mitarbeiter identifiziert.

1. Wir betrachten einen Graph G und sortieren seine n Knoten nach Knotengrad in absteigender Reihenfolge. Nun soll iterativ ein höchstgradiger Knoten inklusive seiner inzidenten Kanten entfernt und die verbleibenden Knoten erneut sortiert

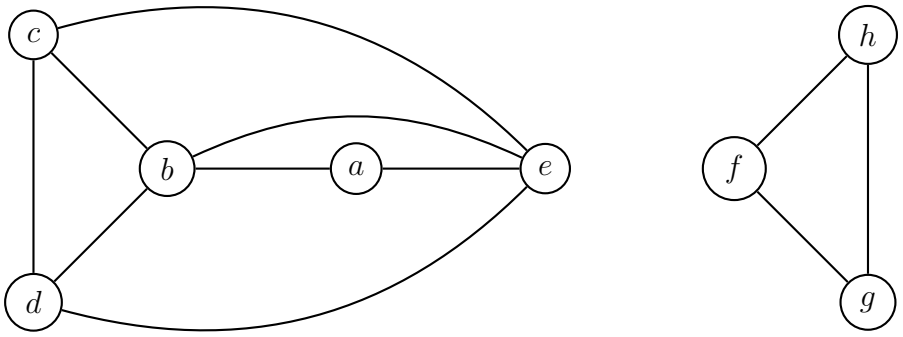
werden. Zeige, dass sich die Reihenfolge der niedriggradigeren Knoten dadurch verändern kann. (1 Punkt)

2. Beschreibe einen Algorithmus, der als Eingabe einen Graph G und eine Zahl k erhält und iterativ einen Knoten mit dem größten Grad (inklusive der inzidenten Kanten) aus dem Graph entfernt, bis die Anzahl der verbleibenden Kanten höchstens k aber möglichst groß ist. (2 Punkte)

3. Im folgenden darfst du annehmen, dass ein Knoten v mit Grad $\deg(v)$ inklusive seiner inzidenten Kanten aus in Zeit $O(\deg(v))$ aus einem Graphen entfernen kannst. Zeige, dass dein Algorithmus eine Laufzeit von $O(n + m \log(n))$ hat. (1 Punkt)

Lösung 4

1. Betrachte den folgenden Graphen:



Die beiden Knoten mit höchstem Grad sind e und b. Löschen wir einen von ihnen, wird der andere der Knoten mit dem höchsten Grad im verbleibenden Graphen. Löschen wir nun den zweiten, haben c und d nur noch Grad 1, während f, g und h unverändert Grad 2 behalten. Damit ändert sich die Sortierung nach Knotengrad, denn c und d waren zuvor vor f, g und h eingeordnet.

2. Sei $G = (V, E)$ mit $|V| = n$ und $|E| = m$. Zuerst bauen wir einen Max-Heap H auf, der alle Knoten in G enthält. Als Prioritäten verwenden wir die Knotengrade. Gilt $k < m$, bestimmen wir mit `popMax` den Knoten v mit höchster Priorität, d.h. mit höchstem Grad, und entfernen ihn mitsamt seiner inzindenten Kanten aus G. Dabei verringern wir für jeden Knoten in $N(v)$ seine Priorität um 1. Dieses Vorgehen wiederholen wir so lange, bis die Anzahl der verbleibenden Kanten den Wert k annimmt oder unterschreitet.

3. Das Aufbauen des Max-Heaps benötigt $\Theta(n)$ Zeit. Da wir stets den Knoten mit dem höchsten Grad entfernen und $k \leq m$ ist, müssen wir maximal m Knoten löschen, d.h. maximal m Mal `popMin` aufrufen. Zudem wird im Verlauf des Algorithmus höchstens m Mal `decPrio` ausgeführt, da dies immer mit dem Löschen

einer Kante verbunden ist. Da wir voraussetzen, dass das Entfernen eines Knoten v in $O(\deg(v))$ möglich ist, benötigen alle Lösch-Operationen zusammen Zeit in $O(m)$. Insgesamt ergibt sich damit eine Laufzeit in $\Theta(n + m \log(n))$.