ETH Zürich HS 2019
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger, Dr. Johannes Lengler
Übungsleitung: Florian Meier

Algorithmen und Komplexität
Lösungsvorschlag zu Übungsblatt 8

Lösungsvorschlag zu Aufgabe 1

Kruskal betrachtet zu Beginn einen Graphen $T = (V, F)$, welcher nur aus den Knoten des Graphen G besteht, das heisst $F = \varnothing$. Dann werden sukzessive die Kanten, aufsteigend nach ihrem Gewicht betrachtet. Falls die Kante keinen Kreis in T schliesst, wird sie in den Graphen T eingefügt, ansonsten wird sie verworfen.

Zu Beginn, solange noch keine Kante in den Graphen betrachtet wurde, ist jeder Knoten für sich eine Zusammenhangskomponente (ZHK). Jede zusätzliche Kante kann nun

(1) zwei Knoten unterschiedlicher ZHK oder

(2) zwei Knoten der selben ZHK verbinden.

Genau dann wenn der 2. Fall eintritt wird ein Kreis geschlossen.

Eine Union-Find Datenstruktur eignet sich hervorragend zur Modellierung dieser Vorgänge. Eine ZHK wird dabei durch die Menge der in ihr enthaltenen Knoten repräsentiert. Wenn nun eine neue Kante $\{u, v\}$ betrachtet wird, rufen wir die Find Opertion auf beide Knoten der Kante auf. Falls $Find(u) = Find(v)$, sind die Knoten in derselben ZHK und wir sind in Fall (2). Ansonsten sind wir in Fall (1): Dann fügen wir $\{u, v\}$ zu T hinzu und vereinigen die beiden Komponenten mit der Operation $Union(Find(u), Find(v))$.

Laufzeit: Um zu testen ob sich u und v in der selben ZHK befinden und um diese gegebenenfalls zu vereinigen benötigen wir mit Union-Find Strukturen logarithmische Laufzeit $O(\log |V|)$. (Zweimaliges Ausführen einer Find Operation und gegebenenfalls eine Union Operation.) Insgesamt ergibt sich also Laufzeit $O(|E| \cdot \log |V|)$. (Für das Sortieren brauchen wir Zeit $O(|E| \cdot \log |E|) = O(|E| \cdot \log |V|)$, da $|E| \leq |V|^2$ und für das sukzessive Einfügen der Kanten Zeit $O(|E| \cdot \log |V|)$).

Mit dem „naiven" Ansatz (modifizierte Breitensuche) benötigt man für jede Kante hingegen linear lange, also Laufzeit $O(|V| + |E|) = O(|V|)$, wobei die Gleichheit hier gilt, weil der auf Kreisfreiheit zu testende Graph immer höchstens einen Kreis hat und deshalb $|E| \leq |V|$ ist. Zusammen mit dem Sortieren ergibt sich also die Laufzeit

$$O(|E| \cdot \log |V| + |E| \cdot |V|) = O(|E| \cdot |V|).$$

Punkteschema:
Korrekte Implementierung von Kruskal mit Union-Find Struktur: 4 Punkte
Korrektheitbegründen (kreisfreiheit testen entspricht zu testen ob u und v in der selben ZSHK sind): 2 Punkte
Laufzeitanalyse: 2 Punkte
Vergleich zu naiver Implementation mit BFS: 2 Punkte
Bei kleineren Fehlern wie Laufzeit fürs Sortieren vergessen oder ungenaue Präsentation der Implementierung jeweils -1 Punkt

Lösungsvorschlag zu Aufgabe 2

a) Die längste aufsteigende Teilsequenz ist $2, 5, 6, 11, 14$.

b) Im folgenden kürzen wir die längste aufsteigende Teilsequenz mit LATS ab. Seien $a_1 < a_2 < \ldots < a_{n\cdot m}$ die Elemente von A austeigend sortiert. Wir definieren L_i als die Länge der LATS von A, welche in a_i endet. Zusätzlich definieren wir die Nachbarschaft $\Gamma(a_i)$ eines Elements a_i als die in A zu a_i benachbarten Elemente. Die LATS die in a_i endet, besteht entweder nur aus a_i oder sie besteht aus a_i angehängt an die LATS einer der Nachbarn der kleiner als a_i ist. Es folgt die Korrektheit folgender Rekursionsformel:

$$L_i = \begin{cases} 1, \text{ falls } a_j > a_i \text{ für alle } a_j \in \Gamma(a_i) \\ \max_{a_j \in \Gamma(a_i): a_i > a_j} \{1 + L_j\} \text{ sonst} \end{cases}$$

Die Formel für L_i lässt sich berechnen, falls L_j für alle Nachbarn a_j von a_i mit $a_j < a_i$ bereits bekannt ist. Darum lassen sich die L_i iterativ für $i = 1, \ldots, mn$ berechnen, wenn wir die a_i in aufsteigend sortierter Reihenfolge betrachten. Die Länge der LATS in A ist dann gleich $\max_{1 \leq i \leq mn}\{L_i\}$. Pseudo-Code:

Sortiere die Elemente von A: $a_1 < \ldots < a_{m\cdot n}$
for $i = 1, \ldots, mn$ **do**
  if $\forall a_j \in \Gamma(a_i)\ :\ a_j > a_i$ **then**
    $L_i = 1$
  else
    $L_i = \max_{a_j \in \Gamma(a_i)\ :\ a_j < a_i}\{1 + L_j\}$
  end if
end for
$L = \max_{1 \leq i \leq mn}\{L_i\}$
return L

Wenn zusätzlich die LATS gefunden werden soll, starten wir in a_i wo die LATS endet. In jedem Schritt suchen wir den Vorgänger von a_i in der LATS (nämlich derjenige Nachbar a_j mit $L_j = L_i - 1$), hänge a_j an die LATS und fahre bei a_j fort.

$i = \arg\max_{1 \leq i \leq mn} L_i$
$\ell = L_i$
$w[\ell] = a_i$
for $k = 1, \ldots, \ell - 1$ **do**
  Wähle $a_j \in \Gamma(a_i)$ sodass $L_j = L_i - 1$
  $w[\ell - k] = a_j$
  $i = j$
end for
return $w[1 \ldots \ell]$

Das Sortieren benötigt Laufzeit $\mathcal{O}(nm \log(mn)$, das Berechnen der Länge der LATS $\mathcal{O}(mn)$, da jede Iteration der For-Schleife in konstanter Zeit ausgeführt werden kann. Das Bestimmen der LATS benötigt Zeit $\mathcal{O}(L) = \mathcal{O}(mn)$. Somit ergibt sich eine Gesamtlaufzeit von $\mathcal{O}(mn \log(mn))$

Punkteschema:
a) 2 Punkte falls korrekte LATS angegeben
b) **Rekursionsformel:** maximal 6 Punkte.
Definition L_i: 1 Punkt
Korrekte Rekursionsformel für L_i: 3 Punkte (bei Ungenauigkeiten, z.B. Fall $L_i = 1$ falls ... vergessen, 1 Punkt pro Fehler Abzug)

Begründung Rekursionsformel: 2 Punkte
Algorithmus zur Bestimmung von L: maximal 5 Punkte (diese Punkte können nur erreicht werden, wenn L_i korrekt definiert ist und die Rekursionsformel einigermassen Sinn macht, d.h. L_i anhand der Nachbarn mit kleineren Werten berechnet wird)
Berechnungsreihenfolge der L_i (anhand aufsteigender Sortierung): 2 Punkte
Ausgabe maximales L_i: 1 Punkt
Laufzeitanalyse: 2 Punkte
Bestimmen der LATS: 3 Punkte
- 1 Punkt pro Fehler wenn die Idee stimmt.

Bemerkung: Die Laufzeit kann auf $\mathcal{O}(mn)$ verbessert werden. Die Berechnung der Länge der LATS nach dem Sortieren benötigt lediglich $\mathcal{O}(mn)$. Der Hauptterm der Laufzeit des oben präsentierten Algorithmus wird vom Sortieren der Elemente verursacht, welches folgendermassen umgangen werden kann. Wir konstruieren einen gerichteten Graphen G mit Knoten $a_1, \ldots, a_{mn}$. Eine Kante (a_i, a_j) befindet sich genau dann im G, falls $a_i > a_j$ und a_i und a_j in A benachbart sind. Die L_i können nun in der Reihenfolge einer Topoligische Sortierung berechnet werden, welche nach Serie 3 in Zeit $\mathcal{O}(mn)$ gefunden werden kann (G enthält $\mathcal{O}(mn)$ Kanten!).

Lösungsvorschlag zu Aufgabe 3

Wir speichern in jedem inneren Knoten v des (a, b)-Baumes die Zahl $\ell(v)$ ab, welche angibt wieviele Schlüssel sich im Teilbaum mit Wurzel v befinden. Wir erklären zuerst, wie die Operationen INSERT, FIND und DELETE angepasst werden müssen, damit ℓ korrekt aktualisiert wird und sie weiterhin Zeit $\mathcal{O}(\log n)$ benötigen.

FIND: Wir belassen die FIND - Operation so wie sie ist. Da sich die Anzahl Schlüssel in keinem Teilbaum ändert, müssen die ℓ Werte nicht aktualisiert werden. Somit bleibt auch die Laufzeit bei $\mathcal{O}(\log n)$.

INSERT: Am Ende der INSERT-Operation werden eventuell Rebalancierungen ausgeführt. Wird ein Knoten v in zwei Knoten v_1 und v_2 zerteilt so bestimmen wir $\ell(v_1)$ und $\ell(v_2)$ indem wir die ℓ Werte ihrer Kinder aufsummieren. Sobald keine Rebalancierungen mehr nötig sind (d.h. der Vater v hat maximal b Kinder), so laufen wir von v bis zur Wurzel und erhöhen den ℓ- Wert jedes besuchten Knotens um 1, da ja die Anzahl Schlüssel in diesen Teilbäumen um 1 grösser wurde. Für alle anderen Knoten ändert sich die Anzahl Schlüssel im Teilbaum nicht. Sowohl für jede Rebalancierung also auch auf dem Weg zurück zur Wurzel brauchen wir jeweils Zeit $\mathcal{O}(1)$ um den neuen ℓ-Wert zu berechnen. Da der Baum Höhe $\mathcal{O}(\log n)$ hat, bleibt die Laufzeit bei $\mathcal{O}(\log n)$.

DELETE: Auch hier müssen am Schluss die ℓ-Werte aktualisiert werden. Falls keine Rebalancierungen nötig sind (der Vater v hat mindestens a Kinder) so reduzieren wir die ℓ Werte auf dem Weg von v zur Wurzel um jeweils 1. Falls v ein Kind von einem Nachbarn u adoptiert, so müssen die ℓ -Werte auf dem Weg von u bis zur Wurzel um 1 reduziert werden. Falls Verschmelzungen stattfinden, so berechnen wir den ℓ Wert des Vaters jeweils indem wir die ℓ -Werte der Kinder aufsummieren. Wie bei INSERT sieht man, dass insgesmat $\mathcal{O}(\log n)$ ℓ-Werte aktualisert werden und somit die Laufzeit $\mathcal{O}(\log n)$ bleibt.

Es ist einfach einzusehen, dass die Aktualisierungen der ℓ-Werte obiger Operationen korrekt sind und dass sich die Anzahl Schlüssel in den restlichen Teilbäumen nicht ändert.

Wir implementieren nun die SELECT(k, T) Operation. Für einen Knoten v seien $v_1, \ldots, v_k$ seine Kinder. Wir definieren die Funktion SELECT(k, v), welche den k-kleinsten Schlüssel im Teilbaum mit Wurzel v findet, folgendermassen. Falls v keine Kinder mehr hat, so geben wir den Schlüssel von v aus. Ansonsten suchen wir das Kind v_i von v mit $v_1 + \ldots + v_{i-1} < k \leq v_1 + \ldots + v_i$, und rufen SELECT$(k - (v_1 + \ldots + v_{i-1}), v_i)$ auf. Um SELECT(k, T) zu berechnen rufen wir SELECT(k, w) auf, wobei w die Wurzel von T ist. Die Korrektheit folgt, da wir jeweils das Kind auswählen, in dessen Teilbaum der k -kleinste Schlüssel enthalten ist. Da wir während der Suche auf jeder Ebene des Baumes konstant viel Zeit benötigen folgt eine Laufzeit von $\mathcal{O}(\log n)$.

Punkteschema:
Idee abspeichern der Werte $\ell(v)$, 2 Punkte
Implementierung von Select, 5 Punkte
Anpassung von Find, 1 Punkt
Anpassung von Insert, 3 Punkte
Anpassung von Delete, 3 Punkte