Universität Karlsruhe
Fakultät für Informatik
ITI Wagner

2. Übungsblatt

Ausgabe: 14. November 2006
Abgabe: 22. November, 15:30 Uhr im ITI Wagner (Informatik-Hauptgebäude, 3. Stock)

Die Bearbeitung in Zweiergruppen ist ausdrücklich erwünscht.

Problem 1: Least Common Ancestor 3pt

Der letzte gemeinsame Vorfahr (Least Common Ancestor LCA) zweier Knoten u und $v \in V$ in einem gerichteten Baum T ist derjenige Knoten w, der sowohl ein Vorfahr von u als auch ein Vorfahr von v ist und der die größten Tiefe in T besitzt (Ein Knoten ist auch sein eigener Vorfahr). Beim Offline-Problem zur Bestimmung des letzten gemeinsamen Vorfahren sind ein gerichteter Baum T und eine beliebige Menge $P \subseteq V \times V$ von ungeordneten Knotenpaaren $\{u, v\}$ in T gegeben, und wir wollen den letzten gemeinsamen Vorfahren jedes Paares in P bestimmen. Um dieses Problem zu lösen, führt der Algorithmus 1 eine Traversierung von T mit dem Anfangsaufruf LCA($wurzel[T]$) aus. Wir nehmen an, dass jeder Knoten vor der Traversierung weiss gefärbt ist.

Algorithmus 1 : LCA(u)

```
1  MAKESET(u)
2  Vorfahr[FIND(u)] ← u
3  Für jedes Kind v von u in T
4  |  LCA(v)
5  |  UNION(u,v)
6  |  Vorfahr[FIND(u)] ← u
7  farbe[u] ← SCHWARZ
8  Für jeden Knoten v mit {u, v} ∈ P
9  |  Wenn farbe[v] = SCHWARZ
10 |  |  print 'Der letzte gemeinsame Vorfahr von' u 'und' v 'ist' Vorfahr[FIND(v)]
```

(a) Zeigen Sie, dass Zeile 10 für jedes Paar $\{u, v\} \in P$ genau einmal ausgeführt wird.

(b) Zeigen Sie, dass die Anzahl der Mengen in der Datenstruktur zur Zeit des Aufrufs LCA(v) gleich der Tiefe von v in T ist (Wurzel habe Tiefe 0).

(c) Beweisen Sie, dass LCA den letzten gemeinsamen Vorfahren von u und v für jedes Paar $\{u, v\} \in P$ korrekt ausgibt.

(d) Analysieren Sie die Laufzeit (möglichst Scharfe obere Schranke für Worst-case) von LCA unter der Annahme, dass wir UNION-FIND aus der Vorlesung verwenden.

Problem 2: Nachbarschaftstratsch 2pt

Gegeben sei eine Menge M von n verschiedenen Personen und dazu eine symmetrische *Entfernung* $d : M \times M \to \mathbb{R}$. Die Personen sollen nun wie folgt in „lokale“ Teilmengen (Tratschgruppen) $C_1, \ldots, C_k \subset M$ partitioniert werden. Zu Beginn bildet jedes Person eine eigene, einelementige Teilmenge $C_i, i = 1, \ldots, n$. Dann werden iterativ immer die beiden Teilmengen mit geringster Entfernung verschmolzen. Der Abstand zweier Teilmengen C_p, C_q ist $\min_{v \in C_p, w \in C_q} d(v, w)$. Der Algorithmus soll abbrechen, falls nur noch eine Teilmenge vorhanden ist oder jedes Paar von Teilmengen mehr Abstand als $D_{\max}$ (Tratschreichweite) hat. Die Entfernungen aller Personenpaare sind vorberechnet und aufsteigend sortiert in einem Array $D = D[1], \ldots, D[n^2]$. Die beiden zu $D[\ell]$ korrespondierenden Personen seien in $V[\ell]$ und $W[\ell]$ gespeichert, es gilt also $D[\ell] = d(V[\ell], W[\ell])$ für alle $\ell \in \{1, \ldots, n^2\}$.

(a) Schreiben Sie unter Verwendung von Union-Find einen Algorithmus für das beschriebene Verfahren in Pseudocode.

(b) Analysieren Sie die asymptotische Laufzeit Ihres Algorithmus und begründen Sie Ihre Antwort.

Problem 3: Alternative zu Heaps 2pt

Gegeben sei eine Anwendung, bei der zusätzlich zu den Prioritätszahlen die einzelnen Elemente noch einen Wert speichern. Ein Beispiel sind die Kanten eines Graphen, wobei der Wert dann der Name (z.B. „e_5“) einer Kante ist, und die Prioritätszahl (Schlüssel) das Gewicht einer Kante (z.B. 3). Es sei von vornherein klar, dass:

- insgesamt nur die $k, k \in O(n)$ verschiedenen Schlüssel $1, \ldots, k$ vorkommen und
- niemals mehr als n Elemente zugleich gespeichert werden.

Formulieren Sie eine Datenstruktur der Größe $O(n)$, und die folgenden Zugriffsmethoden mit dem angegebenen Aufwand:

- Insert(Schlüssel, Wert) in $O(1)$
- Findmax() in $O(k)$
- Remove(Pointer auf Element in Datenstruktur) in $O(1)$

Problem 4: d-Heaps 1pt

Ein d-Heap ist eine Verallgemeinerung des aus der Vorlesung bekannten Heaps. Der einzige Unterschied zwischen einem Heap und einem d-Heap besteht darin, dass in einem d-Heap jeder innere Knoten (bis zu) d unmittelbare Nachfolger hat. Ein „normaler“ Heap ist also ein 2-Heap.

(a) Wie würden Sie einen d-Heap in einem Array speichern?

(b) Geben Sie ein Beispiel für einen 3-Heap mit 11 Werten ($[1, \ldots, 11]$) an. Geben Sie dabei die Baumstruktur und das entsprechende Array an.

(c) Geben Sie eine effiziente Implementierung von Sift-Up für einen d-Heap an. Analysieren Sie die Laufzeit Ihrer Implementierung in Abhängigkeit von d und n.