Universität Karlsruhe
Fakultät für Informatik
ITI Wagner

2. Musterlösung

Problem 1: Das Postamtplatzierungsproblem **

Sei OE $x_1 \leq x_2 \leq \cdots \leq x_n$. Gesucht ist ein Punkt $p = (x, y)$ mit

$$\min_p \sum_{i=1}^{n} w_i d(p, p_i) = \min_p \left(\sum_{i=1}^{n} w_i \left(|x_p - x_i| + |y_p - y_i| \right) \right) \tag{1}$$

$$= \min_p \left(\sum_{i=1}^{n} w_i \left(|x_p - x_i| \right) + \sum_{i=1}^{n} w_i \left(|y_p - y_i| \right) \right) \tag{2}$$

Wir betrachten also zwei unabhängige eindimensionale Probleme, d.h wir suchen

$$\min_x \sum_{i=1}^{n} w_i d(x, x_i) \quad \text{und} \quad \min_y \sum_{i=1}^{n} w_i d(y, y_i).$$

Wir bestimmen im folgenden die x-Koordinate (für die y-Koordinate ergibt sich eine analoge Berechnung):

Falls $x_i = x_j$ für $i \neq j$ gilt, fassen wir die beiden Gewichte zusammen.

Sei $W := 1/2 \sum_{i=1}^{n} w_i$. Für den gewichteten Median x_m der x-Werte gilt:

$$\sum_{x_i < x_m} w_i \leq W \geq \sum_{x_i > x_m} w_i \tag{3}$$

Sei $W_m = \sum_{i=1}^{n} w_i d(x_m, x_i)$. Betrachte OBdA die linke Seite von x_m (rechte Seite ist analog). Es gilt $\sum_{x_i < x_m} w_i + w_m \geq W$. Für jeden Punkt $\tilde{x}$ mit $\tilde{x} = x_m + \Delta$ gilt:

$$\tilde{W} = W_m + \left(\sum_{x_i < x_m} w_i + w_m \right) \cdot \Delta - \left(\sum_{x_i > x_m} w_i \right) \cdot \Delta \tag{4}$$

$$= W_m + \underbrace{\left(\sum_{x_i < x_m} w_i + w_m - \sum_{x_i > x_m} w_i \right)}_{\geq 0 \text{ wegen (3)}} \cdot \Delta \geq W_m \tag{5}$$

Beachte, dass obige Ungleichung auch ähnlich gilt wenn $\tilde{x} \geq x_{m+1}$. Sei dazu $\Delta_{m,m+1} = |x_{m+1} - x_m|$

$$\tilde{W} = W_m + \underbrace{\left(\sum_{x_i < x_m} w_i + w_m - \sum_{x_i > x_{m+1}} w_i \right) \cdot \Delta - \underbrace{w_{m+1} \cdot \Delta_{m,m+1}}_{\text{über } \Delta_{m,m+1} \text{ war } x_{m+1} \text{ rechts}}}_{\geq 0 \text{ wie vorher (sogar schlimmer)}} + \underbrace{w_{m+1} \cdot \underbrace{(\Delta - \Delta_{m,m+1})}_{\geq 0}}_{\text{nun } x_{m+1} \text{ links}} \geq W_m \tag{6}$$

Somit gilt also

$$\forall\, \tilde{x} : \quad \tilde{W} \geq W_m$$

Man finde also mit G-Median (vgl. Übungsblatt 1) in linearer Zeit die gewichteten Mediane x_m und y_m. Der Punkt (x_m, y_m) ist der gesuchte Punkt p.

Problem 2: UNION-FIND ***

Da alle UNION-Operationen vor allen FIND- und MAKESET-Operationen ausgeführt werden, vergrößert sich die Höhe eines Knoten nach den UNION-Operationen nicht mehr. Der Aufwand für die UNION- und MAKESET-Operationen liegt in $O(m)$. Danach werden nur noch FINDs auf eine Menge von Bäumen angewendet.

(a) Zunächst ist klar, dass die Operationen UNION und MAKESET insgesamt einen Aufwand in $O(m)$ haben und höchstens m Knoten für nachfolgende FIND-Operationen zur Verfügung stellen. Wie bei der Analyse des allgemeinen Falles, berechnen wir die Gesamtkosten nach der Ganzheitsmethode. Dazu teilen teilen wir die Kosten für die FIND-Operationen in zwei Teile auf:
Für eine Operation FIND(i) seien $i = i_0, i_1, \ldots, i_{l-1}, i_l$ die Knoten auf dem Weg von i zur Wurzel i_l. Die Pfadkompression sorgt dafür, dass bei einem FIND(i) für alle Knoten außer i_{l-1} und i_l die Höhe auf eins verringert wird. Die konstanten Kosten c_1 für i_l und i_{l-1} ordnen wir der FIND-Operation zu. Die Kosten für i_k mit $0 \leq k \leq l-2$ ordnen wir dem Knoten i_k zu, der bei der FIND-Operation bewegt wird. Da jeder Knoten höchstens einmal bewegt wird, ergibt sich für die Gesamtkosten K aller FIND-Operationen

$$\begin{aligned} K &= \text{Kosten für die FINDs} && + \text{ Kosten für die Knoten} \\ &\leq \underbrace{m \cdot c_1}_{\text{Kosten für die } O(m) \text{ FINDs}} && + \underbrace{m \cdot c_2}_{\#\text{Knoten} \cdot \#\text{Bewegungen pro Knoten}} \\ &= O(m) \end{aligned}$$

(b) Die obige Analyse macht von der *balancing*-Annahme keinen Gebrauch. Deswegen ergibt sich auch hier Gesamtaufwand $O(m)$.

(c) Sei $n \leq m$ die Anzahl der vorangegangenen MAKESET-Operationen. Die Höhe eines Knotens ist hier im schlimmsten Fall in $\Theta(\log n)$ also kann jede FIND-Operation Aufwand $\Theta(\log n)$ haben. Damit ergibt sich im schlechtesten Fall ein Gesamtaufwand von $\Theta(m \log n)$. Setzt man $n = m/2$, dann ist klar dass der Gesamtaufwand auch in $\Theta(m \log m)$ ist.

Problem 3: Tiefenbestimmungsproblem **

Die folgende Funktion erzeugt einen Baum mit einem Knoten.

Algorithmus 1 : MAKE-TREE(v)

Eingabe : Knoten v
Ausgabe : Neuer Baum mit Wurzel v
$d[v] := 0$
$\mathsf{Vor}[i] := -1$

Algorithmus 2 wird als Subroutine der Algorithmen 3 und 4 aufgerufen. Er liefert zu einem Knoten seine Tiefe und die Wurzel seines Baumes. Die Funktion FIND-DEPTH gibt lediglich das erste Argument von FIND zurück. Mit ATTACH wird ein Baum mit Wurzel v an einem beliebigen Knoten r eines anderen Baums angehängt.

Laufzeitanalyse Der Zeitaufwand von MAKE-TREE bzw. FIND-DEPTH ist analog zum Zeitaufwand von MAKESET bzw. FIND. Der Aufwand für ATTACH entspricht dem Aufwand für eine UNION- und eine FIND-Operation. Der Gesamtaufwand liegt also bei $O(2m \cdot G(2m))$. Wegen der Monotonie von G gilt für alle m:

$$G(2m) \leq G(2^m) \leq G(m) + 1 \leq 2G(m).$$

Daraus folgt $G(2m) \in O(G(m))$ und somit liegt der Gesamtaufwand bei $O(m \cdot G(m))$.

Algorithmus 2 : FIND(v)

Eingabe : Knoten v
Ausgabe : Das Paar $(\mathsf{depth}(v), r(v))$ (Tiefe und Wurzel von v)

```
r := v
j := v
depth := d[v]
solange Vor[j] > 0 tue
    j := Vor[v]
    depth := depth + d[j]
    r := j
Wenn r ≠ v                          // Pfadkompression
    j := v
    pdepth := 0
    solange Vor[j] > 0 tue
        temp := Vor[j]
        Vor[j] := r
        d[j] := depth − pdepth
        pdepth := pdepth + d[j]
        j := temp
(depth(v), r(v)) := (depth, r)
```

Algorithmus 3 : FIND-DEPTH(v)

Eingabe : Knoten v
Ausgabe : Die Tiefe $\mathsf{depth}(v)$ von v

```
(r, d) := FIND(i)
depth(v) := d
```

Algorithmus 4 : ATTACH(r, v)

Eingabe : Wurzel v eines Baumes und ein Knoten r
Ausgabe : Der Knoten v wird an r angehängt

```
(d, q) := FIND(r)
z := Vor[r] + Vor[q]                // Größe des neuen Baumes
Wenn −Vor[q] ≥ −Vor[v]
    Vor[r] := q
    d[r] := d[r] + d + 1
    Vor[q] := z
sonst
    Vor[q] := r
    d[r] := d[r] + d + 1
    d[q] := −d[r]
    Vor[r] := z
Vor[v] := r
```

Problem 4: Graustufenbilder **

(a) Der im folgenden angegebene Algorithums bildet für jeden Bildpunkt eine Menge und geht danach die Matrix Spaltenweise von oben nach unten durch. Dabei wird für jedes Element der obere und linke Nachbar in der Matrix (falls vorhanden) auf Äquivalenz geprüft. Sollte sich herausstellen, dass die Bildpunkte äquivalent sind, so werden ihre Mengen vereinigt. Der Index ist eine kanonische Abbildung von $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$, da wir für UNION-FIND $\mathbb{N}$ (1-dimensional) als Schlüsselmenge nutzen wollen.

Algorithmus 5 : Äquivalenzklassenbildung

Eingabe : Die Bildpunke als $(m \times n)$-Matrix mit den Graustufenwerten als Einträge.
Ausgabe : Einteilung der Bildpunkte in Äquivalenzklassen bzgl. der gegebenen Äquivalenzrelation.

```
1  Für i = 1,...,m
2      Für j = 1,...,n
3          INDEX (i,j) := n·(j−1)+i
4          MAKESET (INDEX(i,j))
5  Für j = 1,...,n
6      Für i = 1,...,m
7          Wenn grauwert (i,j)=grauwert (i−1,j)
8              UNION (FIND(INDEX(i,j)),FIND(INDEX(i−1,j)))
9          Wenn grauwert (i,j)=grauwert (i,j−1)
10             UNION (FIND(INDEX(i,j)),FIND(INDEX(i,j−1)))
```

(b) Bei jedem Durchlauf der Schleife in Zeile 6 wird ein Bildpunkt (i_0, j_0) seiner Äquivalenzklasse zugeordnet. Nach jedem solchen Schritt gilt folgende Invariante:

> Jedes Element der Menge $M := \{(i,j) \mid (j < j_0) \vee (j = j_0 \wedge i \leq i_0)\}$ ist seiner Äquivalenzklasse eingeschränkt auf M zugeordnet.

Problem 5: d-Heaps **

(a) Wir betrachten als Beispielswerte die Werte von 1 bis 13. Ein 5-Heap bei dem ein Vaterknoten einen größeren Wert hat als alle Kinderknoten könnte wie in Abb.1 aussehen.

(b) Wie bei binären Heaps sei ein d-Heap ein voller d-ärer Baum. Ein vollständiger d-ärer Baum der Höhe h hat

$$1 + d + d^2 + \cdots + d^h = \sum_{i=0}^{h} d^i = \frac{d^{h+1} - 1}{d - 1} =: n$$

Knoten. Also gilt hier

$$h + 1 = \log_d(n(d-1) + 1).$$

Demnach ist die Höhe $h(n)$ eines Baumes mit n Knoten

$$h(n) = \lceil \log_d((n)(d-1) + 1) \rceil = O(\log_d n).$$

Für festes d gilt

$$h(n) = O(\log_2 n)$$

(c) Die Nummerierung der Arrayelemente beginne wie in der Vorlesung bei 1. Gesucht wird succ(i, j) im

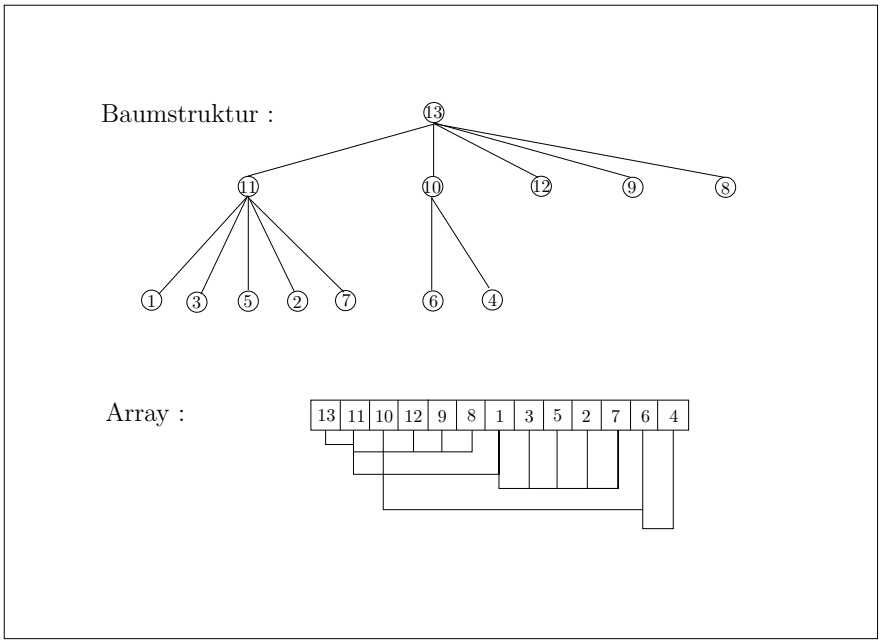

Abbildung 1: Baumstruktur und Arraystruktur eines 5-Heaps

d-Heap. Sei ℓ die Tiefe in der Knoten i liegt (wobei $\ell_{\text{Wurzel}} = 0$). Dann gilt:

$$\begin{aligned}
\text{Knoten im vollst. Baum bis Tiefe } \ell-1 &: \sum_{i=0}^{\ell} d^i = \frac{d^\ell - 1}{d-1} \\
\text{Knoten vor } i \text{ innerhalb Tiefe } \ell &: i - \frac{d^\ell - 1}{d-1} - 1 \\
\text{Knoten vor succ(i, j) innerhalb Tiefe } \ell+1 &: d(i - \frac{d^\ell - 1}{d-1} - 1) \\
\text{Knoten im vollst. Baum bis Tiefe } \ell &: \frac{d^{\ell+1} - 1}{d-1}
\end{aligned}$$

Der Index succ(i, j) ist also:

$$\begin{aligned}
\text{succ(i, j)} &= d(i - \frac{d^\ell - 1}{d-1} - 1) + \frac{d^{\ell+1} - 1}{d-1} + j \\
&= d \cdot i - \frac{d^{\ell+1}}{d-1} + \frac{d}{d-1} - d \cdot 1 + \frac{d^{\ell+1}}{d-1} - \frac{1}{d-1} + j \\
&= d \cdot (i-1) + 1 + j
\end{aligned}$$

Gesucht wird nun pred(k) im d-Heap, aus der Formel für succ(i, j) ergibt sich:

$$\begin{aligned}
& k = d \cdot (i-1) + 1 + j \\
\Rightarrow \quad & i = \frac{k-1-j}{d} + 1 \\
= \; & \underbrace{\frac{k-1}{d} + 1}_{>i \text{ und } \leq i+1 \text{ (wegen Absch. rechts)}} - \underbrace{\frac{j}{d}}_{>0 \text{ und } \leq 1} \in \mathbb{N} \\
\Rightarrow \quad & = \left\lceil \frac{k-1}{d} \right\rceil
\end{aligned}$$

(d) `Heapify` funktioniert ganz analog zum binären Fall:

Algorithmus 6 : Heapify$(A[1 \cdot\cdot n], i)$

Eingabe : Array A, so dass unterhalb vom Index i die Heap-Eigenschaft erfüllt ist
Ausgabe : Array A, so dass die Heap-Eigenschaft bis einschließlich i erfüllt ist.

```
k := Succ(i, 1) − 1
maxindex := k
solange k + 1 <= n tue
    k := k + 1
    Wenn A[k] > A[maxindex]
        maxindex := k
Wenn maxindex ≠ i
    tausche A[i] und A[maxindex]
    rufe Heapify(A, maxindex) auf
```

(e) `Heapify` wird wieder höchstens d_i Mal aufgerufen, wobei d_i die Höhe des Unterbaumes von i sei. Da dieser Unterbaum voll ist, gilt $d_i \in O(\log_d n)$. Somit ist die Gesamtlaufzeit von `Heapify`(A, i) in $O(\log_d n) = O(\log_2 n)$.