Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825

Institut für Theoretische Informatik
Lehrstuhl Prof. Dr. D. Wagner

Übungsblatt 1

Vorlesung Algorithmentechnik im WS 08/09

Problem 1: Amortisierte Analyse

Für die Implementierung benutzen wir zwei Stacks E und D. Der Stack E wird für ENQUEUE- und der Stack D für DEQUEUE-Operationen benutzt. Formal sind die Operationen wie folgt implementiert:

Algorithmus 1 : ENQUEUE(x)

```
1 PUSH(E,x)                                   // O(1)
```

Der Aufwand für das ENQUEUE ist in $\mathcal{O}(1)$.

Algorithmus 2 : DEQUEUE()

```
1 Wenn D = ∅
2 |  solange E ≠ ∅ tue
3 |  |  PUSH(D,POP(E))
4 pop(D)                                      // O(1)
```

Der Aufwand für das DEQUEUE ist in $\mathcal{O}(1)$ falls $D \neq \emptyset$ und in $\mathcal{O}(|E|)$, falls $D = \emptyset$.

1.1 Ganzheitsmethode

- wenn DEQUEUE Aufwand von $|E|$ verursacht, dann müssen vorher $|E|$ ENQUEUE-Operationen stattgefunden haben
- Gesamtaufwand für DEQUEUE-Operationen ist höchstens so groß wie Gesamtaufwand für ENQUEUE-Operationen, d.h. $\mathcal{O}(n)$

$\Rightarrow$ Gesamtaufwand $T(n) = \mathcal{O}(n)$

- Aufwand pro Operation $T(n)/n = \mathcal{O}(1)$

1.2 Buchungsmethode

- definieren Kosten wie folgt

	tatsächliche Kosten	**amortisierte Kosten**
ENQUEUE	1	3
DEQUEUE	1, $\|E\|$	1

- *Idee:* beim Einfügen bezahlt jedes Element das Einfügen in E sofort und hinterlegt Kredit von 2 für Hinzufügen zu D (1 POP, 1 PUSH)

$\Rightarrow$ amortisierte Kosten in $\mathcal{O}(1)$

1.3 Potentialmethode

- sei D_i Datenstruktur nach i-ter Operation,
- Datenstruktur sei zu Beginn leer
- definiere Potential $\Phi: \ D_i \mapsto \Phi(D_i) := 2|E|$
- $\Phi(D_O) = 0$, $\Phi(D_i) \geq \Phi(D_0)$
- im Folgenden $\hat{c}_i$: *amortisierte Kosten*, c_i *reale Kosten* der i-ten Operation
- dann gilt
$$\hat{c_i} := c_i + \Phi(D_i) - \Phi(D_{i-1})$$
- i-te Operation ist ENQUEUE mit $|E| := k$:
$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 2(k+1) - 2k = 3\end{aligned}$$
- i-te Operation ist DEQUEUE mit $|D| > 0$, $|E| := k$:
$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 2k - 2k = 1\end{aligned}$$
- i-te Operation ist DEQUEUE mit $|D| = 0$, $|E| := k$:
$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 2k + 1 + 0 - 2k = 1\end{aligned}$$

Problem 2: Untere Schranken

Betrachte Entscheidungsbäume mit Maximalgrad k, wie in der Übung vorgestellt, als Algorithmenmodell. Wir interessieren uns für die minimale Anzahl an Entscheidungen, die jeder Algorithmus durchführen muss, um den Index i_{max} des maximalen Elementes in der gegebenen Folge zu bestimmen. Jeder Entscheidungsbaum für das Maximum-Problem muss mindestens n Blätter haben, da jeder Index in $\{1, \ldots, n\}$ als Index des maximalen Elementes in Frage kommt. Die Höhe eines Baumes mit n Blättern ist mindestens $\log_k n$. Um dies einzusehen, kann man sich klarmachen, dass jeder Weg von der Wurzel zu einem Blatt verwendet werden kann, um die Ausgabe (den Wert des Blattes) zu kodieren. Ein Baum mit Maximalgrad k und Höhe h kann dann höchstens k^h Blätter haben, d.h. $n \leq k^h$ und somit $h \geq \log_k n$. Damit muss jeder Entscheidungsbaum eine Mindesthöhe von $\log_k n$ haben und es existiert für jeden solchen Baum eine Eingabe, für die mindestens $\log_k n$ Entscheidungen getroffen werden müssen.

Bemerkung: Diese untere Schranke ist sogar scharf, wenn man das allgemeine Entscheidungsbaum-Modell betrachtet. Wie in der Übung gezeigt, kann man aber auch eine realistischere untere Schranke von $n - 1$ Vergleichen beweisen.

Problem 3: Laufzeit rekursiver Funktionen

Betrachte die zweite Laufzeit mit Hilfe des allgemein formulierten Master-Theorems aus der Vorlesung. Es gilt:

$$\sum_{i=1}^{m} \alpha_i^k = \sum_{i=1}^{7} \left(\frac{1}{2}\right)^2 = \frac{7}{4} > 1$$

Somit gilt der dritte Fall:

$$T_A(n) \in \Theta(n^c) \quad \text{wobei für } c \text{ gilt:} \quad \sum_{i=1}^{7} \left(\frac{1}{2}\right)^c = 1 \quad \Rightarrow \quad \left(\frac{1}{2}\right)^c = \frac{1}{7}$$

$$\text{und somit:} \quad 2^c = 7 \quad \Rightarrow \quad c = \log_2 7$$

Für die erste Laufzeit hingegen gilt analog:

$$\sum_{i=1}^{m} \alpha_i^k = \sum_{i=1}^{a} \frac{1}{4}^2 = \frac{a}{16}$$

Somit gilt zunächst folgende Fallunterscheidung:

$$\begin{array}{lll} T_B(n) \in \Theta(n^2) & \text{falls} & a < 16 \\ T_B(n) \in \Theta(n^2 \log n) & \text{falls} & a = 16 \\ T_B(n) \in \Theta(n^c) & \text{falls} & a > 16 \quad \text{wobei wieder gilt} \quad \sum_{i=1}^{a} \left(\frac{1}{4}\right)^c = 1 \end{array}$$

F"ur $a \leq 16$ ist die Laufzeit $T_B(n)$ demnach geringer als $T_A(n)$. Beachte, dass wegen $\sum_{i=1}^{a} \alpha_i^k > 1 = \sum_{i=1}^{a} \alpha_i^c$ stets gilt $c > k$. F"ur $a > 16$ ist eine weitere Fallunterscheidung n"otig, c muss für alle Werte von a berechnet werden. Berechne zun"achst c für $T_B(n)$:

$$\sum_{i=1}^{a} \left(\frac{1}{4}\right)^c = 1 \quad \Rightarrow \quad \left(\frac{1}{4}\right)^c = \frac{1}{a}$$

$$\text{somit gilt:} \quad 4^c = a \quad \Rightarrow \quad c = \log_4 a$$

Vergleiche nun, wann für $T_A(n)$ und $T_B(n)$ derselbe Wert c gilt (Gleichstand):

$$\log_4 a = \log_2 7 \quad \Leftrightarrow$$

$$a = 4^{\log_2 7} = 2^{2 \log_2 7} = (2^{\log_2 7})^2 = 7^2 = 49$$

Somit ist im Θ-Kalkül ein Algorithmus mit der Laufzeit $T_B(n)$ für $a < 49$ asymptotisch schneller als ein Algorithmus mit der Laufzeit $T_A(n)$.

Problem 4: Das Tiefen-Bestimmungs-Problem

Für das Tiefen-Bestimmungs-Problem wird eine Union-Find-Datenstruktur mit weighted Union und Pfadkompression modifiziert. Die Idee besteht darin, in der Union-Find-Datenstruktur nur die Knoten (ohne strukturelle Information über den Baum) zu verwalten und Knoten v in der Union-Find-Datenstruktur mit einem Label $d(v)$ zu annotieren, welches zur Tiefen-Bestimmung verwendet werden kann. Das Label $d(v)$ ist wie folgt definiert:

$$\begin{aligned} d(v) &:= \textsc{Find-Depth}(v) - \textsc{Find-Depth}(w) \quad (i < k) \\ d(w) &:= \textsc{Find-Depth}(w) \end{aligned}$$

wobei w der unmittelbare Vorgänger von v in der Union-Find-Datenstruktur ist (nicht im richtigen Baum).

Die Operation MAKE-TREE erzeugt einen neuen Baum und ist analog zu MAKESET. Da x beim Erzeugen eines neuen Baumes Wurzel ist und Tiefe 0 hat, wird $d[x]$ mit 0 initialisiert.

Algorithmus 3 : MAKE-TREE(x)

Eingabe : Knoten x
Ausgabe : Neuer Baum mit Wurzel x

```
1 d[x] ← 0
2 VOR[x] ← −1
```

Die Operation FIND wurde so modifiziert, dass sie die Label auf dem abgesuchten Pfad aggregiert. Diese Summe ergibt nach Definition der Label genau die Tiefe des von FIND gesuchten Knotens. Zusätzlich wird nicht nur der Repräsentant der Menge (des Baumes), sondern auch die Tiefe des Knotens zurückgegeben. Bei der Pfadkompression müssen die Label angepasst werden. Dazu wird eine Variable PDEPTH verwendet, in der die Partialsumme der Label auf dem aktuellen Pfad gespeichert wird.

Algorithmus 4 : FIND(x)

Eingabe : Knoten x
Ausgabe : Das Paar (DEPTH, r) (Tiefe und Wurzel von x)

```
1  r ← x//                                          Repräsentanten-Suche
2  DEPTH ← d[x]
3  solange VOR[r] > 0 tue
4  |  r ← VOR[r]
5  |  DEPTH ← DEPTH + d[r]
6  i ← x //                                         Pfadkompression
7  PDEPTH ← 0
8  solange VOR[i] > 0 tue
9  |  t ← d[i]
10 |  d[i] ← DEPTH − PDEPTH − d[r]
11 |  PDEPTH ← PDEPTH + t
12 |  temp ← i
13 |  i ← VOR[i]
14 |  VOR[temp] ← r
```

Bemerkung zur Korrektheit: Sei $v_0, \ldots, v_k$ die Folge der Knoten, die FIND(x) durchläuft, d.h. $x = v_0$. Betrachte den i-ten Schleifendurchlauf der Schleife in Zeile 8, d.h. das Label von v_i wird neu berechnet. Wir bezeichnen mit d_{alt} den Wert der Label vor der Find-Operation und mit d_{neu} den Wert des Labels nach der FIND-Operation. Dann gilt : DEPTH=FIND-DEPTH(x) und hat den Wert

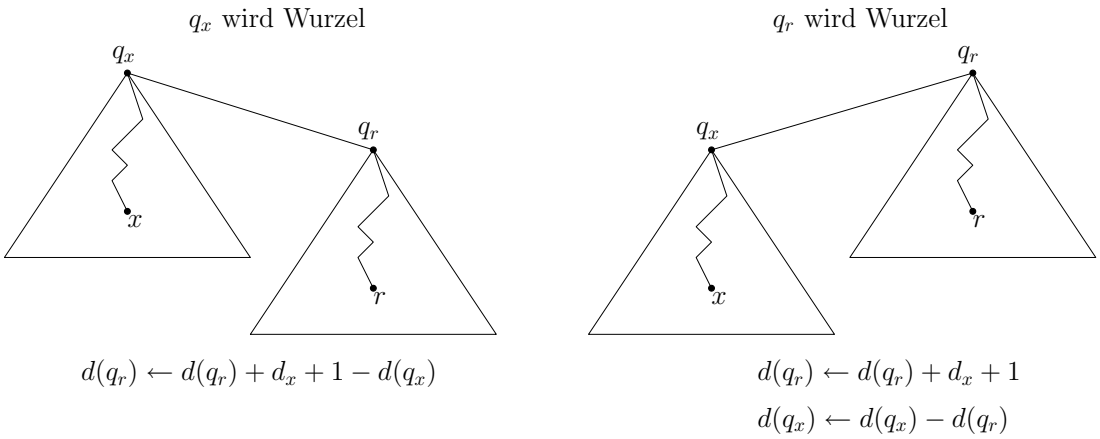

Abbildung 1: Fallunterscheidung bei ATTACH(r, x)

$\sum_{s=0}^{k} d_{alt}[s]$. PDEPTH hat den Wert $\sum_{s=0}^{i-1} d_{alt}[s]$. Damit gilt

$$\begin{aligned} d_{neu}[v_i] &= \text{DEPTH} - \text{PDEPTH} - d[r] \\ &= \sum_{s=0}^{k} d_{alt}[s] - \sum_{s=0}^{i-1} d_{alt}[s] - d[r] \\ &= \sum_{s=i}^{k} d_{alt}[s] - d_{alt}[r] \\ &= \text{FIND-DEPTH}(v_i) - d_{alt}[r] \\ &= \text{FIND-DEPTH}(v_i) - \text{FIND-DEPTH}(r) \end{aligned}$$

ATTACH(r, x) ist eine Modifikation der Weighted-Union Operation aus der Vorlesung und hängt den Baum mit Wurzel r an den Knoten x eines anderen Baumes. Es sei darauf hingewiesen, dass weder r noch x Wurzeln in der Union-Find-Datenstruktur sein müssen. Daher bestimmt ATTACH zunächst die Repräsentanten von x und r. Anschließend werden die Mengen gemäß Weighted-Union vereinigt und es werden neue Labels berechnet (siehe auch Abbildung 1).

Algorithmus 5 : ATTACH(r, x) (modifiziertes Union)

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

```
1  (d_x, q_x) ← FIND(x)
2  (d_r, q_r) ← FIND(r)
3  z ← VOR[q_x] + VOR[q_r]
4  Wenn |VOR[q_r]| < |VOR[q_x]|                          q_x wird Wurzel
5  |  VOR[q_r] ← q_x und VOR[q_x] ← z
6  |  d(q_r) ← d(q_r) + d_x + 1 − d(q_x)
7  sonst                                                 q_r wird Wurzel
8  |  VOR[q_x] ← q_r und VOR[q_r] ← z
9  |  d(q_r) ← d(q_r) + d_x + 1
10 |  d(q_x) ← d(q_x) − d(q_r)
```

Schließlich wird für die Operation FIND-DEPTH lediglich FIND auf gerufen und die gefundene Tiefe

ausgegeben.

Algorithmus 6 : FIND-DEPTH(x)

Eingabe : Knoten x
Ausgabe : Die Tiefe DEPTH(x) von x

```
1 (d, r) ← FIND(i)
2 DEPTH(x) ← d
```

Problem 5: Berechnung starker Zusammenhangskomponenten

Zunächst war zu zeigen, dass $\rightleftharpoons$ eine Äquivalenzrelation ist. Dies ist für die Korrektheit des Algorithmus notwendig.

- *Reflexivität:* $u \rightleftharpoons u$, da Weg der Länge 0 existiert
- *Symmetrie:* $u \rightleftharpoons v \Rightarrow v \rightleftharpoons u$, nach Definition
- *Transitivität:* $u \rightleftharpoons v$, $v \rightleftharpoons w \Rightarrow u \rightleftharpoons w$, da Weg von u über v nach w und umgekehrt existiert

Bemerkung: $u \rightleftharpoons w$ gdw. u und v liegen auf einem gerichteten Kreis

Für die Berechnung starker Zusammenhangskomponenten wird eine (rekursive) Tiefensuche modifiziert. Die Idee besteht darin, iterativ gerichtete Kreise zu "kontrahieren", die von der Tiefensuche gefunden werden, da alle Knoten auf einem gerichteten Kreis nach obiger Bemerkung in der selben starken Zusammenhangskomponente liegen. Es wird ein zusätzlicher Stack verwendet, um Information darüber zu verwalten, welche Knoten auf einem gerichtetn Kreis liegen. Wenn $v_0, \ldots, v_k$ die Folge der Knoten auf dem aktuellen Tiefensuche-Pfad sind, dann liegen auf dem Stack Elemente $w_0, \ldots, w_r$ mit folgender Eigenschaft: Für jedes $i = 1, \ldots, n$ existiert ein w_j, so dass FIND(v_i) = FIND(w_j) und falls $i < j$ und FIND(v_i) = FIND(w_a) sowie FIND(v_j) = FIND(w_b) gilt mit $a \neq b$, dann gilt $a < b$. D.h. auf dem Stack liegen Repräsentanten von $v_0, \ldots, v_k$ in der entsprechenden Reihenfolge und kein Repräsentant kommt mehrfach vor.

Im Unterschied zur klassischen (rekursiven) Tiefensuche werden nun nicht nur Knoten, sondern Mengen (von äquivalenten) Knoten gefärbt. Wenn MOD-DFS-VISIT einen Knoten zum ersten mal besucht, wird die Menge, in der dieser liegt, grau gefärbt, was bedeutet, dass dieser Knoten offen ist, bis die Menge, in der er liegt schließlich schwarz gefärbt wird. Eine solche Menge wird erst dann schwarz gefärbt (abgeschlossen), wenn alle Nachfolger aller Knoten in der Menge betrachtet wurden.

Algorithmus 7 : STARKE-ZUSAMMENHANGSKOMPONENTEN

Daten : Stack S

```
1 Für v ∈ V
2 |  MAKESET(v)
3 |  π(w) ← NIL
4 |  COLOR[FIND(v)] ← WHITE
5 Für v ∈ V
6 |  Wenn COLOR[FIND(v)] = WHITE
7 |  |  MOD-DFS-VISIT(v)
```

Wenn MOD-DFS-VISIT einen neuen Knoten betrachtet, wird nachgesehen, ob die Menge, in der dieser Knoten liegt, noch grau ist. Ist dies der Fall, so gibt es in dieser Menge einen Knoten, der auf

dem aktuellen Tiefensuche-Pfad noch offen ist. Da alle Knoten in der betrachteten Menge äquivalent sind, gibt es auch einen Pfad von dem aktuell gefundenen Knoten zu diesem offenen Knoten und damit wurde ein Kreis gefunden. Dieser wird dann kontrahiert, d.h. die Knoten auf dem Kreis werden zu einer Menge vereinigt. Beachte, dass auf dem Kreis bereits kontrahiert Knoten liegen können.

Algorithmus 8 : MOD-DFS-VISIT(v, S)

```
1  COLOR[FIND(v)] ← GRAY
2  Wenn FIND(v) ≠ FIND(PEEK(S))
3      PUSH(S, v)
4  Für (v, w) ∈ E
5      Wenn COLOR[FIND(w)] = WHITE                          neuer Knoten
6          π(w) ← v
7          MOD-DFS-VISIT(w, S)
8      sonst, wenn COLOR[FIND(w)] = GRAY                    Kreis gefunden
9          KONTRAHIERE-KREIS(w, S)
10 Wenn FIND(π(v)) ≠ FIND(PEEK(S))                          Stack aufräumen
11     COLOR[FIND(v)] ← BLACK
12     POP(S)
```

KONTRAHIERE KREIS benutzt die auf dem Stack vorgehaltene Information über noch offene Knoten, um die Knoten auf dem gefundenen Kreis zu kontrahieren. Dabei ist w der Knoten auf dem Kreis, der in der Tiefensuche als erstes angetroffen wurde. Daher liegt dieser auf dem Stack am tiefsten. KONTRAHIERE KREIS nimmt alle Knoten bis einschließlich w vom Stack und vereinigt diese Mengen zu einer neuen Menge, die dann grau gefärbt wird.

Algorithmus 9 : KONTRAHIERE-KREIS(w, S)

```
1  x ← POP(S)
2  solange FIND(w) ≠ FIND(x) tue
3      UNION(FIND(w), FIND(x))
4      x ← POP(S)
5  COLOR[FIND(w)] ← GREY
6  PUSH(S, w)
```