Mengen / Union-Find-Strukturen

Gegeben: Datensätze, partitioniert in (disjunkte) Mengen, wobei jede Menge durch einen in ihr enthaltenen Datensatz repräsentiert werde.

Ziel: Datenstruktur, die die folgenden drei Operationen möglichst effizient realisiert:

`MakeNewSet(`x`)`	$\rightarrow$	Füge neuen Datensatz x ein; dieser bildet eine neue (einelementige) Menge
`Find(`x`)`	$\rightarrow$	Gebe Repräsentanten der Menge aus, zu der x gehört
`Union(`x`, `y`)`	$\rightarrow$	Vereinige die beiden Mengen mit Repräsentanten x und y, wobei x Repräsentant der neuen Menge sei.

Realisierung: Jede Menge wird durch einen zur Wurzel hin gerichteten Baum (sog. intree) dargestellt, an der Wurzel notieren wir den Repräsentanten der Menge.

2.3.1 Grundversion

`MakeNewSel(`x`):`	mache x zur Wurzel eines neuen (einelementigen) Baumes
`Find(`x`):`	Laufe von x zur Wurzel r, gebe Repräsentanten von r aus
`Union(`x`, `y`):`	Hänge Wurzel des kleineren der beiden Bäume (“kleiner” $\hat{=}$ gesamte Anzahl der Knoten im Baum) an die Wurzel des größeren an.

<u>Lemma:</u> Für jeden in der Union-Find-Struktur enthaltenen Baum T gilt

$$\text{size}(T) \geq 2^{\text{height}(T)}$$

wobei size$(T)\hat{=}$ Anzahl der Knoten in T und height(T) die Höhe von T sei.

2.3.2 Path Compression

Idee: Ändere `Find`(x) so ab, daß alle Knoten auf dem Pfad von x zur Wurzel r *direkt* unter die Wurzel gehängt werden. (Dadurch erhöht sich die Laufzeit von `Find` nur um einen konstanten Faktor!)

Satz: Ausgehend von einer leeren Datenstruktur benötigen k beliebige Operationen, darunter genau n `MakeNewSet`, höchstens $\mathcal{O}(k \log^* n)$ Schritte.

Zur Erinnerung:

$$\log^* n = \min\{i > 1 | \underbrace{\log \log \ldots \log}_{i \text{ mal}} n \leq 1\}$$

(Für alle “normalen” Probleme ist $\log^* n$ immer $\leq$ 6)

Für den Beweis des Satzes führen wir eine amortisierte Laufzeitanalyse durch. Dazu definieren wir uns zunächst eine geeignete Potentialfunktion:

- Definiere $a_0, a_1, a_2, \ldots$ durch $\quad a_0 = 0, \quad a_{i+1} = 2^{a_i}$.
  (Die Funktion $n \mapsto a_n$ ist (cum grano salis) die Umkehrung von $\log^* n$: es gilt $\log^*(a_n) = n$.)

- Ordne jedem Element x drei Werte zu:

  rank(x) = Höhe des Unterbaums mit Wurzel x — in der Datenstruktur *ohne* Path Compression;

  class(x) = $i \quad \Longleftrightarrow \quad a_i \leq \text{rank}(x) < a_{i+1}$

  dist(x) = Länge des kürzesten Pfades von x zu einem Knoten y mit class$(y) >$ class(x) bzw. zur Wurzel — in der Datenstruktur *mit* Path Compression;

- Potential: $\Phi = c \cdot \sum_x \text{dist}(x)$ für eine geeignet gewählte Konstante $c > 0$.

Union-Find-Strukturen: Anwendungsbeispiel

Eine Standard-Anwendungen ist:

<u>`Kruskals Algorithmus`</u>

Eingabe: zshgd. Graph $G = (V, E)$; Gewichte $w[e]$;

Ausgabe: Kantenmenge T eines min. spann. Baums in G.

Sortiere die Kantenmenge so, daß gilt:

$$w[e_1] \leq w[e_2] \leq \ldots \leq w[e_m];$$

$T := \emptyset$;

for all $v \in V$ **do** MakeNewSet(v);

for i := 1 **to** m **do**

 Bestimme $u, v \in V$ mit $e_i = \{u, v\}$;

 if Find(u) $\neq$ Find(v) **then**

  $T := T \cup \{e_i\}$;

  Union(Find(u), Find(v));

Laufzeit: $\mathcal{O}(m \log n + m \log^* n)$, wobei der erste Term entfällt, wenn die Kanten bezüglich ihrer Gewichte vorsortiert sind.