CoNe Freiburg

IIF INSTITUT FÜR INFORMATIK FREIBURG

Graphentheorie
18 – Union Find Datenstruktur

Dr. Sven Köhler
Rechnernetze und Telematik
Technische Fakultät
Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Union Find

Operationen der Datenstruktur:

- $\mathrm{MakeSets}(n)$: Erzeugt n diskunkte Mengen $\{i\}$ mit $i \in [1, n]$.
- $\mathrm{Find}(x)$: ermittelt die Identität der Menge welche Element x enthält.
  Als Identität einer Menge dient ein Element der Menge.
- $\mathrm{Union}(x, y)$: vereinigt Mengen welche x bzw. y enthalten.

Anwendung in dieser Vorlesung:

- Kruskal startet mit n Zusammenhangkomponenten
- Wird $e = [v_i, v_j]$ dem Spannbaum hinzugefügt, so wird $\mathrm{Union}(i, j)$ aufgerufen.
- Eine Kante $[v_i, v_j]$ schließt einen Kreis, wenn $\mathrm{Find}(i) = \mathrm{Find}(j)$.

Test auf Kreise ist sehr effizient!

Union Find – Beispiel für $n = 6$

Initialisierung:

- $\mathrm{MakeSets}()$

Zustand: $\{\mathbf{1}\}\ \{\mathbf{2}\}\ \{\mathbf{3}\}\ \{\mathbf{4}\}\ \{\mathbf{5}\}\ \{\mathbf{6}\}$

Operationen:

- $\mathrm{Union}(1, 2)$
- $\mathrm{Union}(1, 4)$
- $\mathrm{Union}(3, 5)$

Ergebnis: $\{1, \mathbf{2}, 4\}\ \{3, \mathbf{5}\}\ \{\mathbf{6}\}$

- $\mathrm{Find}(1) = 2$
- $\mathrm{Find}(4) = 2$
- $\mathrm{Find}(3) = 5$
- $\mathrm{Find}(6) = 6$

Union Find – Beispiel für $n = 6$

Zustand: $\{1, \mathbf{2}, 4\}\ \{3, \mathbf{5}\}\ \{\mathbf{6}\}$

Operation:

- $\mathrm{Union}(1, 3)$

Ergebnis: $\{1, \mathbf{2}, 3, 4, 5\}\ \{\mathbf{6}\}$

- $\mathrm{Find}(1) = 2$
- $\mathrm{Find}(2) = 2$
- $\mathrm{Find}(4) = 2$
- $\mathrm{Find}(3) = 2$
- $\mathrm{Find}(5) = 2$
- $\mathrm{Find}(6) = 6$

Union Find – Beispiel

Zustand: $\{1, \mathbf{2}, 3, 4, 5\}\ \{\mathbf{6}\}$

Operation:

- $\mathrm{Union}(5, 6)$

Neuer Zustand: $\{1, \mathbf{2}, 3, 4, 5, 6\}$

- $\mathrm{Find}(1) = 2$
- $\mathrm{Find}(2) = 2$
- $\mathrm{Find}(4) = 2$
- $\mathrm{Find}(3) = 2$
- $\mathrm{Find}(5) = 2$
- $\mathrm{Find}(6) = 2$

Union Find – Kruskal

Algorithmus 6.1 Algorithmus von Kruskal mit Union Find

Eingabe: $G = (V, E)$, $c : E \to \mathbb{R}$, $V = \{v_1, v_2, \ldots, v_n\}$
Ausgabe: Kantenmenge E_F des MSF

Sortiere Kanten nach Kosten: $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$
$\mathrm{MakeSets}(n)$
$E_F := \emptyset$

for $i = 1, 2, \ldots, m$ **do**
    Seien v_x, v_y die Endknoten von e_i
    if $\mathrm{Find}(x) \neq \mathrm{Find}(y)$ **then**
        $E_F := E_F \cup \{e_i\}$
        $\mathrm{Union}(x, y)$

return E_F

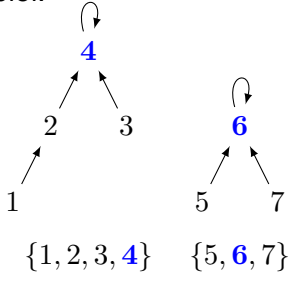
Union Find – Baumstrukturen

Mengen werden als Wurzelbäume gespeichert.
Wurzel ist Identität der Menge.

Darstellung in diesem Kapitel:

- Alle Kanten zeigen zur Wurzel.
- Wurzel hat sich selbst als Elternknoten.
- Für jeden Knoten wird lediglich Elternknoten gespeichert.

Beispiel:

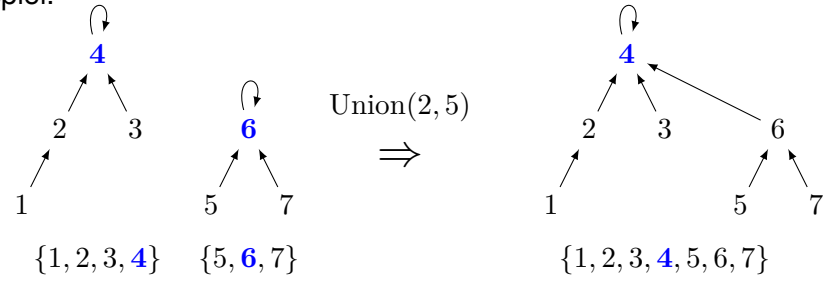


Union Find – Union By Size

Bäume können schnell vereinigt werden:

- Finde Wurzeln der beiden Mengen
- Mache Wurzel des kleineren Baumes
  zu Kind der Wurzel des größeren Baumes

Beispiel:



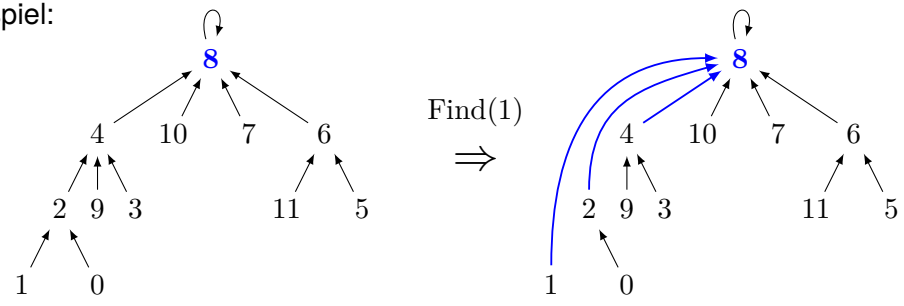
Union Find – Path Compression

Sei x ein Knoten und P der Weg von x zur Wurzel.

$\mathrm{Find}(x)$ mit Path Compression:

- Weg P wird zweimal abgeschritten
- Durch P berührte Knoten zeigen danach direkt auf Wurzel
- Zukünftige Aufrufe von $\mathrm{Find}(x)$ haben danach Laufzeit $\mathcal{O}(1)$
- Auch für andere Knoten wird $\mathrm{Find}(x)$ schneller

Beispiel:



Union Find – MakeSets Operation

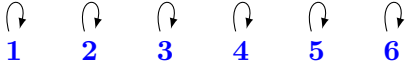
Algorithmus 18.1 $\mathrm{MakeSets}(n)$

Eingabe: $n \in \mathbb{N}$

Alloziere Array $p[]$ der Größe n ▷ Zeiger auf Elternknoten
Alloziere Array $s[]$ der Größe n ▷ Größe der Bäume

for each $i \in \{1, 2, \ldots, n\}$ **do**
  $p[i] := i$ ▷ Jeder Knoten ist Wurzel
  $s[i] := 1$ ▷ Jeder Baum enthält einen Knoten

Ergebnis für $n = 6$:



Union Find – Union Operation

Algorithmus 18.2 $\mathrm{Union}(x, y)$

Eingabe: $x, y \in [1, n]$

$r_x := \mathrm{Find}(x)$
$r_y := \mathrm{Find}(y)$
if $r_x = r_y$ **then**
    return ▷ x und y sind Teil desselben Baumes

if $s[r_x] >= s[r_y]$ **then**
    $p[r_y] := r_x$ ▷ r_y wird Kind der Wurzel r_x
    $s[r_x] := s[r_x] + s[r_y]$ ▷ Aktualisiere Größe des Baumes
else
    $p[r_x] := r_y$ ▷ r_x wird Kind der Wurzel r_y
    $s[r_y] := s[r_x] + s[r_y]$ ▷ Aktualisiere Größe des Baumes

Union Find – Find Operation

Algorithmus 18.3 $\mathrm{Find}(x)$

Eingabe: $x \in [1, n]$

$r := x$
while $p[r] \neq r$ **do** ▷ Laufe bis zur Wurzel
    $r := p[r]$

while $p[x] \neq r$ **do** ▷ Path Compression
    $y = p[x]$
    $p[x] := r$
    $x = y$

return r

Wir vernachlässigen Updates der Baumgrößen.
Lediglich der Wert der Wurzel muss stimmen.

Union Find – Komplexität

Satz

Betrachte $\mathrm{MakeSets}(n)$ gefolgt von m $\mathrm{Union}/\mathrm{Find}$-Operationen.
Der Speicherbedarf der Datenstruktur ist $\mathcal{O}(n)$.
Die Laufzeit von $\mathrm{MakeSets}()$ ist $\mathcal{O}(n)$.
Die Gesamtlaufzeit der $\mathrm{Union}/\mathrm{Find}$-Operationen ist $\mathcal{O}(m \log^* n)$.

$\log^*(x) =$ wie oft $\log$ anwenden, bis Ergebnis ≤ 1 ist?

Beispiel: $\log\log\log\log(65536) = 1$ bzw. $2^{\left(2^{\left(2^2\right)}\right)} = 65536$

$$0 = \log^*(1) \qquad 3 = \log^*(16)$$
$$1 = \log^*(2) \qquad 4 = \log^*(65536)$$
$$2 = \log^*(4) \qquad 5 = \log^*(2^{65536})$$

Insbesondere $\log^*(n) \leq 5$ für alle $n \leq 10^{19728}$

Union Find – Komplexität

Wir betrachten $\mathrm{MakeSets}(n)$ gefolgt von m $\mathrm{Union}/\mathrm{Find}$–Operationen.

Die Operationen werden in identischer Reihenfolge auf zwei Datenstrukturen angewendet:

- Die Datenstruktur F_U mit deaktivierter Path Compression
- Die Datenstruktur F_C mit aktivierter Path Compression

Beobachtung

Zu allen Zeitpunkten gilt:

- Die Bäume in F_U und F_C repräsentieren die gleichen Mengen.
- Die Bäume in F_U und F_C haben die gleichen Wurzeln.
- Die Werte $s[x]$ in F_U und F_C sind identisch.

Union Find – Komplexität

Wir betrachten $\mathrm{MakeSets}(n)$ gefolgt von m $\mathrm{Union}/\mathrm{Find}$–Operationen.

Sei M die Menge der von den $\mathrm{Union}/\mathrm{Find}$–Operationen betroffenen Knoten.

Beobachtung

- Alle Knoten in $[1, n] \setminus M$ sind in einem Baum der Größe 1.
- Die Knoten werden durch keine Find-Operation besucht.

Wir entfernen alle Knoten in $[1, n] \setminus M$ aus F_U und F_C.

Es verbleiben jeweils $n_M = |M| \leq \min(n, 2m)$ Knoten.

Wir zeigen: zusammen haben alle m Operationen Laufzeit $\mathcal{O}(m \log^* n_M)$.

Union Find – Ränge

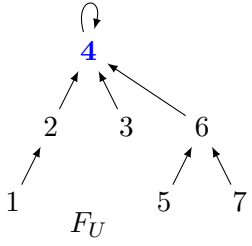
Definition

Der *Rang* eines Knotens x ist definiert als $r(x) = \lfloor \log s[x] \rfloor$.

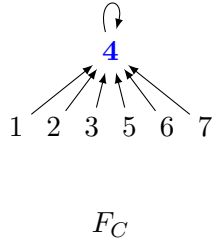
Beobachtung

In F_U gibt es $s[x] \geq 2^{r(x)}$ Knoten, die x als Vorfahren haben.

$\Rightarrow$ Es gibt höchstens $\frac{n_M}{2^r}$ Knoten mit Rang r.



F_U



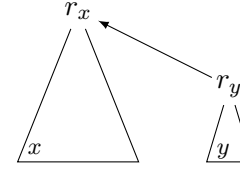
F_C

$s[4] = 7$
$s[6] = 3$
$s[2] = 2$
$s[3] = 1$
$s[1] = 1$

Union Find – Ränge

Lemma 18.1
Für F_U und F_C gilt folgende Invariante:
Entlang eines Weges steigt der Rang der Knoten streng monoton.

Proof.
$\mathrm{Union}(x, y)$ vereinigt zwei Bäume mit Wurzeln r_x und r_y.



Fall 1) $r(r_x) > r(r_y)$:
r_y wird Kind von r_x und $r(r_x)$ erhöht sich eventuell.
Somit gilt weiterhin $r(r_x) > r(r_y)$.

Fall 2) $r(r_x) = r(r_y) = r$:
Beide Bäume haben mind. 2^r Knoten. O.B.d.A. gilt $s[r_x] \geq s[r_y]$.
r_y wird Kind von r_x und es gilt $s[r_x] \geq 2^{r+1}$ und somit $r + 1 = r(r_x) > r(r_y)$.

Zu jedem Weg P_C in F_C gibt es einen Weg P_U in F_U, so dass die durch P_C berührten Knoten in P_U in der gleichen Reihenfolge auftauchen. □

Union Find – Buckets

Wir betrachten Buckets der Form $[r, 2^r)$.
Der erste Bucket ist $[0, 2^0) = [0, 1)$.
Der auf $[r, 2^r)$ folgende Bucket ist $[2^r, 2^{2^r})$.

Bucket $[r, 2^r)$ enthält alle Knoten x mit $r \leq r(x) < 2^r$,
d.h. er enthält alle Knoten mit $s[x] \in [2^r, 2^{2^r})$.

Bucket	Ränge $r(x)$	Baumgrößen $s[x]$
$[0, 2^0)$	0	1
$[1, 2^1)$	1	$2-3$
$[2, 2^2)$	$2-3$	$4-15$
$[4, 2^4)$	$4-15$	$16-65535$

Beobachtung

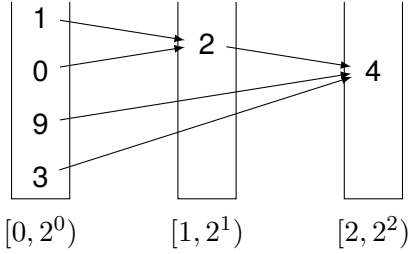
Bucket $[r, 2^r)$ enthält höchstens $\sum_{i=r}^{2^r-1} \frac{n_M}{2^i} = \frac{n_M}{2^r} \sum_{i=r}^{2^r-1} \frac{1}{2^{i-r}} \leq \frac{2n_M}{2^r}$ Knoten.

Union Find – Buckets

Laufzeit von Union wird durch die zwei Find-Aufrufe dominiert.
Gesamtlaufzeit ist proportional zu den von Find benutzen Kanten.

Wir unterteilen die Kanten in drei Klassen:

1) Kanten inzident zu einer Wurzel
2) Kanten deren Endknoten in unterschiedlichen Buckets liegen
3) Kanten deren Endknoten im selben Bucket liegen



$$s[0] = 1 \qquad r(0) = 0$$
$$s[1] = 1 \qquad r(1) = 0$$
$$s[2] = 3 \qquad r(2) = 1$$
$$s[4] = 6 \qquad r(4) = 2$$

Union Find – Komplexität

Lemma 18.2
Pro Find-Operation werden $\mathcal{O}(1)$ Kanten der Klasse 1 benutzt.
Pro Find-Operation werden $\mathcal{O}(\log^* n_M)$ Kanten der Klasse 2 benutzt.

Proof.
Jeder Weg zur Wurzel enthält höchstens eine Kante inzident zur Wurzel.

Nach Lemma 18.1 steigt der Rang monoton entlang jedes Weges.
Damit verlaufen Kanten immer von einem Bucket zu einem größeren Bucket.
Nur die ersten $\log^*(n_M + 1)$ Buckets können Knoten enthalten. Damit enthält jeder Weg $\mathcal{O}(\log^* n_M)$ Kanten der Klasse 2. □

Zwischenstand:
Gesamtzahl an Kanten bisher (ohne Klasse 3): $\mathcal{O}(m \log^* n_M)$.

Union Find – Komplexität

Lemma 18.3
Alle Find-Operationen benutzen zusammen $\mathcal{O}(n_M \log^* n_M)$ Kanten der Klasse 3.

Proof.
Betrachte Bucket $B = [r, 2^r)$ und einen Knoten u mit $r(u) \in B$.
Seien $(u, v_0), (u, v_1), (u, v_2), \ldots$ die in u startenden genutzten Kanten.

Path Compression ersetzt nach Lemma 18.1 die Kante (u, v_i) durch (u, v_{i+1}) mit $r(v_{i+1}) > r(v_i)$. (Hinweis: Kanten können aus Klasse 1 in Klasse 3 zurückkehren)

$B = [r, 2^r)$ enthält höchstens $\frac{2n_M}{2^r}$ Knoten und höchstens 2^r verschiedene Ränge.
Es werden also höchstens $\frac{2n_M}{2^r} 2^r = 2n_M$ Kanten mit Endknoten in B benutzt.

Nur die ersten $\log^*(n_M + 1)$ Buckets können Knoten enthalten.
Insgesamt werden also $\mathcal{O}(n_M \log^* n_M)$ Kanten aus Klasse 3 benutzt. □

Union Find – Komplexität

Satz

Betrachte $\mathrm{MakeSets}(n)$ gefolgt von m $\mathrm{Union}/\mathrm{Find}$-Operationen.
Die Gesamtlaufzeit der $\mathrm{Union}/\mathrm{Find}$-Operationen ist $\mathcal{O}(m \log^* n)$.

Proof.

Die Laufzeit einer Union-Operation ist dominiert durch die beiden Find-Aufrufe.

Von allen Find-Operationen werden zusammen

- $\mathcal{O}(m)$ Kanten der Klasse 1,
- $\mathcal{O}(m \log^* n_M)$ Kanten der Klasse 2 und
- $\mathcal{O}(n_M \log^* n_M)$ Kanten der Klasse 3

benutzt.

In Summe ergebt dies $\mathcal{O}(m \log^* n)$ Kanten da $n_M \leq \min(n, 2m)$. □