Die Operation DecreaseKey($\mathcal{H}, v, k$) (diese Operation ersetzt, falls $k < key(v)$, $key(v)$ durch k):

1. sei B_i der Binomialbaum in $\mathcal{H}$, der den Knoten v enthält
2. falls $k < key(v)$, ersetze $key(v)$ durch k
3. stelle, falls nötig, die Heap-Bedingung auf dem Pfad von v zur Wurzel von B_i wieder her, indem, solange nötig, der Schlüssel eines Knotens mit dem seines Vaters ausgetauscht wird

Falls $\mathcal{H}$ n Schlüssel enthält, beträgt die Laufzeit der DecreaseKey-Operation offensichtlich $O(\log n)$.

6. Mengendarstellungen — Union-Find-Strukturen

Gegeben ist eine (endliche) Menge S von Objekten, die in Klassen X_i partitioniert ist:

$$S = X_1 \uplus X_2 \uplus \ldots \uplus X_l$$

Für jede Klasse X_i gibt es dabei einen Repräsentanten $r_i \in X_i$.

Es wird eine Datenstruktur gesucht, die die folgenden Operationen effizient unterstützt:

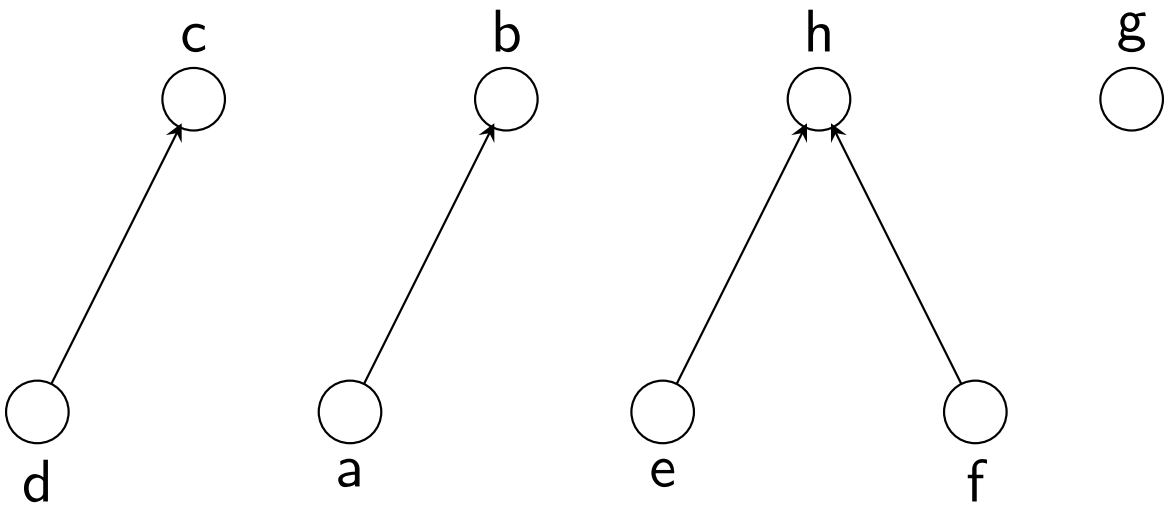
1. Init(S): jedes Element $\in S$ bildet eine eigene Klasse mit sich selbst als Repräsentant
2. Union(r, s): vereinige die beiden Klassen mit r bzw. s als Repräsentanten, wähle r als Repräsentant der neuen Klasse (die beiden alten Klassen verschwinden)
3. Find(x): bestimme zu $x \in S$ den Repräsentanten der eindeutigen aktuellen Klasse, die x enthält

Eine entsprechende Datenstruktur nennt man auch Union-Find-Struktur.

Eine triviale Implementierung, in der zu jedem Element aus S sein Repräsentant (bzw. ein Verweis darauf) gespeichert wird, benötigt im worst case Laufzeit $\Omega(|S|)$ für die Union-Operation.

Um logarithmische Laufzeit zu erreichen, realisieren wir die Datenstruktur als Vereinigung von zur Wurzel hin gerichteten Bäumen. Dabei bilden die Repräsentanten die Wurzeln und die restlichen Elemente der Klasse die restlichen Knoten.

Beispiel 206



Die Operationen Find und Union werden wie folgt implementiert:

Find(x) verfolge von x aus den Pfad zur Wurzel und gib diese aus; die worst-case Laufzeit ist proportional zur maximalen Höhe eines Baums in der Union-Find-Struktur.

Union(r, s) hänge die Wurzel des niedrigeren Baums als neues Kind der Wurzel des anderen Baums ein; vertausche notfalls r und s, damit r an der Wurzel des neuen Baums zu liegen kommt.

Wir nennen die bei der Union-Operation verfolgte Strategie gewichtete Vereinigung. Um sie effizient implementieren zu können, wird für jeden Baum der Union-Find-Struktur seine Höhe mitgeführt.

Lemma 207

Sei T ein Baum einer Union-Find-Struktur mit gewichteter Vereinigung. Dann gilt für seine Höhe $h(T)$

$$h(T) \leq \mathrm{ld}(|T|)\,,$$

wobei $|T|$ die Anzahl der Knoten in T bezeichnet.

Beweis:

Beweis durch Induktion über die Höhe.

Induktionsanfang: Für Bäume der Höhe 0 oder 1 stimmt die Behauptung.

Lemma 207

Sei T ein Baum einer Union-Find-Struktur mit gewichteter Vereinigung. Dann gilt für seine Höhe $h(T)$

$$h(T) \leq \mathrm{ld}(|T|)\,,$$

wobei $|T|$ die Anzahl der Knoten in T bezeichnet.

Beweis:

Induktionsschluss: Sei o.B.d.A. $h(T_r) \geq h(T_s)$. Ist $h(T_r) > h(T_s)$, so ist die neue Höhe $= h(T_r)$ und die Behauptung erfüllt.
Ist $h(T_r) = h(T_s)$, so ist die neue Höhe $= h(T_r) + 1$, und es gilt:

$$\begin{aligned}
\mathrm{ld}(|T_r| + |T_s|) &\geq \mathrm{ld}(2\min\{|T_r|, |T_s|\}) \\
&= 1 + \mathrm{ld}(\min\{|T_r|, |T_s|\}) \\
&\geq 1 + h(T_s) \qquad\qquad \text{gemäß I.A.}
\end{aligned}$$

□

Satz 208

In einer Union-Find-Datenstruktur mit gewichteter Vereinigung lässt sich die Union*-Operation in Zeit $O(1)$, die* Find*-Operation in Zeit $O(\log n)$ durchführen.*

Beweis:

Die Kosten der Find-Operation sind durch die Tiefe der Bäume beschränkt. □

Bei einer Operation Find(x) durchlaufen wir im zu x gehörigen Baum den Pfad von x zur Wurzel r. Nach Erreichen von r können wir alle Kanten auf diesem Pfad direkt auf r umlenken und damit zukünftige Find-Operationen beschleunigen.

Bemerkung: Eine andere Variante ist, für jeden Knoten y auf diesem Pfad die Kante von y zu seinem Vater auf seinen Großvater umzulenken. Dadurch wird der Pfad i.W. durch zwei Pfade der halben Länge ersetzt, die Umsetzung kann jedoch bereits beim Durchlaufen des Pfades von x aus zu r erfolgen.

Strategien wie die gerade beschriebenen zur Verkürzung von Find-Suchpfaden nennt man Pfadkompression (engl. path compression).

Satz 209

- *Union-Find mit Pfadkompression (und ohne gewichtete Vereinigung) erfordert Zeit* $\Theta(\log n)$ *pro Operation im worst-case.*
- *Union-Find mit gewichteter Vereinigung und mit Pfadkompression benötigt für eine (beliebige) Folge von* m Union- *und* Find-*Operationen Zeit* $O(m \log^* n)$.
- *Union-Find mit gewichteter Vereinigung und mit Pfadkompression benötigt für eine (beliebige) Folge von* m Union- *und* Find-*Operationen Zeit* $O(m \cdot \alpha(m,n))$. *Dabei ist* $\alpha(m,n)$ *eine Inverse der Ackermannfunktion, definiert durch*

$$\alpha(m,n) := \min\{i;\ A\left(i, \left\lfloor \frac{m}{n} \right\rfloor\right) \geq \operatorname{ld} n\}$$

Beweis:

Ohne Beweis. □

7. Graphenalgorithmen

7.1 Kürzeste Pfade

Gegeben: Digraph $G = (V, A)$, $|V| = n$, $|A| = m$, mit einer Längenfunktion $d : A \to \mathbb{Q}^+$, sowie ein Startknoten $s \in V$.
Gesucht: für alle $t \in V \setminus \{s\}$ die Entfernung (bzw. ein kürzester Pfad) von s nach t.

Im Folgenden bezeichne für einen Knoten $v \in V$ $\Gamma^+(v)$ die von v aus mittels einer Kante erreichbare Nachbarschaft von v:

$$\Gamma^+(v) = \{w \in V;\ (v, w) \in A\}$$

Der Algorithmus $\textsc{Dijkstra}(G, d, s)$:

$W := \{s\}$

for all $v \in V$ **do**

$$\rho[v] := \begin{cases} 0 & \text{falls } v = s \\ d(s,v) & \text{falls } v \in \Gamma^+(s) \\ \infty & \text{sonst} \end{cases}$$

$$\text{pred}[v] := \begin{cases} s & \text{falls } v \in \Gamma^+(s) \\ \text{nil} & \text{sonst} \end{cases}$$

while $W \neq V$ **do**

bestimme $x_0 \in V \setminus W$ so, dass $\rho[x_0] = \min\{\rho[v];\ v \in V \setminus W\}$

$W := W \cup \{x_0\}$

for all $v \in \Gamma^+(x_0) \cap (V \setminus W)$ **do**

if $\rho[v] > \rho[x_0] + d(x_0, v)$ **then**

$\rho[v] := \rho[x_0] + d(x_0, v)$; $\text{pred}[v] := x_0$

return ρ, pred

Satz 210

Der Algorithmus DIJKSTRA *bestimmt, für s und für alle $t \in V$, einen kürzesten Pfad bzw. die Länge eines solchen von s nach t.*

Bemerkung:

Wir nennen das entsprechende algorithmische Problem das single source shortest path Problem (sssp).

Beweis:

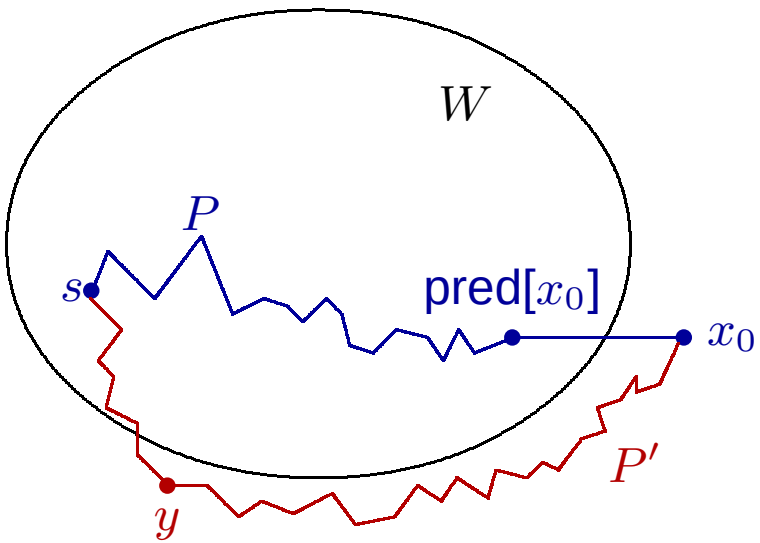
Wir zeigen folgende Invarianten:

1. $\forall v \in W$:
   $\rho[v] =$ Länge eines kürzesten Pfades von s nach v
2. $\forall v \in V \setminus W$:
   $\rho[v] =$ Länge eines kürzesten Pfades von s nach v, der als innere Knoten nur solche aus W enthält
3. die im Feld pred gespeicherten Verweise liefern jeweils die genannten Pfade

Induktionsanfang: Die Invarianten sind durch die im Algorithmus erfolgte Initialisierung gewährleistet.

Beweis:

Induktionsschritt: Betrachte den Schleifendurchlauf mit $W := W \cup \{x_0\}$.
Es ist gemäß I.A. $\rho[x_0]$ gleich der Länge eines kürzesten Pfades von s nach x_0, der als innere Knoten nur solche aus (dem alten) W enthält.



Angenommen, es gäbe einen kürzeren Pfad P'. Sei $y \neq x_0$ der erste Knoten auf P' außerhalb W. Dann gilt nach Wahl von x_0: $\rho[y] \geq \rho[x_0]$, also

$$\text{Länge}(P') \geq \rho[y] \geq \rho[x_0] = \text{Länge}(P)\,,$$

und damit Widerspruch!

Wir verwenden hier essentiell, dass $d \geq 0$. Die beiden anderen Invarianten sind damit klar.

□

Eine naive Implementierung von Dijkstra's Algorithmus (mit ρ als Array) ergibt eine Laufzeit $O(n^2)$.

Implementiert man dagegen ρ als Priority Queue (wobei jedes $v \in V$ mit dem Schlüssel $\rho[v]$ eingetragen wird), so ergibt sich

Satz 211

Für die Laufzeit des DIJKSTRA*-Algorithmus für das Single-Source-Shortest-Path-Problem gilt:*

	# Aufrufe	*Kosten pro Aufruf*	
		BinHeaps	*FibHeaps*
Insert	n	$O(\log n)$	$O(1)$
ExtractMin	$n-1$	$O(\log n)$	$O(\log n)$
DecreaseKey	$\leq m$	$O(\log n)$	$O(1)$
Insgesamt		$O((n+m)\log n)$	$O(m+n\log n)$

Gegeben: Digraph $G = (V, A)$, $V = \{v_1, \ldots, v_n\}$, $|A| = m$, mit einer Längenfunktion $d : A \to \mathbb{Q}^+$.
Gesucht: für alle $s, t \in V$ die Entfernung (bzw. ein kürzester Pfad) von s nach t.

Bemerkung:
Wir nennen das entsprechende algorithmische Problem das all pairs shortest path Problem (apsp).

Der Algorithmus FLOYD-WARSHALL(G, d):

for all $i, j \in \{1, \ldots, n\}$ **do**

$$D^0[i,j] := \begin{cases} 0 & \text{falls } i = j \\ d(v_i, v_j) & \text{falls } (v_i, v_j) \in A \\ \infty & \text{sonst} \end{cases}$$

for $k = 1$ **to** n **do**

$\forall i, j$: $D^k[i,j] := \min\{D^{k-1}[i,j], D^{k-1}[i,k] + D^{k-1}[k,j]\}$

return D^n

Satz 212

Der Floyd-Warshall-Algorithmus berechnet in Zeit $O(n^3)$ die kürzeste Entfernung für alle $s, t \in V$.

Beweis:

Es ist leicht zu sehen, dass $D^k[i,j]$ die Länge eines kürzesten Pfades von v_i nach v_j ist, der als *innere* Knoten nur solche $\in \{v_1, \ldots, v_k\}$ benützt (i und j selbst können $> k$ sein).

Damit ergibt sich die Korrektheit des Algorithmus durch Induktion.

Die Schranke für die Laufzeit ist offensichtlich. □

Bemerkungen

1. Der Floyd-Warshall-Algorithmus funktioniert auch mit negativen Kantenlängen, solange der Graph keine Kreise negativer Länge enthält (im letzteren Fall ist der kürzeste Abstand nicht überall wohldefiniert, sondern wird $-\infty$). Das Vorhandensein negativer Kreise kann an den Werten der Diagonale von D^n erkannt werden.
2. Man beachte die Ähnlichkeit des Algorithmus zum Algorithmus für die Konstruktion eines regulären Ausdrucks zu einem gegebenen endlichen Automaten.
3. Für ungerichtete Graphen mit trivialer Längenfunktion (also Kantenlänge $= 1$) liefert Breitensuche eine effizientere Methode!

7.2 Transitive Hülle

Gegeben: Digraph $G = (V, A)$, $|V| = n$, $|A| = m$.
Gesucht: die transitive Hülle $G^* = (V, A^*)$ von G, wobei gilt:

$$A^* = \{(v, w) \in V \times V;\ \text{es gibt in } G \text{ einen Pfad von } v \text{ nach } w\}$$

Bemerkungen

1. Für Digraphen kann der Floyd-Warshall-Algorithmus benutzt werden. Die Kante (v, w) ist in A^* gdw die Entfernung von v nach w endlich ist.
2. Für die Transitive Hülle gibt es effizientere Algorithmen, die in der Vorlesung EA besprochen werden.
3. In ungerichteten Graphen entspricht die Berechnung der Transitiven Hülle der Bestimmung der Zusammenhangskomponenten.