Vorlesung Algorithmentechnik

Marcus Krug

Lehrstuhl für Algorithmik I
Institut für Theoretische Informatik
Universität Karlsruhe (TH)

30.10.2008

Offline-Min Problem

Gegeben sei eine Menge $M = \emptyset$. Führe eine Folge Q von n Operationen vom Typ

$$\begin{array}{lll} \textsc{Insert}[i]: & M := M \cup \{i\} & \text{oder} \\ \textsc{Extract-Min}: & M := M \setminus \{\min M\} & \end{array}$$

aus, wobei i aus der Menge $\{1, \ldots, n\}$ ist. Eine Operation INSERT[i] trete für jedes i höchstens einmal in der Folge auf.

Aufgabe: Zu einer gegebenen Folge Q wollen wir alle i finden, die durch eine Operation EXTRACT-MIN entfernt werden und die entsprechende EXTRACT-MIN-Operation angeben.

Notation

- Schreibe die Folge Q als

$$Q_1 E Q_2 E \dots E Q_{k+1},$$

  wobei Q_j für $1 \leq j \leq k+1$ nur aus INSERT-Operationen besteht (möglicherweise $Q_j = \emptyset$);

- E stehe für eine EXTRACT-MIN-Operation.
- Die Anzahl der EXTRACT-MIN sei also k.

Datenstruktur

• Für den UNION-FIND-Algorithmus initialisieren wir (paarweise disjunkte) Mengen

$$M_j := \{i : \text{INSERT}[i] \text{ liegt in } Q_j\}$$

für $1 \leq j \leq k+1$.

• Erzeuge doppelt verkettete Listen der Werte j, für die eine Menge M_j existiert

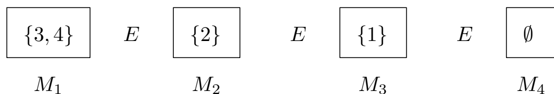
≫ Benutze dazu Arrays PRED (predecessor) und SUCC (successor)

≫ Zu Beginn sei $\text{PRED}[j] = j - 1$ für $2 \leq j \leq k+1$ und SUCC $[j] = j + 1$ für $1 \leq j \leq k$.

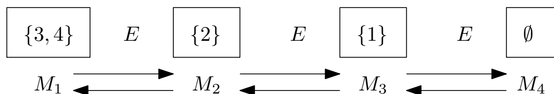
Datenstruktur

$\{3,4\}$ E $\{2\}$ E $\{1\}$ E $\emptyset$

Datenstruktur



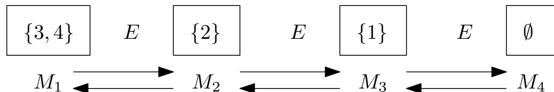
Datenstruktur



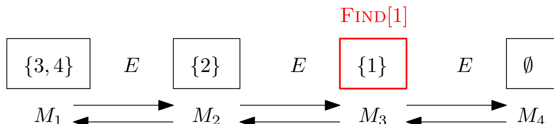
Algorithmus 1 : Offline-Min

```
1  für i = 1 bis n tue
2      j ← FIND[i]
3      wenn j ≤ k dann
4          Schreibe „i ist entfernt worden im j-ten EXTRACT-MIN“
5          UNION(j, SUCC[j], SUCC[j])
6          SUCC[PRED[j]] ← SUCC[j]
7          PRED[SUCC[j]] ← PRED[j]
```

Beispiel

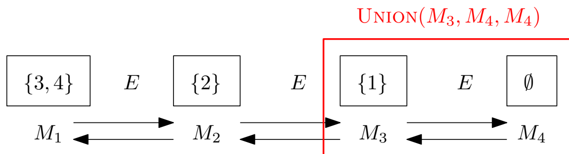


Beispiel



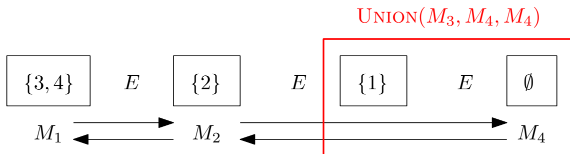
$\text{FIND}[1] = M_3,\ 3 \leq k = 3 \Rightarrow 3.\ \text{EXTRACT-MIN}$

Beispiel



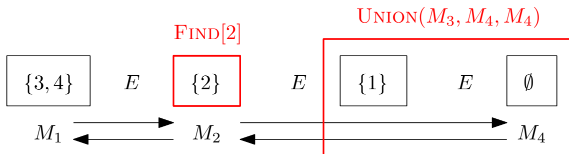
$\text{FIND}[1] = M_3$, $3 \leq k = 3 \Rightarrow 3.$ EXTRACT-MIN

Beispiel



$$\text{Find}[1] = M_3,\ 3 \leq k = 3 \Rightarrow 3.\ \text{Extract-Min}$$

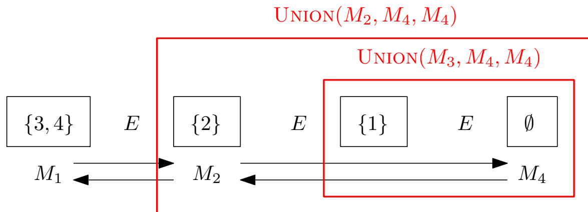
Beispiel



$\textsc{Find}[1] = M_3,\ 3 \leq k = 3 \Rightarrow 3.\ \textsc{Extract-Min}$

$\textsc{Find}[2] = M_2,\ 2 \leq k = 3 \Rightarrow 2.\ \textsc{Extract-Min}$

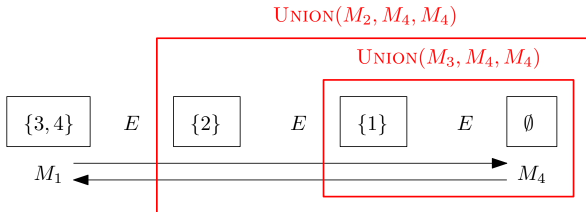
Beispiel



$\text{Find}[1] = M_3,\ 3 \leq k = 3 \Rightarrow 3.\ \text{Extract-Min}$

$\text{Find}[2] = M_2,\ 2 \leq k = 3 \Rightarrow 2.\ \text{Extract-Min}$

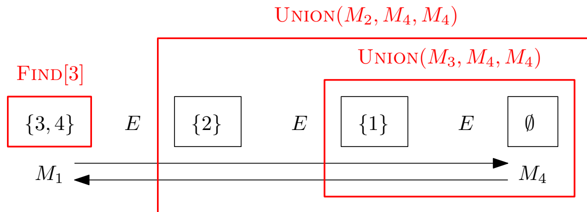
Beispiel



Find[1] $= M_3$, $3 \leq k = 3 \Rightarrow 3.$ Extract-Min

Find[2] $= M_2$, $2 \leq k = 3 \Rightarrow 2.$ Extract-Min

Beispiel

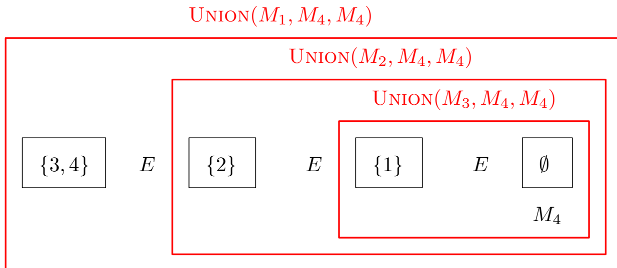


$\text{FIND}[1] = M_3,\ 3 \leq k = 3 \Rightarrow 3.\ \text{EXTRACT-MIN}$

$\text{FIND}[2] = M_2,\ 2 \leq k = 3 \Rightarrow 2.\ \text{EXTRACT-MIN}$

$\text{FIND}[3] = M_1,\ 1 \leq k = 3 \Rightarrow 1.\ \text{EXTRACT-MIN}$

Beispiel

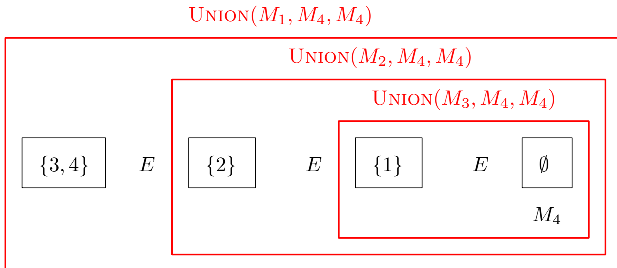


Find[1] $= M_3$, $3 \leq k = 3 \Rightarrow 3.$ Extract-Min

Find[2] $= M_2$, $2 \leq k = 3 \Rightarrow 2.$ Extract-Min

Find[3] $= M_1$, $1 \leq k = 3 \Rightarrow 1.$ Extract-Min

Beispiel



FIND$[1] = M_3$, $3 \leq k = 3 \Rightarrow 3.$ EXTRACT-MIN

FIND$[2] = M_2$, $2 \leq k = 3 \Rightarrow 2.$ EXTRACT-MIN

FIND$[3] = M_1$, $1 \leq k = 3 \Rightarrow 1.$ EXTRACT-MIN

FIND$[4] = M_4$, $4 \not\leq k = 3 \Rightarrow 4$ wurde nicht entfernt

Analyse

Bemerkung. Eine Folge von n Operationen vom Typ UNION und FIND, beginnend mit einer Menge disjunkter Teilmengen einer Menge mit $\mathcal{O}(n)$ Elementen, ist natürlich ebenfalls in $\mathcal{O}(n \cdot G(n))$ (bzw. $\mathcal{O}(n \cdot \alpha(n, n)))$ ausführbar.

$\Rightarrow T_{OM}(n) = \mathcal{O}(n\alpha(n, n))$

Bemerkung. OFFLINE-MIN ist sogar in $\mathcal{O}(n)$, da die UNION-FIND-Folge zu den Spezialfällen gehört, die in Linearzeit ausführbar sind.

Weitere Anwendung von Union-Find

Definition (Endlicher Automat)

Ein endlicher Automat $\mathcal{A}$ besteht aus

- einem endlichen Alphabet Σ
- einer endlichen Zustandsmenge Q
- einem Anfangszustand $q_0 \in Q$
- einer Menge von Endzuständen $F \subseteq Q$ und
- einer Zustandsübergangsfunktion $\delta : \; Q \times \Sigma \rightarrow Q$.

Σ^* bezeichne die Menge aller Wörter endlicher Länge (incl. ε).

Weitere Anwendung von Union-Find

Die Zustandsübergangsfunktion δ lässt sich erweitern zu einer Funktion

$$\begin{aligned} \delta : Q \times \Sigma^* &\longrightarrow Q \\ \delta(q, wa) &:= \delta(\delta(q, w), a) \\ \delta(q, \varepsilon) &:= q \end{aligned}$$

- Ein Automat $\mathcal{A}$ akzeptiert $w \in \Sigma^*$ gdw $\delta(q_0, w) \in F$.
- $L(\mathcal{A})$ ist die Sprache der Wörter, die von $\mathcal{A}$ akzeptiert werden.
- $\mathcal{A}_1$ und $\mathcal{A}_2$ heißen äquivalent ($\mathcal{A}_1 \equiv \mathcal{A}_2$), falls $L(\mathcal{A}_1) = L(\mathcal{A}_2)$.

Weitere Anwendung von Union-Find

Problem (Äquivalenzproblem endlicher Automaten)

Gegeben: endliche Automaten $\mathcal{A}_1, \mathcal{A}_2$

Frage: gilt $L(\mathcal{A}_1) = L(\mathcal{A}_2)$, d.h. sind $\mathcal{A}_1$ und $\mathcal{A}_2$ äquivalent?

Kann mit Union-Find effizient entschieden werden.

Äquivalenz endlicher Automaten

Seien $\mathcal{A}_1 := (Q_1, \Sigma, \delta_1, s_1, F_1)$ und $\mathcal{A}_2 := (Q_2, \Sigma, \delta_2, s_2, F_2)$

Eigenschaften

- $q_1 \in Q_1$ und $q_2 \in Q_2$ heißen äquivalent $(q_1 \equiv q_2)$ falls

$$\delta_1(q_1, w) \in F_1 \Leftrightarrow \delta_2(q_2, w) \in F_2 \quad \forall w \in \Sigma^*$$

- es gilt $q_1 \equiv q_2$ gdw

$$\delta_1(q_1, a) \equiv \delta_2(q_2, a) \quad \forall a \in \Sigma$$

- weiter gilt $q_1 \not\equiv q_2$ für alle $q_1 \in F_1,\ q_2 \in Q_2 \setminus F_2$
- $\mathcal{A}_1 \equiv \mathcal{A}_2$ gdw. $s_1 \equiv s_2$

Äquivalenz endlicher Automaten

Vorgehensweise

- initialisiere Zustände von $Q_1 \cup Q_2$ als einelementige Mengen
- Annahme $s_1 \equiv s_2$
- führe simultane Tiefensuche in $\mathcal{A}_1$ und $\mathcal{A}_2$ durch
- vereinige dabei Mengen "angeblich" äquivalenter Zustände

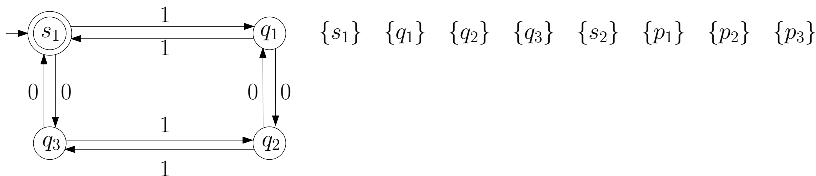
Korrektheit

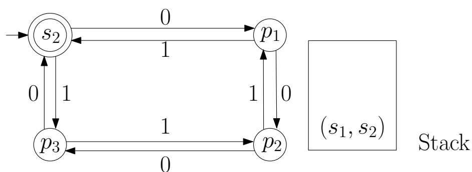
- wenn Annahme ($s_1 \equiv s_2$) stimmt, werden äquivalente Zustände zu Mengen zusammengefasst
- d.h. keine Menge enthält $q_1 \in F_1$ und $q_2 \in Q_2 \setminus F_2$
- andernfalls ist derartige Menge Beweis für Nicht-Äquivalenz

Algorithmus 2 : Äquivalenz endlicher Automaten

```
1  S ← ∅                                              /* Stack */
2  für q ∈ Q1 ∪ Q2 tue
3  │  MAKESET(q)
4  PUSH((s1, s2))
5  solange S ≠ ∅ tue
6  │  (q1, q2) ← POP(S)
7  │  wenn FIND[q1] ≠ FIND[q2] dann
8  │  │  UNION(FIND[q1], FIND[q2])
9  │  │  für a ∈ Σ tue
10 │  │  │  PUSH(δ1(q1, a), δ2(q2, a))
```

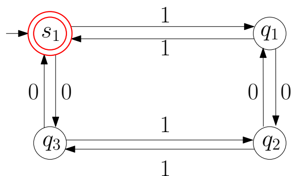
Beispiel





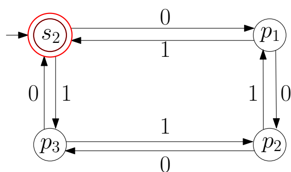
Stack

Beispiel



$\{s_1\}$ $\{q_1\}$ $\{q_2\}$ $\{q_3\}$ $\{s_2\}$ $\{p_1\}$ $\{p_2\}$ $\{p_3\}$

$\{s_1, s_2\}$ $\{q_1\}$ $\{q_2\}$ $\{q_3\}$ $\{p_1\}$ $\{p_2\}$ $\{p_3\}$

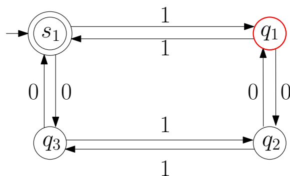


(q_1, p_3)

(q_3, p_1)

Stack

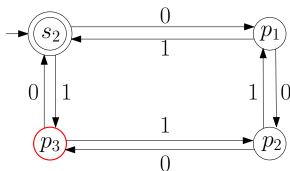
Beispiel



$\{s_1\}$ $\{q_1\}$ $\{q_2\}$ $\{q_3\}$ $\{s_2\}$ $\{p_1\}$ $\{p_2\}$ $\{p_3\}$

$\{s_1, s_2\}$ $\{q_1\}$ $\{q_2\}$ $\{q_3\}$ $\{p_1\}$ $\{p_2\}$ $\{p_3\}$

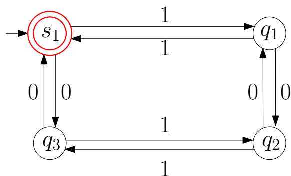
$\{s_1, s_2\}$ $\{q_1, p_3\}$ $\{q_2\}$ $\{q_3\}$ $\{p_1\}$ $\{p_2\}$



(s_1, p_2)
(q_2, s_2)
(q_3, p_1)

Stack

Beispiel

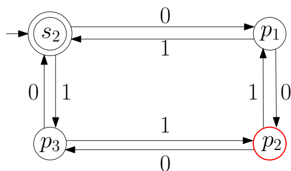


$\{s_1\}$ $\{q_1\}$ $\{q_2\}$ $\{q_3\}$ $\{s_2\}$ $\{p_1\}$ $\{p_2\}$ $\{p_3\}$

$\{s_1, s_2\}$ $\{q_1\}$ $\{q_2\}$ $\{q_3\}$ $\{p_1\}$ $\{p_2\}$ $\{p_3\}$

$\{s_1, s_2\}$ $\{q_1, p_3\}$ $\{q_2\}$ $\{q_3\}$ $\{p_1\}$ $\{p_2\}$

$\{s_1, s_2, p_2\}$ $\{q_1, p_3\}$ $\{q_2\}$ $\{q_3\}$ $\{p_1\}$



(q_2, s_2)
(q_3, p_1)

Stack

Analyse

Laufzeit

- ≫ $n := |Q_1| + |Q_2|$, $|\Sigma| = k$
- ≫ höchstens $n - 1$ Union-Operationen
- ≫ nach jeder Union-Operation werden k Paare auf den Stack gelegt
- ⇒ höchstens $\mathcal{O}(k(n-1)+1)$ Find-Operationen (Anzahl ist proportional zur Anzahl der Paare, die auf den Stack gelegt werden)
- ≫ falls k konstant ist, ist Anzahl der Union-Find-Operationen in $\mathcal{O}(n)$
- ⇒ Laufzeit in $\mathcal{O}(n\alpha(n,n))$

Grundbegriffe

Konvention

- $G = (V, E)$ Graph
- n Anzahl Knoten $(= |V|)$
- m Anzahl Kanten $(= |E|)$
- C Kreis
- T Baum
- F Wald

Spannbäume minimalen Gewichts

Problem (Minimaler Spannbaum)

Gegeben: zusammenhängender Graph $G = (V, E)$,
Gewichtsfunktion $c : E \longrightarrow \mathbb{R}$

Gesucht: spannender Baum $T = (V, E')$ mit minimalem Gewicht

$$c(T) := \sum_{e \in E'} c(e)$$

Schnitte in Graphen

Definition (Schnitt)

Sei $G = (V, E)$ Graph.

- Ein Schnitt ist eine Partition $(S, V \setminus S)$ der Knotenmenge.
- Die Kante $\{u, v\}$ kreuzt den Schnitt, falls
  - $u \in S$ und $v \in V \setminus S$
- alternativ: Schnitt als Menge von Kanten, die Partition von V induziert

Färbungsmethode von Tarjan

Vorgehensweise

- färbe Kanten iterativ rot und grün
- färbe Kanten grün, die zum Spannbaum gehören sollen
- grüne Kanten induzieren spannenden Wald

Färbungsmethode von Tarjan

Grüne Regel

- wähle Schnitt $(S, V \setminus S)$ in G, der nicht von grüner Kante gekreuzt wird
- wähle Kante e minimalen Gewichts, die den Schnitt kreuzt
- färbe e grün

Rote Regel

- wähle Kreis, der keine rote Kante enthält
- wähle Kante e maximalen Gewichts auf dem Kreis
- färbe e rot

Färbungsmethode von Tarjan

Algorithmus 3 : Färbungsmethode von Tarjan

Eingabe : Graph mit gewichteten Kanten

Ausgabe : Aufspannender Baum minimalen Gewichts in Form der grünen Kanten

1 **solange** *noch eine der beiden Regeln anwendbar* **tue**

2   Wende die grüne oder die rote Regel an

Färbungsinvariante

Färbungsinvariante
Es gibt einen aufspannenden Baum minimalen Gewichts, der alle grünen Kanten und keine rote Kante enthält.

Satz 2.6 (Färbungsinvariante)

Sei G zusammenhängender Graph, auf den die Färbungsmethode angewendet wird. Dann bleibt die Färbungsinvariante erhalten.

Beweis: Induktion über Anzahl der Färbungsschritte.

Korrektheit

Satz 2.7 (Färbungsmethode)

Die Färbungsmethode färbt alle Kanten eines zusammenhängenden Graphen rot oder grün.

2 Anwendungen

- Algorithmus von Kruskal
- Algorithmus von Prim

Algorithmus von Kruskal

- Beispiel für Färbungsmethode
- Eingabe: Graph mit gewichteten Kanten
- Sortiere die Kanten nach ihrem Gewicht in nicht–absteigender Reihenfolge
- Durchlaufe die sortierten Kanten der Reihe nach und wende folgenden Färbungsschritt an:
  - Beide Endknoten der Kante liegen in unterschiedlichen grünen Baum, färbe sie grün
  - sonst färbe sie rot
  - benutze Union-Find für die Verwaltung der Bäume
- Ausgabe: Aufspannender Baum minimalen Gewichts in Form der grünen Kanten

Algorithmus 4 : Algorithmus von Kruskal mit Union Find

Eingabe : Graph $G = (V, E)$ mit gewichteten Kanten

Ausgabe : Aufspannender Baum minimalen Gewichts in Form der grünen Kanten

1	$\textsc{Grün} \leftarrow \emptyset$	
2	$(e_1, \ldots, e_m) \leftarrow sort(E, \leq)$	$\mathcal{O}(m \log m)$
3	**für** $i = 1$ **bis** n **tue**	$\mathcal{O}(n)$
4	$\quad \textsc{Makeset}(i)$	$\mathcal{O}(1)$
5	**für** $k = 1$ **bis** m **tue**	$\mathcal{O}(m \log n)$
6	$\quad \{i, j\} \leftarrow e_k$	
7	$\quad$ **wenn** $\textsc{Find}(i) \neq \textsc{Find}(j)$ **dann**	$\mathcal{O}(\log n)$
8	$\quad\quad \textsc{Union}(\textsc{Find}(i), \textsc{Find}(j))$	$\mathcal{O}(1)$
9	$\quad\quad \textsc{Grün} \leftarrow \textsc{Grün} \cup \{\{i, j\}\}$	

Beispiel: Tafel

Algorithmus von Kruskal

Genauere Analyse:

- Sortieren der m Kanten
  $\rightsquigarrow \mathcal{O}(m \log m)$
- Folge von maximal $n + 3m = \mathcal{O}(m)$ Union, Find und Makeset Operationen mit n Elementen
  $\rightsquigarrow \mathcal{O}(m \cdot \alpha(m, n))$

$\rightsquigarrow$ bei vorsortierter Kantenliste “fast linear”

Algorithmus von Prim

Motivation

- Algorithmus von Kruskal wird durch Sortieren dominiert
- Algorithmus von Prim ist unter bestimmten Voraussetzungen besser

Algorithmus von Prim

Vorgehensweise

- betrachte beliebigen Startknoten als (grünen) Baum
- wähle ungefärbte Kante minimalen Gewichts, die genau einen Endknoten im grünen Baum hat, sog. begrenzende Kante
- färbe diese Kante grün

Bemerkung zur Korrektheit

Menge der Kanten mit genau einem Endknoten im grünen Baum bilden einen Schnitt $\Rightarrow$ Algorithmus erhält Färbungsinvariante

Implementation

d-Heap Datenstruktur

- naheliegende Verallgemeinerung eines Binary Heaps (2-Heap)
- innere Knoten haben Grad d (statt 2)
- Algorithmen analog zu Binary Heaps

- $\textsc{Insert}(H, x)$ in $\mathcal{O}(\log_d n)$
- $\textsc{DeleteMin}(H)$ in $\mathcal{O}(d \log_d n)$
  Beachte: hier min-Heap
- $\textsc{DecreaseKey}(H, x, k)$ in $\mathcal{O}(\log_d n)$
  Beachte: k darf nicht größer als aktueller Schlüsselwert sein

Implementation

Datenstrukturen

- $\text{KEY}[v]$: Array, das Kosten zur Anbindung von v enthält
  - ∞ Knoten wurde noch nicht betrachtet
  - $-\infty$ Knoten gehört zum grünen Baum
- $\text{GRÜN}[v]$: Array, das für jeden Knoten v die Kante e enthält, über die v an den günen Baum angebunden wurde
- H Priority Queue (d-Heap) für die Knoten, gewichtet mit KEY

```
1  für v ∈ V tue
2     KEY[v] ← ∞
3  v ← s
4  solange v ist definiert tue
5     KEY[v] ← −∞
6     für Kanten {v, w} inzident zu v tue
7        wenn KEY[w] = ∞ dann
8           KEY[w] ← c({v, w})
9           GRÜN[w] ← {v, w}
10          INSERT(H, w)
11       sonst
12          wenn c({v, w}) <KEY[w] dann
13             KEY[w] ← c({v, w})
14             GRÜN[w] ← {v, w}
15             DECREASEKEY(H, w, c({v, w}))
16    v ← DELETEMIN(H)
```

Analyse

Laufzeit

≫ Beobachtung: KEY ist monoton fallend

⇒ jeder Knoten wird genau einmal in H eingefügt und einmal gelöscht, d.h. verursacht höchstens ein INSERT und ein DELETEMIN

⇝ $\mathcal{O}(n \cdot (\log_d n + d \log_d n)) = \mathcal{O}(nd \log_d n)$

≫ jede Kante verursacht höchstens ein DECREASEKEY

⇝ $\mathcal{O}(m \log_d n)$

≫ Gesamtaufwand $T_{Prim} \in \mathcal{O}((nd + m) \log_d n)$

Analyse

Laufzeit (Fortsetzung)

- ≫ Gesamtaufwand $T_{Prim} \in \mathcal{O}((nd + m) \log_d n)$
- ≫ wähle $d := \lceil 2 + m/n \rceil$
- ⇒ $T_{Prim} \in \mathcal{O}(m \log_{2+m/n} n)$
- ≫ für $m \in \Omega(n^{1+\varepsilon})$ gilt
- ≫ $T_{Prim} \in \mathcal{O}(m/\varepsilon)$

Fazit

- ≫ Kantenliste nicht sortiert
- ⇒ Prim ist schneller für dichte Graphen, d.h. für Graphen mit $m \in \Omega(n^{1+\varepsilon}),\ \varepsilon > 1$