SS 2005

Einführung in die Informatik IV

Ernst W. Mayr

Fakultät für Informatik
TU München

http://www14.in.tum.de/lehre/2005SS/info4/index.html.de

11. Juli 2005

Satz 211

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. □

Satz 211

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).

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).

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 212

- *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*

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

Beweis:
Ohne Beweis. □

Satz 212

- *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*

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

Beweis:

Ohne Beweis. □

6. Graphenalgorithmen

6.1 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:

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

6. Graphenalgorithmen

6.1 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

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

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

Der Algorithmus 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 213

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).

Satz 213

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

Beweis:

Wir zeigen folgende Invarianten:

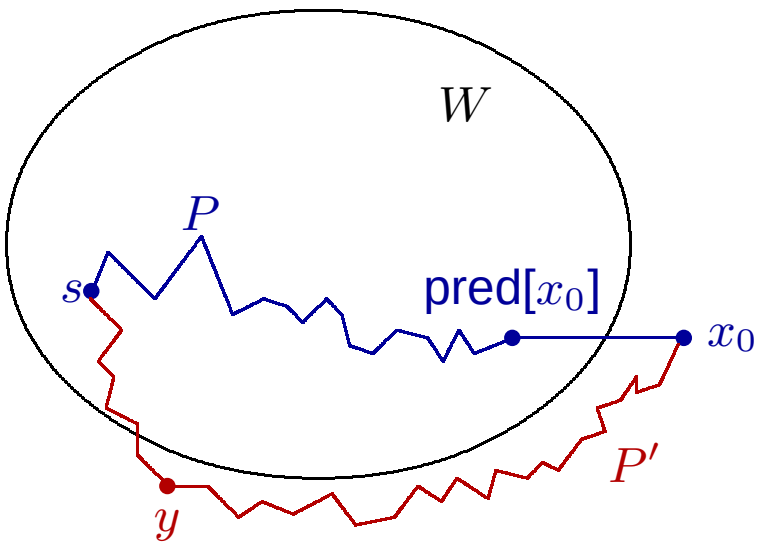
1. $\forall v \in W$:
   $\rho[v] =$ Länge eines kürzesten Pfades von s nach v
2. $\forall v \in V \setminus W$:
   $\rho[v] =$ Länge eines kürzesten Pfades von s nach v, der als innere Knoten nur solche aus W enthält
3. die im Feld pred gespeicherten Verweise liefern jeweils die genannten Pfade

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

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

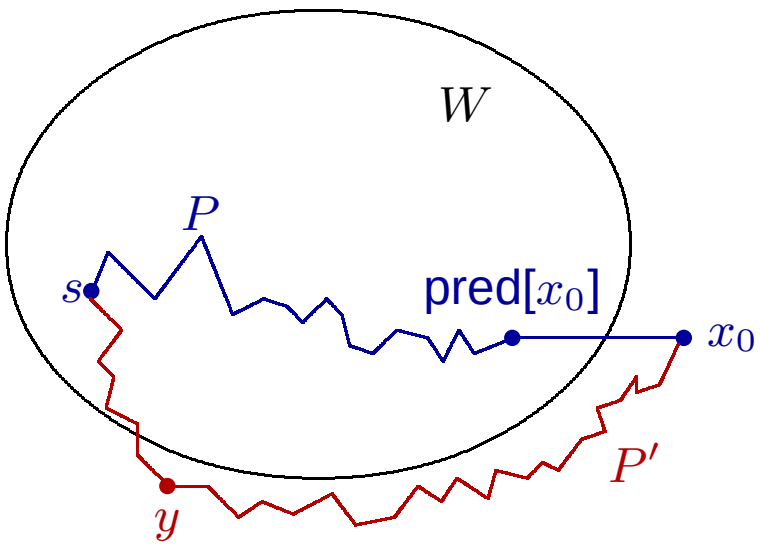
und damit Widerspruch!

□

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

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

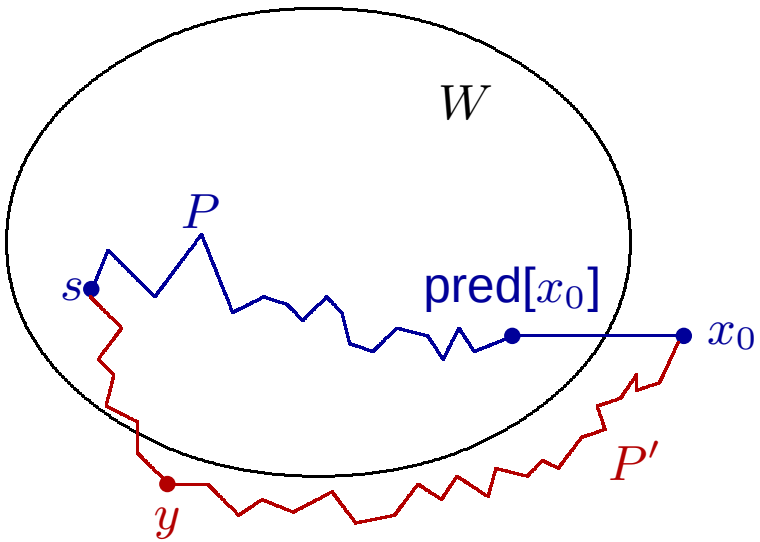
und damit Widerspruch!

Wir verwenden hier essentiell, dass $d \geq 0$.

□

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!

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 214

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

	Anzahl 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		$O((n+m)\log n)$	$O(m+n\log n)$

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 214

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

	Anzahl 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).

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 215

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

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 215

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

Der Algorithmus FLOYD-WARSHALL(G, d):

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

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

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

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

return D^n

Satz 215

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

Der Algorithmus FLOYD-WARSHALL(G, d):

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

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

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

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

return D^n

Satz 215

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

Der Algorithmus FLOYD-WARSHALL(G, d):

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

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

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

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

return D^n

Satz 215

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. □

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. □

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!

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!

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!

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!

6.2 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.

6.2 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.

6.2 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.

6.2 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.

6.2 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.

6.3 Minimale Spannbäume

Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph mit einer Gewichtsfunktion $w : E \to \mathbb{Q}^+$. Das Gewicht eines Teilgraphen $G' = (V', E')$ von G ist dann $w(G') = \sum_{e \in E'} w(e)$.

Definition 216

Ein minimaler Spannbaum von $G = (V, E)$ ist ein Spannbaum von G (also ein *Baum* (V, E') mit $E' \subseteq E$) mit minimalem Gewicht unter allen Spannbäumen von G.

Anwendungen: Verdrahtungs- und Routingprobleme, Zuverlässigkeit von Netzwerken

6.3 Minimale Spannbäume

Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph mit einer Gewichtsfunktion $w : E \to \mathbb{Q}^+$. Das Gewicht eines Teilgraphen $G' = (V', E')$ von G ist dann $w(G') = \sum_{e \in E'} w(e)$.

Definition 216

Ein minimaler Spannbaum von $G = (V, E)$ ist ein Spannbaum von G (also ein *Baum* (V, E') mit $E' \subseteq E$) mit minimalem Gewicht unter allen Spannbäumen von G.

Anwendungen: Verdrahtungs- und Routingprobleme, Zuverlässigkeit von Netzwerken

6.3 Minimale Spannbäume

Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph mit einer Gewichtsfunktion $w : E \to \mathbb{Q}^+$. Das Gewicht eines Teilgraphen $G' = (V', E')$ von G ist dann $w(G') = \sum_{e \in E'} w(e)$.

Definition 216

Ein minimaler Spannbaum von $G = (V, E)$ ist ein Spannbaum von G (also ein *Baum* (V, E') mit $E' \subseteq E$) mit minimalem Gewicht unter allen Spannbäumen von G.

Anwendungen: Verdrahtungs- und Routingprobleme, Zuverlässigkeit von Netzwerken

Lemma 217 (rote Regel)

Sei $G = (V, E)$, $e \in E$ schwerste Kante auf einem Kreis C in G. Dann gibt es einen minimalen Spannbaum von G, der e nicht enthält (d.h. dann ist jeder minimale Spannbaum von $G - e$ auch minimaler Spannbaum von G). Ist e die einzige schwerste Kante auf einem Kreis C in G, dann ist e in keinem minimalen Spannbaum von G enthalten.

Beweis:

Sei e schwerste Kante in C, und sei M ein minimaler Spannbaum von G, der e enthält. Durch Entfernen von e zerfällt M in zwei Teilbäume, die, da C ein Kreis ist, durch eine geeignete Kante $f \neq e$ aus C wieder verbunden werden können. Der dadurch entstehende Spannbaum M' von G enthält e nicht und hat das Gewicht

$$w(M') = w(M) - w(e) + w(f)\,.$$

Falls e einzige schwerste Kante in C ist, wäre $w(M') < w(M)$ im Widerspruch zur Tatsache, dass M minimaler Spannbaum von G ist. □

Beweis:

Sei e schwerste Kante in C, und sei M ein minimaler Spannbaum von G, der e enthält. Durch Entfernen von e zerfällt M in zwei Teilbäume, die, da C ein Kreis ist, durch eine geeignete Kante $f \neq e$ aus C wieder verbunden werden können. Der dadurch entstehende Spannbaum M' von G enthält e nicht und hat das Gewicht

$$w(M') = w(M) - w(e) + w(f)\,.$$

Falls e einzige schwerste Kante in C ist, wäre $w(M') < w(M)$ im Widerspruch zur Tatsache, dass M minimaler Spannbaum von G ist. □

Der Algorithmus $\textsc{Kruskal}(V, E, w)$:

sortiere E aufsteigend: $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$
initialisiere Union-Find-Struktur für V
$T := \emptyset$
for $i = 1$ **to** m **do**
  bestimme die Endpunkte v und w von e_i
  if Find(v) $\neq$ Find(w) **then**
    $T := T \cup \{e_i\}$
    Union(Find(v),Find(w))
return T

Der Algorithmus $\textsc{Kruskal}(V, E, w)$:

sortiere E aufsteigend: $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$
initialisiere Union-Find-Struktur für V
$T := \emptyset$
for $i = 1$ **to** m **do**
  bestimme die Endpunkte v und w von e_i
  if Find$(v) \neq$ Find(w) **then**
    $T := T \cup \{e_i\}$
    Union(Find(v),Find(w))
return T

Der Algorithmus $\textsc{Kruskal}(V, E, w)$:

sortiere E aufsteigend: $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$
initialisiere Union-Find-Struktur für V
$T := \emptyset$
for $i = 1$ **to** m **do**
  bestimme die Endpunkte v und w von e_i
  if Find$(v) \neq$ Find(w) **then**
    $T := T \cup \{e_i\}$
    Union(Find(v),Find(w))
return T

Satz 218

Kruskal's Algorithmus bestimmt einen minimalen Spannbaum des zusammenhängenden Graphen $G = (V, E)$ in Zeit $O(m \log n)$.

Beweis:

Find$(v) =$ Find(w) gdw die Kante (v, w) schwerste Kante auf einem Kreis in G ist. Die Korrektheit des Algorithmus folgt damit aus dem vorigen Lemma.

Die Laufzeit für das Sortieren beträgt $O(m \log n)$ ($m \geq n - 1$, da G zusammenhängend). Bei geeigneter Implementierung (gewichtete Vereinigung) ist die Zeitkomplexität jeder Find-Operation $O(\log n)$. □

Satz 218

Kruskal's Algorithmus bestimmt einen minimalen Spannbaum des zusammenhängenden Graphen $G = (V, E)$ in Zeit $O(m \log n)$.

Beweis:

Find$(v) =$ Find(w) gdw die Kante (v, w) schwerste Kante auf einem Kreis in G ist. Die Korrektheit des Algorithmus folgt damit aus dem vorigen Lemma.

Die Laufzeit für das Sortieren beträgt $O(m \log n)$ ($m \geq n - 1$, da G zusammenhängend). Bei geeigneter Implementierung (gewichtete Vereinigung) ist die Zeitkomplexität jeder Find-Operation $O(\log n)$. □

Satz 218

Kruskal's Algorithmus bestimmt einen minimalen Spannbaum des zusammenhängenden Graphen $G = (V, E)$ in Zeit $O(m \log n)$.

Beweis:

Find$(v) =$ Find(w) gdw die Kante (v, w) schwerste Kante auf einem Kreis in G ist. Die Korrektheit des Algorithmus folgt damit aus dem vorigen Lemma.

Die Laufzeit für das Sortieren beträgt $O(m \log n)$ ($m \geq n - 1$, da G zusammenhängend). Bei geeigneter Implementierung (gewichtete Vereinigung) ist die Zeitkomplexität jeder Find-Operation $O(\log n)$. □