

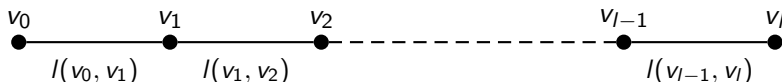
Teil 5: Algorithmen auf Graphen

Kürzeste Wege

Kürzeste Wege

In vielen Anwendungen muss man die kürzeste Verbindung zwischen zwei Knoten eines Graphen berechnen. Dabei wird die Länge eines Weges in einem gerichteten, gewichteten Graphen $G = (V, E; l)$ als Summe der Längen seiner Kanten definiert, d.h. Für einen Weg $\omega = (v_0, \dots, v_k)$ gilt

$$l(\omega) = l(v_0, \dots, v_k) := \sum_{i=1}^k l(v_{i-1}, v_i).$$



Bemerkung

Sei $\omega(u, w_1, \dots, w_{l-1}, v)$ ein kürzester Weg von u nach v . Dann ist für jedes w_i der Weg (u, w_1, \dots, w_i) ein kürzester Weg.

Kürzeste Wege

Definition (Distanz)

Sei $G = (V, E; l)$ ein gerichteter, gewichteter Graph. Die **Distanz** $d(u, v)$ zwischen zwei Knoten $u, v \in V$ ist das Minimum aller Längen von Wegen von u nach v , falls ein solcher Weg existiert und ∞ sonst. D.h.

$$d(u, v) := \min(\{l(\omega) \mid \omega \text{ ist ein Weg von } u \text{ nach } v\} \cup \{\infty\})$$

Problem (Shortest-Path)

Eingabe: Ein gerichteter, gewichteter Graph $G = (V, E; l)$ und zwei Knoten $u, v \in V$.

Berechne die Distanz $d(u, v)$.

Variationen des Problems

Wir werden zwei Varianten des Problems behandeln

Problem (All-Pair-Shortest-Paths (APSP))

Eingabe: Ein gerichteter, gewichteter Graph $G = (V, E; l)$.

Berechne die Distanzen $d(u, v)$ für alle Paare (u, v) von Knoten.

Problem (Single-Source-Shortest-Paths (SSSP))

Eingabe: Ein gerichteter, gewichteter Graph $G = (V, E; l)$ und ein Knoten $s \in V$.

Berechne alle Distanzen $d(s, v)$ für alle Knoten $v \in V$.

Ungewichtete Graphen

Auf ungewichteten Graphen können wir SSSP durch eine einfache Breitensuche lösen. Dazu führen wir **Distanzlabel** $d(v)$ für jeden Knoten mit.

Zu Beginn setzen wir $d(s) = 0$.

Anschließend wird immer, wenn ein Knoten v vom Knoten u aus zur Warteschlange hinzugefügt wird, sein Distanzlabel auf $d(v) = d(u) + 1$ gesetzt.

Satz

Auf einem ungewichteten, gerichteten Graphen $G = (V, E)$ läßt sich SSSP für jeden Knoten $s \in V$ mittels Breitensuche in Zeit $O(|V| + |E|)$ lösen.

Negative Längen und Kreise negativer Länge

In seiner vollen Allgemeinheit, d.h. $l(e) \in \mathbb{R}$ oder $l(e) \in \mathbb{Z}$ ist Shortest Paths **NP-vollständig**.

Die Ursache dafür ist die mögliche Existenz von **negativen Kreisen**, d.h. geschlossenen Wegen mit einer negativen Gesamtlänge. Sind keine solchen Kreise vorhanden, können wir das Problem in polynomieller Laufzeit lösen.

Aus diesem Grund beschränken wir uns in der Regel auf nicht-negative Kantengewichte, d.h. für jede Kante e gilt $l(e) \geq 0$.

Negative Längen und Kreise negativer Länge

Lemma

Besitzt $G = (V, E; l)$ keine Kreise negativer Länge, so existiert für jedes Paar u, v von Knoten ein einfacher Weg ω mit $l(\omega) = d(u, v)$.

Beweis.

Sei $\omega = (v_0, v_1, \dots, v_{k-1}, v_k)$ ein kürzester Weg von v_0 nach v_k , der nicht einfach ist. Sei v_i der erste Knoten auf diesem Weg, der mehr als einmal durchlaufen wird. D.h. es existiert ein j mit $i < j \leq k$ und $v_i = v_j$.

Damit ist $\omega' := (v_0, \dots, v_i, v_{j+1}, \dots, v_l)$ ein Weg mit weniger Knoten und es gilt

$$l(\omega) = l(\omega') + l(v_i, \dots, v_j) \geq l(\omega') \geq l(\omega).$$

Damit ist auch ω' ein kürzester Weg von v_0 nach v_l .

Is ω' einfach, sind wir fertig. Ansonsten fahren wir mit dem Entfernen von Kreisen fort. □

Teil 5: Algorithmen auf Graphen
Kürzeste Wege

APSP: Der Algorithmus von Floyd-Warshall

Die Grundidee

Wir gehen davon aus, dass der betrachtete Graph $G = (V, E; l)$ keine Kreise negativer Länge enthält.

Der APSP-Algorithmus von Floyd-Warshall ist dem Algorithmus von Warshall zur Berechnung der transitiven Hülle sehr ähnlich.

Wir werden Matrizen $A^{(k)} = (a_{ij}^{(k)})$ berechnen, so dass $a_{ij}^{(k)}$ die Länge eines kürzesten Weges von i nach j durch Zwischenknoten aus $\{1, \dots, k\}$ ist.

Wenn kein solcher Weg existiert ist $a_{ij}^{(k)}$ gleich ∞ .

Wie oben gezeigt, können wir davon ausgehen, dass jeder von uns betrachtete kürzeste Weg ein einfacher Weg ist.

Die Grundidee

Ein kürzester Weg ω von i nach j durch Knoten aus $\{1, \dots, k\}$ hat eine der beiden folgenden Formen:

- ω läuft nicht durch k , d.h. $l(\omega) = a_{ij}^{(k-1)}$,
- oder ω läuft durch k .

Da wir von einfachen Wegen ausgehen, besteht ω im zweiten Fall aus zwei Teilwegen, einem von i nach k und einem von k nach j , wobei beide Teilwege nicht durch k laufen.

Da ω ein kürzester Weg ist, müssen auch beide Teile kürzeste Wege sein.

Beide Fälle kombiniert ergeben somit:

$$a_{ij}^{(k)} = \min \left(a_{ij}^{(k-1)}, a_{ik}^{(k-1)} + a_{kj}^{(k-1)} \right)$$

(Detaillierter Beweis als Übungsaufgabe)

Algorithmus (Algorithmus von Floyd-Warshall)

Eingabe: Ein gerichteter, gewichteter Graph $G = (V, E; l)$ ohne Kreise negativer Länge

Ausgabe: Die Distanzmatrix $D = (d(i, j))$ von G

für $i := 1, \dots, n$ **tue**

für $j := 1, \dots, n$ **tue**

wenn $(i, j) \in E$ **dann** $D[i, j] := l(i, j)$

sonst wenn $i = j$ **dann** $D[i, j] := 0$

sonst $D[i, j] := \infty$

Ende

Ende

für $k := 1, \dots, n$ **tue**

für $i := 1, \dots, n$ **tue**

für $j := 1, \dots, n$ **tue**

$D[i, j] := \min(D[i, j], D[i, k] + D[k, j])$

Ende

Ende

Ende

Laufzeit und Korrektheit

Die Korrektheit des Algorithmus kann analog zu der des Algorithmus von Warshall für die transitive Hülle gezeigt werden.

Da die drei geschachtelten Schleifen vollständig ausgeführt werden und der Vergleich sowie die Aktualisierung des jeweiligen Matrixeintrages konstante Zeit benötigen, ergibt sich die Gesamtlaufzeit als $O(|V|^3)$.

Satz

Der Algorithmus von Floyd-Warshall berechnet die Distanzen zwischen allen Knotenpaaren eines gerichteten, gewichteten Graphen $G = (V, E; l)$ ohne Kreise negativer Länge in Zeit $O(|V|^3)$.

Detektion von Kreisen negativer Länge

Bislang sind wir davon ausgegangen, dass wir vor dem Ausführen des Algorithmus von Floyd/Warshall wissen müssen, ob der betrachtete Graph G Kreise negativer Länge enthält. Tatsächlich können wir ihre Existenz auch im Nachhinein feststellen.

Wir nehmen an, dass D die Matrix der vom Algorithmus von Floyd/Warshall berechneten „Distanzen“ ist.

Hat G keinen Kreis negativer Länge, so ist $D[i, j] = d(i, j)$, wie bereits gezeigt.

Lemma

Es gilt genau dann $D[i, j] \leq D[i, k] + l(k, j)$ für alle $i, j \in V$ und $(k, j) \in E$, wenn G keinen Kreis negativer Länge enthält.

Beweis (Detektion von Kreisen negativer Länge)

Wenn G keinen Kreis negativer Länge enthält, dann ist D , wie oben gezeigt, die Distanzmatrix von G . Damit gilt

$$D[i, j] \leq D[i, k] + I(k, j)$$

für alle Knoten $i, j \in V$ und alle Kanten $(k, j) \in E$, denn die rechte Seite ist die Länge eines Weges von i nach j über die Kante (k, j) .

Nehmen wir nun an, dass $\omega = (v_0, \dots, v_k = v_0)$ ein Kreis negativer Länge ist und dass die Ungleichung für alle Knoten i, j und alle Kanten $(k, j) \in E$ gilt.

Dann gilt wegen $v_0 = v_k$ und $I(\omega) < 0$

$$\begin{aligned} \sum_{i=1}^k D[v_0, v_i] &\leq \sum_{i=1}^k (D[v_0, v_{i-1}] + I(v_{i-1}, v_i)) \\ &= \sum_{i=1}^k D[v_0, v_i] + I(\omega) < \sum_{i=1}^k D[v_0, v_i] \end{aligned}$$

Dies ist aber offensichtlich ein Widerspruch. □

Detektion von Kreisen negativer Länge

Damit ergibt sich eine Möglichkeit nach dem Abschluß des Algorithmus von Floyd/Warshall zu prüfen, ob Kreise negativer Länge vorliegen.

Algorithmus (Detektion von Kreisen negativer Länge)

Eingabe: *Gerichteter, gewichteter Graph $G(V, E; l)$ und „Distanzmatrix“ D .*

Ausgabe: *True wenn G Kreise negativer Länge enthält, False sonst.*

für alle $u \in V$ tue

für alle $(w, v) \in E$ tue

wenn $D[u, v] > D[u, w] + l(w, v)$ dann Gebe True zurück

Ende

Ende

Gebe False zurück

Teil 5: Algorithmen auf Graphen
Kürzeste Wege

SSSP: Der Algorithmus von Dijkstra

Die Grundidee

$G = (V, E; l)$ sei ein gerichteter Graph mit nicht-negativen Kantengewichten und $s \in V$ sei ein ausgezeichnete Knoten von G . Wir wollen im Folgenden alle Distanzen $d(s, v)$ für $v \in V$ berechnen.

Der Algorithmus von Dijkstra verwaltet eine Menge $S \subseteq V$ von Knoten v , für die die Distanz $d(s, v)$ bereits bekannt ist. Diese Menge wird, beginnend mit $S = \{s\}$, sukzessive erweitert.

Um zu entscheiden, welcher Knoten als nächstes zu S hinzugefügt wird, wird ein **Distanzlabel** $d(v)$ für jeden Knoten $v \in V$ mitgeführt, so dass

- $d(v) \geq d(s, v)$ für alle $v \in V$,
- $d(v) = d(s, v)$ falls $v \in S$.

Die „Distanzen“

Genauer gesagt wird dafür gesorgt, dass für die „Distanzen“ $d(v)$ stets folgendes gilt:

$d(v)$ ist die Länge eines kürzesten Weges (s, u_1, \dots, u_k, v) von s nach v mit $s, u_1, \dots, u_k \in S$.

Wir sagen auch $d(v)$ ist die Länge eines kürzesten Weges von s durch S nach v .

Damit gilt offensichtlich

$$d(v) = \min \{d(u) + l(u, v) \mid u \in S, (u, v) \in E\}$$

Der Algorithmus von Dijkstra erweitert nun die Menge S immer um einen Knoten v mit kleinstem Distanzlabel $d(v)$ und passt anschließend die Label der Knoten in $V \setminus S$ entsprechend der oben stehenden Formel an.

Algorithmus (Abstrakter Dijkstra)

Eingabe: Ein gerichteter, gewichteter Graph $G = (V, E; l)$ mit nicht-negativen Kantengewichten und ein Knoten $s \in V$.

Ausgabe: Ein Feld $d[1 \dots V]$ mit $d(v) = d(s, v)$.

$S := \{s\}$

$d(s) := 0$

für alle $v \in V \setminus \{s\}$ **tue** $d(v) := \infty$

solange $S \neq V$ **tue**

 Wähle einen Knoten $v \in V \setminus S$ mit minimalem $d(v)$

$S := S \cup \{v\}$

für alle $w \in V \setminus S$ und $(v, w) \in E$ **tue**

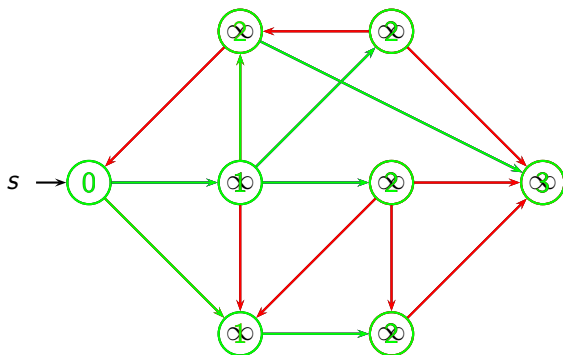
$d(w) := \min \{d(w), d(v) + l(v, w)\}$

// Relaxation

Ende

Ende

Der Algorithmus von Dijkstra am Beispiel



Lemma

Zu jedem Zeitpunkt des Algorithmus von Dijkstra ist $d(v)$ die Länge eines Weges von s nach v oder ∞ , d.h. $d(s, v) \leq d(v)$.

Beweis.

Der Beweis erfolgt per Induktion über die Anzahl der Knoten in S .

Für $|S| = 0$ gilt die Behauptung offensichtlich.

Sei nun $|S| \geq 0$ vor dem Hinzufügen des Knotens v .

Gilt beim Hinzufügen $d(v) = \infty$, so muss $d(w) = \infty$ für alle $w \in V \setminus S$. Damit werden keine Label verändert, so dass die Behauptung auch nach den Korrekturen noch gilt.

Gilt beim Hinzufügen $d(v) < \infty$, dann existiert ein Weg der Länge $d(v)$ von s nach v . Wird nun das Label eines Knotens w im Folgenden verändert, setzen wir $d(w) = d(v) + l(v, w)$ für eine existierende Kante (v, w) . Damit entspricht $d(w)$ anschließend der Länge des Weges von s nach v , verlängert um die Kante (v, w) . □

Satz (Korrektheit des Algorithmus von Dijkstra)

Für jeden Knoten $v \in V$ gilt beim Hinzufügen zu S stets $d(v) = d(s, v)$.

Beweis

Wir führen eine Induktion über die Anzahl $|S|$ der Elemente in S vor dem Hinzufügen durch.

Für $|S| = 0$ wird s mit $d(s) = 0 = d(s, s)$ als nächstes hinzugefügt, so dass die Behauptung trivialerweise gilt.

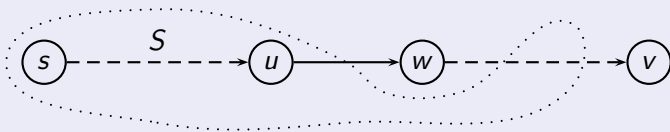
Sei nun $|S| > 0$. Da für jeden Knoten $u \in S$ beim Hinzufügen zu S $d(u) = d(s, u)$ galt, und $d(u)$ später nicht mehr verändert wird, gilt die Gleichung immer noch.

Außerdem stellen wir fest, dass durch die Korrektur der Label, diese höchstens sinken.

Beweis (Korrektheit von Dijkstra - Fortsetzung)

Sei nun v ein Knoten aus $V \setminus S$, so dass $d(v) \leq d(w)$ für alle $w \in V \setminus S$. Nehmen wir an, dass $d(v) > d(s, v)$ gilt. Dann gibt es einen Weg $\omega = (s, u_1, \dots, u_k, v)$ von s nach v mit $l(\omega) = d(s, v) < d(v)$. Damit gilt auch $d(s, u_k) < d(v)$.

Sei nun (u, w) die erste Kante auf diesem Weg mit $u \in S$ und $w \notin S$. Eine solche Kante existiert, da $s \in S$ und $v \notin S$.



Damit ist u vor v ausgewählt worden und nach Induktionsannahme gilt $d(u) = d(s, u)$.

Beweis (Korrektheit von Dijkstra - Fortsetzung)

Beim Hinzufügen von u zu S ist das Label von w überprüft worden, so dass beim Hinzufügen von v gilt:

$$d(w) \leq d(u) + l(u, w) = d(s, u) + l(u, w) = d(s, w).$$

Da w auf einem kürzesten Weg von s nach v liegt, gilt $d(s, w) \leq d(s, v)$, denn alle Kanten haben nicht-negatives Kantengewicht. Außerdem gilt $d(s, w) \leq d(w)$, denn $d(w)$ ist die Länge eines Weges von s nach w .

Dies führt zu

$$d(w) = d(s, w) \leq d(s, v) < d(v)$$

Dies steht aber im Widerspruch zur Auswahl von v als Knoten in $V \setminus S$ mit minimalem Label. □

Die Implementierung des Algorithmus von Dijkstra

Nachdem die Korrektheit des Verfahrens fest steht, bleibt zu beschreiben, wie der Algorithmus im Detail implementiert wird.

Offensichtlich bietet sich eine Priority Queue für die Ermittlung des nächsten Knoten an. Der Schlüssel ist dabei offensichtlich das Distanzlabel $d(v)$.

Die Relaxation der Label erfolgt über die Funktion `decrease_key`.

Algorithmus (Der Algorithmus von Dijkstra)

Eingabe: Ein gerichteter, gewichteter Graph $G = (V, E; l)$ mit nicht-negativen Kantengewichten und ein Knoten $s \in V$.

Ausgabe: Ein Feld $d[1 \dots V]$ mit $d(v) = d(s, v)$.

Daten: Eine Priority Queue Q

$Q.insert(s, 0)$ // Initialisiere die Priority Queue

für alle $v \in V \setminus \{s\}$ **tue**

$Q.insert(v, \infty)$

Ende

solange $Q \neq \emptyset$ **tue** // Solange noch Knoten übrig sind ...

$v := Q.delete_min()$ // ... extrahiere den nächsten und ...

für alle $w \in Q$ mit $(v, w) \in E$ **tue** // ... aktualisiere seine Nachbarn.

$Q.decrease_key(w, d(v) + l(v, w))$

Ende

Ende

Die Laufzeit des Algorithmus von Dijkstra

Die Laufzeit des Algorithmus von Dijkstra hängt wesentlich von der Anzahl und Art der auf der Prioritätswarteschlange ausgeführten Operationen ab.

Es ergeben sich die folgenden ausgeführten Operationen:

- Eine Initialisierung der Priority Queue, d.h. Zeit t_{init} .
- $|V|$ -mal `insert` zu Beginn, d.h. Zeit $|V| \cdot t_{\text{insert}}$.
- $|V|$ -mal `delete_min` um alle Knoten wieder zu extrahieren, d.h. Zeit $|V| \cdot t_{\text{delete_min}}$.
- höchstens $|E|$ -mal `decrease_key`, da das Label eines Knotens für jede Kante höchstens einmal nach unten korrigiert wird, d.h. , d.h. Zeit $|E| \cdot t_{\text{decrease_key}}$.

Damit ergibt sich die Gesamtlaufzeit als:

$$O(t_{\text{init}} + |V| \cdot (t_{\text{min}} + t_{\text{delete_min}}) + |E| \cdot t_{\text{decrease_key}})$$

Die Laufzeit des Algorithmus von Dijkstra

Benutzt man **binäre Heaps** zur Realisierung er gibt sich t_{init} als $O(|V|)$ und alle anderen Laufzeiten als $O(\log |V|)$. Dies führt zu einer Gesamtlaufzeit von

$$O(|V| \log |V| + |E| \log |V|).$$

Mittels **Fibonacci Heaps** erhalten wir eine Gesamtlaufzeit von

$$O(|V| \log |V| + |E|),$$

denn die amortisierten Kosten für maximal $|V|$ Elemente ergeben sich als $O(\log |V|)$ für `insert` und `delete_min` und $O(1)$ für `decrease_key`.

Satz (Laufzeit des Algorithmus von Dijkstra)

Sei $G = (V, E; l)$ ein gerichteter, gewichteter Graph mit nicht-negativen Kantengewichten und $s \in V$ ein ausgezeichneteter Knoten.

Unter Verwendung eines Fibonacci Heaps, berechnet der Algorithmus von Dijkstra die Distanzen $d(s, v)$ für alle Knoten $v \in V$ in der Zeit

$$O(|V| \log |V| + |E|).$$

Bemerkungen

- Der Algorithmus von Dijkstra kann leicht modifiziert werden, um nicht nur die Distanzen, sondern auch kürzeste Wege zu berechnen (Übungsaufgabe).
- Der (modifizierte) Algorithmus von Dijkstra konstruiert einen Spannbaum (Übungsaufgabe).
- Der Algorithmus von Dijkstra extrahiert die Knoten in Reihenfolge steigender Distanz aus der Prioritätswarteschlange, d.h. je näher je früher.
- Zur Berechnung der Distanz $d(s, t)$ zu einem bestimmten Knoten, kann der Algorithmus nach der Extraktion von t abgebrochen werden.

Teil 5: Algorithmen auf Graphen
Kürzeste Wege

SSSP: Der Algorithmus von Hagerup

Die Grundidee

Im Laufe der Zeit wurden verschiedene Algorithmen für das SSSP vorgestellt, die die Laufzeit des Algorithmus von Dijkstra verbessern.

Eine mögliche Verbesserung bezieht sich dabei auf eine Relaxierung der Auswahl des nächsten Knotens.

Um zu einem korrekten Ergebnis zu kommen, muss nicht unbedingt der Knoten mit dem tatsächlich kleinsten Label gewählt werden. Stattdessen ist es auch möglich einen kleinen, kontrollierten Fehler zu machen.

Torben Hagerup stellte 2004 eine Variante des Algorithmus von Dijkstra vor, die zwar im schlechtesten Fall eine zu dem Algorithmus von Dijkstra vergleichbare Laufzeit hat, in relativ allgemeinen Situationen im Mittel aber lineare Laufzeit erreicht.

Die Grundidee

Wir betrachten einen gerichteten, gewichteten Graphen $G = (V, E; l)$ mit reellen Kantengewichten aus $[0, 1]$.

Wie beim Algorithmus von Dijkstra werden die Label $d(v)$ mitgeführt. Allerdings wird als nächster Knoten nicht der mit minimalem Label $d(v)$ gewählt, sondern die Auswahl wird leicht verändert.

Sei $k := \lfloor \log |E| \rfloor$ und $\kappa := \frac{1}{k}$.

Die Knotenmenge zerlegen wir nun anhand des Wertes κ in zwei Teile:

$$V_1 := \{v \in V \mid \text{es existiert ein } u \in V \text{ mit } l(u, v) < \kappa\}$$

$$V_2 := V \setminus V_1$$

D.h. die Knoten in V_1 sind diejenigen, die mindestens eine „kurze“ eingehende Kante besitzen, während die Knoten in V_2 nur „lange“ eingehende Kanten haben.

Die Grundidee

Das neue Label $\hat{d}(v)$ misst nun die Distanz für die Knoten in V_2 in Vielfachen der Grenzlänge κ , d.h.

$$\hat{d}(v) := \begin{cases} d(v) & \text{falls } v \in V_1 \\ \lfloor \frac{d(v)}{\kappa} \rfloor \cdot \kappa & \text{falls } v \in V_2. \end{cases}$$

Damit gilt insbesondere für jeden Knoten $v \in V$:

$$d(v) - \kappa < \hat{d}(v) \leq d(v).$$

Die Grundidee ist somit, dass wir für Knoten, die nur lange eingehende Kanten besitzen, bei der Auswahl einen Fehler von maximal κ machen dürfen.

Algorithmus (Abstrakter Hagerup)

Eingabe: Ein gerichteter, gewichteter Graph $G = (V, E; l)$ mit Kantengewichten in $[0, 1]$ und ein Knoten $s \in V$.

Ausgabe: Ein Feld $d[1 \dots V]$ mit $d(v) = d(s, v)$.

$S := \{s\}$

$d(s) := 0$

für alle $v \in V \setminus \{s\}$ **tue** $d(v) := \infty$

solange $S \neq V$ **tue**

 Wähle einen Knoten $v \in V \setminus S$ mit minimalem $\hat{d}(v)$

$S := S \cup \{v\}$

für alle $w \in V \setminus S$ und $(v, w) \in E$ **tue**

$d(w) := \min \{d(w), d(v) + l(v, w)\}$

Ende

Ende

Satz (Korrektheit des Algorithmus von Hagerup)

Für jeden Knoten $v \in V$ gilt beim Hinzufügen zu S stets $d(v) = d(s, v)$.

Beweis

Wir führen eine Induktion über die Anzahl $|S|$ der Elemente in S vor dem Hinzufügen durch.

Für $|S| = 0$ wird s mit $d(s) = 0 = d(s, s)$ als nächstes hinzugefügt, so dass die Behauptung trivialerweise gilt.

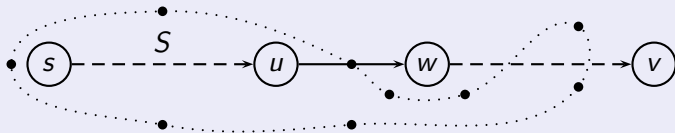
Sei nun $|S| > 0$. Da für jeden Knoten $u \in S$ beim Hinzufügen zu S $d(u) = d(s, u)$ galt, und $d(u)$ später nicht mehr verändert wird, gilt die Gleichung immer noch.

Außerdem stellen wir fest, dass durch die Korrektur der Label, diese höchstens sinken.

Beweis (Korrektheit von Hagerup - Fortsetzung)

Sei nun v ein Knoten aus $V \setminus S$, so dass $d(v) \leq d(w)$ für alle $w \in V \setminus S$. Nehmen wir an, dass $d(v) > d(s, v)$ gilt. Dann gibt es einen Weg $\omega = (s, u_1, \dots, u_k, v)$ von s nach v mit $l(\omega) = d(s, v) < d(v)$. Damit gilt auch $d(s, u_k) < d(v)$.

Sei nun (u, w) die erste Kante auf diesem Weg mit $u \in S$ und $w \notin S$. Eine solche Kante existiert, da $s \in S$ und $v \notin S$.



Damit ist u vor v ausgewählt worden und nach Induktionsannahme gilt $d(u) = d(s, u)$.

Beweis (Korrektheit von Hagerup - Fortsetzung)

Beim Hinzufügen von u zu S ist das Label von w überprüft worden, so dass beim Hinzufügen von v gilt:

$$d(w) \leq d(u) + l(u, w) = d(s, u) + l(u, w) = d(s, w).$$

Da w auf einem kürzesten Weg von s nach v liegt, gilt $d(s, w) \leq d(s, v)$, denn alle Kanten haben Längen ≥ 0 . Außerdem gilt $d(s, w) \leq d(w)$, denn $d(w)$ ist die Länge eines Weges von s nach w .

Gilt $v \in V_2$, so ergibt sich sogar $d(s, v) \geq d(s, w) + \kappa$, denn jede in v eingehende Kante hat mindestens das Gewicht κ .

Dies führt über

$$\begin{aligned} d(v) > d(s, v) &\geq d(s, w) = d(w) \text{ falls } v \in V_1 \\ d(v) > d(s, v) &\geq d(s, w) + \kappa = d(w) + \kappa \text{ falls } v \in V_2 \end{aligned}$$

zu $\hat{d}(v) > \hat{d}(w)$. Dies steht aber im Widerspruch zur Auswahl von v als Knoten in $V \setminus S$ mit minimalem Label. □

Die Anpassung der Prioritätswarteschlange

Um die Bestimmung des Elementes mit minimalem label $\hat{d}(v)$ zu beschleunigen, passen wir die Implementierung der verwendeten Prioritätswarteschlange an.

Die wesentliche Änderung besteht darin, dass wir nicht mehr eine „kontinuierliche“ Priority Queue verwenden, sondern die Queue in **Fächer** unterteilen.

Wir verwenden insgesamt $|V| \cdot k$ Fächer B_i mit $i = 0, \dots, |V| \cdot k - 1$. Dabei enthält B_i alle Knoten v mit $i \leq \frac{d(v)}{\kappa} < i + 1$, d.h. $v \in B_{\lfloor \frac{d(v)}{\kappa} \rfloor}$.

Da jeder kürzeste Weg maximal $|V| - 1$ Kanten enthält, ist $|V|$ eine geeignete obere Schranke für die Distanzlabel $d(v)$. Damit ist zu jedem Zeitpunkt jeder Knoten in einem Fach.

Die Bearbeitung der Fächer

Im Folgenden werden wir beweisen, dass die Fächer in aufsteigender Reihenfolge bearbeitet werden.

Präziser formuliert, werden wir feststellen, dass sobald B_i das nicht-leere Fach mit kleinstem Index ist, sich anschließend stets alle Knoten nur noch in den Fächern B_j mit $j \geq i$ befinden.

Lemma

$\min_{v \in V \setminus S} \hat{d}(v)$ wächst monoton.

Beweis (Monotonie der Label)

Offensichtlich kann durch die Korrektur eines Labels $d(v)$ mittels der Kante (u, v) das neue Label von v nicht kleiner als $d(u)$ werden, denn alle Kanten haben positive Längen.

Für $v \in V_1$ gilt somit $\hat{d}(v) = d(v) > d(u) \geq \hat{d}(u)$.

Für $v \in V_2$ gilt sogar

$$\hat{d}(v) = \left\lfloor \frac{d(v)}{\kappa} \right\rfloor \cdot \kappa \geq \left\lfloor \frac{d(u) + \kappa}{\kappa} \right\rfloor \cdot \kappa \geq \left\lfloor \frac{d(u)}{\kappa} \right\rfloor \cdot \kappa + \kappa > d(u).$$

Damit kann das Minimum der Label $\hat{d}(v)$ von noch nicht bearbeiteten Knoten nicht unter den Wert des gerade extrahierten Knotens sinken, was die Behauptung beweist. □

Die Suche nach dem kleinsten nicht-leeren Fach

Sei B_{i^*} das **aktuelle Fach**, d.h. i^* ist der kleinste Index, so dass B_{i^*} nicht leer ist.

Wir wissen, dass sich im Folgenden alle Knoten nur noch in Fächern B_i mit $i \geq i^*$ befinden werden. Damit müssen wir nach der vollständigen Leerung von B_{i^*} das kleinste nicht-leere Fach nicht im gesamten Bereich suchen, sondern nur unter den Indices $i > i^*$.

Eine mögliche Implementierung dieser Suche besteht darin, den Index i^* sukzessive zu erhöhen, bis wieder ein nicht-leeres Fach erreicht wird.

Diese Art der Suche nach dem nächsten nicht-leeren Fach, würde insgesamt (d.h. über den Verlauf des gesamten Algorithmus) $O(|V| \log |E|)$ Zeit erfordern (entsprechend der Anzahl der Fächer).

Dies lässt sich weiter verringern.

Die Organisation der Fächer

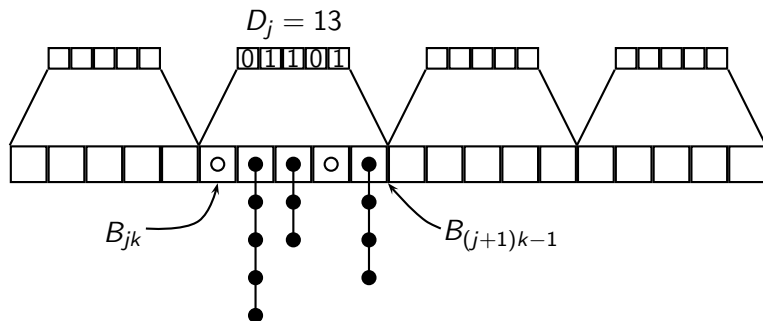
Für $j = 0, \dots, |V| - 1$ fassen wir jeweils die k Fächer B_i mit $jk \leq i < (j + 1)k$ zu einer Gruppe zusammen. Jeder dieser Gruppen wird ein Bitstring D_j der Länge k zugeordnet, den wir als natürliche Zahl interpretieren.

Der Wert dieser Zahl ist gegeben durch

$$D_j = \sum_{\substack{i=0 \\ B_{jk-1+i} \neq \emptyset}}^{k-1} 2^{k-1-i},$$

d.h. das Bit mit Wertigkeit 2^{k-1-i} ist genau dann gesetzt, falls B_{jk-1+i} nicht leer ist.

Die Organisation der Fächer



Die Organisation der Fächer

Durch diese Gruppierung wird das Auffinden des nächsten nicht-leeren Fachs beschleunigt:

- 1 Erhöhe den Zähler j der aktuellen Gruppe, bis $D_j \neq 0$.
- 2 Bestimme das höchste gesetzte Bit in D_j , um das erste nicht-leere Fach in dieser Gruppe zu ermitteln. Dies geschieht durch die Berechnung von $k - 1 - \lfloor \log D_j \rfloor$.

Können die D_j durch ein Speicherwort repräsentiert werden, kann sowohl das Setzen, das Löschen, als auch das Ermitteln des höchsten Bits in Zeit $O(1)$ realisiert werden.

Reicht ein Speicherwort nicht aus, kann man mit vorher in $O(|E|)$ Zeit berechneten Tabellen der Größe $O(|E|)$ arbeiten.

Damit haben wir die Gesamtlaufzeit für die Suchen nach dem nächsten nicht-leeren Fach auf $O(|V| + |E|)$ beschränkt.

Die Organisation der Warteschlange

Im Folgenden müssen wir beschreiben, wie wir die Operationen `decrease_key` und `delete_min` auf der angepassten Prioritätswarteschlange realisieren.

Da die Fächer in aufsteigender Folge bearbeitet werden, können wir stets davon ausgehen, dass es ein **aktuelles Fach** B_{i^*} gibt, d.h. i^* ist der kleinste Index, so dass B_{i^*} nicht leer ist.

Für das aktuelle Fach B_{i^*} verwalten wir

- eine Liste L , die alle Elemente aus $B_{i^*} \cap V_2$ enthält und
- einen (binären) Heap H , der alle Elemente aus $B_{i^*} \cap V_2$ enthält.

Prozedur (delete_min)

Eingabe: Die Queue Q mit aktuellem Fach B_{i^*} , Liste L und Heap H .

Ausgabe: Ein Element aus Q mit minimalem Label $\widehat{d}(v)$.

wenn $H = \emptyset$ und $L = \emptyset$ **dann**

// B_{i^*} ist leer

Suche nächstes nicht-leeres Fach B_j

$i^* := j$

Füge $B_j \cap V_2$ in L ein und $B_j \cap V_1$ in H

Ende

wenn $L \neq \emptyset$ **dann** Gebe ein beliebiges $v \in L$ zurück

sonst Gebe $\text{delete_min}(H)$ zurück

delete_min gibt tatsächlich einen Knoten mit minimalem $\widehat{d}(v)$ zurück, denn alle Knoten in $B_{i^*} \cap V_2$ haben ein minimales Label:

$$\widehat{d}(v) = \lfloor \frac{d(v)}{\kappa} \rfloor \cdot \kappa = i \cdot \kappa.$$

decrease_key

Das Senken eines Labels $d(v)$ führt zu den folgenden Situationen:

- 1 $v \in V_2$ gehört bereits zur aktuellen Liste L . Dann kann $d(v)$ einfach gesenkt werden.
- 2 $v \in V_2$ wird in den aktuelle Liste verschoben.
- 3 $v \in V_1$ gehört bereits zum aktuellen Heap H . Dann wird einfach `decrease_key` ausgeführt.
- 4 $v \in V_1$ wird in den aktuellen Heap verschoben. Dann wird es mittels `insert` eingefügt.
- 5 v wird in ein anderes, noch nicht bearbeitetes Fach verschoben. Dann wird es einfach an das Fach angehängt.

Der erste, zweite und fünfte Fall erfordern lediglich konstante Zeit.

Kritisch sind die Heap-Operationen im dritten und vierten Fall.

decrease_key

Da jeder Knoten aus V_1 höchstens einmal in den Heap H eingefügt wird (entweder er war bereits im aktuellen Fach oder er wurde dahin verschoben), ergeben sich insgesamt $|V_1|$ insert -Operationen.

Außerdem stellen wir folgendes fest:

Lemma

Wenn für einen Knoten $v \in H$ das Label $d(v)$ durch die Kante (u, v) gesenkt wird, dann gilt $l(u, v) < \kappa$.

Beweis

Der Knoten u muss ebenfalls aus dem aktuellen Fach B_{i^} stammen.*

Nehmen wir nun an, dass $l(u, v) \geq \kappa$ gilt. Damit ergibt sich bei der Entnahme von u $d(v) > d(u) + l(u, v) \geq d(u) + \kappa$.

Beweis (Fortsetzung)

Damit wäre v aber in Fach

$$\left\lfloor \frac{d(v)}{\kappa} \right\rfloor > \left\lfloor \frac{d(u) + \kappa}{\kappa} \right\rfloor = \left\lfloor \frac{d(u)}{\kappa} \right\rfloor + 1$$

und nicht im aktuellen Fach.



Als Konsequenz aus diesem Lemma, können wir feststellen, dass `decrease_key` auf binären Heaps höchstens für jede Kante (u, v) mit $l(u, v)$ ausgeführt wird.

Sei E_1 die Menge dieser Kanten in G .

Die Laufzeit des Algorithmus von Hagerup

Damit ergeben sich die folgenden Laufzeiten für den Algorithmus von Hagerup:

- Insgesamt $O(|V| + |E|)$ für die Operationen auf den Listen L für alle $v \in V$.
- Insgesamt $O(|V_1| \log |V_1|)$ für insert-Operationen auf den Heaps.
- Insgesamt $O(|E_1| \log |V_1|)$ für decrease_key-Operationen auf den Heaps.
- Insgesamt $O(|V| + |E|)$ für alle anderen Operationen.
- Insgesamt $O(|V| + |E|)$ für die Suchen nach dem aktuellen Fach.

Damit ist die Gesamtlaufzeit gegeben durch

$$O(|V| + |E| + (|V_1| + |E_1|) \log |V_1|) = O(|V| + |E| + |E_1| \log |V_1| + |V| \log |E|),$$

wobei $|E_1| \geq |V_1|$ ausgenutzt wurde.

Satz (Laufzeit des Algorithmus von Hagerup)

Sei $G = (V, E; l)$ ein gerichteter, zusammenhängender, gewichteter Graph mit $l(e) \in [0, 1]$ für alle Kanten $e \in E$, und $s \in V$ ein ausgezeichnete Knoten.

Sei $k = \lfloor \log |E| \rfloor$ und $\kappa = \frac{1}{k}$ und $E_1 := \{e \in E \mid l(e) < \kappa\}$ und $V_1 := \{v \in V \mid l(u, v) < \kappa \text{ für eine Kante } (u, v) \in E\}$. Dann berechnet der Algorithmus von Hagerup alle Distanzen $d(s, v)$ in Zeit $O(|V| + |E| + |E_1| \log |V|)$.

Sind die Kantengewichte gleichmäßig über $[0, 1]$ verteilt, ergibt sich eine mittlere Laufzeit von $O(|V| + |E|)$.

Beweis.

Die Laufzeit im schlechtesten Fall ist klar.

Für die mittlere Laufzeit beobachten wir, dass

$$E(|E_1|) = |E| \cdot \kappa = \frac{|E|}{\lfloor \log |E| \rfloor}$$

gilt. Dies führt zu der folgenden Laufzeit im Mittel:

$$O(|V| + |E| + \frac{|E|}{\log |E|} \log |V|) = O(|V| + |E| + \frac{|E|}{\log |E|} \log |E|) = O(|V| + |E|).$$

