

Kürzeste Wege Algorithmen und Datenstrukturen

Institut für Informatik
Universität zu Köln
SS 2009

Teile 1 und 2

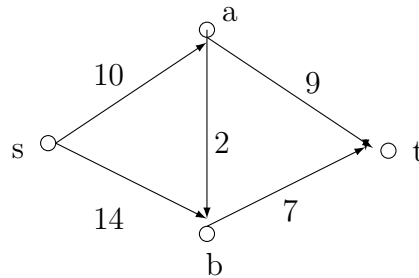
Inhaltsverzeichnis

1	Kürzeste Wege	2
1.1	Voraussetzungen	2
1.2	Dijkstra-Algorithmus	2
1.3	Korrektheit des Algorithmus	3
1.4	Abschätzung der Laufzeit	4
2	Datenstrukturen	5
2.1	Fibonacci-Heaps	5
2.1.1	Einige Operationen auf F-heaps	6
2.1.2	Amortisierte Kosten der Operationen	9
2.2	Buckets	13

1 Kürzeste Wege

1.1 Voraussetzungen

Gegeben sei ein gerichteter Graph $G = (V, E)$ mit Knotenmenge V und Kantenmenge E und Entfernungen $c(v, w) \geq 0$ für $(v, w) \in E$. Zusätzlich sei ein Knoten $s \in V$ ausgezeichnet.



Definition 1.1. Ein **Weg** P von $v \in V$ nach $w \in V$ ist eine Folge v_0, v_1, \dots, v_k von Knoten, so dass $v_0 = v, v_k = w$ und $(v_i, v_{i+1}) \in E$ für $0 \leq i \leq k - 1$. Der Weg hat die **Länge** $c(P) := \sum_{i=0}^{k-1} c(v_i, v_{i+1})$.

Gesucht: Kürzeste Wege von s zu allen anderen Knoten.

Idee des Verfahrens: Wir vergrößern schrittweise eine Menge M von Knoten, für deren Elemente $v \in M$ wir bereits einen kürzesten Weg von s nach v gefunden haben. Allen anderen Knoten $v \notin M$ ordnen wir die Länge des bisher gefundenen kürzesten Weges zu.

1.2 Dijkstra-Algorithmus

Wir bezeichnen mit $Dist(v)$ die Länge des (bisher gefundenen) kürzesten Weges von s nach v und mit $Vor(v)$ den Vorgänger von v auf einem solchen Weg.

- (1) Setze $Dist(s) := 0, M := \{s\}$
- (2) Für $v \in V$ mit $(s, v) \in E$ setze $Dist(v) := c(s, v)$ und $Vor(v) := s$
- (3) Für $v \in V$ mit $(s, v) \notin E$ setze $Dist(v) := +\infty$ und $Vor(v) := \emptyset$
- (4) Bestimme $u \notin M$ mit $Dist(u) = \min\{Dist(v) : v \notin M\}$.
- (5) Falls $Dist(u) = \infty$, dann **STOP**. Andernfalls setze $M = M \cup \{u\}$.
- (6) Für alle $v \notin M$ mit $(u, v) \in E$
Falls $Dist(v) > Dist(u) + c(u, v)$ dann setze:
 $Dist(v) = Dist(u) + c(u, v)$
 $Vor(v) = u$
- (7) Falls $M \neq V$, dann gehe zu 4, sonst **STOP**.

Die Menge M scheint sich „kreisförmig“ um den Startknoten s auszubreiten. Sei $d(u)$ der Wert von $Dist(u)$ zu dem Zeitpunkt, zu dem u in die Menge M aufgenommen wird.

1.3 Korrektheit des Algorithmus

Lemma 1.2. *Wird u vor v in M aufgenommen, so gilt $d(u) \leq d(v)$.*

Beweis. Angenommen, es existieren Knoten u und v , so dass u vor v aufgenommen wird, aber $d(u) > d(v)$ gilt. Unter allen diesen Knoten v wähle den als ersten in M aufgenommenen. In dem Moment, in dem u aufgenommen wird, gilt $d(u) = \text{Dist}(u) \leq \text{Dist}(v)$. Da per Definition $d(u)$ nicht mehr verändert wird, muss später $\text{Dist}(v)$ verringert worden sein. Dies kann nur passieren, wenn ein Knoten w in M aufgenommen und $\text{Dist}(v) = \text{Dist}(w) + c(w, v)$ gesetzt wird.

Sei w der letzte solche Knoten, d.h. es gilt $d(v) = d(w) + c(w, v)$. Nach Wahl von v gilt aber $d(u) \leq d(w)$. Da $c(w, v) \geq 0$, folgt $d(v) \geq d(u)$, im Widerspruch zur Voraussetzung. \square

Wenn das Verfahren korrekt ist, muss am Schluss gelten, dass $\text{Dist}(v) \leq \text{Dist}(u) + c(u, v)$, denn ansonsten wäre es kürzer, über u und die Kante (u, v) nach v zu gehen. Das folgende Lemma zeigt, dass unser Verfahren zumindest diese notwendige Bedingung erfüllt.

Lemma 1.3. *Nach Beendigung des Verfahrens gilt für alle Kanten $(u, v) \in E$:*

$$\text{Dist}(v) \leq \text{Dist}(u) + c(u, v).$$

Beweis. Die Aussage gilt sicherlich direkt nach der Iteration, in der u aufgenommen wird. Wenn sie zu einem späteren Zeitpunkt nicht mehr gilt, muss $\text{Dist}(u)$ verringert worden sein, da $\text{Dist}(v)$ nicht wächst. D.h. es gilt $\text{Dist}(u) < d(u)$. Dies kann wiederum nur passiert sein, als ein Knoten w in M aufgenommen und $\text{Dist}(u) = d(w) + c(w, u)$ gesetzt wurde. Nach dem ersten Lemma: $d(w) \geq d(u)$ und $c(w, u) \geq 0 \rightsquigarrow \text{Dist}(u) \geq d(u)$, im Widerspruch zur Voraussetzung. \square

Daraus folgt unmittelbar die Korrektheit des Verfahrens:

Satz 1.4. *Der Dijkstra-Algorithmus berechnet kürzeste Wege von s zu allen anderen erreichbaren Knoten.*

Beweis. Wir zeigen per Induktion über die Anzahl der Knoten auf einem kürzesten Weg, dass die notwendige Bedingung auch hinreichend ist. Die Aussage ist sicherlich richtig für s . Sei jetzt v ein Knoten und u ein Vorgänger auf einem kürzesten Weg. Per Induktion können wir annehmen, dass $\text{Dist}(u)$ die Länge eines kürzesten Weges von s nach u ist. Da $\text{Dist}(v) \leq \text{Dist}(u) + c(u, v)$, ist dann $\text{Dist}(v)$ die Länge eines kürzesten Weges nach v . \square

Aus dem zweiten Lemma folgt:

- Sobald u in M aufgenommen wird, ist ein kürzester Weg gefunden worden.
- Der Wert $\text{Dist}(u)$ ändert sich nicht mehr.
- Es reicht daher, eine im allgemeinen kleinere Menge U von Knoten dynamisch zu verwalten.
- U besteht aus allen Knoten, für die bereits ein Weg gefunden wurde, der jedoch noch nicht der kürzeste sein muss.

- U wird auch als **Front** bezeichnet.

Satz 1.5. *Der Dijkstra-Algorithmus endet, nachdem für alle von s aus erreichbaren Knoten die kürzesten Wege gefunden worden sind.*

Beweis. Nach Satz 1.4 berechnet der Dijkstra-Algorithmus einen kürzesten Weg für alle von s erreichbaren Knoten. Nachdem auch für den letzten dieser Knoten ein solcher Weg berechnet wurde, befindet sich der Algorithmus in Schritt (7).

1.Fall: In Schritt (7) gilt $M = V$. Dann stoppt der Algorithmus nach Schritt (7).

2.Fall: In Schritt (7) gilt $M \neq V$. Da bereits für alle Knoten ein kürzester Weg berechnet worden ist, befinden sich in $Q := V \setminus M$ nur noch Knoten, die von s und damit insbesondere von M aus nicht erreichbar sind. Für alle $v \in Q$ gilt also: $Dist(v) = \infty$. Bei der Bestimmung des Minimums in Schritt (4) erhält man folglich einen Knoten u mit $Dist(u) = \infty$. Nach der Vorschrift aus (5) hält demnach der Algorithmus. \square

1.4 Abschätzung der Laufzeit

Dijkstra-Algorithmus

- (1) $Dist(s) := 0, U = \{s\}$
- (2) for $v \in V \setminus \{s\}$ do
- (3) $Dist(v) = +\infty, Vor(v) = Null$
- (4) endfor
- (5) while $U \neq \emptyset$
- (6) choose $u \in U$ with $Dist(u)$ minimal (**finde Min**)
- (7) $U := U \setminus \{u\}$ (**lösche Min**)
- (8) for all $(u, v) \in E$
- (9) if $Dist(u) + c(u, v) < Dist(v)$ then
- (10) $Dist(v) = Dist(u) + c(u, v), Vor(v) = u$, (**verringere**)
- (11) $U := U \cup \{v\}$ (**füge ein**)
- (12) endif
- (13) endfor
- (14) endwhile

Lemma 1.6. *Die Laufzeit des Dijkstra-Algorithmus ist*

$$\mathcal{O}(n \cdot \max\{ \mathcal{O}(\text{finde Min}), \mathcal{O}(\text{lösche Min}), \mathcal{O}(\text{füge ein}) \}) + m \cdot \mathcal{O}(\text{verringere}).$$

Beweis: Die while-Schleife kann höchstens $n = |V|$ mal ausgeführt werden, da jeder Knoten höchstens einmal als Minimum auftreten kann. Die darin enthaltene for-Schleife kann aber auch nur einmal für jede Kante durchlaufen werden. Somit:

n Ausführungen von *finde Min*, *lösche Min*, *füge ein*,

m Ausführungen von *verringere*.

\square

2 Datenstrukturen

Für die Implementierung des Dijkstra-Algorithmus benötigen wir Datenstrukturen, um den Graphen, die Distanz ($Dist(v)$) und den Vorgänger ($Vor(v)$) eines Knotens sowie die Front (Menge U) effizient zu verwalten. Der Graph kann z.B. in einer Adjazenzliste abgespeichert werden:

Feld Tail $[1, \dots, n]$: Tail(i) zeigt auf Beginn einer verketteten Liste, die die Endknoten j und Kosten von Kanten $(i, j) \in E$ enthält.

Die Distanz und der Vorgänger eines Knotens können in Felder $Dist[1 \dots n]$, $Vor[1 \dots n]$ abgelegt werden.

Die Verwaltung der Front verlangt eine Datenstruktur, die die folgenden Operationen unterstützt.

- finde Minimum
- lösche Minimum
- füge ein
- verringere Inhalt eines gegebenen Elements.

Hier gibt es viele verschiedene Möglichkeiten. Wir wollen nun einige vorstellen und zeigen, wie sich ihre Verwendung auf die Laufzeit auswirkt.

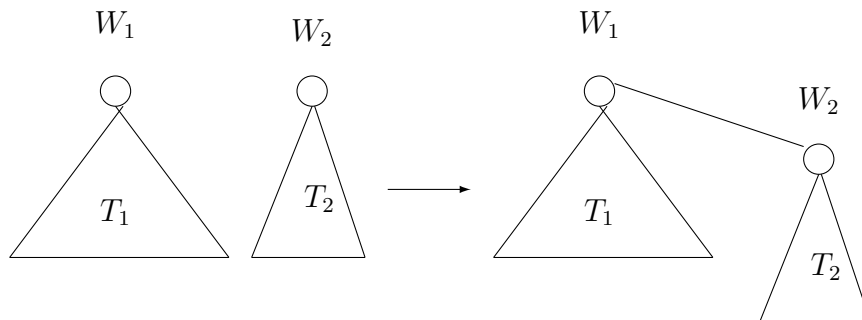
2.1 Fibonacci-Heaps

Definition 2.1. Ein **heap-geordneter Baum** ist ein Wurzelbaum mit Heapstruktur, d.h. der Schlüssel eines Knotens ist höchstens so groß wie die Schlüssel seiner Söhne.

Definition 2.2. Der **Rang** $rang(x)$ eines Knotens x ist die Anzahl der Söhne von x .

Definition 2.3. Seien T_1 und T_2 zwei heap-geordnete Bäume. Die **link-Operation** verschmilzt die beiden Bäume zu einem heap-geordneten Baum T .

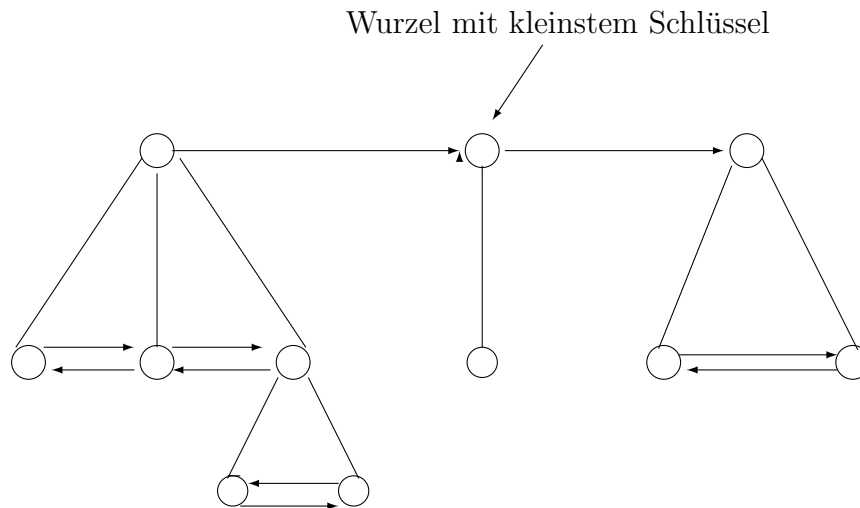
Implementation der Link-Operation: Vergleiche die Inhalte der Wurzeln r_1 von T_1 und r_2 von T_2 . Sei o.B.d.A. Inhalt von $r_1 \leq$ Inhalt von r_2 . Mache r_2 zum Sohn von r_1 .



$\Rightarrow \mathcal{O}(1)$.

Definition 2.4. Ein **Fibonacci-Heap** ist eine Familie von (schlüssel)-disjunkten heap-geordneten Bäumen mit

- Vater-Sohn-Zeiger
- Sohn-Vater-Zeiger
- Brüder in doppelt-verketteter Liste
- Wurzeln in zirkulärer Liste
- zusätzlicher Zeiger auf die Wurzel mit kleinstem Schlüssel.

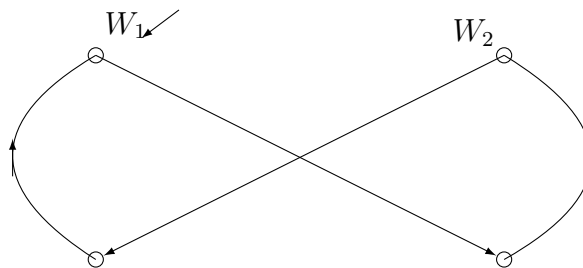
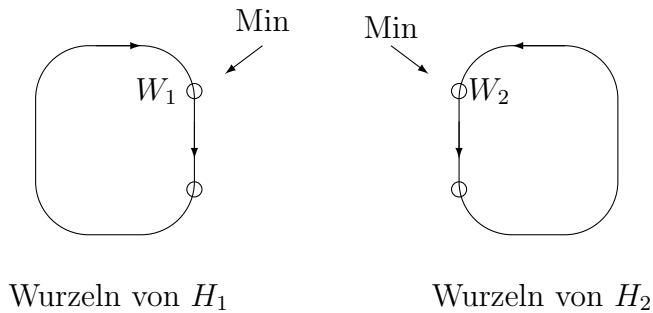


In jedem Knoten sind gespeichert:

- 4 Pointer für die Verwaltung des Heaps
- der Rang $\text{rang}(x)$
- ein weiteres Bit für “markiert” bzw. “unmarkiert”, dessen Bedeutung weiter unten erklärt wird.

2.1.1 Einige Operationen auf F-heaps

- **Vereinige:** Vereinige die zirkulären Wurzellisten zu einer zirkulären Wurzelliste und setze den Zeiger auf die minimale Wurzel neu. Diese Operation lässt sich wie folgt in $\mathcal{O}(1)$ Schritten durchführen. Seien H_1, H_2 zwei F-heaps mit minimalen Wurzeln W_1, W_2 , wobei o.b.d.A. Inhalt von $W_1 \leq$ Inhalt von W_2 .



- **Füge Schlüssel i in F-Heap H_1 ein:** Diese Operation ist ein Spezialfall der Vereinige-Operation und lässt sich in $\mathcal{O}(1)$ Schritten ausführen:
 1. erzeuge einen F-Heap H_2 , der aus i besteht
 2. vereinige H_1 und H_2
- **Finde Minimum:** durch Pointer in $\mathcal{O}(1)$
- **Lösche Minimum:** Wenn wir das Minimum löschen, müssen wir die neue “Wurzelliste” nach dem neuen Minimum durchsuchen. Um nicht bei verschiedenen Löschschritten dieselben Knoten als Minimumskandidaten vorzufinden, ist es ratsam, mit dem Auffinden des neuen Minimums gleichzeitig einen neuen heap-geordneten Baum aufzubauen. Der folgende Algorithmus leistet dies:
 1. lösche Minimum
 2. konkateniere die Liste der übrigen Wurzeln mit der Liste der Söhne des alten Minimums zu einer neuen Wurzelliste.
 3. gehe die neue Wurzelliste durch, um das neue Minimum zu bestimmen.
 4. falls keine zwei Wurzeln den gleichen Rang haben, Stop.

-
5. andernfalls seien T_1 und T_2 zwei Bäume mit Wurzeln r_1, r_2 und $\text{rang}(r_1) = \text{rang}(r_2)$. Führe $\text{link}(T_1, T_2)$ aus und gehe zu 3.

Implementation von Schritt 3 und 4:

Wir definieren ein Sortierfeld, Sortfeld $[0 : \text{Max Rang}]$, mit der Eigenschaft, dass Sortfeld (i) auf eine Wurzel mit Rang i zeigt. Max Rang ist der größte vorkommende Rang. Wir werden später zeigen, dass $\text{Max Rang} = \lceil \log n \rceil$ ist.

```

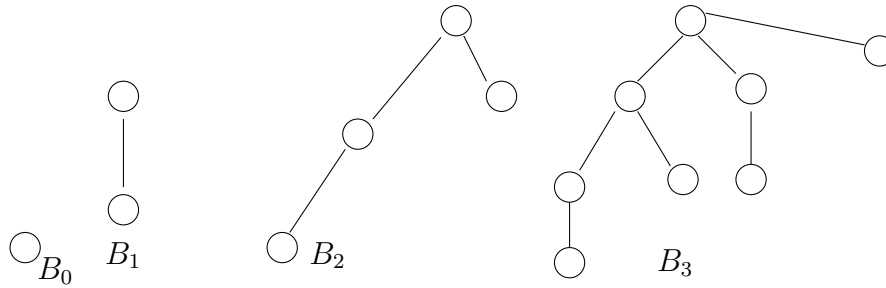
Sortfeld  $(i) = \text{Nil}$  für  $0 \leq i \leq \text{MaxRang}$ .
While Wurzelliste  $\neq \emptyset$  do
    entferne die erste Wurzel  $x$  aus der Liste
    setze AktWurzel :=  $x$ 
    setze AktRang :=  $\text{rang}(x) = (\# \text{ der Söhne von } x)$ 
    if Schlüssel  $(x) < \text{Schlüssel}(Min)$  then  $Min := x$ 
    while  $\text{Sortfeld}(\text{AktRang}) \neq \text{Nil}$  do
        link (AktWurzel, Sortfeld(AktRang))
        setze AktWurzel auf die Wurzel der vereinigten Bäume
        Sortfeld (AktRang) := Nil
        AktRang := AktRang + 1
    endwhile
    Sortfeld (AktRang) := AktWurzel
endwhile
For  $i := 0$  to MaxRang do
    if Sortfeld  $(i) \neq \text{Nil}$  then füge  $\text{Sortfeld}(i)$  in Wurzelliste ein
endfor

```

Lemma 2.5. Die Laufzeit für "lösche Min" beträgt $\mathcal{O}(\text{MaxRang} + \text{Anzahl aller link-Operationen})$ □

Beobachtung 2.6. Wenn wir auf einen anfänglich leeren F -heap nacheinander und mehrmals die Operationen "Einfügen", "Finde min", "lösche min" anwenden, so sind zu jedem Zeitpunkt die Bäume des F -heaps **binomial**, d.h.:

$$\begin{aligned}
 B_0 &= \{r_0\} \\
 B_k &= \text{link}(B_{k-1}, B_{k-1})
 \end{aligned}$$



Denn: lediglich “lösche min” verändert die Bäume und der obige Algorithmus behält die binomiale Struktur bei.

Lemma 2.7. (i) $|B_k| = 2^k$

(ii) Die Wurzel von B_k hat k Söhne

(iii) $x \in B_k \Rightarrow \text{rang}(x) \leq k$.

Beweis: Induktion über k .

(i) $|B_k| = 2|B_{k-1}| = 2 \cdot 2^{k-1}$.

(ii) # Söhne von Wurzel von $B_k = 1 + \#$ Söhne von B_{k-1} .

(iii) $x \in B_{k-1}$ oder x ist die Wurzel von B_k . □

2.1.2 Amortisierte Kosten der Operationen

Wir haben bisher worst-case-Abschätzungen für die Laufzeiten unserer Operationen durchgeführt. Diese Operationen werden mehrmals hintereinander ausgeführt. Es ist daher durchaus möglich, dass wir bei einer Ausführung Arbeit investieren, die sich bei späteren Ausführungen durch eine geringere Laufzeit bezahlt machen. Die Analyse wird üblicherweise mit amortisierten Kosten gemacht. Sei dazu zu jedem Fibonacci-Heap H eine nichtnegative Zahl $\text{potential}(H)$ gegeben, die wir später geeignet spezifizieren.

Definition 2.8. Einer Operation Op , die den F -Heap H_1 in den F -Heap H_2 überführt, ordnen wir wie folgt **amortisierte Kosten** zu:

$$\text{amortisierteKosten}(Op) = \text{Laufzeit}(Op) + \text{potential}(H_2) - \text{potential}(H_1)$$

Bemerkung 2.9. Wir starten mit einem leeren F -Heap und führen eine Folge von Operationen Op_1, Op_2, \dots, Op_k durch, wobei Op_i den F -Heap H_{i-1} in den F -Heap H_i überführt. Dann ergeben sich die amortisierten Gesamtkosten als Summe der Laufzeiten + $\text{potential}(H_k)$. Da $\text{potential}(H_k) \geq 0$, gilt stets:

$$\text{Laufzeit} \leq \text{amortisierte Kosten}$$

Somit stellen die amortisierten Kosten eine obere Schranke für die Laufzeit dar. Es reicht also, die amortisierten Kosten zu berechnen, die meist besser als die Laufzeiten abgeschätzt werden können. Dabei ist es natürlich wichtig eine geeignete Potentialfunktion zu wählen.

Definition 2.10. Sei H ein F -Heap mit Bäumen T_1, \dots, T_k . Das **Potential** von H ist gegeben durch:

$$\text{potential}(H) = k = \text{Anzahl der Bäume}$$

Wir wollen jetzt die Operationen finde Min, füge ein und lösche Min noch einmal amortisiert abschätzen.

- **füge ein:** amortisierte Kosten $\mathcal{O}(1)$ (da die Anzahl der Bäume gleich bleibt)
- **lösche Min:** durch das Löschen der Wurzel in H_1 erhöht sich die Anzahl der Bäume in H_2 um höchstens $\log n \geq \text{rang}(\text{Wurzel})$ und jeder link-Schritt verringert die Anzahl um eins. Somit:

$$\text{potential}(H_2) \leq \text{potential}(H_1) + \log n - \text{Anzahl der link-Operationen}$$

Entsprechend ergeben sich die amortisierten Kosten von "lösche Min" als höchstens (vergl. Lemma 2.5:

$$(\log n + \text{Anzahl der link-Operationen}) + (|\text{Bäume in } H_1| + \log n - \text{Anzahl der link-Operationen}) - |\text{Bäume in } H_1| = \mathcal{O}(\log n).$$

- **finde Min:** amortisierte Kosten $\mathcal{O}(1)$ (da die Anzahl der Bäume gleich bleibt)
- **verringere:** Gegeben ein Zeiger auf das Element i , verringere den Schlüsselwert von i um $\Delta \geq 0$: Wir können versuchen, Δ zu subtrahieren und danach die Heap-Struktur zu reparieren. Dies würde zu einer Laufzeit von $\mathcal{O}(\log n)$ führen und damit die Gesamtlaufzeit gegenüber den $(a, 2a)$ -Bäumen nicht verbessern. Wir wählen daher folgenden Ansatz:

Subtrahiere Δ vom Schlüssel in i
 Löse die Vater-Sohn-Verbindung von i zu seinem Vater und von i zu seinen Brüdern
 Verringere den Rang des Vaters
 Nimm i in die Wurzelliste auf
 Schreibe Minimalzeiger fort.

in $\mathcal{O}(1)$

Aber: "verringere" verändert die Struktur, insbesondere die Binomialstruktur! Damit ist die Laufzeitabschätzung für lösche Min so nicht mehr gültig. Wir führen daher eine Zusatzregel ein, die dafür sorgen wird, dass die Laufzeitschranke auch weiterhin gelten wird (siehe Lemma 2.13).

Zusatzregel:

1. wird ein Knoten x in einem link-Schritt Sohn eines anderen Knotens, lösche, wenn vorhanden, die Markierung von x .

2. wird die Verbindung zwischen x und seinem Vater $p(x)$ gelöst und ist $p(x) \neq$ Wurzel:

- (a) $p(x)$ unmarkiert $\Rightarrow p(x)$ wird markiert.
- (b) $p(x)$ markiert \Rightarrow löse Verbindung $p(x), p(p(x))$.

Bemerkung: Die Markierung zählt die von einem Knoten abgeschnittenen Söhne und begrenzt deren Anzahl.

Vorsicht: der letzte Schritt kann propagieren.

Werden bei einer “verringere”-Operation die Verbindungen zwischen $p(x)$ und $p^2(x)$ bis hinauf zu $p^k(x)$ und p^{k+1} gelöst, so bewirkt die “verringere”-Operation k **Folgeschnitte**. Die Laufzeit ergibt sich damit als \mathcal{O} (Anzahl der Folgeschnitte).

Bemerkung:

Durch die Zusatzregel haben wir erreicht, dass weiterhin die Größe eines jeden Baumes exponentiell im Rang der Wurzel ist (Korollar 2.12). Überlegen Sie, dass dies die einzige Eigenschaft binomialer Bäume war, die wir für die Laufzeitabschätzung von lösche Min benutzt haben, so dass diese weiterhin gültig ist (vergl. Lemma 2.13). Andererseits läuft “verringere” auf den ersten Blick jetzt nicht mehr in $\mathcal{O}(1)$. Hat eine “verringere”-Operation jedoch Folgeschnitte ausgelöst, verkleinert sich für viele andere Knoten die Anzahl der Folgeschnitte, die “verringere” auslösen könnte. Wir sind also in der typischen Situation, in der die Laufzeit mit Hilfe amortisierter Kosten und einer geeigneten Potentialfunktion besser abgeschätzt werden kann. Wir werden zeigen, dass bei mehreren Heap-Operationen die Summe der Folgeschnitte, die von einzelnen “verringere”-Operationen hervorgerufen werden, durch die Anzahl der “verringere”-Operationen beschränkt ist.

Lemma 2.11. *Sei x ein Knoten in einem F-heap und y_1, \dots, y_k die augenblicklichen Söhne von x in der Reihenfolge ihrer Link-Verbindungen zu x . Dann ist $\text{rang}(y_i) \geq i - 2$.*

Beweis: Zum Zeitpunkt als y_i zum Sohn von x wurde, hatte x mindestens $i - 1$ Söhne, nämlich y_1, \dots, y_{i-1} . Gemäß der Link-Operation, hatten x und y_i den gleichen Rang, d.h. beide hatten Rang mindestens $i - 1$. Nach der Link-Operation kann y_i höchstens einen Sohn verloren haben, da sonst die Verbindung zwischen x und y_i gekappt worden wäre. \square

Korollar 2.12. *Ein Knoten vom Rang k in einem F-Heap hat mindestens $F_{k+2} \geq \tau^k$ Nachfolger (sich selbst eingeschlossen, $F_0 = 0, F_1 = 1, F_{k+1} = F_k + F_{k-1}$ und $\tau = (1 + \sqrt{5})/2$ der goldene Schnitt).*

Beweis: Sei S_k die kleinstmögliche Anzahl von Nachfolgern eines Knotens vom Rang k in einem F-Heap. Offensichtlich: $S_0 = 1$, und $S_1 = 2$. Allgemein besteht ein Baum mit Wurzel vom Rang k zumindest aus der Wurzel, dem “ältesten” Sohn und den Teilbäumen unter den $k - 1$ “jüngeren” Söhnen. Nach obigem Lemma ist dann $S_k \geq 2 + \sum_{i=2}^k S_{i-2}$, für $k \geq 2$. Es ist: $F_{k+2} = \sum_{i=2}^k F_i + 2$ für $k \geq 2$, denn $2 + \sum_{i=2}^k F_i = 2 + \sum_{i=2}^{k-1} F_i + F_k = F_{k+1} + F_k = F_{k+2}$ per Induktion. Somit $S_k \geq F_{k+2}$ per Induktion und entsprechend $F_{k+2} \geq \tau^k$. \square

Lemma 2.13. *(Amortisierte Kosten von “lösche Min”) Die amortisierten Kosten einer “lösche Min”-Operation in einem F-Heap, der durch die oben beschriebenen Operationen “finde Min”, “füge ein”, “lösche Min”, “verringere” verändert wird, betragen $\mathcal{O}(\log n)$*

Beweis: Für die Laufzeitabschätzung von “lösche Min”, haben wir den maximalen Rang mit Hilfe der binomialen Bäume durch $\log n$ abgeschätzt. Dies ist jetzt nicht mehr möglich. Sei k der Rang des Minimums, dann gilt nach Korollar 2.12 $\tau^k \leq n \Leftrightarrow \frac{\log_2 n}{\log_\tau n} \leq n$ und somit

$$\max \text{Rang} \leq \frac{\log n}{\log \tau} \leq 1.5 \log n$$

□

Sei $ck, c \in \mathbb{R}_+$ eine Laufzeitabschätzung für die “verringere”-Operation mit k Folgeschnitten.

Definition 2.14. $Potential(H) = c(\text{Anzahl der Bäume} + 2 * \text{Anzahl der markierten Nicht-wurzelknoten})$.

Lemma 2.15. Bei einer “verringere”-Operation mit k Folgeschnitten erhöht sich das Potential um höchstens $c(3 - k)$.

Beweis: Da k Folgeschnitte ausgeführt werden, waren $p(x), \dots, p^k(x)$ markierte Nichtwurzeln, die zusammen mit x zu Wurzeln werden. Zudem ist $p^{k+1}(x)$ Wurzel oder unmarkiert. Im letzteren Fall wird sie zu einer markierten Nicht-Wurzel. Die Anzahl der Bäume erhöht sich so um $k + 1$ und die Anzahl der markierten Nichtwurzeln sinkt um mindestens $k - 1$. Damit ergibt sich eine Potentialänderung um höchstens $c(3 - k)$. □

Lemma 2.16. Die amortisierten Kosten einer “verringere”-Operation in einem F -Heap, der durch die oben beschriebenen Operationen “finde Min”, “füge ein”, “lösche Min” und “verringere” verändert wird, betragen $\mathcal{O}(1)$.

Beweis: Es bleibt lediglich die Aussage für die “verringere”-Operation zu zeigen. Sei dazu k die Anzahl der Folgeschnitte.

$$\begin{aligned} \text{Amortisierte Kosten} &\leq ck + \text{Potentialänderung} \\ &\leq ck + c(3 - k) \\ &= 3c = \mathcal{O}(1) \end{aligned}$$

□

Satz 2.17. Wenn wir mit einem leeren F -Heap starten und eine Folge von Heap-Operationen ausführen, so ist die Laufzeit durch die amortisierte Zeit beschränkt.

Beweis: Beide Potentialfunktionen sind positiv und am Anfang = 0 und somit folgt die Behauptung mit Bemerkung 2.10. □

Satz 2.18. Die Laufzeit eines Dijkstra-Algorithmus, der die Front mit Hilfe eines F -Heaps verwaltet, beträgt $\mathcal{O}(m + n \log n)$.

Beweis: Nach Satz 2.17 ist die Laufzeit durch die amortisierten Kosten beschränkt. Diese ergeben sich für jede Operation als Anzahl der Ausführungen dieser Operation mal ihren amortisierten Kosten. Insgesamt führen wir höchstens n füge-ein, n finde-Min, n lösche-Min und m verringere Operationen durch, also ergibt sich eine Laufzeit von $\mathcal{O}(n \cdot 1 + n \cdot 1 + n \log n + m \cdot 1) = \mathcal{O}(m + n \log n)$. □

2.2 Buckets

Lemma 2.19. Sei $d := \min\{\text{dist}(u) : u \in U\}$ und $C = \max\{c(e) : e \in E\}$. Für alle $v \in U$ gilt $d \leq \text{dist}(v) \leq d + C$.

Beweis: Induktion über die while-Schleife, wobei wir mit d_k, dist_k, U_k die entsprechenden Größen vor Ausführung der k -ten Iteration bezeichnen.

Die Aussage ist offensichtlich richtig für $k = 1$. Sei u im Schritt k das Element mit minimaler Distanz und $v \in U_k$. Nach Wahl von u ist $d_k \leq \text{dist}_k(v)$. Ist $v \in U_{k-1}$, so gilt per Induktion, dass $\text{dist}_k(v) \leq \text{dist}_{k-1}(v) \leq d_{k-1} + C \leq d_k + C$. Ist $v \notin U_{k-1}$, so ist $\text{dist}_k(v) = d_k + c(u, v) \leq d_k + C$. \square

Wir können daher die Menge U in $C + 1$ doppelt verketteten Listen Eimer $(0), \dots, \text{Eimer}(C)$ verwalten, wobei die Liste $\text{Eimer}(k)$ alle Knoten $v \in U$ enthält, für die gilt $k = \text{dist}(v) \bmod (C + 1)$. Zusätzlich benutzen wir wieder einen Zeiger $\text{assign}(v)$, der von v auf dasjenige Listenelement der Liste der Eimer zeigt, das v enthält.

$$\begin{aligned} & \text{Eimer}(0), \dots, \text{Eimer}(i), \dots, \text{Eimer}(C) \text{ mit} \\ & \text{Dist}(v) \bmod (C + 1) = i \Leftrightarrow v \in \text{Eimer}(i) \end{aligned}$$

- **füge ein:** Füge v in den Eimer $\text{Dist}(v) \bmod (C + 1)$ ein.
- **Finde Min:** Sei i_0 die Nummer des Eimers, der das letzte Minimum enthielt

```

i = 0
While Eimer  $((i_0 + i) \bmod (C + 1)) = \emptyset$  und  $i \leq C$  do i = i + 1.
If i = C + 1
    then Stop (Front ist leer)
else
    das Minimum befindet sich im Eimer  $(i_0 + i) \bmod (C + 1)$ .
endif

```

- **lösche Min:** Entferne das Minimum aus seinem Eimer.
- **Verringere:** Entferne v aus dem Eimer $\text{OldDist}(v) \bmod (C + 1)$ und füge v in den Eimer $\text{NewDist}(v) \bmod (C + 1)$ ein.

Lemma 2.20. Die Laufzeit des Dijkstra-Algorithmus mit Buckets beträgt $\mathcal{O}(n \cdot |C| + m)$.

Beweis Füge-ein, Verringere und Lösche-Min können auf doppelt verketteten Listen in $\mathcal{O}(1)$ ausgeführt werden. Die Schleife in Find Min wird maximal $C + 1$ mal durchlaufen also ist Find Min in $\mathcal{O}(|C|)$ und die Behauptung folgt mit Lemma 2.6. \square