
Kapitel 3: Greedy-Algorithmen

Greedy*-Algorithmen sind anwendbar bei **Konstruktionsaufgaben** zum **Finden einer optimalen Struktur**.

Sie finden eine Lösung, die sie schrittweise aufbauen, ohne zurückzusetzen. Es werden dabei nicht mehr Teillösungen konstruiert als unbedingt nötig.

* greedy (*engl.*): gierig.

3.1 Zwei Beispiele

Beispiel 1: Hörsaalbelegung

Gegeben: Veranstaltungsort (Hörsaal), Zeitspanne $[T_0, T_1]$ und eine Menge von n Aktionen (Vorlesungen oder ähnliches), durch Start- und Endzeit spezifiziert:

$$[s_i, f_i), \text{ für } 1 \leq i \leq n.$$

Gesucht: Belegung des Hörsaals, die **möglichst viele** Ereignisse mit disjunkten Zeitspannen stattfinden lässt.

Formal: Finde Menge $A \subseteq \{1, \dots, n\}$, so dass alle $[s_i, f_i)$, $i \in A$, disjunkt sind und $|A|$ maximal ist.

Ansätze, die nicht funktionieren:

- Zuerst kurze Ereignisse planen
- Immer ein Ereignis mit möglichst früher Anfangszeit wählen

Trick: Bearbeite Ereignisse nach **wachsenden Schlusszeiten**. O.B.d.A.: Veranstaltungen nach Schlusszeiten aufsteigend sortiert (Zeitaufwand $O(n \log n)$), also:

$$f_1 \leq f_2 \leq \dots \leq f_n.$$

Wiederhole: Wähle die **zulässige** Aktion mit der kleinsten Schlusszeit und füge sie zum Belegungsplan hinzu.

Eine Aktion ist **zulässig**, wenn ihre Startzeit mindestens so groß wie die Schlusszeit der letzten schon geplanten Veranstaltung ist.

Algorithmus Greedy Scheduling (GS)

Eingabe: $[T_0, T_1), [s_1, f_1), \dots, [s_n, f_n)$ nichtleere reelle Intervalle, $[s_i, f_i) \subseteq [T_0, T_1)$

Ausgabe: Maximal großes $A \subseteq \{1, \dots, n\}$ mit $[s_i, f_i), i \in A$ disjunkt

Sortiere Intervalle gemäß f_1, \dots, f_n aufsteigend;

$A \leftarrow \{1\};$

$f_{last} \leftarrow f_1;$

for i **from** 2 **to** n **do**

if $s_i \geq f_{last}$ **then** (* $[s_i, f_i)$ zulässig *)

$A \leftarrow A \cup \{i\};$

$f_{last} \leftarrow f_i;$

return A

Satz 3.1.1

Der Algorithmus Greedy Scheduling hat lineare Laufzeit (bis auf die Sortierkosten von $O(n \log n)$) und löst das Hörsaalplanungsproblem optimal.

Beweis: Laufzeit: Sortieren kostet Zeit $O(n \log n)$; der restliche Algorithmus hat offensichtlich Laufzeit $O(n)$.

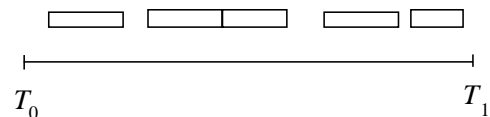
Korrektheit: Wir behaupten: Algorithmus GS liefert auf Inputs mit Größe n eine optimale Lösung, und beweisen dies durch **vollständige Induktion**.

Der Fall $n = 1$ ist trivial.

Sei nun $n > 1$.

I.V.: GS liefert für Inputs der Länge $n' < n$ eine optimale Lösung.

Ind.-Schritt: Sei $B \subseteq \{1, \dots, n\}$ eine **optimale** Lösung, $|B| = r$. Die zu B gehörigen Intervalle können auf einer Zeitachse etwa wie folgt angeordnet sein. (Im Beispiel: $r = 5$.)



1. Beobachtung: Es gibt eine Lösung B' , die mit dem Intervall $[s_1, f_1)$ (dem ersten Schritt des Greedy-Algorithmus) startet und ebenfalls r Ereignisse hat. (Also ist B' auch optimal.)

Wieso? Setze

$$B' := (B \setminus \{\min(B)\}) \cup \{1\}.$$

Weil die Intervalle gemäß Schlusszeiten aufsteigend sortiert sind, gilt $f_1 \leq f_{\min(B)}$.

2. Beobachtung: Die Menge $B \setminus \{\min(B)\}$ **löst das Teilproblem (*)**, das durch $[f_1, T_1)$, $\{[s_i, f_i) \mid s_i \geq f_1\}$ gegeben ist, **optimal**, das heißt:

in $[f_1, T_1)$ können maximal $r - 1$ Ereignisse untergebracht werden.

Wieso? Sonst würden wir $[s_1, f_1)$ mit einer besseren Lösung für (*) zu einer besseren Lösung für das Gesamtproblem kombinieren:

Widerspruch zur Optimalität von B .

3. Beobachtung: Algorithmus GS auf Eingabe $\{[s_i, f_i) \mid 1 \leq i \leq n\}$ hat ab Iteration für $i = 2$ **genau dasselbe Verhalten** wie wenn man GS auf $[f_1, T_1)$, $\{[s_i, f_i) \mid s_i \geq f_1\}$ starten würde. Nach I.V. liefert also dieser Teil des Algorithmus eine optimale Lösung mit $r - 1$ Ereignissen für (*).

Also liefert Greedy Scheduling insgesamt eine optimale Lösung der Größe r .

□

Beispiel 2: Fraktionales („teilbares“) Rucksackproblem

Gegeben: n Objekte mit positiven **Volumina** a_1, \dots, a_n und positiven **Nutzenwerten** c_1, \dots, c_n , sowie eine Volumenschranke b .

Gesucht: Vektor $(\lambda_1, \dots, \lambda_n) \in [0, 1]^n$, so dass

$$\lambda_1 a_1 + \dots + \lambda_n a_n \leq b$$

und

$$\lambda_1 c_1 + \dots + \lambda_n c_n \text{ maximal.}$$

Veranschaulichung: Dieb stiehlt Säckchen mit Edelmetallkrümeln, beliebig teilbar. Nutzen pro Volumen unterschiedlich für unterschiedliche Materialien. Was soll er in seinen Rucksack mit Volumen b packen, um den Wert zu maximieren?

Beim 0-1-Rucksackproblem werden nur 0-1-Vektoren mit $\lambda_i \in \{0, 1\}$ zugelassen.

(Mitteilung im Vorgriff auf BuK, 5. Sem.: Die $\{0, 1\}$ -Version ist NP-vollständig, besitzt also wahrscheinlich keinen effizienten Algorithmus.)

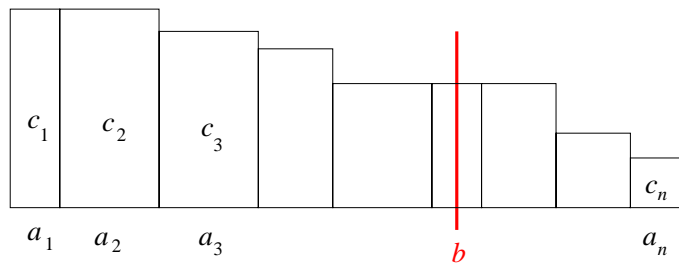
Das fraktionale Rucksackproblem ist mit einem Greedy-Algorithmus in Zeit $O(n \log n)$ lösbar.

Kern der Lösungsidee: Berechne „**Nutzendichte**“

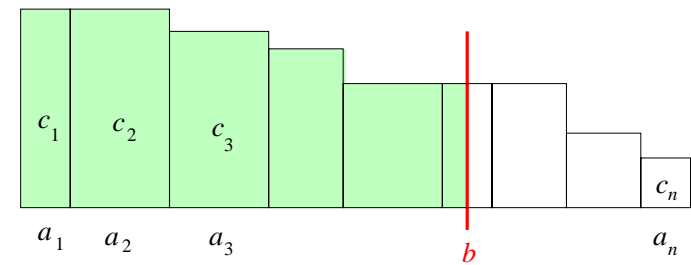
$$d_i = c_i / a_i, 1 \leq i \leq n,$$

und sortiere die Objekte gemäß d_i fallend.

Nehme von vorne beginnend möglichst viele ganze Objekte, bis schließlich das letzte Objekt teilweise genommen wird, so dass die Gewichtsschranke vollständig ausgenutzt wird.



Input.



Vom Greedy-Algorithmus gelieferte Lösung.

Algorithmus Greedy Fractional Knapsack (GFKS)

for i **from** 1 **to** n **do**

$d_i \leftarrow c_i/a_i$;

$\lambda_i \leftarrow 0$;

Sortiere Objekte gemäß d_i fallend;

$i \leftarrow 0$;

$r \leftarrow b$; (* r ist verfügbares Rest-Volumen *)

while $r > 0$ **do**

$i++$;

if $a_i \leq r$

then $\lambda_i \leftarrow 1$; $r \leftarrow r - a_i$;

else $\lambda_i \leftarrow r/a_i$;

$r \leftarrow 0$;

Satz 3.1.2

Der Algorithmus GFKS ist korrekt und hat Laufzeit $O(n \log n)$.

Beweis: Laufzeit: klar.

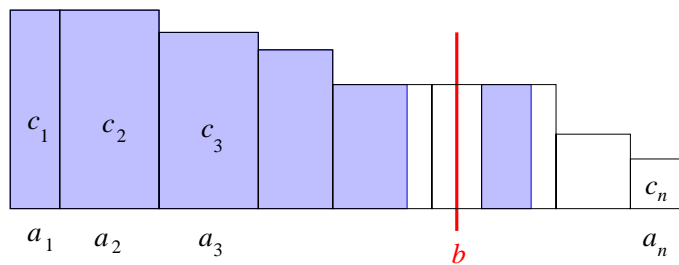
Korrektheit: Wieder „Austausch der ersten Entscheidung“.

Sei $(\lambda_1, \dots, \lambda_n) \in [0, 1]^n$ die Ausgabe des Algorithmus.

Sei $(\lambda'_1, \dots, \lambda'_n) \in [0, 1]^n$ eine optimale Lösung.

Wir benutzen wieder Induktion über n .

I.A.: $n = 1$. Dann liefert der Algorithmus offensichtlich die optimale Lösung: Packer Objekt Nummer 1 ganz oder bis zu dem Bruchteil, der in den Rucksack passt.



Eine andere optimale Lösung.

I.S.: $n > 1$. – Wenn $\lambda_1 < 1$, dann kann das nur daran liegen, dass das erste Objekt (mit maximaler Nutzendichte) nicht in den Rucksack passt. Dann liefert der Algorithmus das korrekte Ergebnis.

Wenn $1 = \lambda_1 = \lambda'_1$, wenden wir die Induktionsvoraussetzung auf das Teilproblem $x' = (a_2, \dots, a_n, c_2, \dots, c_n, b - a_1)$ an. GFKS läuft in Schleifendurchläufen 2 bis n ebenso wie GFKS auf x' , liefert also für dieses Teilproblem eine optimale Lösung.

Weiter muss $(\lambda'_2, \dots, \lambda'_n)$ eine optimale Lösung sein. (Wieso?)

Also haben die Lösungen $(\lambda'_1, \dots, \lambda'_n)$ und $(\lambda_1, \dots, \lambda_n)$ denselben Wert.

Merkwürdiger Sonderfall:

$1 = \lambda_1 > \lambda'_1$: Dann muss $\lambda'_1 a_1 + \dots + \lambda'_n a_n = b$ sein.

Reduziere $\lambda'_2, \dots, \lambda'_n$ so zu $\lambda''_2, \dots, \lambda''_n$, dass

$$(\lambda'_2 - \lambda''_2)a_2 + \dots + (\lambda'_n - \lambda''_n)a_n = (1 - \lambda'_1)a_1,$$

und setze $\lambda''_1 = 1$. Dann ist $(\lambda''_1, \dots, \lambda''_n)$ zulässig und hat einen Nutzenwert

$$\lambda''_1 c_1 + \dots + \lambda''_n c_n \geq \lambda'_1 c_1 + \dots + \lambda'_n c_n.$$

Die letzte Summe war aber optimal, also ist $(\lambda''_1, \dots, \lambda''_n)$ auch optimale Lösung.

Nun können wir argumentieren wie oben, weil $\lambda''_1 = 1$ ist. \square

Charakteristika der Greedy-Methode:

1. Der erste Schritt der Greedy-Lösung ist nicht falsch. Es gibt eine optimale Lösung, die als Fortsetzung dieses ersten Schrittes konstruiert werden kann.
2. „Prinzip der optimalen Substruktur“: Entfernt man aus einer optimalen Lösung die erste(n) Komponente(n), so bleibt als Rest die optimale Lösung eines Teilproblems.
3. Der Korrektheitsbeweis wird mit vollständiger Induktion geführt.

3.2 Kürzeste Wege mit einem Startknoten: Der Algorithmus von Dijkstra

Definition

- Ein *gewichteter Digraph* ist ein Tripel $G = (V, E, c)$, wo (V, E) ein Digraph und $c : E \rightarrow \mathbb{R}_0^+$ ist. c steht für „*cost*“; $c(v, w)$ kann als „Kosten“ oder „Länge“ der Kante (v, w) interpretiert werden.
- Ein gerichteter Weg $p = (v_0, v_1, \dots, v_k)$ in G hat **Kosten/Länge**

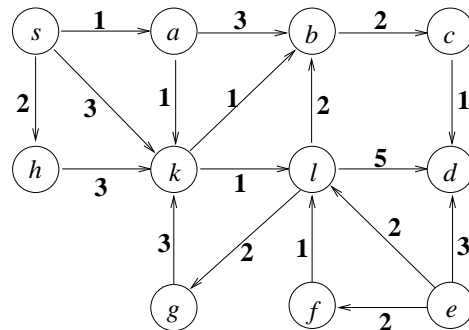
$$c(p) = \sum_{i=1}^k c(v_{i-1}, v_i).$$

- Der **(gerichtete) Abstand** von $v, w \in V$ ist

$$d(v, w) := \min\{c(p) \mid p \text{ Weg von } v \text{ nach } w\}$$

(= ∞ , falls kein Weg von v nach w existiert).

Beispiel:



$$c((s, a, b, c)) = 6, \quad c((s, a, k, b, c)) = 5, \quad d(s, c) = 5, \\ d(s, e) = \infty.$$

Hier betrachten wir einen Algorithmus für das Problem

„Single-Source-Shortest-Paths“

Kürzeste Wege von einem Startknoten aus.

Gegeben: gewichteter Digraph $G = (V, E, c)$ mit **nichtnegativen Kantenkosten** und $s \in V$.

Gesucht ist für jedes $v \in V$ der Abstand $d(s, v)$ und im Fall $d(s, v) < \infty$ ein Weg von s nach v der Länge $d(s, v)$.

Der **Algorithmus von Dijkstra** löst dieses Problem.

(Aussprache: „Deiks-tra“.)

(Zündende) Idee:

Eine Kante (v, w) wird als „Einbahnstraßen-Zündschnur“ der Länge $c(v, w)$ gedacht.

Die Zündschnüre brennen mit konstanter Geschwindigkeit 1 pro Sekunde.

Zum Zeitpunkt $t_0 = 0$ halten wir ein Zündholz an den Knoten s .

Alle Zündschnüre, die Kanten (s, v) entsprechen, beginnen zu brennen.

Das nächste interessante Ereignis: Zum Zeitpunkt $t = \min\{c(s, v) \mid (s, v) \in E\}$ erreicht das Feuer (mindestens) einen Knoten v .

Vereinbarung: Wenn das Feuer einen neuen Knoten v erreicht, beginnen ohne Verzögerung alle Zündschnüre zu brennen, die zu Kanten (v, w) gehören.

Wenn später das Feuer über andere Schnüre nochmals bei v ankommt, passiert nichts mehr.

Veranschaulichung: Spätere Folien. Ignoriere Notationen.

„Klar“:

Das Feuer erreicht den Knoten v genau zum Zeitpunkt $d(s, v)$.

Denn: Der Zeitraum $[0, d(s, v)]$ genügt genau, damit das Feuer einen kürzesten Weg durchwandern kann.

Wir bilden diese Idee algorithmisch nach. Dabei bemerken wir, dass nur die n Zeitpunkte $d(s, v)$, $v \in V$, wirklich interessant sind.

Daher: n Runden.

Dazwischen wandert das Feuer irgendwelche Kanten entlang, ohne dass im Prinzip viel passiert (selbst wenn ein schon erreichter Knoten nochmals erreicht wird).

O.B.d.A.: $V = \{1, \dots, n\}$.

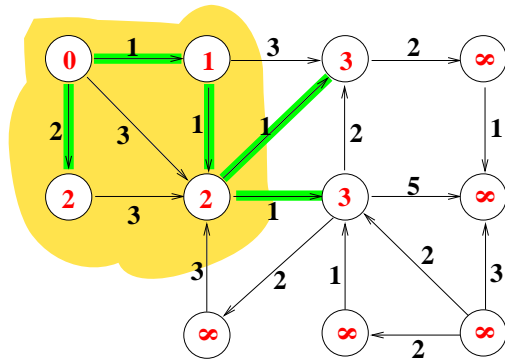
Der Algorithmus arbeitet in n Runden, eine für jeden Knoten, (in der Reihenfolge der $d(s, v)$!).

Der Algorithmus erzeugt eine (Daten-)Struktur und hält sie durch die Runden aufrecht.

V ist in zwei disjunkte Mengen S und $V - S$ zerlegt (veränderlich).

(Die Knoten in S wurden schon vom Feuer erreicht, die Knoten in $V - S$ noch nicht.)

Beispiel:



Während des Ablaufs wird pro Runde ein Knoten zu S hinzugefügt. (Der nächste Knoten, der vom Feuer erreicht wird.)

Runde 0: Anfangs ist $S = \{s\}$, man hat also stets $s \in S$.

Ein Hilfsarray speichert Informationen zur Verwaltung der Abstände:

$\text{dist}[1..n]$: speichert eine obere Schranke für $d(s, v)$.

Genauere Idee:

$\text{dist}[v]$ ist die Länge eines kürzesten S -Weges von s nach v .

Ein **S -Weg** nach v ist ein Weg $p = (s = v_0, v_1, \dots, v_{t-1}, v_t = v)$ mit $v_0, v_1, \dots, v_{t-1} \in S$.

(Alle Knoten auf dem Weg bis auf den letzten hat das Feuer schon erreicht; momentan wandert das Feuer die Kante (v_{t-1}, v) entlang; oder auch Knoten v ist schon erreicht.)

Wenn es keinen S -Weg nach v gibt, ist $\text{dist}[v] = \infty$.

- Aus Sicht von $v \in S$:

Das Feuer hat v schon erreicht; es gilt $\text{dist}[v] = d(s, v)$.

- Aus Sicht von $v \in V - S$, mit $\text{dist}[v] = \infty$:

Das Feuer hat noch keinen Knoten w erreicht, so dass (w, v) eine Kante ist.

- Aus Sicht von $v \in V - S$, mit $\text{dist}[v] < \infty$:

Das Feuer hat v noch nicht erreicht; es gibt aber einen Knoten w , so dass die Zündschnur von w nach v schon brennt.

Wann wird das Feuer v **spätestens** erreichen?

$$\text{dist}[v] = \min\{\text{dist}[w] + c(w, v) \mid w \in S, (w, v) \in E\}.$$

Wenn $\text{dist}[v]$ für alle $v \in V - S$ die richtige Bedeutung hat, kann man ablesen, welchen Knoten das Feuer als nächstes erreicht, nämlich

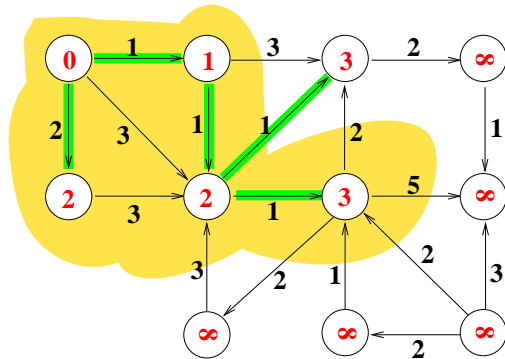
einen Knoten u , der $\text{dist}[v]$ minimiert.

(Es könnte auch mehrere solche Knoten geben. Man wählt einen davon.)

Wir fügen Knoten u zu S hinzu (er brennt an).

Was ist noch zu tun?

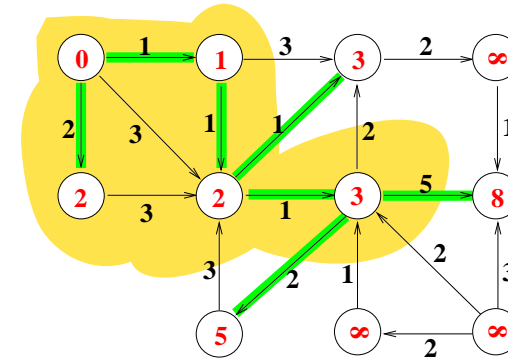
Beispiel:



Ab jetzt können S -Wege auch den Knoten u benutzen.

Wir werden später sehen, dass es nur auf **neue** S -Wege ankommt, bei denen u der letzte Knoten in S ist.

Beispiel:



Das heißt, es sind nur Knoten $v \in V - S$ betroffen, die von u aus über eine Kante (u, v) erreichbar sind.

Für solche Knoten müssen wir eventuell den `dist`-Wert aktualisieren:

$$\text{dist}[v] \leftarrow \min\{\text{dist}[v], \text{dist}[u] + c(u, v)\}.$$

(Das Feuer wandert jetzt auch entlang der Kante (u, v) .)

Der bisher entwickelte Algorithmus berechnet schon die Länge der kürzesten Wege (also die *Zeitpunkte*, zu denen das Feuer jeden Knoten erreicht).

Algorithmus Dijkstra-Distanzen (G, s) (Rohfassung)

Eingabe: gewichteter Digraph $G = (V, E, c)$, Startknoten s

Ausgabe: Länge der kürzesten Wege von s zu den Knoten in G

- (1) $S \leftarrow \{s\}$;
- (2) $\text{dist}[s] \leftarrow 0$;
- (3) **for** $v \in V - \{s\}$ **do** $\text{dist}[v] \leftarrow \infty$;
- (4) **for** $v \in V - \{s\}$ mit $(s, v) \in E$ **do**
- (5) $\text{dist}[v] \leftarrow c(s, v)$;
- (6) **while** $\exists u \in V - S: \text{dist}[u] < \infty$ **do**
- (7) $u \leftarrow$ ein solcher Knoten u mit minimalem $\text{dist}[u]$;
- (8) $S \leftarrow S \cup \{u\}$;
- (9) **for** $v \in V - S$ mit $(u, v) \in E$ **do**
- (10) $\text{dist}[v] \leftarrow \min\{\text{dist}[v], \text{dist}[u] + c(u, v)\}$;
- (11) **Ausgabe:** Das Array $\text{dist}[1..n]$.

Behauptung: Der Algorithmus gibt in $\text{dist}[v]$ für jeden Knoten $v \in V$ den Wert $d(s, v)$ aus.

Beweis: Wir zeigen durch Induktion über Runden die folgende Invariante:

(1) Für $v \in S$ ist $\text{dist}[v] = d(s, v)$.

(2) für $v \notin S$ ist $\text{dist}[v]$ Länge eines kürzesten S -Weges von s nach v , falls ein solcher Weg existiert; sonst ist $\text{dist}[v] = \infty$.

Nach der 1. Runde (Zeilen (1)–(5)) gilt die Invariante.

Nehmen wir nun an, die Invariante gilt nach Runde $t - 1$.

Es folgt Durchlauf Nummer t durch die **while**-Schleife.

1. Fall: Es gibt keinen Knoten $v \in V - S$ mit $\text{dist}[v] < \infty$.

\Rightarrow es gibt keinen Durchlauf t ; die Datenstruktur ändert sich nicht mehr.

Es gibt aber auch keinen S -Weg mehr, der zu einem Knoten $v \in V - S$ führt.

D. h.: Die Knoten in $V - S$ sind von s aus nicht erreichbar, es gilt $d(s, v) = \infty$ für diese Knoten.

Für die Knoten in S gilt $\text{dist}[s] = d(s, v)$ nach I.V.

Aussagen (1) und (2) gelten.

2. Fall: In Zeile (7) wird ein Knoten $u \in V - S$ gewählt. Dieses u soll zu S hinzugefügt werden.

Wir müssen zeigen (1): $\text{dist}[u] = d(s, u)$.

Nach I.V. ist $\text{dist}[u] < \infty$ Länge eines S -Weges von s nach u , also gilt bestimmt $\text{dist}[u] \geq d(s, u)$.

Sei nun $p = (s = v_0, v_1, \dots, v_t = u)$ ein beliebiger Weg von s nach u .

p beginnt in S und endet in $V - S$, also gibt es ein r mit:

$s = v_0, v_1, \dots, v_{r-1} \in S, v_r \notin S$.

Letzteres ist ein S -Weg p' von s nach v_r mit Länge

$$\begin{aligned} c(p') &= c(v_0, v_1) + \dots + c(v_{r-1}, v_r) \\ &\geq d(s, v_{r-1}) + c(v_{r-1}, v_r) \stackrel{\text{I.V.}}{=} \text{dist}[v_{r-1}] + c(v_{r-1}, v_r). \end{aligned}$$

Die letzte Summe wird aber in Zeile (7) bei der Suche nach dem Minimum berücksichtigt – das heißt:

$$c(p) \geq c(p') \geq \text{dist}[v_{r-1}] + c(v_{r-1}, v_r) \geq \text{dist}[u].$$

Achtung:

Wir benutzen hier, dass die Kantenlängen nicht negativ sind.

Das heißt, dass der S -Weg von s nach u keinesfalls länger ist als p .

Da p beliebig war, hat der S -Weg von s nach u mit Länge $\text{dist}[u]$ kleinstmögliche Länge, und $\text{dist}[u] = d(s, u)$.

Damit gilt Aussage (1) auch für den Knoten u , der neu zu S hinzukommt.

Aussage (2):

Wir überlegen: Wenn $v \notin S \cup \{u\}$ und $(u, v) \notin E$ ist, dann kann kein $(S \cup \{u\})$ -Weg von s nach v den Knoten u benutzen. Für solche Knoten v gilt also (2) für $(S \cup \{u\})$ direkt nach I.V.

Nun sei $v \notin S \cup \{u\}$ und $(u, v) \in E$.

Es könnte nun neue $(S \cup \{u\})$ -Wege nach v geben, die (u, v) als letzte Kante haben.

Zeilen (9) und (10) testen, ob hierdurch ein kürzerer $(S \cup \{u\})$ -Weg entsteht, und ersetzen $\text{dist}[v]$ durch den besseren Wert, falls nötig.

Dadurch bleibt auch Invariante (2) erhalten. \square

Wir wollen aber eigentlich nicht nur die Distanzen $d(s, v)$, sondern kürzeste Wege berechnen.

Idee: für jeden Knoten v merken wir uns, über welche Kante (w, v) das Feuer den Knoten v erreicht hat.

Wenn wir diese „Vorgänger“-Information benutzen und von v immer weiter rückwärts laufen, erhalten wir einen kürzesten Weg.

Am einfachsten lässt sich diese Information erstellen, wenn man auch gleich für jeden Knoten mit $\text{dist}[v] < \infty$ einen Knoten $w \in S$ notiert, der der letzte S -Knoten auf einem kürzesten S -Weg nach v ist.

Datenstruktur: $p[1..n]$: speichert Vorgänger für v in S (falls es einen gibt).

Initialisierung in Runde 1:

(4) **for** $v \in V - \{s\}$ mit $(s, v) \in E$ **do**

(5a) $\text{dist}[v] \leftarrow c(s, v)$;

(5b) $p[v] \leftarrow s$;

In den späteren Runden:

Prozedur Update(u)

(9) **for** $v \in V - S$ mit $(u, v) \in E$ **do**

(10a) $dd \leftarrow \text{dist}[u] + c(u, v)$;

(10b) **if** $dd < \text{dist}[v]$ **then**

(10c) $\text{dist}[v] \leftarrow dd$;

(10d) $p[v] \leftarrow u$;

Dijkstra(G, s)

Eingabe: gewichteter Digraph $G = (V, E, c)$, Startknoten s

Ausgabe: Länge $d(s, v)$ der kürzesten Wege, Vorgängerknoten $p(v)$

(1) $S \leftarrow \{s\}$; $p[s] \leftarrow -2$;

(2) $\text{dist}[s] \leftarrow 0$;

(3) **for** $v \in V - \{s\}$ **do** $\text{dist}[v] \leftarrow \infty$; $p[v] \leftarrow -1$;

(4) **for** $v \in V - \{s\}$ mit $(s, v) \in E$ **do**

(5) $\text{dist}[v] \leftarrow c(s, v)$; $p[v] \leftarrow s$;

(6) **while** $\exists v \in V - S: \text{dist}[v] < \infty$ **do**

(7) $u \leftarrow$ ein solcher Knoten v mit minimalem $\text{dist}[v]$;

(8) $S \leftarrow S \cup \{u\}$;

(9) **for** $v \in V - S$ mit $(u, v) \in E$ **do**

(10a) $dd \leftarrow \text{dist}[u] + c(u, v)$;

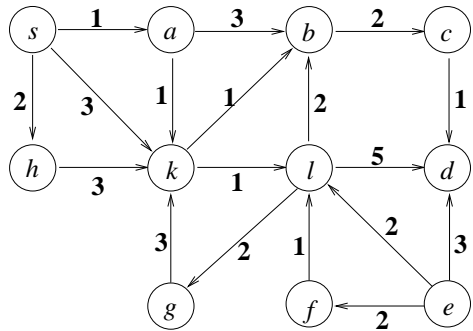
(10b) **if** $dd < \text{dist}[v]$ **then**

(10c) $\text{dist}[v] \leftarrow dd$;

(10d) $p[v] \leftarrow u$;

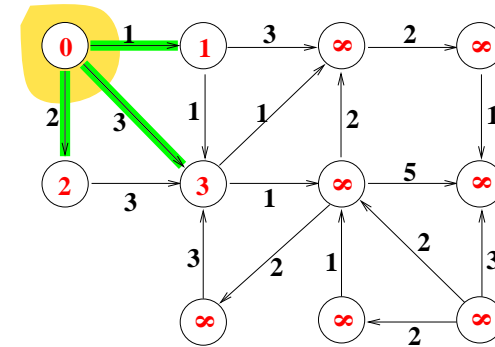
(11) **Ausgabe:** $\text{dist}[1..n]$ und $p[1..n]$.

Ablauf des Algorithmus von Dijkstra



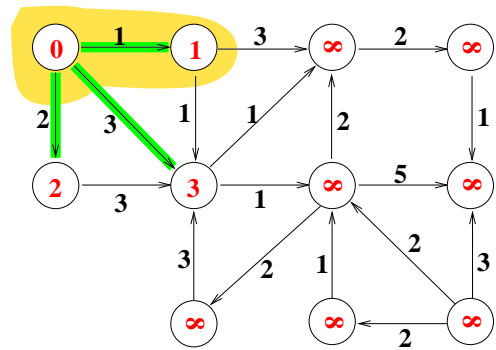
Der Ausgangsgraph.

Ablauf des Algorithmus von Dijkstra



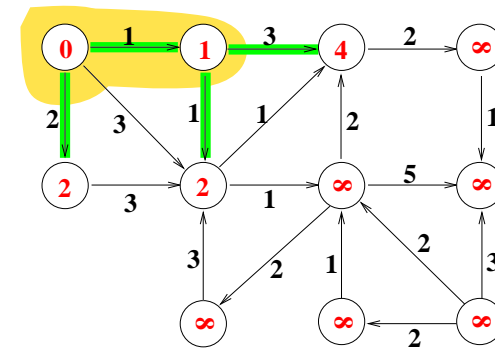
Nach der Initialisierung (Zeilen (1)–(5))

Ablauf des Algorithmus von Dijkstra



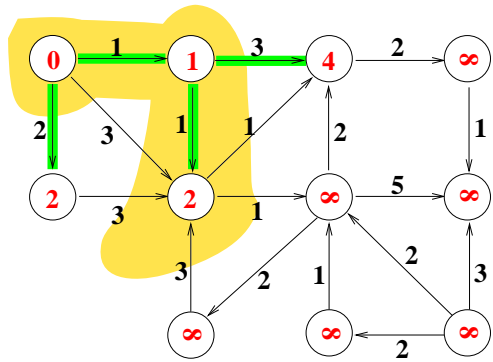
$u = a$, Zeile (8)

Ablauf des Algorithmus von Dijkstra



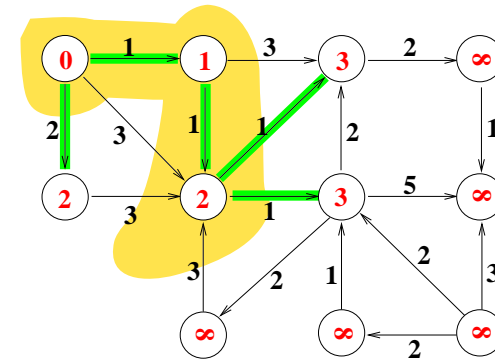
$u = a$, Zeilen (9)–(10d)

Ablauf des Algorithmus von Dijkstra



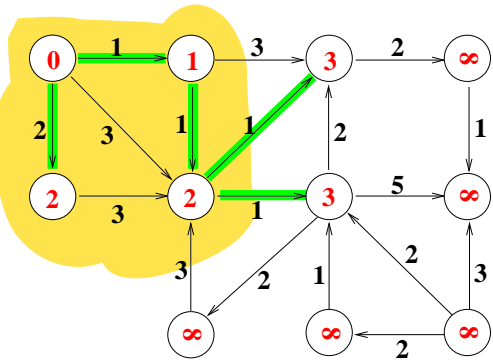
$u = k$, Zeile (8)

Ablauf des Algorithmus von Dijkstra



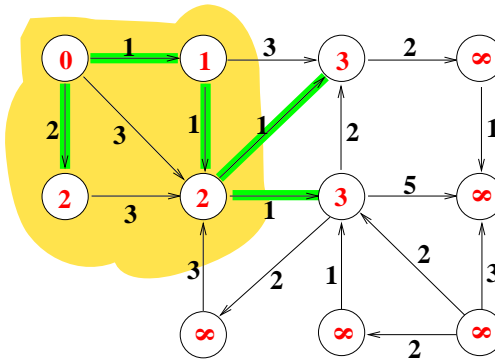
$u = k$, Zeilen (9)–(10d)

Ablauf des Algorithmus von Dijkstra



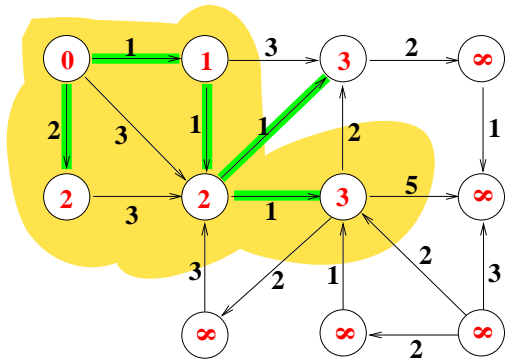
$u = h$, Zeile (8)

Ablauf des Algorithmus von Dijkstra



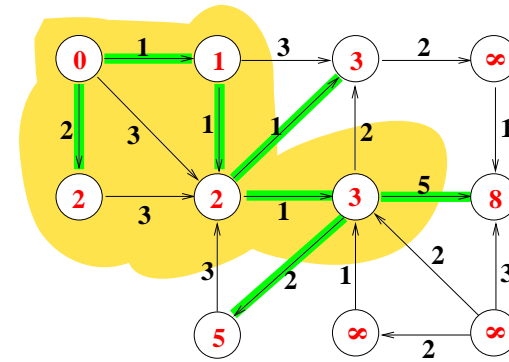
$u = h$, (9)–(10d) ändern nichts

Ablauf des Algorithmus von Dijkstra



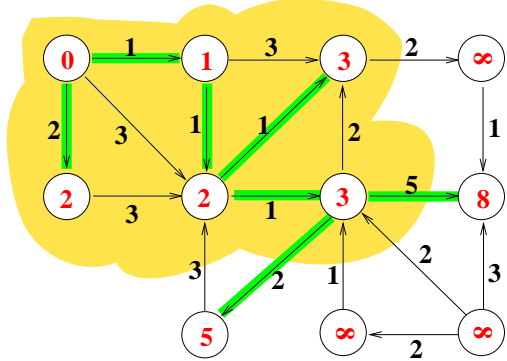
$u = l$, Zeile (8)

Ablauf des Algorithmus von Dijkstra



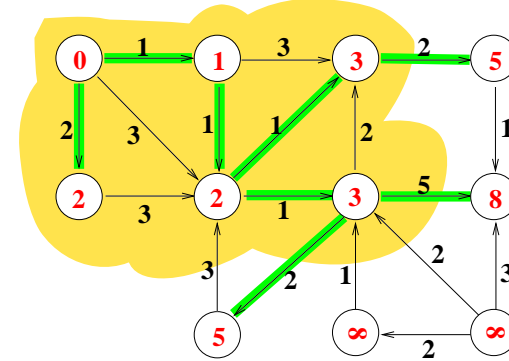
$u = l$, Zeilen (9)–(10d)

Ablauf des Algorithmus von Dijkstra



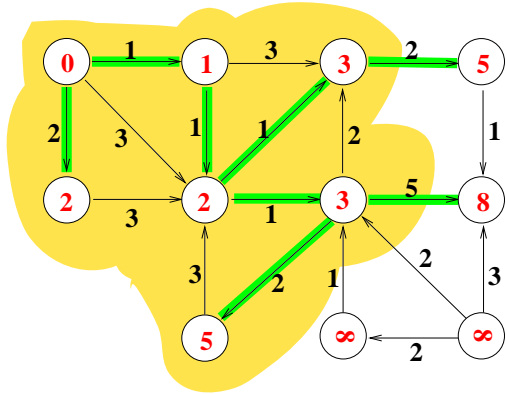
$u = b$, Zeile (8)

Ablauf des Algorithmus von Dijkstra



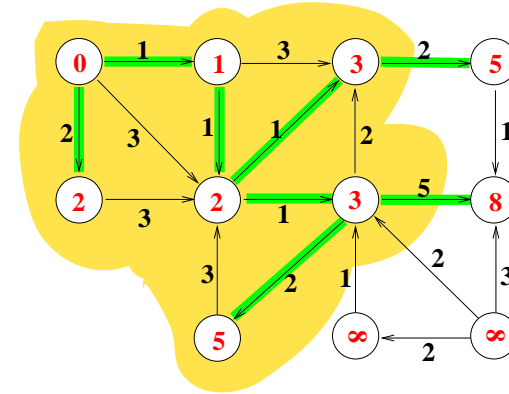
$u = b$, Zeilen (9)–(10d)

Ablauf des Algorithmus von Dijkstra



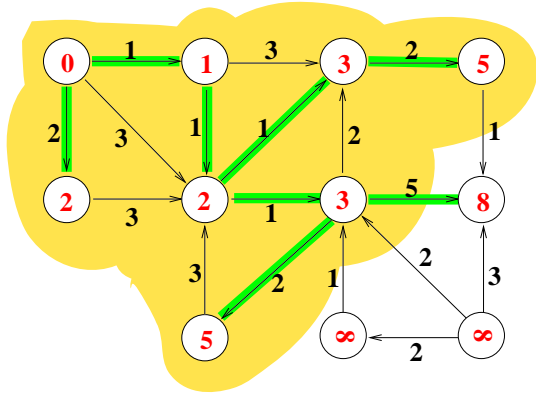
$u = g$, Zeile (8)

Ablauf des Algorithmus von Dijkstra



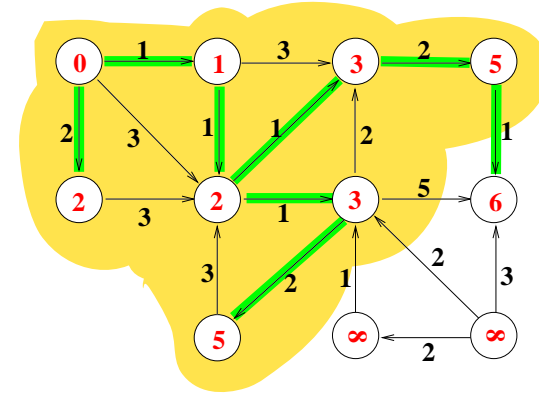
$u = g$, Zeilen (9)–(10d) ändern nichts

Ablauf des Algorithmus von Dijkstra



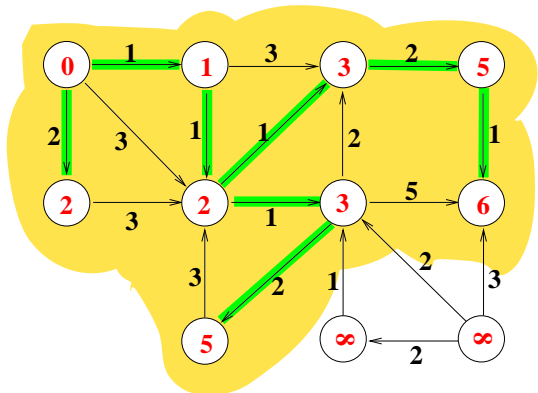
$u = c$, Zeile (8)

Ablauf des Algorithmus von Dijkstra



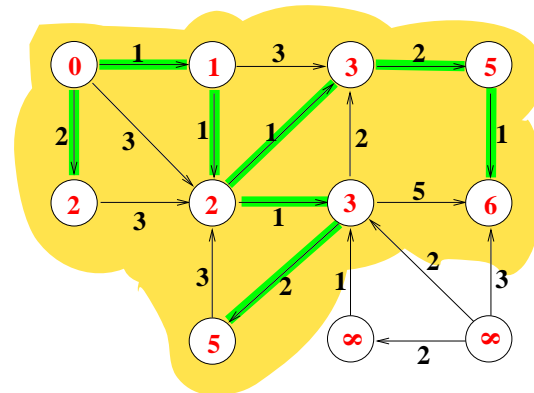
$u = c$, Zeilen (9)–(10d)

Ablauf des Algorithmus von Dijkstra



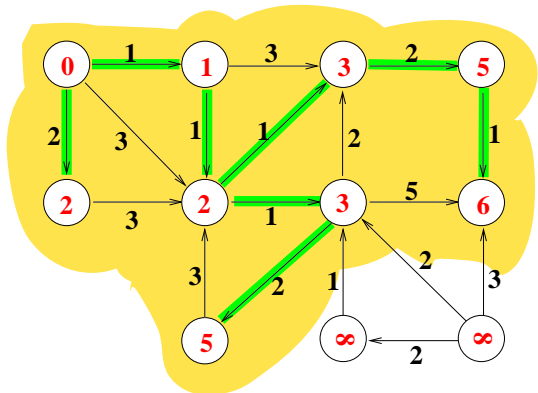
$u = d$, Zeile (8)

Ablauf des Algorithmus von Dijkstra



$u = d$, Zeilen (9)–(10d) ändern nichts.

Ablauf des Algorithmus von Dijkstra



Alle $v \in V - S$ erfüllen $\text{dist}[v] = \infty$: **Ende.**

Nach dem Algorithmus ist klar, dass $p[v] \neq -1$ („undefiniert“) genau dann gilt, wenn $\text{dist}[v] < \infty$.

$p[v] = -2$ („Startknoten, hat keinen Vorgänger“) gilt nur für $v = s$.

Behauptung: Zusätzlich zu (1) und (2) erfüllt der Algorithmus von Dijkstra die folgende Invarianten:

(3) Wenn $v \in S$, dann ist $p[v] \neq -1$ und für $v \neq s$ ist $p[v]$ letzter Knoten auf einem kürzesten Weg von s nach v ; ein solcher verläuft vollständig in S .

(4) Wenn $v \notin S$, dann gilt: $p[v] \neq -1$ genau dann wenn $\text{dist}[v] < \infty$ gilt.

In diesem Fall: $p[v] \in S$ und $p[v]$ ist letzter S -Knoten auf einem S -Weg von s nach v kürzester Länge $\text{dist}[v]$.

Beweis von (3) und (4): Induktion über Runden.

Nach Runde 1 sind nur s und die Knoten v mit $(s, v) \in E$ betroffen; die Aussagen sind klar.

Nun betrachten wir Runde t durch die **while**-Schleife.

1. Fall: Es gibt keinen Knoten $v \in V - S$ mit $\text{dist}[v] < \infty$.

\Rightarrow es gibt keinen Durchlauf t ; die Datenstruktur ändert sich nicht mehr.

(3) und (4) gelten nach I.V.

2. Fall: Es gibt einen Knoten $v \in V - S$ mit $\text{dist}[v] < \infty$.

Der Algorithmus wählt ein solches v mit minimalem $\text{dist}[v]$, genannt u .

Nach I.V. (4) ist $w = p[u] \in S$ letzter Knoten auf einem S -Weg kürzester Länge $\text{dist}[u]$. Nach (1), angewendet auf u , ist dies ein kürzester Weg von s nach u .

Damit gilt (3) auch für u .

Auch (4) folgert man aus der Induktionsvoraussetzung und (2) (Übung).

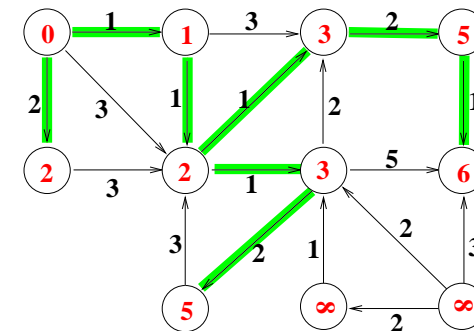
Ergebnis:

Wenn der Algorithmus von Dijkstra anhält, führen die $p[v]$ -Verweise von jedem Knoten v aus entlang eines kürzesten Weges (zurück) zu s .

Da die $p[v]$ -Verweise keinen Kreis bilden können (mit dem Schritt $S \leftarrow S \cup \{u\}$ wird der Verweis vom neuen S -Knoten u auf den S -Knoten $p[u]$ endgültig fixiert), bilden die Kanten $(p[v], v)$ einen Baum, einen

Baum der kürzesten Wege.

Beispiel:



Die Laufzeitanalyse erfordert noch einige Implementierungsdetails, für die Suche nach dem Knoten $v \in V - S$ mit minimalem $\text{dist}[v]$.

Priority-Queue – Vorrangwarteschlange

Eine einfache Implementierung (mittels Heaps) ergibt: Zeit $O(\log n)$ für Minimumssuche und Updates in einer Priority-Queue mit maximal n Einträgen.

Vorläufige Mitteilung:

Der Algorithmus von Dijkstra mit der oben angedeuteten konkreten Implementierung benötigt Zeit $O((n + e) \log n)$.

Wieso steht der Algorithmus von Dijkstra unter der Überschrift „Greedy-Algorithmen“?

Er hält eine sehr geschickte Datenstruktur bereit,

wählt dann aber in jeder Runde nach dem Greedy-Muster als nächsten einzubeziehenden Knoten einen, dessen $\text{dist}[v]$ -Wert (also Länge des besten S -Weges) minimal ist.

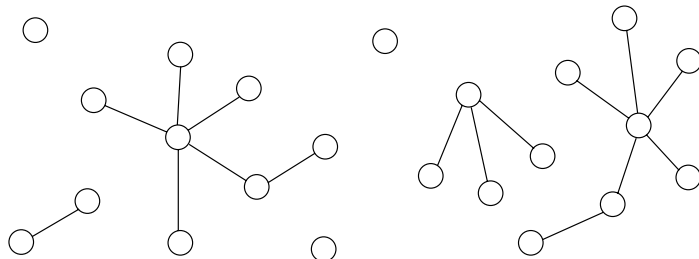
Zum Glück: Der kürzeste S -Weg ist dann auch ein kürzester Weg im gesamten Graphen.

3.3 Minimale Spannbäume: Algorithmus von Jarník/Prim

Definition

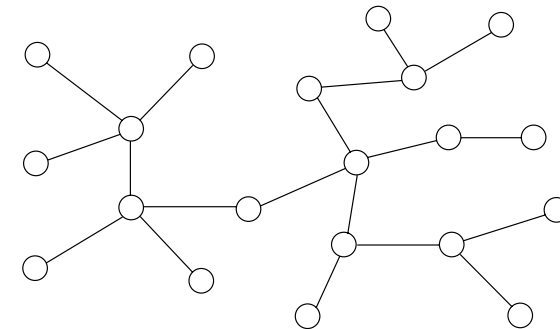
(a) Ein Graph $G = (V, E)$ heißt **kreisfrei**, wenn er **keinen Kreis** besitzt.

Beispiel: Ein kreisfreier Graph:

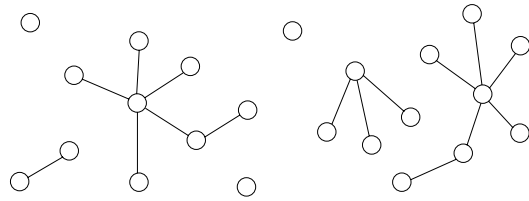


(b) Ein Graph G heißt ein **freier Baum** (oder nur **Baum**), wenn er zusammenhängend und kreisfrei ist.

Beispiel: Ein (freier) Baum:



20 Knoten, 19 Kanten.



Anmerkung: Die Zusammenhangskomponenten eines kreisfreien Graphen sind freie Bäume. Kreisfreie Graphen heißen daher auch **(freie) Wälder**.

Lemma Sei $G = (V, E)$ ein Graph mit $n = |V|$ und $m = |E|$. Dann gilt:

- (a) Wenn G kreisfrei ist, gilt $m \leq n - 1$.
- (b) Wenn G zusammenhängend ist, gilt $m \geq n - 1$.

Fundamental-Lemma über freie Bäume

Wenn $G = (V, E)$ ein Graph ist, mit Knotenzahl $n = |V|$ und Kantenzahl $m = |E|$, dann sind folgende Aussagen äquivalent:

- (a) G ist ein Baum.
- (b) G ist kreisfrei und $m = n - 1$.
- (c) G ist zusammenhängend und $m = n - 1$.
- (d) Zu jedem Paar u, v von Knoten gibt es genau einen einfachen Weg von u nach v .
- (e) G ist kreisfrei, aber das Hinzufügen einer beliebigen weiteren Kante erzeugt einen Kreis (G ist „maximal kreisfrei“).
- (f) G ist zusammenhängend, aber das Entfernen einer beliebigen Kante erzeugt einen nicht zusammenhängenden Restgraphen (G ist „minimal zusammenhängend“).

Beweis des Fundamentallemmas: Für Interessierte in einer eigenen Notiz nachzulesen (siehe Webseite).

Wir benutzen das Fundamental-Lemma über freie Bäume in der folgenden Weise:

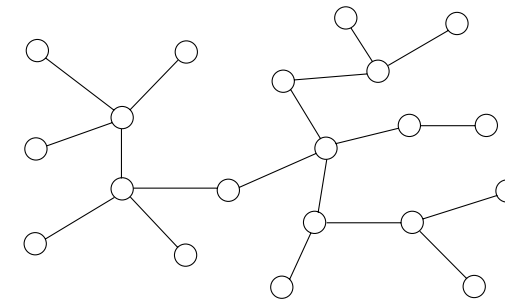
- (1) Wenn G ein Baum ist, erfüllt G alle genannten Eigenschaften (a)–(f).
- (2) Wenn wir eine der Eigenschaften nachweisen, hat sich G als Baum erwiesen.

Aussagen, die aus dem Fundamental-Lemma über Bäume folgen:

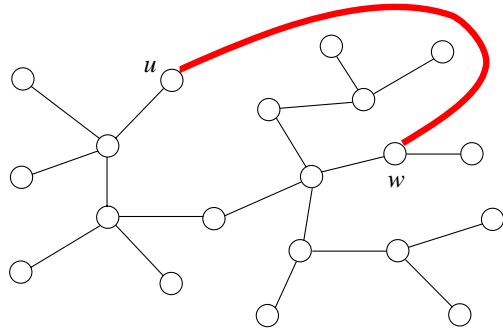
G sei Baum mit n Knoten. Dann gilt:

- (1) G hat $n - 1$ Kanten.

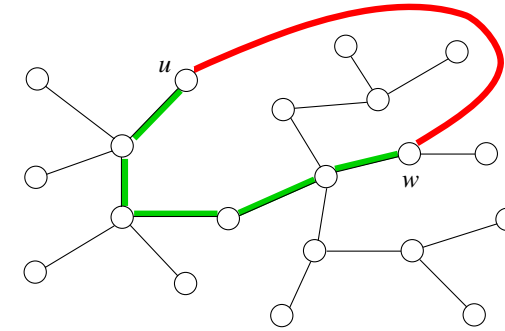
- (2) Wenn man zu G eine Kante (u, w) hinzufügt, entsteht genau ein Kreis (aus (u, w) und dem eindeutigen **Weg** von u nach w in G).



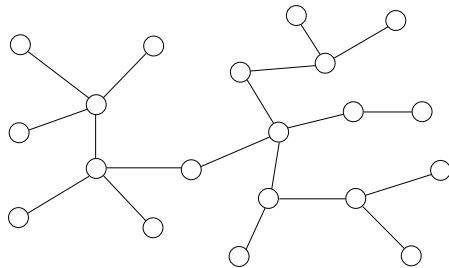
(2) Wenn man zu G eine Kante (u, w) hinzufügt, entsteht genau ein Kreis
 (aus (u, w) und dem eindeutigen **Weg** von u nach w in G).



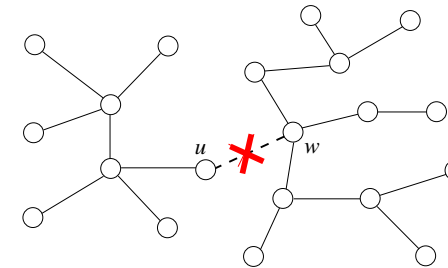
(2) Wenn man zu G eine Kante (u, w) hinzufügt, entsteht genau ein Kreis
 (aus (u, w) und dem eindeutigen **Weg** von u nach w in G).



(3) Wenn man aus G eine Kante (u, w) streicht, zerfällt der Graph in 2 Komponenten
 $U = \{v \in V \mid v \text{ von } u \text{ aus über Kanten aus } E - \{(u, w)\} \text{ erreichbar}\};$
 $W = \{v \in V \mid v \text{ von } w \text{ aus über Kanten aus } E - \{(u, w)\} \text{ erreichbar}\}.$



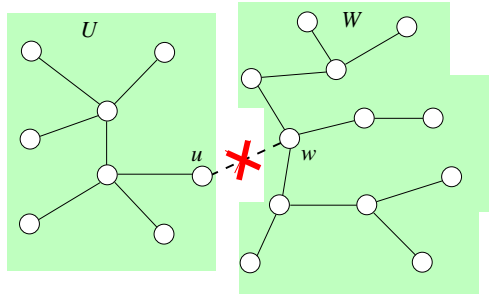
(3) Wenn man aus G eine Kante (u, w) streicht, zerfällt der Graph in 2 Komponenten
 $U = \{v \in V \mid v \text{ von } u \text{ aus über Kanten aus } E - \{(u, w)\} \text{ erreichbar}\};$
 $W = \{v \in V \mid v \text{ von } w \text{ aus über Kanten aus } E - \{(u, w)\} \text{ erreichbar}\}.$



(3) Wenn man aus G eine Kante (u, w) streicht, zerfällt der Graph in 2 Komponenten

$U = \{v \in V \mid v \text{ von } u \text{ aus über Kanten aus } E - \{(u, w)\} \text{ erreichbar}\};$

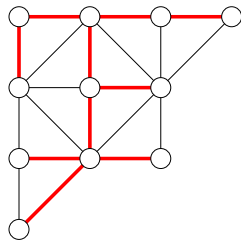
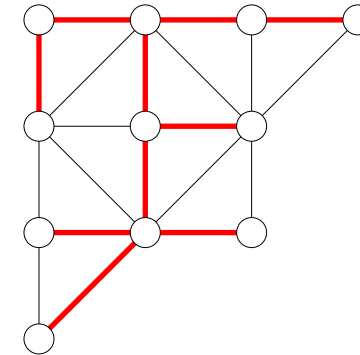
$W = \{v \in V \mid v \text{ von } w \text{ aus über Kanten aus } E - \{(u, w)\} \text{ erreichbar}\}.$



Definition

Es sei $G = (V, E)$ ein zusammenhängender Graph. Eine Menge $T \subseteq E$ von Kanten heißt ein **Spannbaum** für G , wenn (V, T) ein Baum ist.

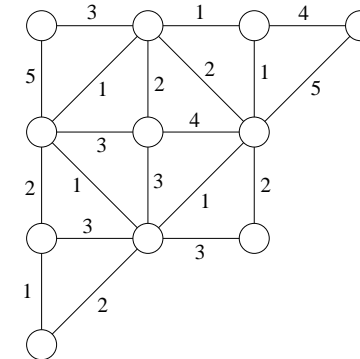
Beispiel:



Klar: Jeder zusammenhängende Graph hat einen Spannbaum.
(Iterativer Prozess: Starte mit E . Solange es Kanten gibt, die auf einem Kreis liegen, entferne eine solche Kreiskante.
Der Zusammenhang bleibt stets erhalten; schließlich muss der Rest-Graph kreisfrei sein.)

Definition

Es sei $G = (V, E, c)$ ein **gewichteter Graph**, d.h. $c: E \rightarrow \mathbb{R}$ ist eine „Gewichtsfunktion“ oder „Kostenfunktion“.

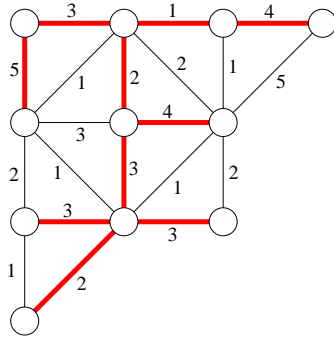


Kantenkosten modellieren:
Herstellungskosten – Leitungsmiete – . . .

(a) Jeder Kantenmenge $E' \subseteq E$ wird durch

$$c(E') := \sum_{e \in E'} c(e)$$

ein **Gesamtgewicht** zugeordnet.



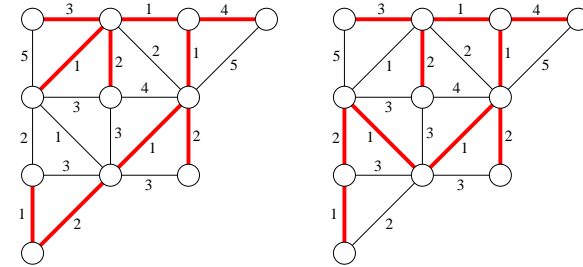
Gesamtgewicht: $3 + 1 + 4 + 5 + 2 + 4 + 3 + 3 + 3 + 2 = 30$.

(b) Sei G zusammenhängend. Ein Spannbaum $T \subseteq E$ für G heißt ein **minimaler Spannbaum**, wenn

$$c(T) = \min\{c(T') \mid T' \text{ Spannbaum von } G\},$$

d.h. wenn er minimale Kosten unter allen Spannbäumen hat.

Abkürzung: **MST** („minimum spanning tree“).



Zwei minimale Spannbäume, jeweils mit Gesamtgewicht 18.

Klar: Jeder Graph besitzt einen MST.

(Es gibt nur endlich viele Spannbäume.)

Achtung: Es kann mehrere verschiedene MSTs geben (die alle dasselbe Gesamtgewicht haben).

Aufgabe: Zu gegebenem $G = (V, E, c)$ finde einen MST T .

Hier: **„Algorithmus von Jarník/Prim“**

„Algorithmus von Kruskal“

Algorithmenparadigma: **„greedy“** („gierig“)

Baue Lösung **Schritt für Schritt** auf

(hier: wähle eine Kante für T nach der anderen)

Treffe in jedem Schritt die **(lokal) günstigste Entscheidung**

Algorithmus von Jarník/Prim: Ähnlichkeit zum Dijkstra-Algorithmus.

S: Menge von Knoten. Enthält die „bisher erreichten“ Knoten.

R: Menge von Kanten. Enthält die „bisher gewählten“ Kanten.

(1) Wähle einen beliebigen (Start-)Knoten $s \in V$.

$$S \leftarrow \{s\};$$

$$R \leftarrow \emptyset;$$

(2) Wiederhole $(n - 1)$ -mal:

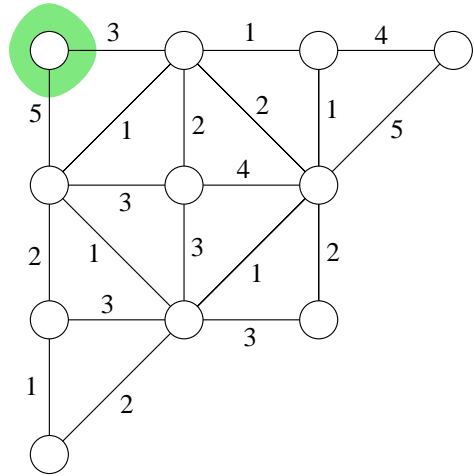
Finde $w \in S$ und $u \in V - S$, so dass $c(w, u)$ **minimal** unter allen Werten $c(w', u')$, $w' \in S$, $u' \in V - S$, ist.

$$S \leftarrow S \cup \{u\};$$

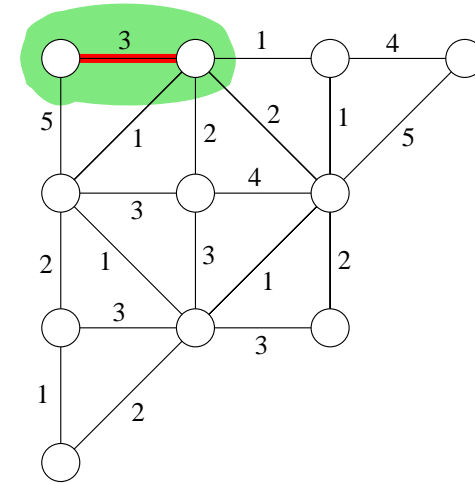
$$R \leftarrow R \cup \{(w, u)\};$$

(3) Ausgabe: R .

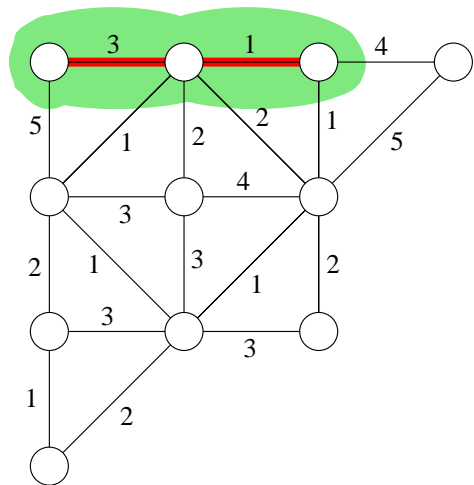
Beispiel (Jarník/Prim):



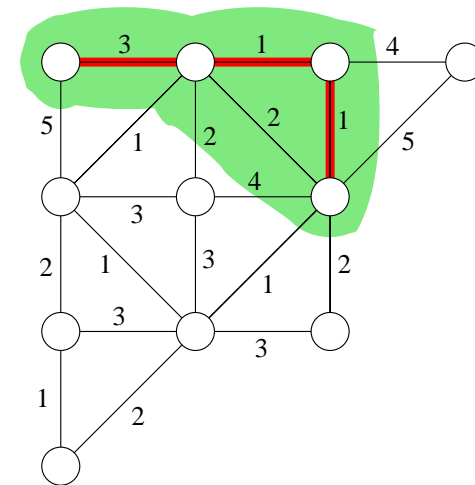
Beispiel (Jarník/Prim):



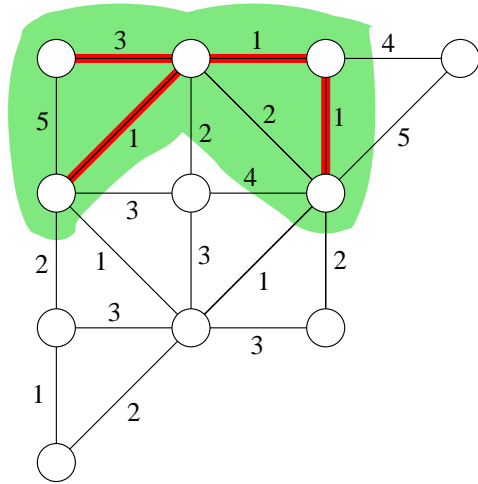
Beispiel (Jarník/Prim):



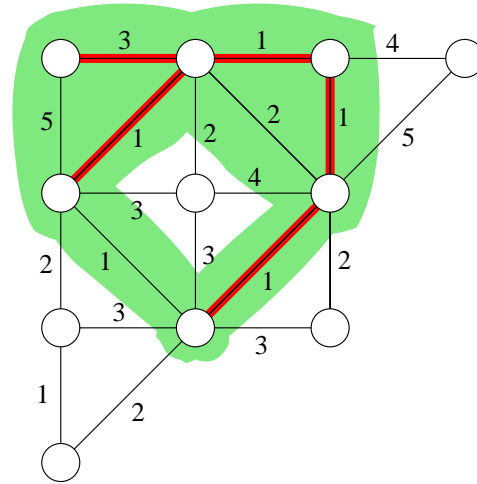
Beispiel (Jarník/Prim):



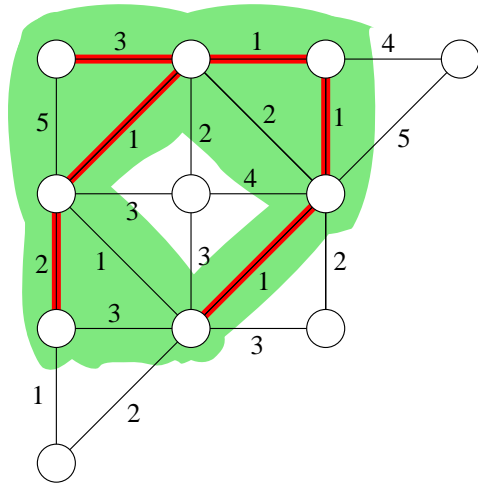
Beispiel (Jarník/Prim):



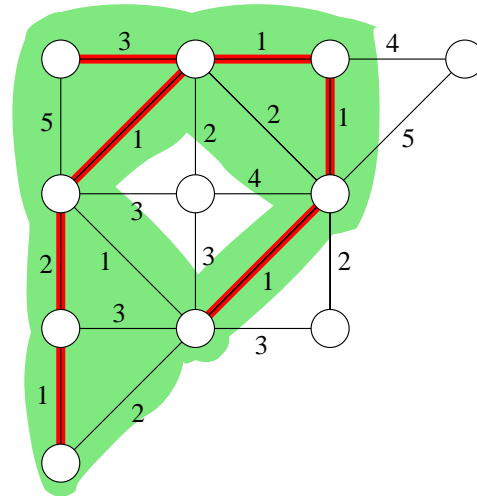
Beispiel (Jarník/Prim):



Beispiel (Jarník/Prim):



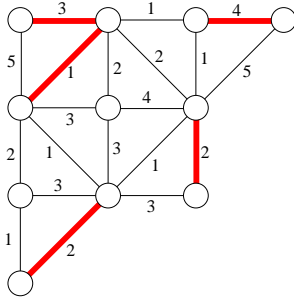
Beispiel (Jarník/Prim):



Die Schnitteigenschaft

Definition

Eine Menge $R \subseteq E$ heißt **erweiterbar** (zu einem MST), wenn es einen MST T mit $R \subseteq T$ gibt.

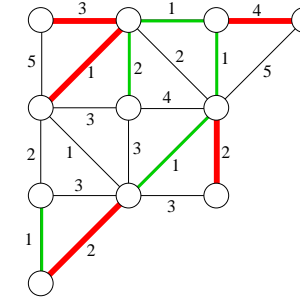


R ist erweiterbare Menge, denn . . .

Die Schnitteigenschaft

Definition

Eine Menge $R \subseteq E$ heißt **erweiterbar** (zu einem MST), wenn es einen MST T mit $R \subseteq T$ gibt.



. . . es gibt einen MST T , der R erweitert.

Die Schnitteigenschaft

Behauptung (Schnitteigenschaft):

Wenn $R \subseteq E$ erweiterbar ist und $(S, V - S)$ ein Schnitt, so dass keine Kante aus R von S nach $V - S$ verläuft, und wenn $e = (v, w)$, $v \in S$, $w \in V - S$ eine Kante ist, die den Wert $c((u, x))$, $u \in S$, $x \in V - S$, **minimiert**, dann ist auch $R \cup \{e\}$ erweiterbar.

Die Schnitteigenschaft

Beweis der Schnitteigenschaft:

Sei $R \subseteq E$, sei $T \supseteq R$ ein MST;

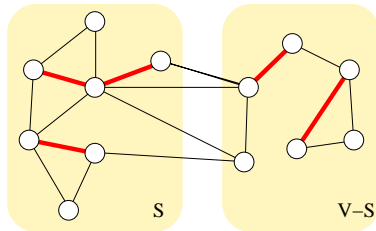
sei $(S, V - S)$ ein Schnitt, so dass keine Kante aus R von S nach $V - S$ verläuft;

sei $e \in S \times (V - S)$ mit minimalem $c(e)$.

1. Fall: $e \in T$. Dann gilt $R \cup \{e\} \subseteq T$, also ist $R \cup \{e\}$ zulässig, fertig.

Die Schnitteigenschaft

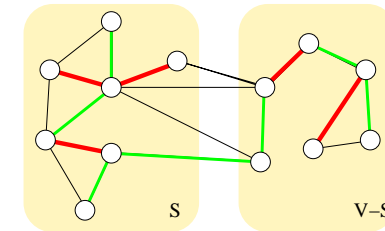
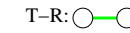
2. Fall: $e \notin T$.



Eine erweiterbare Menge R .

Die Schnitteigenschaft

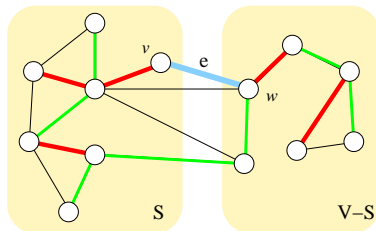
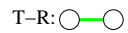
2. Fall: $e \notin T$.



MST T mit $R \subseteq T$.

Die Schnitteigenschaft

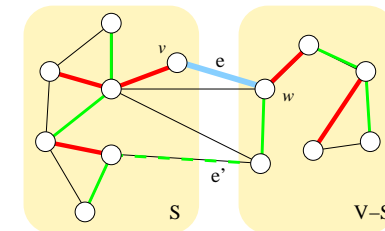
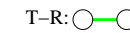
2. Fall: $e \notin T$.



$e = (v, w)$ minimiert $c((u, x))$, $u \in S$, $x \in V - S$.

Die Schnitteigenschaft

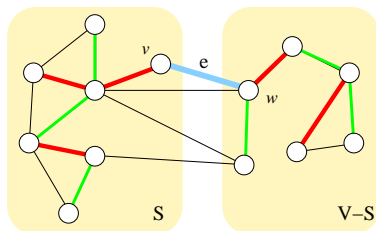
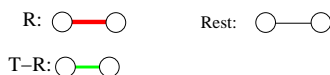
2. Fall: $e \notin T$.



Weg in T von v nach w wechselt von S nach $V - S$ bei Kante e' .
Es entsteht ein Kreis, auf dem e und e' liegen.

Die Schnitteigenschaft

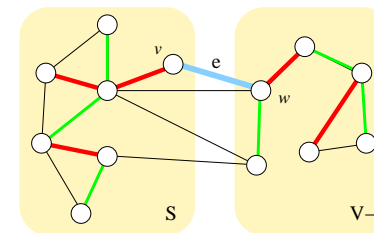
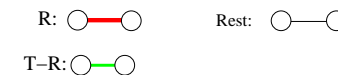
2. Fall: $e \notin T$.



Entferne e' , füge e hinzu: $T_e := (T - \{e'\}) \cup \{e\}$.
 Neuer Spannbaum $T_e \supseteq R \cup \{e\}$.

Die Schnitteigenschaft

2. Fall: $e \notin T$.



Weil $c(e) \leq c(e')$ (Minimalität von $c(e)$ über den Schnitt $(S, V - S)$): $c(T_e) \leq c(T)$, also ist T_e wieder ein MST.

Also: $R \cup \{e\}$ ist erweiterbar, q.e.d.

Korrektheit des Algorithmus von Jarník/Prim:

R_i : Kantenmenge (Größe i), die nach Runde i in R steht.

S_i : Knotenmenge (Größe $i + 1$), die nach Runde i in S steht.

Weil in jeder Runde ein Knoten und eine Kante hinzukommt, und man an die schon vorhandene Knotenmenge anschließt, ist jedes R_i kreisfrei und zusammenhängend, also ein Baum. Die Kanten von R_i verbinden genau die Knoten von S_i .

Zu zeigen: R_{n-1} ist ein MST für G .

Induktionsbehauptung IB(i):

R_i ist erweiterbar.

(Dies beweisen wir durch Induktion über $i = 0, 1, \dots, n - 1$.)

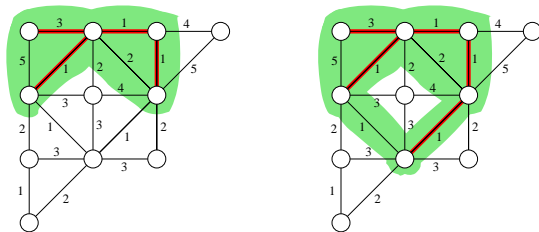
Dann besagt IB($n - 1$), dass $T \supseteq R_{n-1}$ ist für einen MST T . Weil alle Spannbäume $n - 1$ Kanten haben und R_{n-1} auch $n - 1$ Kanten hat, ist $T = R_{n-1}$, also ist die Ausgabe R_{n-1} ein MST.

IB(i): R_i ist erweiterbar.

I.A.: $i = 0$. — $R_0 = \emptyset$, und es gibt einen MST T für G .

I.V.: $1 \leq i \leq n - 1$ und R_{i-1} ist erweiterbar.

I.S.: $(S_{i-1}, V - S_{i-1})$ ist ein Schnitt, und keine Kante von R_{i-1} überquert diesen Schnitt.



Der Algorithmus von Jarník/Prim wählt unter allen Kanten, die den Schnitt überqueren, eine billigste Kante e , und bildet $R_i := R_{i-1} \cup \{e\}$.

Mit der Schnitteigenschaft folgt: R_i ist erweiterbar. \square

Einschub: Hilfs-Datenstrukturen

- **Priority-Queues** – Vorrangwarteschlangen

für: – Algorithmus von Dijkstra

– Algorithmus von Jarník/Prim

- **Union-Find-Datenstrukturen (später)**

für: – Algorithmus von Kruskal

Priority Queues

oder Vorrangwarteschlangen

Idee: (Daten mit) Schlüssel aus sortiertem Universum $(U, <)$ werden eingefügt und entnommen.

Beim Entnehmen wird immer der Eintrag mit dem kleinsten Schlüssel gewählt.

Schlüssel $\hat{=}$ „Prioritäten“

– je kleiner der Schlüssel, desto höher die Priorität –
Wähle stets den Eintrag mit höchster Priorität.

Anwendungen:

1) Prozessverwaltung in Multitask-System

Jeder Prozess hat einen Namen (eine „ID“) und eine Priorität (Zahl in \mathbb{N}).

(Üblich: **kleinere Zahlen** bedeuten **höhere Prioritäten**.)

Aktuell rechenbereite Prozesse befinden sich in der Prozesswarteschlange. Bei Freiwerden eines Prozessors (z. B. durch Unterbrechen eines Prozesses) wird einer der Prozesse mit höchster Priorität aus der Warteschlange entnommen und weitergeführt.

Anwendungen:

2) „Discrete Event Simulation“

System von Aktionen soll auf dem Rechner simuliert werden. Jeder **ausführbaren** Aktion A ist ein Zeitpunkt $t_A \in [0, \infty)$ zugeordnet.

Die Ausführung einer Aktion A kann neue Aktionen A' erzeugen oder ausführbar machen, mit neuen Zeitpunkten $t_{A'} > t_A$.

Ein Schritt: Wähle diejenige noch nicht ausgeführte Aktion mit dem frühesten Ausführungszeitpunkt und führe sie aus.

3) Innerhalb von Algorithmen (Dijkstra, Jarník/Prim, u. a.).

Bei Priority Queues zu berücksichtigen:

Mehrere Objekte mit derselben Priorität sind möglich, die man nicht identifizieren darf.

Mehrere identische Objekte sind möglich, die man nicht identifizieren darf:

Einträge sind nicht „blanke“ Schlüssel, sondern **mit Prioritäten versehene Datensätze**.

Beim Entnehmen: Alle Exemplare des kleinsten Schlüssels gleichberechtigt, einer gewinnt.

Beispiel: Datensatz: (n, L) , $n \in \mathbb{N}$,

$L \in \{A, \dots, Z\}$, Schlüssel ist der Buchstabe L .

Operation	innerer Zustand	Ausgabe
<i>empty</i>	\emptyset	–
<i>insert</i> (1, D)	(1, D)	–
<i>insert</i> (2, B)	{(1, D), (2, B)}	–
<i>insert</i> (3, D)	{(1, D), (2, B), (3, D)}	–
<i>insert</i> (4, E)	{(1, D), (2, B), (3, D), (4, E)}	–
<i>insert</i> (5, G)	{(1, D), (2, B), (3, D), (4, E), (5, G)}	–
<i>extract_min</i>	{(1, D), (3, D), (4, E), (5, G)}	(2, B)
<i>extract_min</i>	{(1, D), (4, E), (5, G)}	(3, D)
<i>insert</i> (4, E)	{(4, E), (1, D), (2, B), (3, D), (4, E)}	–
<i>insert</i> (2, D)	{(4, E), (2, D), (1, D), (4, E), (5, G)}	–
<i>extract_min</i>	{(4, E), (1, D), (4, E), (5, G)}	(2, D)
<i>isempty</i>	{(4, E), (1, D), (4, E), (5, G)}	<i>false</i>

Formale Beschreibung (siehe AuD-Vorlesung):

Datentyp: SimplePriorityQueue (Einfache Vorrangwarteschlange)

Signatur:

Sorten:

Keys

Data

PrioQ (* „Priority Queues“ *)

Boolean

Operatoren:

empty : \rightarrow *PrioQ*

isempty : *PrioQ* \rightarrow *Boolean*

insert : *Keys* \times *Data* \times *PrioQ* \rightarrow *PrioQ*

extract_min: *PrioQ* \rightarrow *Keys* \times *Data* \times *PrioQ*

Mathematisches Modell:

Sorten:

Keys : $(U, <)$ (* totalgeordnetes „Universum“ *)

Data : D (* Menge von „Datensätzen“ *)

Boolean : $\{false, true\}$

PrioQ : die Menge aller endlichen Multimengen $P \subseteq U \times D$

Operationen:

empty(\emptyset) = \emptyset (* die leere Multimenge *)

$isempty(P) = \begin{cases} true, & \text{für } P = \emptyset \\ false, & \text{für } P \neq \emptyset \end{cases}$

$insert(x, d, P) = P \cup \{(x, d)\}$ (als Multimenge),

$extract_min(P) = \begin{cases} \text{undefiniert,} & \text{für } P = \emptyset \\ (x_0, d_0, P'), & \text{für } P \neq \emptyset, \end{cases}$

wobei im zweiten Fall

(x_0, d_0) ein Element in P ist mit

$x_0 = \min\{x \mid \exists d: (x, d) \in P\}$,

und $P' := P - \{(p_0, x_0)\}$

(als Multimenge).

Standardimplementierung: (Binäre) Heaps

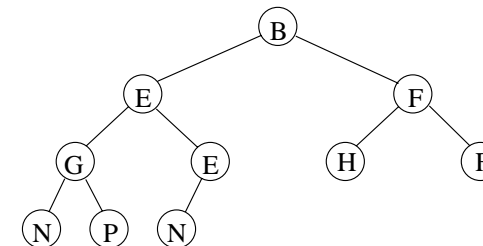
Heaps als Hilfsmittel bei **Heapsort**: bekannt aus AuD.

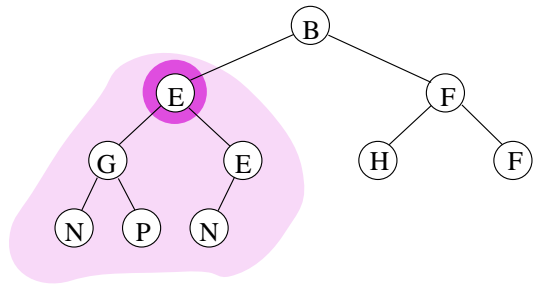
Definition

Ein Binärbaum T mit Knotenbeschriftungen $m(v)$ aus $(U, <)$ heißt **min-heapgeordnet** (oft: **heapgeordnet**),

wenn für jeden Knoten v gilt:

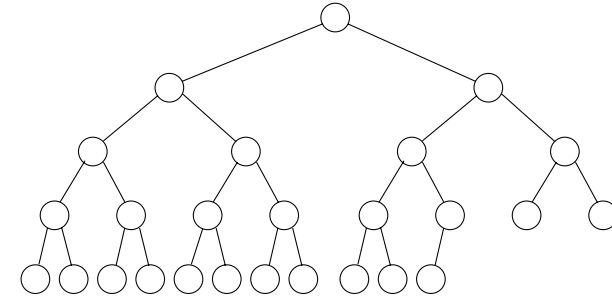
w Kind von $v \Rightarrow m(v) \leq m(w)$.





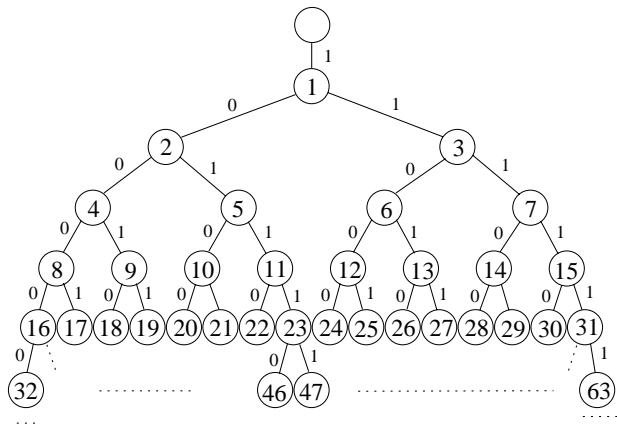
NB: T heapgeordnet \Rightarrow in Knoten v steht der minimale Eintrag des Teilbaums T_v mit Wurzel v .

Ein linksvollständiger Binärbaum:
 Alle Levels $j = 0, 1, \dots$ voll (jeweils 2^j Knoten) bis auf die tiefste.
 Das tiefste Level l hat von links her gesehen eine ununterbrochene Folge von Knoten.

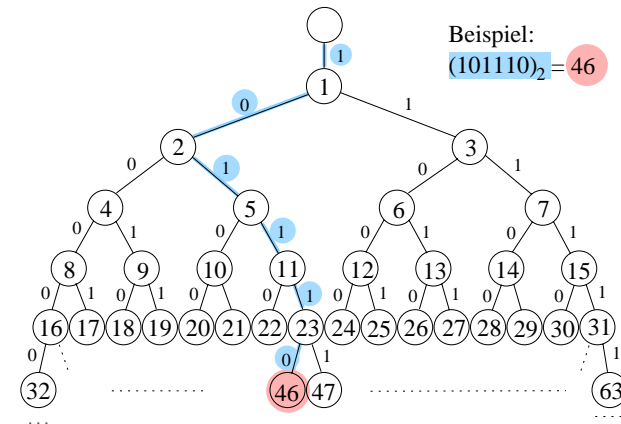


Können gut als Arrays dargestellt werden!

Nummerierung von Knoten in unendlichem, vollständigen Binärbaum gemäß **Breitendurchlauf-Anordnung**:



Nummerierung von Knoten in unendlichem, vollständigen Binärbaum gemäß **Breitendurchlauf-Anordnung**:



Ist $s_1 \cdots s_l \in \{0, 1\}^l$ die Markierung auf dem Weg von der Wurzel zu Knoten i auf Level l , so ist $1s_1 \cdots s_l$ die **Binärdarstellung** von i .

- Knoten 1 ist die Wurzel;
- Knoten $2i$ ist linkes Kind von i ;
- Knoten $2i + 1$ ist rechtes Kind von i ;
- Knoten $i \geq 2$ hat Vater $\lfloor i/2 \rfloor$.

Damit: Array-Darstellung für linksvollständige Binäräume:
Speichere Einträge in Knoten $1, \dots, n$ in Array $A[1..n]$.
Spart den Platz für die Zeiger!

Kombiniere Heapbedingung und Array-Darstellung:

Definition

Ein (Teil-)Array $A[1..k]$ mit Einträgen aus $(U \times D, <)$ heißt ein **Min-Heap** (oder **Heap**), wenn der entsprechende linksvollständige Binärbaum mit k Knoten (Knoten i mit $A[i]$ beschriftet) (min-)heapgeordnet ist.

D. h.: $A[1..k]$ ist ein **Heap**, falls für $1 \leq i \leq k$ gilt:
 $2i \leq k \Rightarrow A[i].key \leq A[2i].key$ und
 $2i + 1 \leq k \Rightarrow A[i].key \leq A[2i + 1].key$.

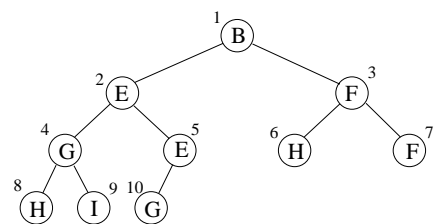
Beachte: Diese Definition erwähnt den (gedachten) Binärbaum nicht mehr explizit.

$U = \{A, B, C, \dots, Z\}$ mit der Standardordnung.

Im Beispiel: Daten weggelassen.

Ein Min-Heap und der zugeordnete Baum ($k = 10$):

1	2	3	4	5	6	7	8	9	10	11	12
B	E	F	G	E	H	F	H	I	G	*	*



Implementierung einer priority Queue: Neben Array $A[1..m]$:

Pegel k : Aktuelle Zahl k von Einträgen.

Überlaufprobleme werden ausgeklammert.

(Verdoppelungsstrategie, falls nötig.)

empty(m): (* Zeitaufwand: $O(1)$ oder $O(m)$ *)

Lege Array $A[1..m]$ an;

(* Jeder Eintrag ist ein Paar (key, data). *)

$k \leftarrow 0$.

isempty(): (* Zeitaufwand: $O(1)$ *)

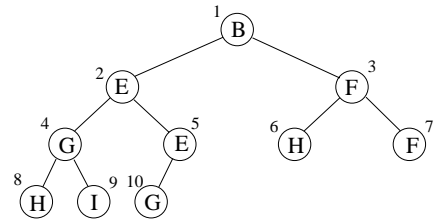
return($k = 0$);

extractMin: Implementierung von $extract_min(P)$:

Ein Eintrag mit minimalem Schlüssel steht in der Wurzel, d. h. in Arrayposition 1.

Entnehme $A[1]$ (hier „B“) und gib es aus.

1	2	3	4	5	6	7	8	9	10	11	12
B	E	F	G	E	H	F	H	I	G	*	*



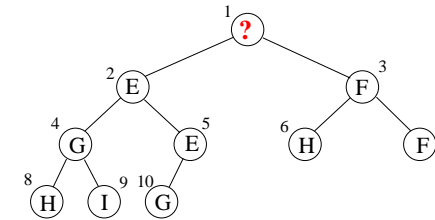
„Loch“ im Binärbaum.

Implementierung von $extract_min(P)$:

Ein Eintrag mit minimalem Schlüssel steht in der Wurzel, d. h. in Arrayposition 1.

Entnehme $A[1]$ (hier „B“) und gib es aus.

1	2	3	4	5	6	7	8	9	10	11	12
?	E	F	G	E	H	F	H	I	G	*	*



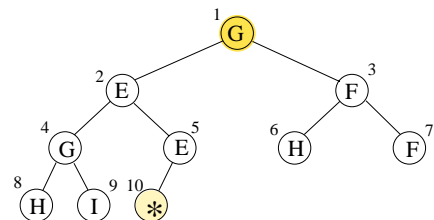
„Loch“ im Binärbaum.

Implementierung von $extract_min(P)$:

Ein Eintrag mit minimalem Schlüssel steht in der Wurzel, d. h. in Arrayposition 1.

Entnehme $A[1]$ (und gib es aus).

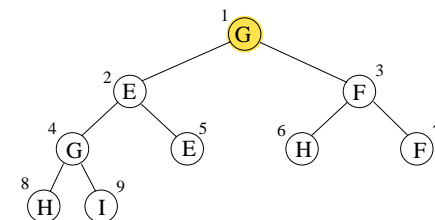
1	2	3	4	5	6	7	8	9	10	11	12
G	E	F	G	E	H	F	H	I	*	*	*



„Loch“ im Binärbaum – „stopfen“ mit dem letzten Eintrag.

bleibt Aufgabe: Heaps reparieren

1	2	3	4	5	6	7	8	9	10	11	12
G	E	F	G	E	H	F	H	I	*	*	*

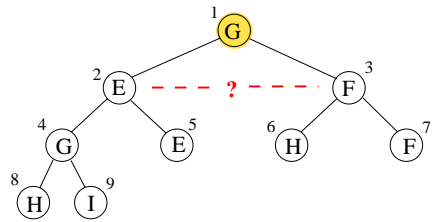


„Abwärts-Fast-Heap“ bis auf Position 1 (Wurzel), d. h.:

Wenn man den Eintrag in $A[1]$ verkleinert (hier z. B. auf C), entsteht ein Heap.

Heaps reparieren

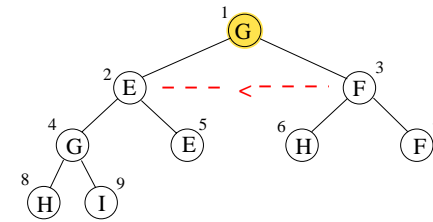
1	2	3	4	5	6	7	8	9	10	11	12
G	E	F	G	E	H	F	H	I	*	*	*



Vergleich der beiden Kinder des „Fehler“-Knotens.

Heaps reparieren

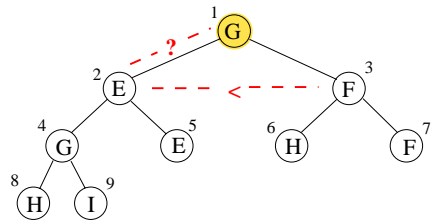
1	2	3	4	5	6	7	8	9	10	11	12
G	E	F	G	E	H	F	H	I	*	*	*



Vergleich der beiden Kinder des „Fehler“-Knotens.

Heaps reparieren

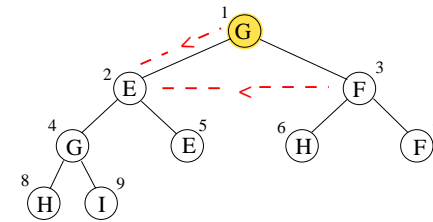
1	2	3	4	5	6	7	8	9	10	11	12
G	E	F	G	E	H	F	H	I	*	*	*



Vergleich des „kleineren“ Kinds mit dem „Fehler“-Knoten.

Heaps reparieren

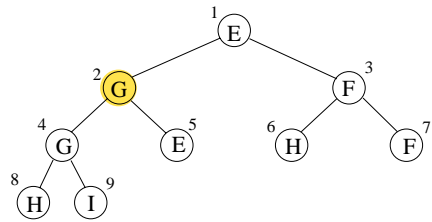
1	2	3	4	5	6	7	8	9	10	11	12
G	E	F	G	E	H	F	H	I	*	*	*



Vergleich des „kleineren“ Kinds mit dem „Fehler“-Knoten.

Heaps reparieren

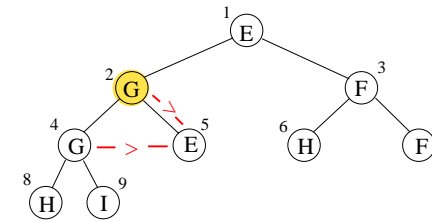
1	2	3	4	5	6	7	8	9	10	11	12
E	G	F	G	E	H	F	H	I	*	*	*



Vertauschen des „kleineren“ Kinds mit dem „Fehler“-Knoten.
Abwärts-Fast-Heap, Fehler in Knoten ein Level tiefer.

Heaps reparieren

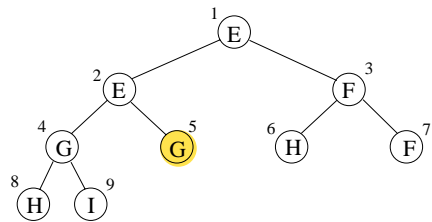
1	2	3	4	5	6	7	8	9	10	11	12
E	G	F	G	E	H	F	H	I	*	*	*



Vergleich des „kleineren“ Kinds mit dem „Fehler“-Knoten.

Heaps reparieren

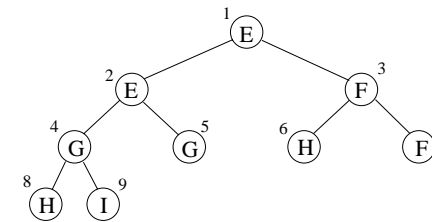
1	2	3	4	5	6	7	8	9	10	11	12
E	E	F	G	G	H	F	H	I	*	*	*



Vertauschen des „kleineren“ Kinds mit dem „Fehler“-Knoten.
Abwärts-Fast-Heap, Fehler in Knoten ein Level tiefer.
„Fehler“-Knoten hat keine Kinder.

Heaps reparieren

1	2	3	4	5	6	7	8	9	10	11	12
E	E	F	G	G	H	F	H	I	*	*	*



Kein Fehler mehr: Reparatur beendet.

Prozedur bubbleDown($1, k$) (* Heapreparatur in $A[1..k]$ *)
 (* Ausgangspunkt: $A[1..k]$ ist Abwärts-Fast-Heap, möglicher Fehler in $A[1]$ *)

- (1) $j \leftarrow 1$; $m \leftarrow 2$; $done \leftarrow false$;
- (2) **while not done and** $m + 1 \leq k$ **do**
 (* Abwärts-Fast-Heap, möglicher Fehler in $A[j]$, $A[j]$ hat 2 Kinder *)
- (3) **if** $A[m+1].key < A[m].key$
- (4) **then** (* $A[m]$: „kleineres“ Kind *)
- (5) $m \leftarrow m + 1$;
- (6) **if** $A[m].key < A[j].key$
- (7) **then** (* Abwärts-Fast-Heap, möglicher Fehler in $A[j]$ *)
- (8) vertausche $A[m]$ mit $A[j]$; $j \leftarrow m$; $m \leftarrow 2 * j$;
- (9) **else** (* fertig, kein Fehler mehr *)
- (10) $done \leftarrow true$;
- (11) **if not done then**
 (* Abwärts-Fast-Heap, möglicher Fehler in $A[j]$, maximal ein Kind *)
- (12) $m \leftarrow 2 * j$;
- (13) **if** $m \leq k$ **then**
- (14) **if** $A[m].key < A[j].key$
- (15) **then** vertausche $A[m]$ mit $A[j]$;

3 Folien Wiederholung aus AuD: Zum Nachlesen.

Korrektheit: **Wenn** vor dem Aufruf von **bubbleDown**($1, k$) das Teilarray $A[1..k]$ ein Abwärts-Fast-Heap war, mit möglichem Fehler in $A[1]$, d. h.:

es gibt einen Schlüssel $x \leq A[1].key$ derart, dass durch Ersetzen von $A[1].key$ durch x ein Heap entsteht,

dann ist nachher $A[1..k]$ ein Heap.

Beweis: Man zeigt folgende Behauptung **durch Induktion über die Schleifendurchläufe:**

(IB _{j}) Zu Beginn des Durchlaufs, in dem j die Zahl j enthält, ist $A[1..k]$ ein Abwärts-Fast-Heap mit möglichem Fehler in $A[j]$.

Der I.A. ist klar.

Betrachte Schleifendurchlauf, in dem j die Zahl j enthält.

1. Fall: In Zeile (8) bzw. (15) wird vertauscht.

Wir betrachten den Fall

$2j + 1 \leq k$ und $A[m+1].key \geq A[m].key$, also $m = 2j$.

(Andere Fälle analog.)

Nach der Vertauschung, vor der Änderung von j , gilt:

$A[j].key \leq A[2j].key \leq A[2j + 1].key$

(wegen der Bestimmung von m).

Weiter: Falls $j > 1$, gilt $A[j].key \geq A[\lfloor j/2 \rfloor].key$, wegen der Definition von „Abwärts-Fast-Heap“

(vor der Vertauschung war $A[m].key \geq A[\lfloor j/2 \rfloor].key$).

Weiter: Im Unterbaum unter dem „größeren“ Kind $A[2j + 1]$ ist alles unverändert, also ist dort die Heapbedingung erfüllt.

Nur $A[2j].key$ kann im Unterbaum unter $A[2j]$ zu groß sein.

Also: Abwärts-Fast-Heap mit möglichem Fehler in $A[2j]$, das ist (IB _{$2j$}).

2. Fall: In Zeile (8) bzw. (15) wird nicht vertauscht.

Dann gilt: Die Schleife endet.

$A[j].key \leq A[2j].key$, $A[2j + 1].key$ (falls $2j + 1 \leq k$).

Es muss auch $A[j].key \geq A[\lfloor j/2 \rfloor].key$ gelten, wegen der Definition von „Abwärts-Fast-Heap“ ($A[j].key$ kann nicht „zu klein“ sein).

Also ist $A[1..k]$ ein **Heap**.

In jedem Schleifendurchlauf wird der Inhalt von j immer mindestens verdoppelt; daher terminiert die Schleife.

Kosten:

... von Aufruf **bubbleDown**(1, k):

$O(1) +$

Θ (Anzahl der Schlüsselvergleiche „ $A[.] .key < A[.] .key$ “).

V_k : Anzahl der Schlüsselvergleiche im schlechtesten Fall.

In jedem Schleifendurchlauf **2** Vergleiche.

j_r : Inhalt von j in Runde $r = 1, 2, 3, \dots$ Verdopplung!

$j_1 = 1 = 2^0, j_r \geq 2^{r-1}$.

Vergleiche finden nur statt, falls $k \geq 2 \cdot j_r \geq 2^r$, d. h. $r \leq \log k$.

$\Rightarrow V_k \leq 2 \cdot \log k$.

Prozedur extractMin

(* Entnehmen eines minimalen Eintrags aus Priority-Queue *)

(* Ausgangspunkt: Pegel k , $A[1..k]$ ist Heap, $k \geq 1$ *)

- (1) $x \leftarrow A[1] .key; d \leftarrow A[1] .data;$
- (2) $A[1] \leftarrow A[k];$
- (3) $k--;$
- (4) **if** $k > 1$ **then** **bubbleDown**(1, k);
- (5) **return** (x, d);

Korrektheit: klar wegen Korrektheit von **bubbleDown**(1, k).

Zeitaufwand: $O(\log(k))$.

Implementierung von **insert**(x, d):

Voraussetzung: $A[1..k]$ ist Heap; $1 \leq i \leq k < m$.

$k++;$

$A[k] \leftarrow (x, d)$.

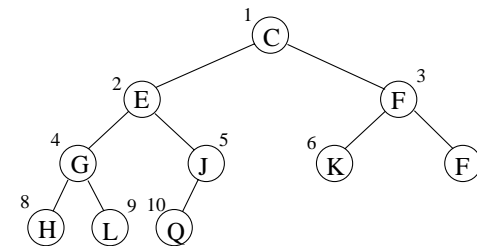
An der Stelle k ist nun eventuell die Heapeigenschaft gestört (x zu klein).

Wir nennen $A[1..k]$ einen „**Aufwärts-Fast-Heap**“ mit möglicher Fehlerstelle k .

Heißt: Es gibt einen Schlüssel $y \geq A[k] .key$, so dass ein Heap entsteht, wenn man $A[k] .key$ durch y ersetzt.

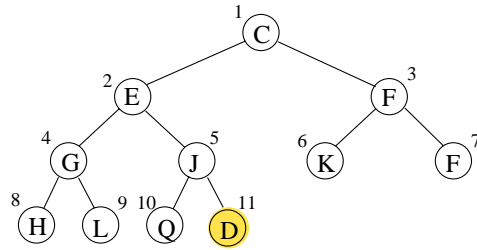
Heap:

1	2	3	4	5	6	7	8	9	10	11	12
C	E	F	G	J	K	F	H	L	Q	*	*



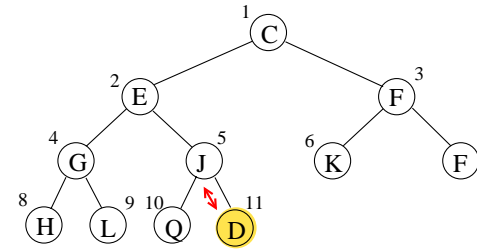
Einfügen von „D“ an Stelle $k = 11$.

1	2	3	4	5	6	7	8	9	10	11	12
C	E	F	G	J	K	F	H	L	Q	D	*



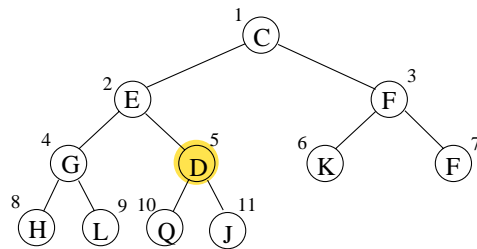
Heapreparatur bei Aufwärts-Fast-Heaps: **bubbleUp**.

1	2	3	4	5	6	7	8	9	10	11	12
C	E	F	G	J	K	F	H	L	Q	D	*



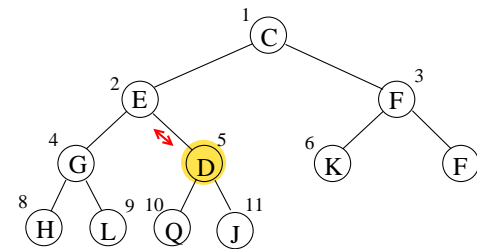
Heapreparatur bei Aufwärts-Fast-Heaps: **bubbleUp**.

1	2	3	4	5	6	7	8	9	10	11	12
C	E	F	G	D	K	F	H	L	Q	J	*



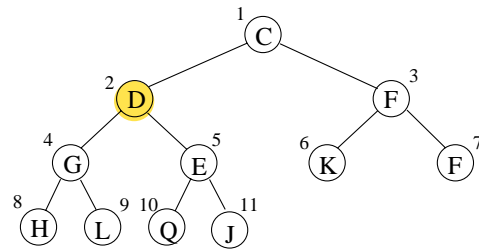
Heapreparatur bei Aufwärts-Fast-Heaps: **bubbleUp**.

1	2	3	4	5	6	7	8	9	10	11	12
C	E	F	G	D	K	F	H	L	Q	J	*



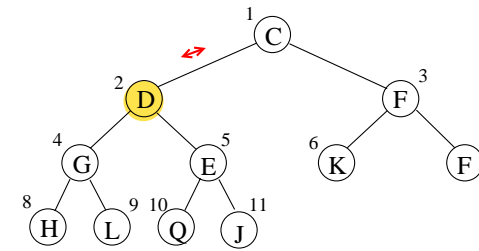
Heapreparatur bei Aufwärts-Fast-Heaps: **bubbleUp**.

1	2	3	4	5	6	7	8	9	10	11	12
C	D	F	G	E	K	F	H	L	Q	J	*



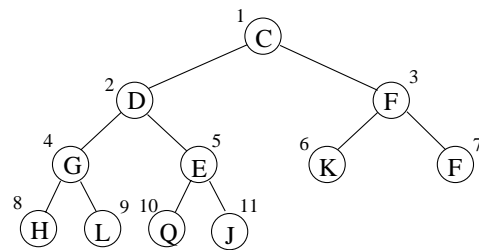
Heapreparatur bei Aufwärts-Fast-Heaps: **bubbleUp**.

1	2	3	4	5	6	7	8	9	10	11	12
C	D	F	G	E	K	F	H	L	Q	J	*



Heapreparatur bei Aufwärts-Fast-Heaps: **bubbleUp**.

1	2	3	4	5	6	7	8	9	10	11	12
C	D	F	G	E	K	F	H	L	Q	J	*



Prozedur bubbleUp(i) (* Heapreparatur ab A[i] nach oben *)

- (1) $j \leftarrow i$;
- (2) $h \leftarrow j \text{ div } 2$;
- (3) $x \leftarrow A[j].\text{key}$; $d \leftarrow A[j].\text{data}$;
- (4) **while** $h \geq 1$ **and** $x < A[h].\text{key}$ **do**
- (5) $A[j] \leftarrow A[h]$;
- (6) $j \leftarrow h$;
- (7) $h \leftarrow j \text{ div } 2$;
- (8) $A[j] \leftarrow (x, d)$;

Auf dem Weg von A[i] zur Wurzel werden alle Elemente, deren Schlüssel größer als x (= der neue Eintrag in A[i]) ist, um eine Position (auf dem Weg) nach unten gezogen.

Eintrag A[i] landet in der freigewordenen Position.

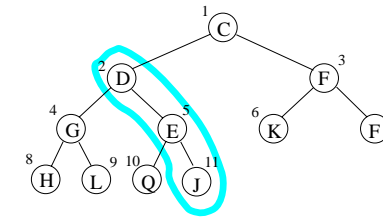
Behauptung

Wenn $A[1..k]$ ein Aufwärts-Fast-Heap ist mit möglichem Fehler an Stelle i ,

dann ist nach Ausführung von $\text{bubbleUp}(i)$ das Array $A[1..k]$ ein Heap.

Der Zeitaufwand ist $O(\log i)$, die Anzahl der Schlüsselvergleiche ist höchstens das Level von i im Baum, also die Anzahl der Bits in der Binärdarstellung $\text{bin}(i)$, d.h. $\lceil \log(i+1) \rceil$.

1	2	3	4	5	6	7	8	9	10	11	12
C	D	F	G	E	K	F	H	L	Q	J	*

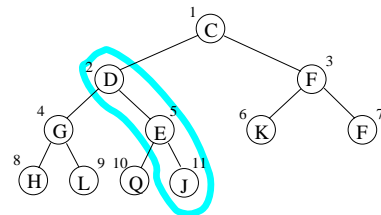


Beweis: Wenn sich ein Eintrag ändert, dann liegt er auf dem Weg von i zur Wurzel.

Das „Hochziehen“ von $A[i]$ wirkt wie bei Sortieren durch Einfügen:

nach $\text{bubbleUp}(i)$ sind die Elemente auf diesem Weg (fallend) geordnet.

1	2	3	4	5	6	7	8	9	10	11	12
C	D	F	G	E	K	F	H	L	Q	J	*



Falls $A[j]$ durch $A[\lfloor j/2 \rfloor]$ ersetzt wird, ist nachher $A[j]$ **kleiner als seine beiden Kinder**.

Beide Kinder der Position ℓ , an die x gelangt, enthalten Schlüssel, die $\geq x$ sind: der Schlüssel **auf** dem Weg (vorher in $A[\ell]$), weil er mit x verglichen wird; der Schlüssel im Kind von ℓ **neben** dem Weg, weil er \geq dem alten Wert $A[\ell]$ ist.

Prozedur $\text{insert}(x, d)$

(* Einfügen eines neuen Eintrags in Priority-Queue *)

(* Ausgangspunkt: Pegel k , $A[1..k]$ ist Heap, $k < n$ *)

- (1) **if** $k = m$ **then** „Überlauf-Fehler“;
- (2) $k++$;
- (3) $A[k] \leftarrow (x, d)$;
- (4) $\text{bubbleUp}(k)$.

Korrektheit: klar wegen Korrektheit von $\text{bubbleUp}(k)$.

Zeitaufwand: $O(\log(k))$.

Wir können **bubbleUp**(i) sogar für eine etwas allgemeinere Operation verwenden (Korrektheitsbeweis gilt weiter):

Wir ersetzen einen beliebigen Schlüssel im Heap (Position i) durch einen kleineren.

Mit **bubbleUp**(i) kann die Heapeigenschaft wieder hergestellt werden.

Prozedur decreaseKey(x, i)

(* (Erniedrigen des Schlüssels an Arrayposition i auf x) *)

- (1) **if** $A[i].key < x$ **then** Fehlerbehandlung;
- (2) $A[i].key \leftarrow x$;
- (3) **bubbleUp**(i).

(* Zeitaufwand: $O(\log(i))$ *)

SimplePriorityQueue plus „decreaseKey“: **Priority Queue.**

Satz

Der Datentyp “Priority Queue” kann mit Hilfe eines Heaps implementiert werden.

Dabei erfordern **empty** und **isempty** (und das Ermitteln des kleinsten Eintrags) konstante Zeit (bzw. $O(m)$, wenn ein Array der Größe m initialisiert werden muss)

und **insert**, **extractMin** und **decreaseKey** benötigen jeweils Zeit $O(\log n)$.

Dabei ist n jeweils der aktuelle Pegelstand, also die Anzahl der Einträge in der Priority Queue.

Technisches Problem:

Wie soll man der Datenstruktur „mitteilen“, **welches Objekt gemeint ist**, wenn *decrease_key* auf einen Eintrag angewendet werden soll?

Bei (binärem) Heap: Positionen der Einträge im Array ändern sich die ganze Zeit, durch die durch **insert** und **extractMin** verursachten Verschiebungen im Array.

Technisch unsauber (widerspricht dem Prinzip der Kapselung einer Datenstruktur):

dem Benutzer stets mitteilen, an welcher Stelle im Array ein Eintrag sitzt.

Besser: Erweiterung der Datenstruktur:

Objekt (x, d) erhält bei Ausführung von **insert**(x, d) eine **eindeutige „Identität“** p zugeteilt, die sich **nie** ändert, und über die dieses Objekt angesprochen werden kann.

Damit können auch **verschiedene Kopien** eines Datensatzes **unterschieden** werden!

p wird dem Benutzer als Reaktion auf die Einfügung mitgeteilt.

Der Benutzer kann dann irgendwann den Schlüssel eines Eintrags mit Identität p ändern.

Technische Realisierung:

(1) Identität p ist direkter Zeiger/Referenz auf das Objekt im Heap. Im Heaparray werden nur Zeiger/Referenzen auf diese Objekte gehalten, so dass sie nie verschoben werden müssen.

Das Objekt im Heap enthält seine **tatsächliche** Position im Heap-Array als Komponente. (**Nachteil wieder:** Kapselung der Datenstruktur ist durchbrochen – der Benutzer kann im Prinzip direkt auf die Objekte zugreifen.)

Oder:

(2) Eine beliebige (laufende) Nummer wird vergeben (z. B. durch Durchzählen der Einfügungen).

Die Datenstruktur führt selbst in einem weiteren Array für jede dieser Nummern die Position des entsprechenden Objekts im eigentlichen Heap-Array mit. Das Objekt im Heap enthält seine Identität und seine **tatsächliche** Position im Heap-Array als Komponenten. (Nachteil: Platzbedarf des Hilfsarrays entspricht der Anzahl der Einfügungen, nicht der maximalen Größe der Pr.Qu.)

Sonder-Übungsaufgabe: Wie kann man arrangieren, dass freigewordene Nummern wieder vergeben werden? Welches Verhalten ist dann vom Benutzer zu verlangen?

Beispiel: Datensatz = Schlüssel = Eintrag aus $\{A, \dots, Z\}$.

Wir vergeben **laufende Nummern** als Identitäten.

Der **Benutzer** der Datenstruktur muss sicherstellen, dass Identitäten nicht mehr benutzt werden, nachdem das entsprechende Objekt (durch ein *extract_min*) wieder aus der Datenstruktur verschwunden ist.

Runde	Operation	innerer Zustand	Ausgabe
1	<i>empty</i>	\emptyset	–
2	<i>insert(D)</i>	{(1,D)}	1
3	<i>insert(B)</i>	{(1,D), (2,B)}	2
4	<i>insert(D)</i>	{(1,D), (2,B), (3,D)}	3
5	<i>insert(E)</i>	{(1,D), (2,B), (3,D), (4,E)}	4
6	<i>insert(G)</i>	{(1,D), (2,B), (3,D), (4,E), (5,G)}	5
7	<i>decrease_key(5,D)</i>	{(1,D), (2,B), (3,D), (4,E), (5,D)}	–

Runde	Operation	innerer Zustand	Ausgabe
...		{(1,D), (2,B), (3,D), (4,E), (5,D)}	
8	<i>extract_min</i>	{(1,D), (3,D), (4,E), (5,D)}	(2,B)
9	<i>extract_min</i>	{(1,D), (4,E), (5,D)}	(3,D)
10	<i>decrease_key(4,B)</i>	{(1,D), (4,B), (5,D)}	–
11	<i>extract_min</i>	{(1,D), (5,D)}	(4,B)
12	<i>insert(F)</i>	{(1,D), (5,D), (6,F)}	6
13	<i>decrease_key(6,C)</i>	{(1,D), (5,D), (6,C)}	–
14	<i>extract_min</i>	{(1,D), (5,D)}	(6,C)
15	<i>extract_min</i>	{(1,D)}	(5,D)
16	<i>extract_min</i>	\emptyset	(1,D)
17	<i>isempty</i>	\emptyset	true.

In Runde 13 wäre zum Beispiel die Operation *decrease_key(3,A)* nicht legal, da das (eindeutig bestimmte) Objekt mit Identität 3 in Runde 9 aus der Datenstruktur verschwunden ist.

Datentyp: PriorityQueue (Vorrangwarteschlange)

Signatur:

Sorten:

Keys

Data

Id (* „Identitäten“ *)

PrioQ (* „Priority Queues“ *)

Boolean

Operatoren:

empty : \rightarrow *PrioQ*

isempty : *PrioQ* \rightarrow *Boolean*

insert : *Keys* \times *Data* \times *PrioQ* \rightarrow *PrioQ* \times *Id*

extract_min: *PrioQ* \rightarrow *Id* \times *Keys* \times *Data* \times *PrioQ*

decrease_key : *Id* \times *Keys* \times *PrioQ* \rightarrow *PrioQ*

Mathematisches Modell:

Keys : $(U, <)$

(* totalgeordnetes „Universum“ *)

Data : D (* Menge von „Datensätzen“ *)

Id : $\mathbb{N} = \{0, 1, 2, \dots\}$

(* unendliche Menge möglicher „Identitäten“ *)

Boolean : $\{false, true\}$

PrioQ : die Menge aller Funktionen $f: X \rightarrow U \times D$,
wobei $X = Def(f) \subseteq \mathbb{N}$ endlich ist

empty($()$) = \emptyset (* die *leere Funktion* *)

$isempty(f) = \begin{cases} true, & \text{für } f = \emptyset \\ false, & \text{für } f \neq \emptyset \end{cases}$

$insert(x, d, f) = (f \cup \{(p, (x, d))\}, p),$
für ein $p \in \mathbb{N} - Def(f)$

$extract_min(f) = \begin{cases} \text{undefiniert}, & \text{für } f = \emptyset \\ (p_0, x_0, d_0, f'), & \text{für } f \neq \emptyset, \end{cases}$

wobei im zweiten Fall

p_0 ein Element in $Def(f)$ ist mit

$(x_0, d_0) = f(p_0)$

mit $x_0 = \min\{x \mid \exists p \exists d: (p, (x, d)) \in f\}$,

und $f' := f - \{(p_0, (x_0, d_0))\}$.

Implementierung

Variante 1:

Typ (Klasse) **entry** hat Komponenten

x : key; d : data; p : integer

Wenn e ein solches Objekt ist, haben seine Komponenten folgende Bedeutung:

$e.x$: Schlüssel; $e.d$: Daten; $e.pos$: Position im Heaparray
 $A[1..m]$: array of entry, die (momentan) den Eintrag e enthält.

Das Heaparray $A[1..m]$ enthält in den Positionen $1, \dots, k$ **Zeiger/Referenzen** auf die momentan im Heap gespeicherten Objekte. Als Identitäten fungieren Zeiger/Referenzen auf die Objekte selbst.

empty(m):

Lege Array $A[1..m]$: array of entry an,
Initialisierung nicht nötig.

$k \leftarrow 0$.

(* Zeitaufwand: $O(1)$ *)

isempty():

return($k = 0$);

(* Zeitaufwand: $O(1)$ *)

Implementierung von $extract_min(P)$:

Ein Eintrag mit minimalem Schlüssel steht in der Wurzel, d. h. in Arrayposition $A[1]$. Dieser wird ausgegeben, der Platz an Position 1 wird frei; er wird mit dem Objekt von Platz k aufgefüllt. Dann: Heapreparatur.

Hilfsprozedur:

exchange(i, j):

Prozedur exchange(i, j) (* Vertausche $A[i]$ und $A[j]$ *)

(* Ausgangspunkt: $1 \leq i, j \leq k; i \neq j$. *)

- (1) **pt**: entry (* Hilfszeiger/-referenz *)
- (2) **pt** $\leftarrow A[i]$;
- (3) $A[i] \leftarrow A[j]$;
- (4) $A[i].pos \leftarrow i$;
- (5) $A[j] \leftarrow pt$;
- (6) $A[j].pos \leftarrow j$;

Prozedur bubbleDown($1, k$) (* Heapreparatur in $A[1..k]$ *)

(* Ausgangspunkt: $A[1..k]$ ist Abwärts-Fast-Heap, möglicher Fehler in $A[1]$ *)

- (1) $j \leftarrow 1; m \leftarrow 2; done \leftarrow false$;
- (2) **while not done and** $m + 1 \leq k$ **do**
(* Abwärts-Fast-Heap, möglicher Fehler in $A[j]$, $A[j]$ hat 2 Kinder *)
- (3) **if** $A[m+1].x < A[m].x$
- (4) **then** (* $A[m]$: „kleineres“ Kind *)
- (5) $m \leftarrow m + 1$;
- (6) **if** $A[m].x < A[j].x$
- (7) **then** (* Abwärts-Fast-Heap, möglicher Fehler in $A[j]$ *)
- (8) **exchange**(m, j); $j \leftarrow m; m \leftarrow 2 * j$;
- (9) **else** (* fertig, kein Fehler mehr *)
- (10) $done \leftarrow true$;
- (11) **if not done then**
(* Abwärts-Fast-Heap, möglicher Fehler in $A[j]$, maximal ein Kind *)
- (12) $m \leftarrow 2 * j$;
- (13) **if** $m \leq k$ **and** $A[m].x < A[j].x$
- (14) **then exchange**(m, j);

Prozedur extractMin

(* Entnehmen eines minimalen Eintrags aus Priority-Queue *)

(* Ausgangspunkt: Pegel k , $A[1..k]$: array of entry
ist Heap, $k \geq 1$ *)

- (1) **ee** $\leftarrow A[1]$;
- (2) $A[1] \leftarrow A[k]$;
- (3) $A[1].pos \leftarrow 1$;
- (4) $k--$;
- (5) **if** $k > 1$ **then bubbleDown**($1, k$);
- (6) **return** (ee, ee.x, ee.d);

Zeitaufwand: $O(\log(k))$.

Implementierung von $\text{insert}(x, d)$:

Voraussetzung:

$A[1..k]$: array of entry ist Heap; $1 \leq i \leq k < m$.

$k++$;

$A[k].x \leftarrow x$; $A[k].d \leftarrow d$;

An der Stelle k ist nun eventuell die Heapeigenschaft gestört ($A[k].x$ zu klein). – Reparatur mit **bubbleUp**.

Prozedur bubbleUp(i) (* Heapreparatur ab $A[i]$ nach oben *)

(1) $j \leftarrow i$;

(2) $h \leftarrow j \div 2$;

(3) **while** $h \geq 1$ **and** $A[j].x < A[h].x$ **do**

(4) **exchange**(j, h);

Prozedur insert(x, d)

(* Einfügen eines neuen Eintrags in Priority-Queue *)

(* Ausgangspunkt: Pegel k , $A[1..k]$ ist Heap, $k < n$ *)

(1) **if** $k = m$ **then** „Überlauf-Fehler“;

(2) $k++$;

(3) $A[k].x \leftarrow x$; $A[k].d \leftarrow d$; $A[k].pos \leftarrow k$;

(4) **bubbleUp**(k).

Korrektheit: wie vorher.

Zeitaufwand: $O(\log(k))$.

Allgemeiner: **decreaseKey**

Wir ersetzen einen beliebigen Schlüssel im Heap, gegeben durch eine Referenz, durch einen kleineren.

Mit **bubbleUp**(i) kann die Heapeigenschaft wieder hergestellt werden.

Prozedur decreaseKey(x, e)

(* (Erniedrigen des Schlüssels in Heapeintrag e auf x) *)

(1) **if** $e.x < x$ **then** Fehlerbehandlung;

(2) $e.x \leftarrow x$;

(3) **bubbleUp**($e.pos$).

(* Zeitaufwand: $O(\log(i))$, wenn $i = e.pos$. *)

SimplePriorityQueue plus „decreaseKey“: **Priority Queue**.

Satz

Der Datentyp “Priority Queue” kann mit Hilfe eines Heaps implementiert werden.

Dabei erfordern **empty** und **isempty** (und das Ermitteln des kleinsten Eintrags) konstante Zeit (bzw. $O(m)$, wenn ein Array der Größe m initialisiert werden muss)

und **insert**, **extractMin** und **decreaseKey** benötigen jeweils Zeit $O(\log n)$.

Dabei ist n jeweils der aktuelle Pegelstand, also die Anzahl der Einträge in der Priority Queue.

Nachteil der gewählten Implementierung:

Die „Außenwelt“ erhält mit den Zeigern direkten Zugriff auf die interne Darstellung der Heapeinträge.

Verbesserung: **Übung**.

Implementierungsdetails im Algorithmus von Dijkstra:

Wir nehmen an, dass $G = (V, E, c)$ mit $V = \{1, \dots, n\}$ in Adjazenzlistendarstellung gegeben ist.

Die Kantengewichte $c(e)$ stehen in den Adjazenzlisten bei den Kanten.

Für jeden Knoten $v \in V - S$ wollen wir immer wissen:

- 1) die Länge $\text{dist}[v]$ des (bzw. eines) kürzesten S -Weges von s nach v , falls ein solcher existiert (sonst: $\text{dist}[v] = \infty$)
- 2) den (bzw. einen) (Vorgänger-)Knoten $p(v) = u \in S$ mit $\text{dist}[v] = \text{dist}[u] + c(u, v)$, falls ein solcher existiert.
p: array[1..n] of integer: Vorgänger, in eigenem Array.

Verwalte die Knoten $v \in V - S$ mit Werten $\text{dist}[v] < \infty$ mit den $\text{dist}[v]$ -Werten als **Schlüssel** in einer **Priority-Queue PQ**.

Wenn $\text{dist}[v] = \infty$, ist v (noch) nicht in der **PQ**.

inS: array[1..n] of Boolean (Knoten in S).

id: array[1..n] of entry (für Identitäten).

Dijkstra-mit-Priority-Queue(G, s)

Eingabe: gewichteter Digraph $G = (V, E, c)$, $V = \{1, \dots, n\}$, Startknoten s ;

Ausgabe: Länge $d(s, v)$ der kürzesten Wege, Vorgängerknoten $p(v)$

Hilfsstrukturen: **PQ:** eine (leere) Priority-Queue; inS, p, id: s.o.

- (1) **for** v **from** 1 **to** n **do** inS[v] \leftarrow false; p[v] \leftarrow -1;
- (2) inS[s] \leftarrow true; p[s] \leftarrow -2;
- (3) dist[s] \leftarrow 0;
- (4) **for** Knoten v mit $(s, v) \in E$ **do**
- (5) dist[v] \leftarrow $c(s, v)$; p[v] \leftarrow s; id[v] \leftarrow **PQ.insert**(dist[v], v);
- (6) **while not** PQ.isempty **do**
- (7) (id, d, u) \leftarrow PQ.extractMin; (* Identität, Distanz, Knotenname *)
- (8) inS[u] \leftarrow true;
- (9) **for** Knoten v mit $(u, v) \in E$ **and not** inS[v] **do**
- (10) dd \leftarrow dist[u] + $c(u, v)$;
- (11) **if** p[v] \geq 0 **and** dd < dist[v] **then**
- (12) PQ.decreaseKey(id[v], dd); p[v] \leftarrow u;
- (13) **if** p[v] = -1 (* v vorher nicht erreicht *) **then**
- (14) dist[v] \leftarrow dd; p[v] \leftarrow u; id[v] \leftarrow **PQ.insert**(dd, v);
- (15) **Ausgabe:** dist[1..n] und p[1..n].

Aufwand:

Die Priority-Queue realisieren wir als Heap.

Maximale Anzahl von Einträgen: $n - 1$.

Initialisierung:

$\text{deg}(s)$ Einfügungen, Kosten $O(\text{deg}(s) \cdot \log n)$.

Es gibt $\leq n - 1$ Durchläufe durch die **while**-Schleife mit Organisationsaufwand jeweils $O(1)$:

$O(n)$ für die Schleifenorganisation.

In Schleifendurchlauf Nummer i , wo u_i zu S hinzugefügt wird: PQ.extractmin kostet Zeit $O(\log n)$.

Durchmustern der $\text{deg}(u_i)$ Nachbarn von u_i :

Jedes PQ.insert oder PQ.decreaseKey kostet Zeit $O(\log n)$.

– Insgesamt:

$$\begin{aligned} n \cdot O(\log n) + \sum_{0 \leq i < n} O(\text{deg}(u_i) \cdot \log n) \\ = O\left(n \log n + \log n \cdot \left(\sum_{0 \leq i < n} \text{deg}(u_i)\right)\right) = O(n \log n + m \log n), \end{aligned}$$

wobei $n = |V|$ (Knotenzahl), $m = |E|$ (Kantenzahl).

Satz

Der **Algorithmus von Dijkstra** mit Verwendung einer heap-basierten Priority-Queue ermittelt kürzeste Wege von Startknoten s aus in einem Digraphen $G = (V, E, c)$ in Zeit $O((n + m) \log n)$.

Implementierungsdetails im Algorithmus von Jarník/Prim:

Wir nehmen an, dass $G = (V, E, c)$ mit $V = \{1, \dots, n\}$ in Adjazenzlistendarstellung gegeben ist.

Die Kantengewichte $c(e)$ stehen in den Adjazenzlisten bei den Kanten.

Für jeden Knoten $v \in V - S$ wollen wir immer wissen:

- 1) die Länge $d(v)$ der billigsten Kante (u, v) , $u \in S$, falls eine existiert: in $\text{dist}[v]$ („Abstand von S “)
- 2) den (einen) (Vorgänger-)Knoten $p(v) = u \in S$ mit $c(u, v) = d(v)$, falls ein solcher existiert.

Falls es von S keine Kante nach v gibt, setzen wir $d(v) = \infty$ und $p(v) = -1$.

Verwalte die Knoten $v \in V - S$ mit Werten $d(v) < \infty$ mit den $d(v)$ -Werten als **Schlüssel** in einer **Priority-Queue PQ**.

Wenn $d(v) = \infty$, ist v (noch) nicht in der **PQ**.

Jarnik/Prim-mit-Priority-Queue(G, s)

Eingabe: gewichteter Graph $G = (V, E, c)$, $V = \{1, \dots, n\}$, Startknoten s ;

Ausgabe: Ein MST für G .

Hilfsstrukturen: PQ: eine (leere) Priority-Queue; inS, p, id: wie oben

- (1) **for** v **from** 1 **to** n **do** inS[v] \leftarrow false; p[v] \leftarrow -1;
- (2) inS[s] \leftarrow true; p[s] \leftarrow -2;
- (3) **for** Knoten v mit $(s, v) \in E$ **do**
- (4) dist[v] \leftarrow $c(s, v)$; p[v] \leftarrow s ; id[v] \leftarrow PQ.insert(dist[v], v);
- (5) **while not** PQ.isEmpty **do**
- (6) $(id, d, u) \leftarrow$ PQ.extractMin; (* Identität, Distanz, Knotenname *)
- (7) inS[u] \leftarrow true;
- (8) **for** Knoten v mit $(u, v) \in E$ **and not** inS[v] **do**
- (9) dd \leftarrow $c(u, v)$; (* einziger Unterschied zu Dijkstra! *)
- (10) **if** p[v] \geq 0 **and** dd < dist[u] **then**
- (11) PQ.decreaseKey(id[v], dd); p[v] \leftarrow u ;
- (12) **if** p[v] = -1 (* v vorher nicht erreicht *) **then**
- (13) dist[v] \leftarrow dd; p[v] \leftarrow u ; id[v] \leftarrow PQ.insert(dd, v);
- (14) **Ausgabe:** Kantenmenge $T = \{(v, p[v]) \mid v \in S - \{s\}\}$.

Zeitanalyse: Exakt wie für den Dijkstra-Algorithmus.

Satz

Der Algorithmus von Jarník/Prim mit Verwendung einer heap-basierten Priority-Queue ermittelt einen minimalen Spannbaum für $G = (V, E, c)$ in Zeit $O((n + m) \log n)$.

Mitteilung

Es gibt eine Implementierung der Datenstruktur **Priority Queue** mit folgenden Laufzeiten:

empty und *isEmpty* (und das Ermitteln des kleinsten Eintrags) kosten konstante Zeit;

extractMin kostet Zeit $O(\log n)$;

n Einfügungen kosten Zeit $O(n \log n)$;

m *decreaseKey*-Operationen benötigen **zusammen** Zeit $O(m)$.

„Fibonacci-Heaps“

(fortgeschritten, siehe z. B. [Cormen et al.]

Mitteilung

Der Algorithmus von Dijkstra, in Kombination mit einer PQ-Implementierung über **Fibonacci-Heaps**, ermittelt kürzeste Wege von einem Startknoten aus in Zeit $O(m + n \log n)$.

Mitteilung

Der Algorithmus von Jarník/Prim, in Kombination mit einer PQ-Implementierung über **Fibonacci-Heaps**, ermittelt einen minimalen Spannbaum in Zeit $O(m + n \log n)$.

Nur sehr dünn besetzte Graphen, mit weniger als $n \log n$ Kanten, benötigen nichtlineare Laufzeit!