Algorithmische Graphentheorie

Matthias Jantzen

SoSe 2003

Donnerstags, 10:15-11:45 Uhr in C-221

Literatur

- Gritzmann/Brandenburg: Das Geheimnis des kürzesten Weges, Springer (2002).
  [nett, einfachst, günstig, eher LK-Mathe Niveau]
- Turau: Algorithmische Graphentheorie, Addison Wesley (1996).
- Klein: Algorithmische Geometrie, Addison Wesley (1997).
- Berg et al.: Computational Geometry, Springer (1997).
  [stellt einige Anwendungsfelder vor]
- Bang-Jensen/Gutin: Digraphs: Theory, Algorithms and Applications, Springer (2001).
- Johnsonbaugh: Discrete Mathematics, Prentice Hall (2001).
- Ottmann/Widmeyer: Algorithmen und Datenstrukturen, B·I· (1992).
  [gut verständlich, Aufbau ähnlich zu Sedgewick]
- Sedgewick: Algorithmen, Addison Wesley (2002).
- Edelsbrunner: Algorithms in Combinatorial Geometry, Springer (1987).
  [zusätzliche Informationen]
- Brassard/Bradley: Fundamentals of Algorithms, Prentice Hall (1996).
  [an Designtechniken orientiert, Grundlagen á la F3]
- Baase/van Gelder: Computer Algorithms, Addison Wesley (2000).
  [die wichtigsten Grundlagen und Algorithmen. Mehr Stoff, als in F3 enthalten]

Algorithmische Graphentheorie

- Suchen in Graphen
- Färbungen von Graphen
- Kürzeste Wege
- Transitive Hülle
- Aufspannende Bäume
- minimale Spannbäume
- beste Zuordnungen finden
- Flüsse in Netzwerken
- einfache Zusammenhangs-komponenten
- strenger Zusammenhang
- Rundreise Probleme
- Plättbarkeit
- ...

versus

Algorithmische Geometrie (computational geometry)

- Schnitte von Strecken
- Schnitte von Polygonen
- konvexe Hülle einer Punktmenge
- nächster Nachbar
- minimale Abstand in einer Punktmenge
- Voronoi Diagramme
- Punktidentifikation
- Kontur von Projektionen
- Sichtbarkeits Polygon
- Triangulationen
- Umrissbestimmung (shape determination)
- Bewegungsplanung für Roboter
- ...

Angaben, die stets zur Dokumentation von Algorithmen gehören sollten!

- **Voraussetzungen:**
  - Eingabebereich, Ausgabebereich und Einschränkungen davon.

- **Korrektheit:**
  - Beschreibt der Algorithmus wirklich den Prozess, den man realisieren möchte?

- **Adäquatheit:**
  - Ist der Prozess, der durch den Algorithmus realisiert wird, eine angemessene Lösung des gestellten Problems?

Angaben, die stets zur Dokumentation von Algorithmen gehören sollten!

- **Zuverlässigkeit und Sicherheit:**
  - Passiert in allen Situationen, in denen der Algorithmus benutzt wird, etwas Sinnvolles?
  - Was geschieht bei unsinnigen Eingaben?
  - Gibt es sinnvolle Ausgaben, obwohl die Eingaben unsinnig waren?
  - Was geschieht, wenn der den Algorithmus ausführende Prozessor fehlerhaft arbeitet?

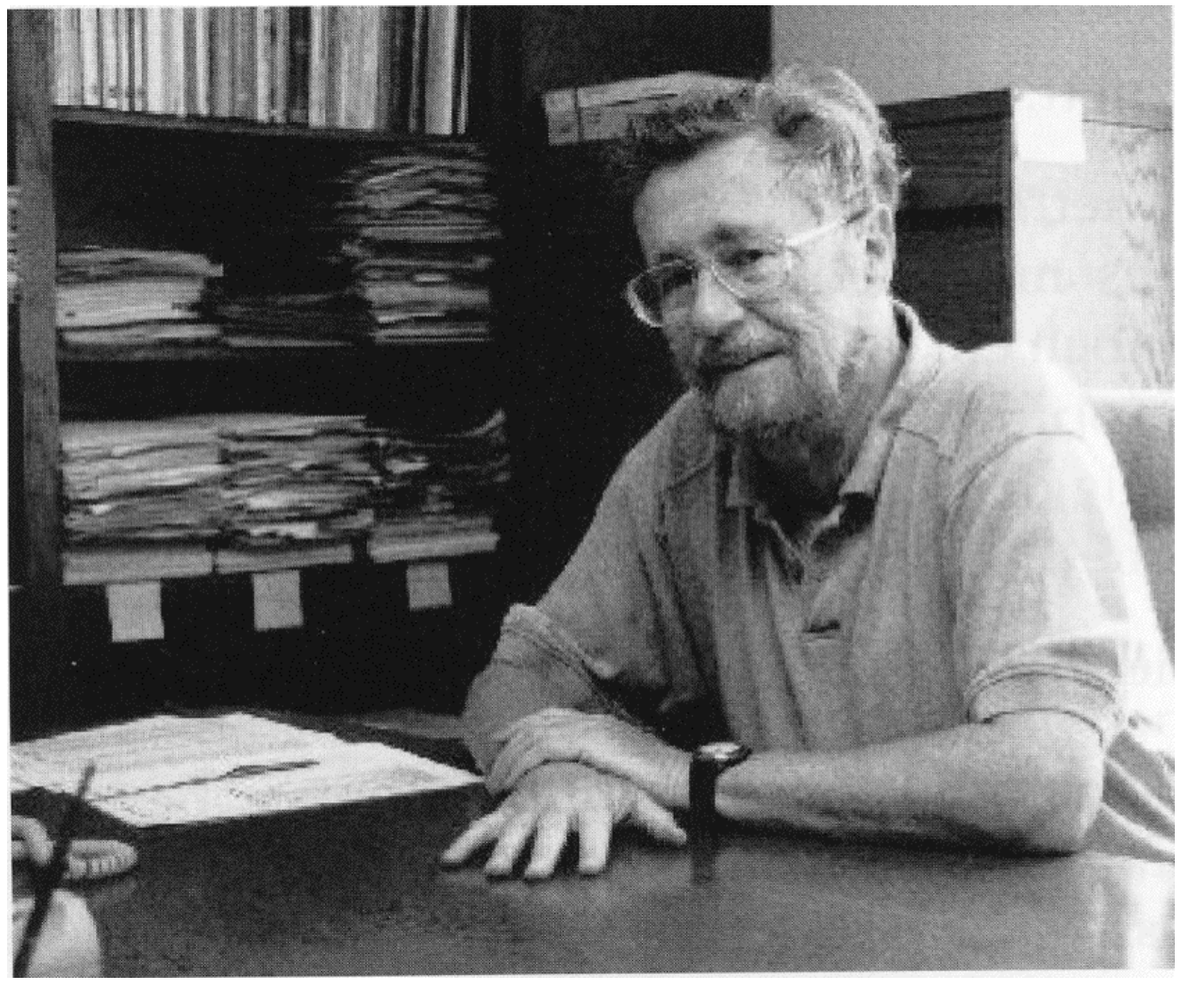
- **Aufwand:**
  - Zeit- und Platzbedarf im schlechtesten oder mittleren Fall, Datendarstellung!

Kürzester Weg vom Start zum Ziel

In der Ebene und auf der Kugel ist die kürzeste Entfernung zweier Punkte die Luftlinienentfernung.

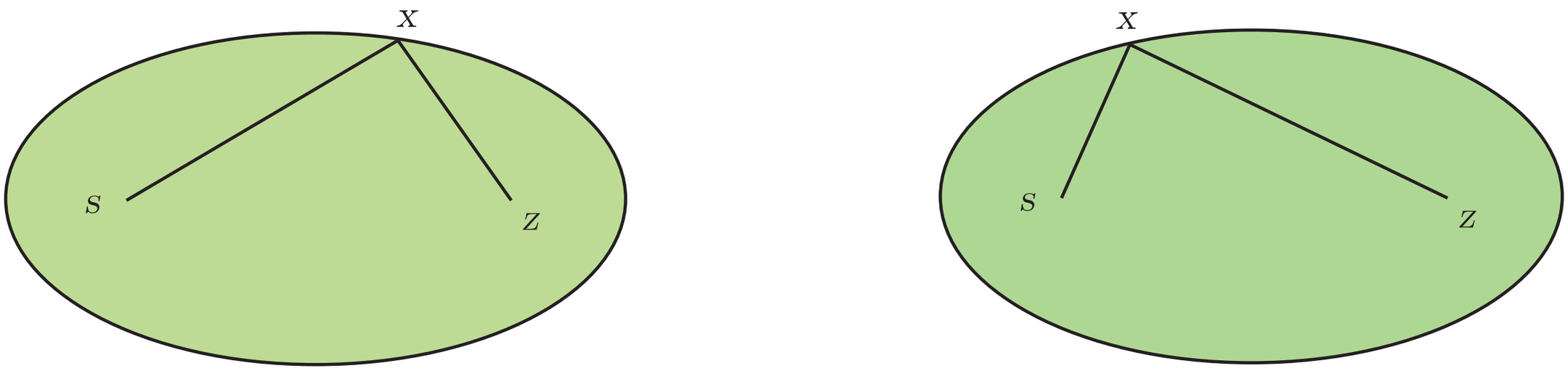
Wenn im ungerichteten Graph die Kanten mit positiven Entfernungsmaßen bewertet sind, so gibt es den Algorithmus von Dijkstra (1959 publiziert), um einen kürzesten Weg zu finden!
Für gerichtete Graphen (*Digraph*) kann dieses Verfahren angepasst werden.

Dieses Verfahren geht nicht mehr bei kantenbewerteten Digraphen mit beliebigen, ganzzahligen (*integral*) Gewichten.



Edsger Wybe Dijkstra

Die Gärtnermethode



Wenn schon ein Weg von s nach z der Länge $w(s,x) + w(x,z)$ bekannt ist brauchen nur Zwischenpunkte x innerhalb der Ellipse besucht zu werden!

Dieses Wissen kann aber nur in Strukturen ausgenutzt werden, in denen die Dreicksungleichung $w(s,x) + w(x,z) \geq w(s,z)$ gilt, also eine Metrik definiert werden kann.

Wenn die Kantengewichte aber Kosten, Nutzen, oder sonst etwas bedeuten, geht das i.A. nicht mehr!

Der Algorithmus von Dijkstra benötigt Kantengewichte aus $\mathbb{N}$!

Wenn Kantengewichte aus $\mathbb{Z}$ sind, dann wird alles noch schwieriger.

Kürzester Weg nach Dijkstra in ungerichteten, kantengewichteten Graphen

$input$: ungerichteter Graph $G{:=}(V,\ E),\ s{,}z \in V$, Gewichtung $w{:}E \to \mathbb{N}$.

$output$: ein kürzester Weg von s nach z und dessen Länge $d(z)$.

$begin\ d(s) \leftarrow 0$

$\quad for\ all\ v \in V \setminus \{s\}\ do$

$\quad\quad d(v) \leftarrow \infty;$ % T sind die Knoten, zu denen noch *kein*

$\quad\quad T \leftarrow V$ % kürzester Weg von s aus gefunden wurde.

$\quad end\ for$

$\quad\quad while\ z \in T\ do$

$\quad\quad begin$

$\quad\quad$ ☺ wähle $v^* \in T$ mit $d(v^*) = min\{d(v) |\ v \in T\};$

$\quad\quad\quad T \leftarrow T \setminus \{v^*\};$

$\quad\quad\quad for\ each\ v \in T\ with\ \{v^*{,}v\} \in E\ do$

$\quad\quad\quad d(v) \leftarrow min\{d(v),\ d(v^*){+}w(\{v^*, v\})\}$ % Neue Bewertungen

$\quad\quad\quad end\ for$ % der Auswahlkandidaten

$\quad end$

end

Mit $d(v)$ wird der kürzeste Weg von s nach v notiert. Dieser ist ∞, wenn es (bisher) keinen Weg gibt. In T bleiben alle Knoten, zu denen bisher kein kürzester Weg gefunden wurde, Knoten in T mit Bewertung $\neq\infty$ sind Kandidaten (Randknoten) für Hinzunahme zu einem möglichen kürzesten Weg.

♠ ***Induktionsbehauptung:***
Beim i-ten Besuch der Zeile ☺, ist $d(v)$ Abstand eines kürzesten Weges von s nach v.

Basis:
i=1 bedeutet $d(s)$=0 und $d(v) = \infty$ für alle anderen Knoten. Also wird in Zeile ☺ der Knoten s gewählt, und von s nach s ist $d(s) = 0$ die Länge eines kürzesten Weges!

Induktionsschritt:
Unter der Annahme, dass ♠ für alle $i < k$ gilt, wählen wir im k-ten Besuch von ☺ den Knoten $v \in T$ mit minimalem $d(v)$. Wir zeigen nun:
Wenn ein Weg von s nach w existiert, mit $d(w) < d(v)$, dann gilt $w \notin T$. Falls dennoch $w \in T$, dann gibt es Pfad $s \rightarrow^* u \rightarrow x \rightarrow^* w$, mit $u \notin T$ und x erster Knoten in T. Weil $u \notin T$ ist, wurde u in Zeile ☺ in einem vorangegangenen Schritt ausgewählt, und $d(u)$ ist nach ***Ind.An.*** die Länge eines kürzesten Weges.

Daher gilt $d(x) \leq d(u) + w(\{u,x\}) \leq d(w) < d(v)$. Beachte, dass $x \rightarrow^* w$ keine negative Länge hat. Mithin ist $d(v)$ nicht minimal unter den Auswahlkandidaten, $d(x)$ ist kleiner! Aus diesem Widerspruch folgt $w \notin T$.

Auch kann es keinen kürzeren Weg $s \rightarrow^* v$ gegeben, denn v wäre dann früher aus-gewählt und nicht mehr in T gewesen. Somit ist der gefundene Weg tatsächlich, wie in ♠ behauptet, einer mit minimaler Entfernung $d(v)$, und der *Algorithmus ist korrekt*!

Jeder kürzeste Weg ist immer zugleich ein Pfad,
d.h. ein Weg, in dem jeder Knoten nur einmal besucht wird!!

Bisher nur die Information, dass ein minimaler Weg von s nach z die Länge $d(z)$ hat, nur kennen wir den Weg noch nicht: **Es fehlt hier die Ausgabe!**
Jedem Knoten v, der aus T entfernt wird, fügen wir den Knoten $pre(v)$ von dem aus v erreicht wird, als Inschrift $pre(v).d(v)$ bei.

Dazu ersetze die Zeile im „$for\ each\ \ldots\ end\ for$“-Block (nach Zeile ☺) durch:

$if\ d(v^*)+w(\{v^*,v\})\} < d(v)$
$then\ d(v) \leftarrow d(v^*)+w(\{v^*,v\});\ pre\{v\} \leftarrow v^*$
$end\ if$

Aus dieser Information wird Pfad rückwärts zusammengesetzt!

Die *for*-Schleife der Initialisierung benötigt $O(n)$ Schritte, n die Zahl der Knoten.

Zeile ☺ wird höchstens n-mal aufgerufen, genauso wie die darin befindliche, folgende *for*-Schleife, was $O(n^2)$ für den gesamten Algorithmus ergibt.

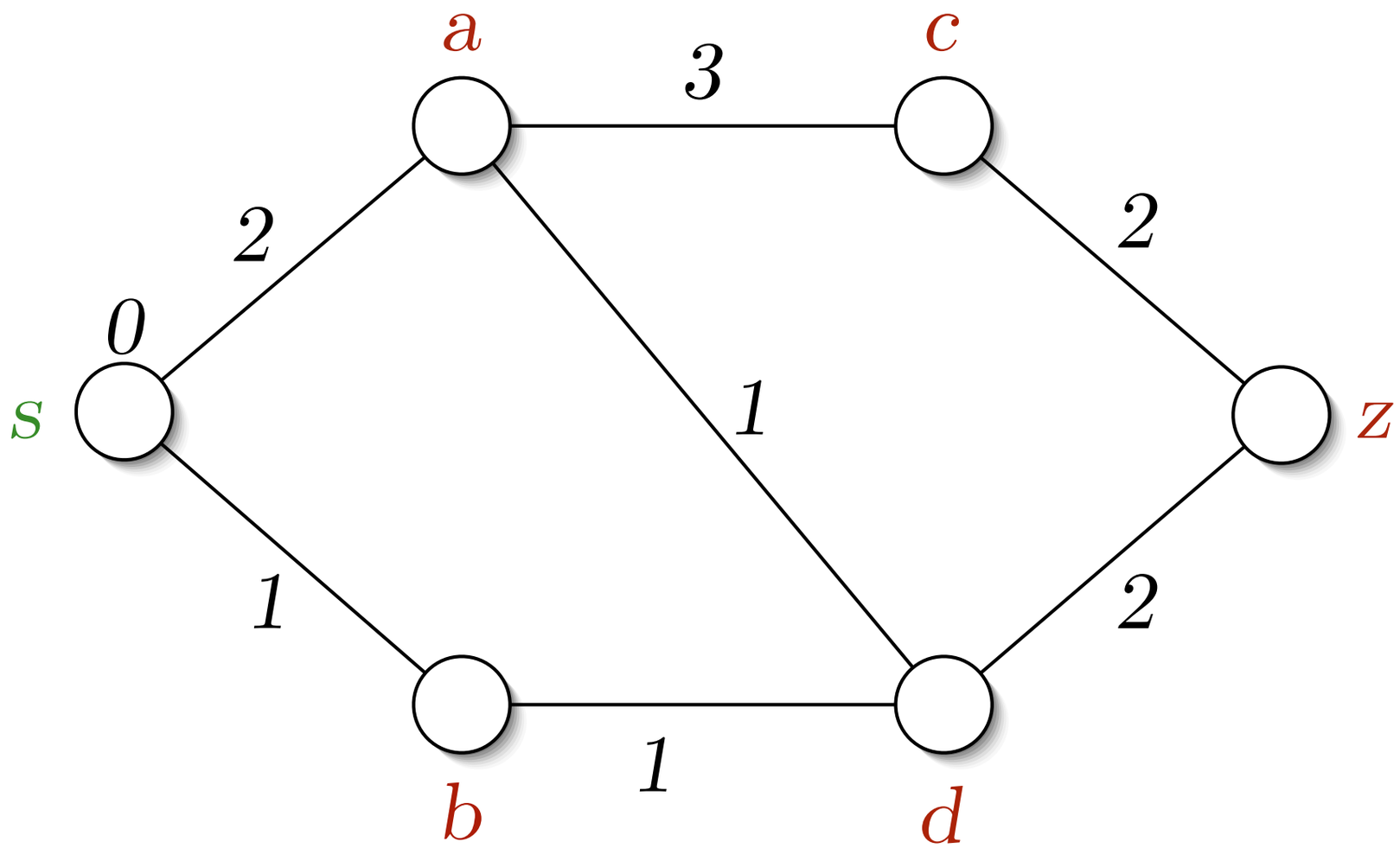
Es geht auch nicht besser, denn im vollständigen Graph K_n, sind stets alle Knoten benachbart und müssen angeschaut werden!

Der Algorithmus von Dijkstra ist optimal!

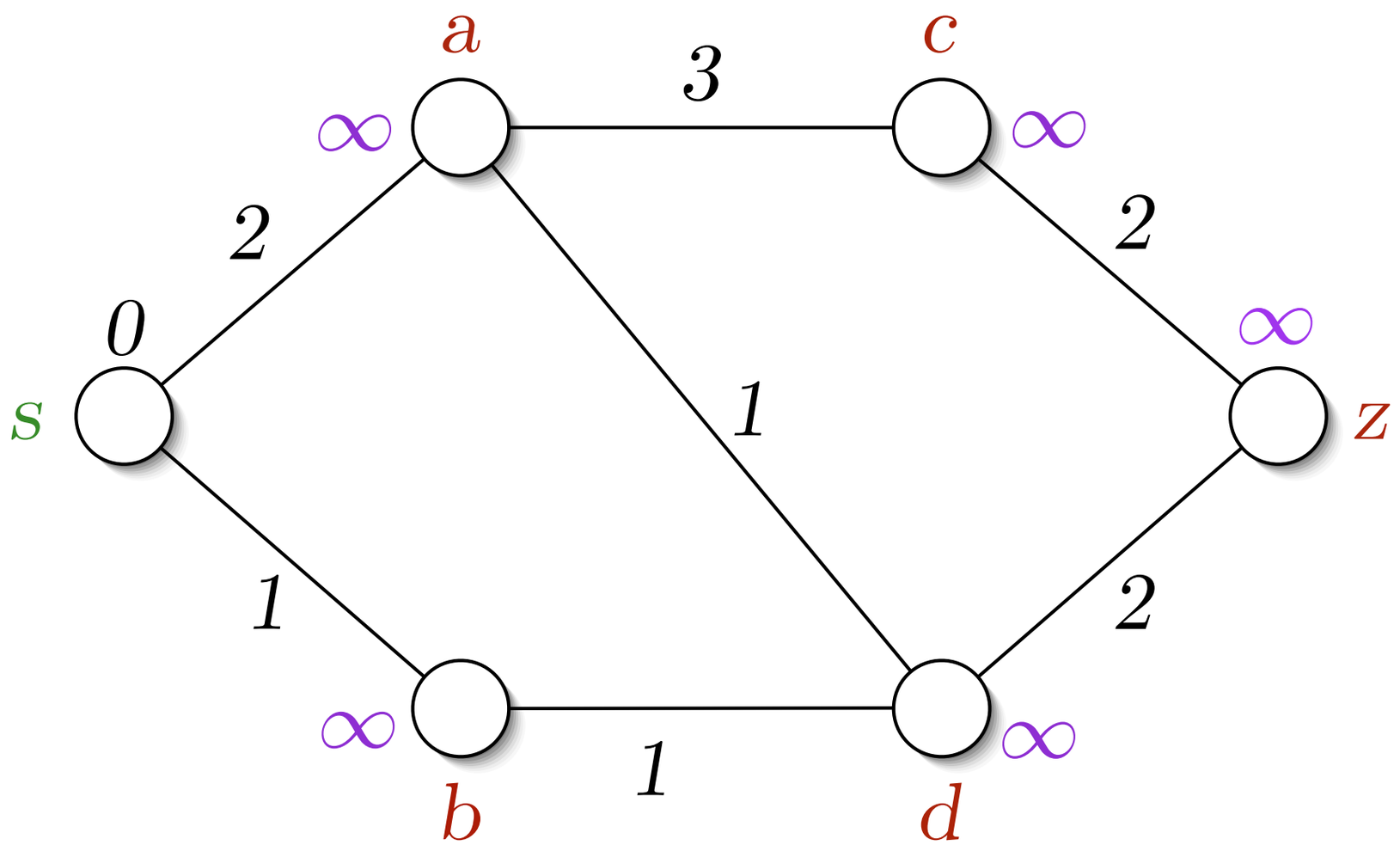
Man kann bei dünnen Graphen ($|E|$ viel kleiner als $|V|$) durch geschickte Datenstrukturen, wie Fibonacci-Heaps, zu Laufzeiten der Ordnung $|E| + |V| \cdot \ell og |V|$ kommen.

Auch ergeben sich sofort Varianten, die alle kürzesten Pfade von s aus zu allen Knoten bestimmen, und solche, die Digraphen (gerichtete Graphen) betrachten. $\Rightarrow Literatur$

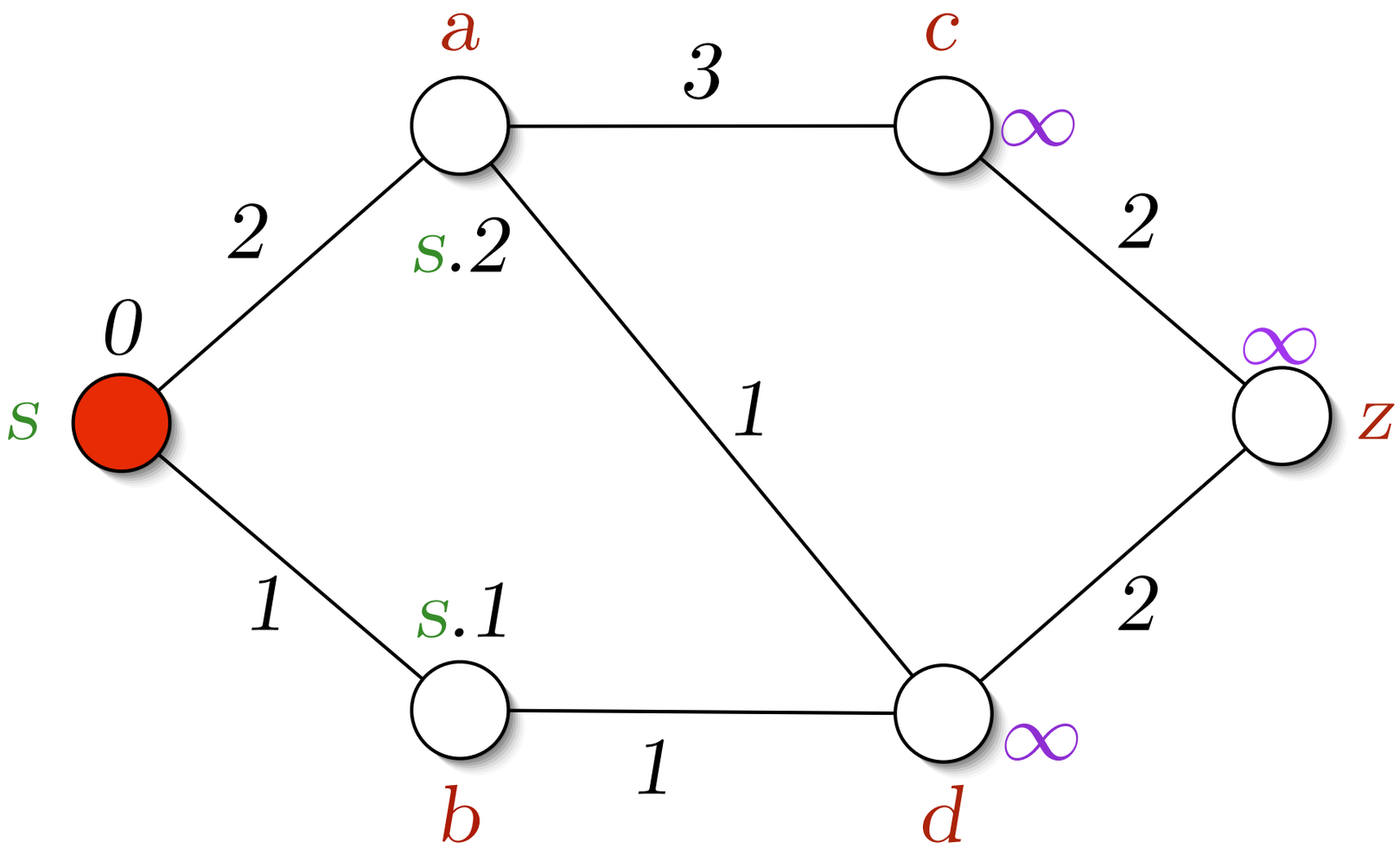
Ein Beispiel
-1-



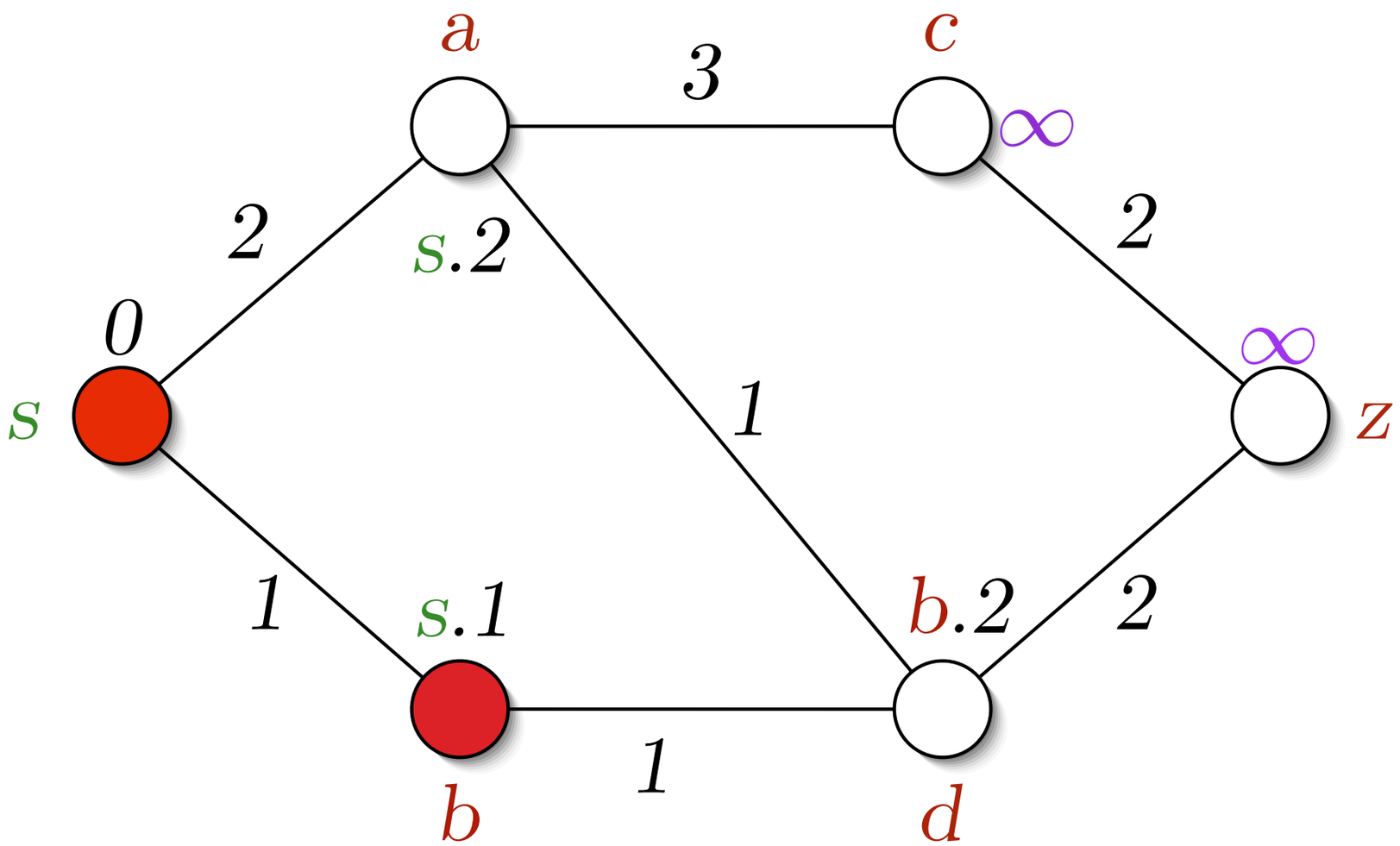
Ein Beispiel
-2-



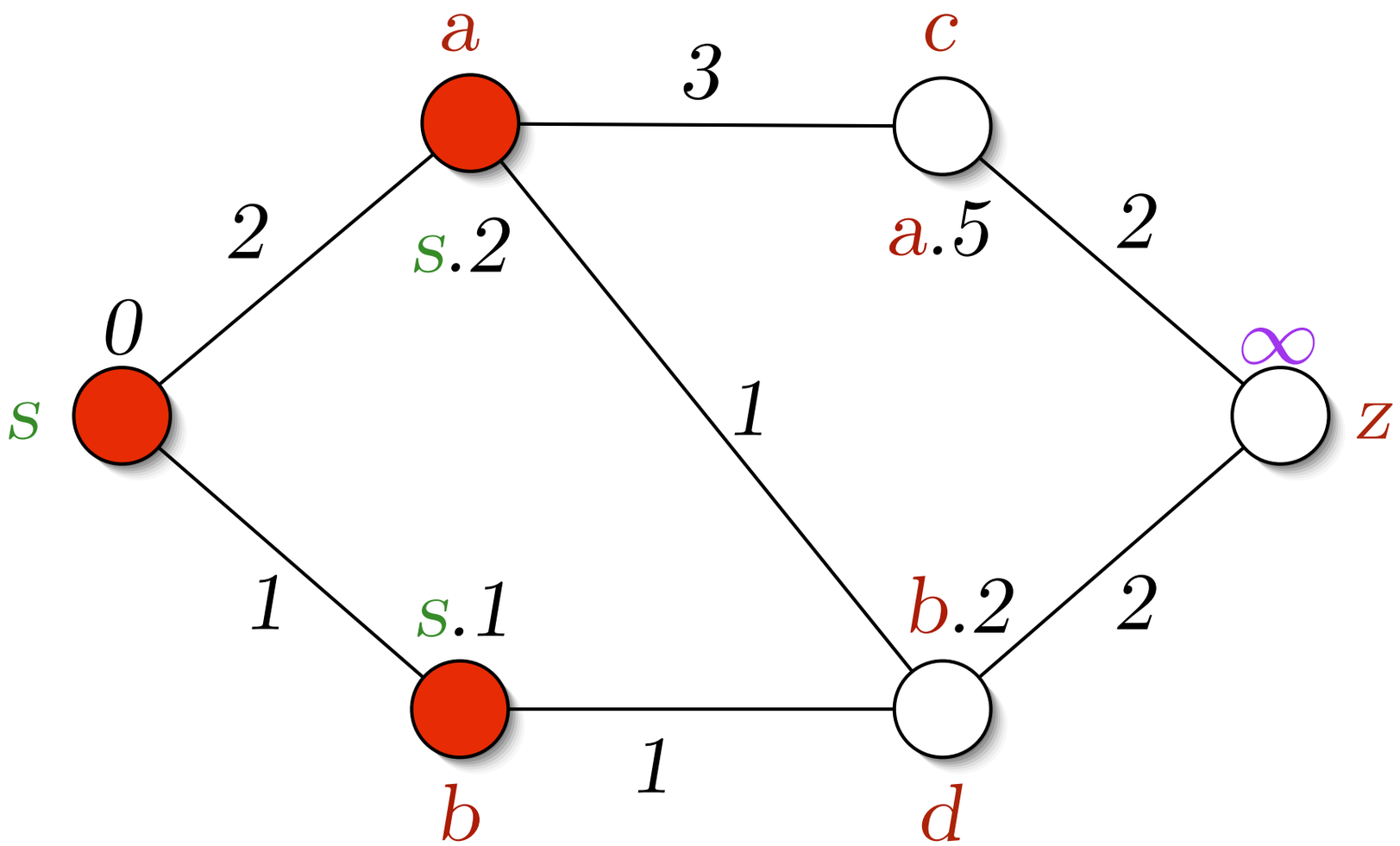
Ein Beispiel
-3-



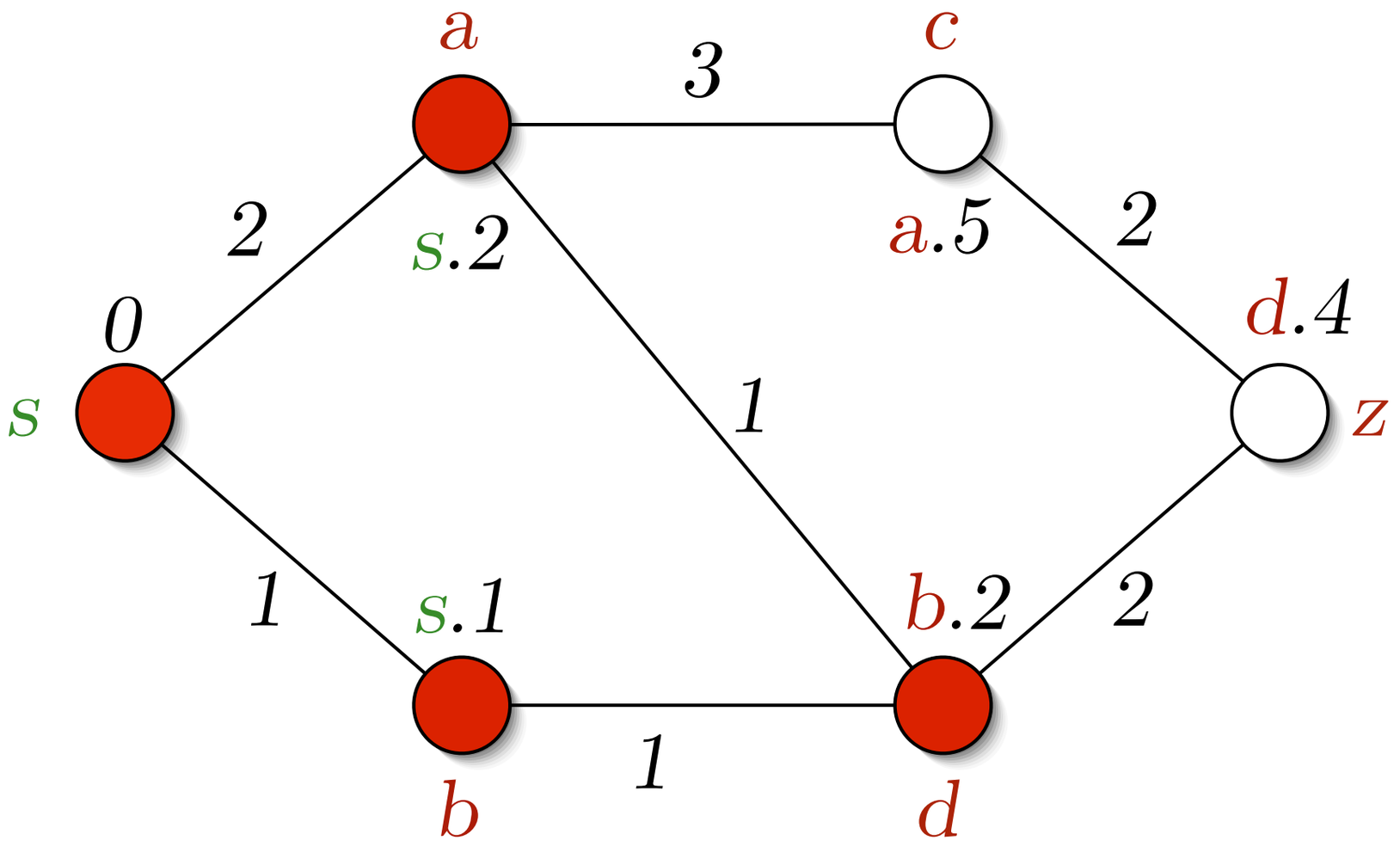
Ein Beispiel
-4-



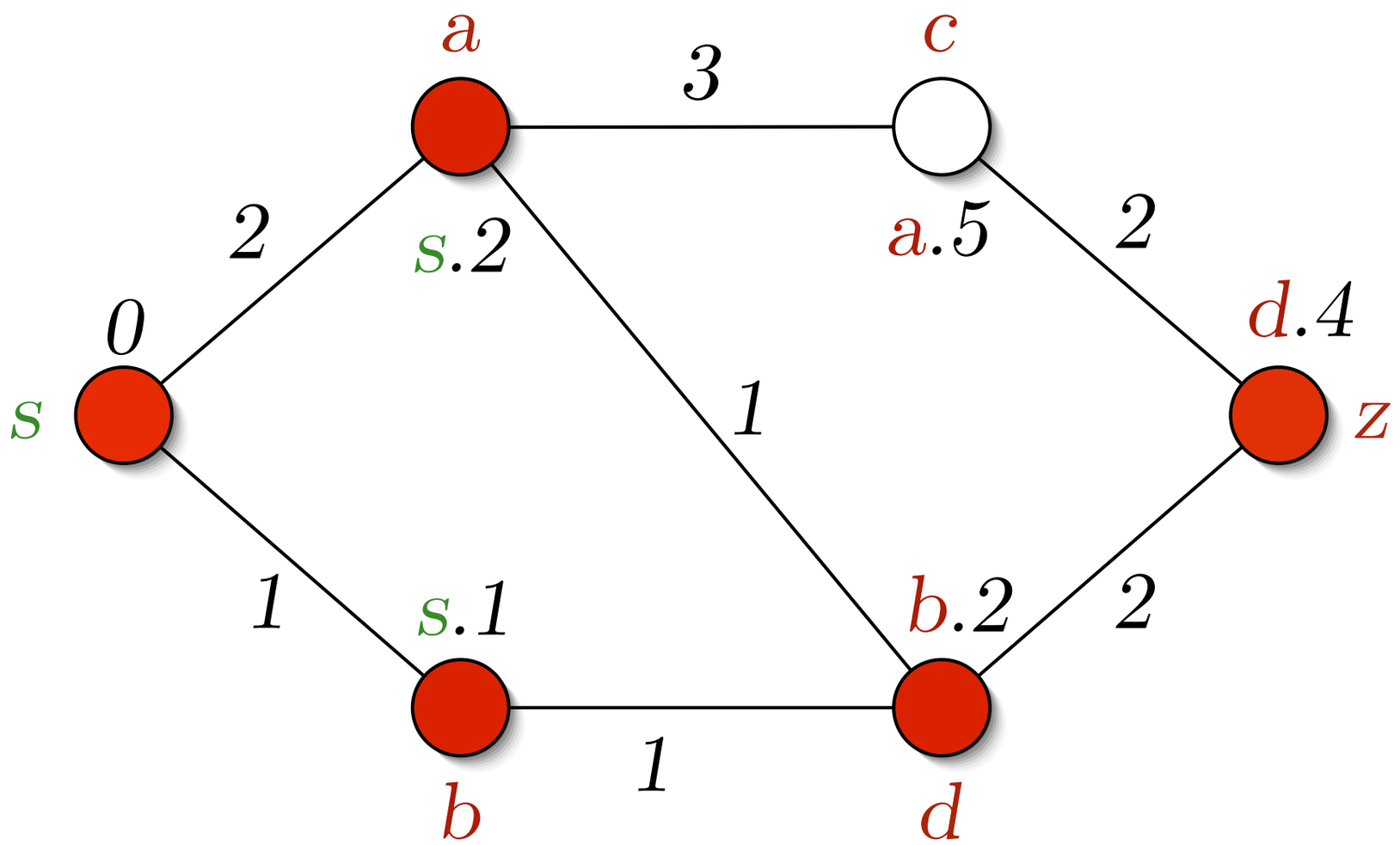
Ein Beispiel
-5-



Ein Beispiel
-6-



Ein Beispiel



Alle kürzesten Wege von einem Startknoten zu allen anderen Knoten

Beachte:

Ein Weg ist eine verbundene Kantenfolge.
Ein Pfad ist eine Weg, in dem jeder Knoten nur einmal besucht wird.

Das Verfahren von Dijkstra kann mit der gleichen Komplexität auch zum Bestimmen aller kürzesten Pfade von einem Startpunkt aus verwendet werden!

Es entsteht ein sogenannter Kürzeste-Wege-Baum

Anwendungsbeispiele:

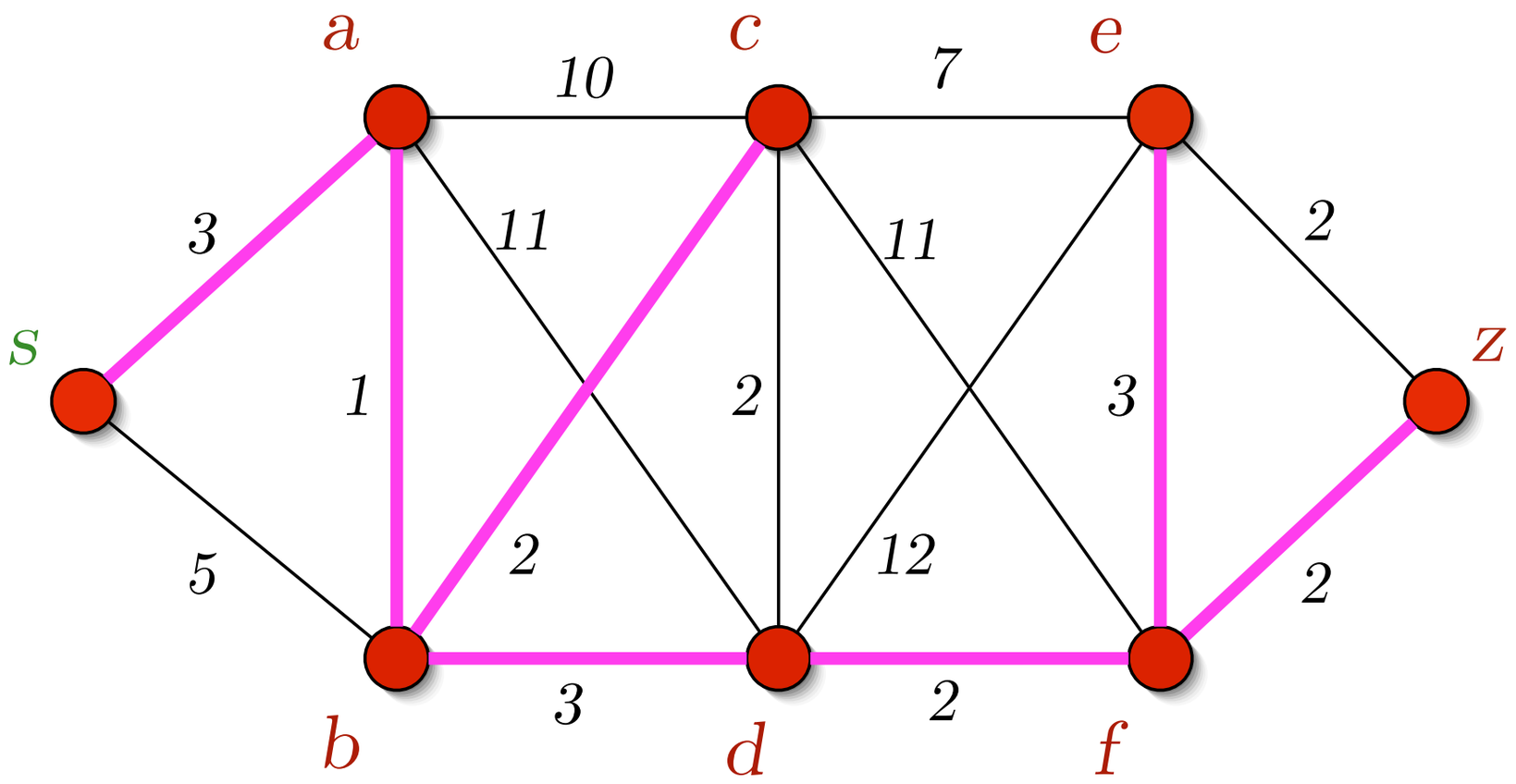
- Routenplaner im PKW (Ziel als Startknoten, um vom abseits liegenden Punkt wieder auf den richtigen Weg zu finden!)
- Bahnauskunft

Jeder kürzeste Wegebaum ist ein aufspannender Baum (Gerüst) des Graphen.

Ist das immer auch ein minimales Gerüst? Nein!

Wieso nicht?

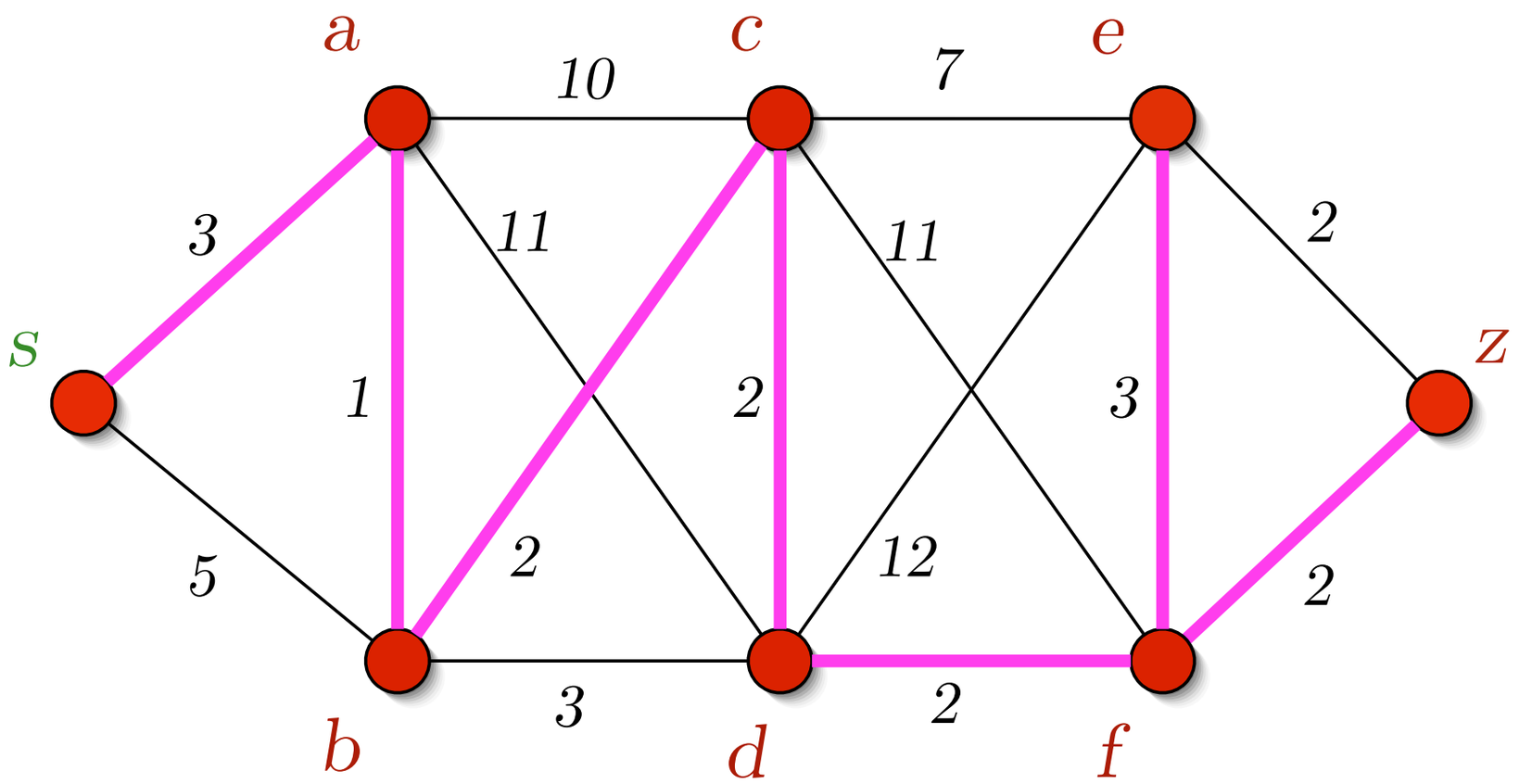
Ein Graph mit Kürzeste-Wege-Baum nach Dijkstra:



Spannbaumgewicht = 16

Aber es geht mit kleinerem Gewicht!

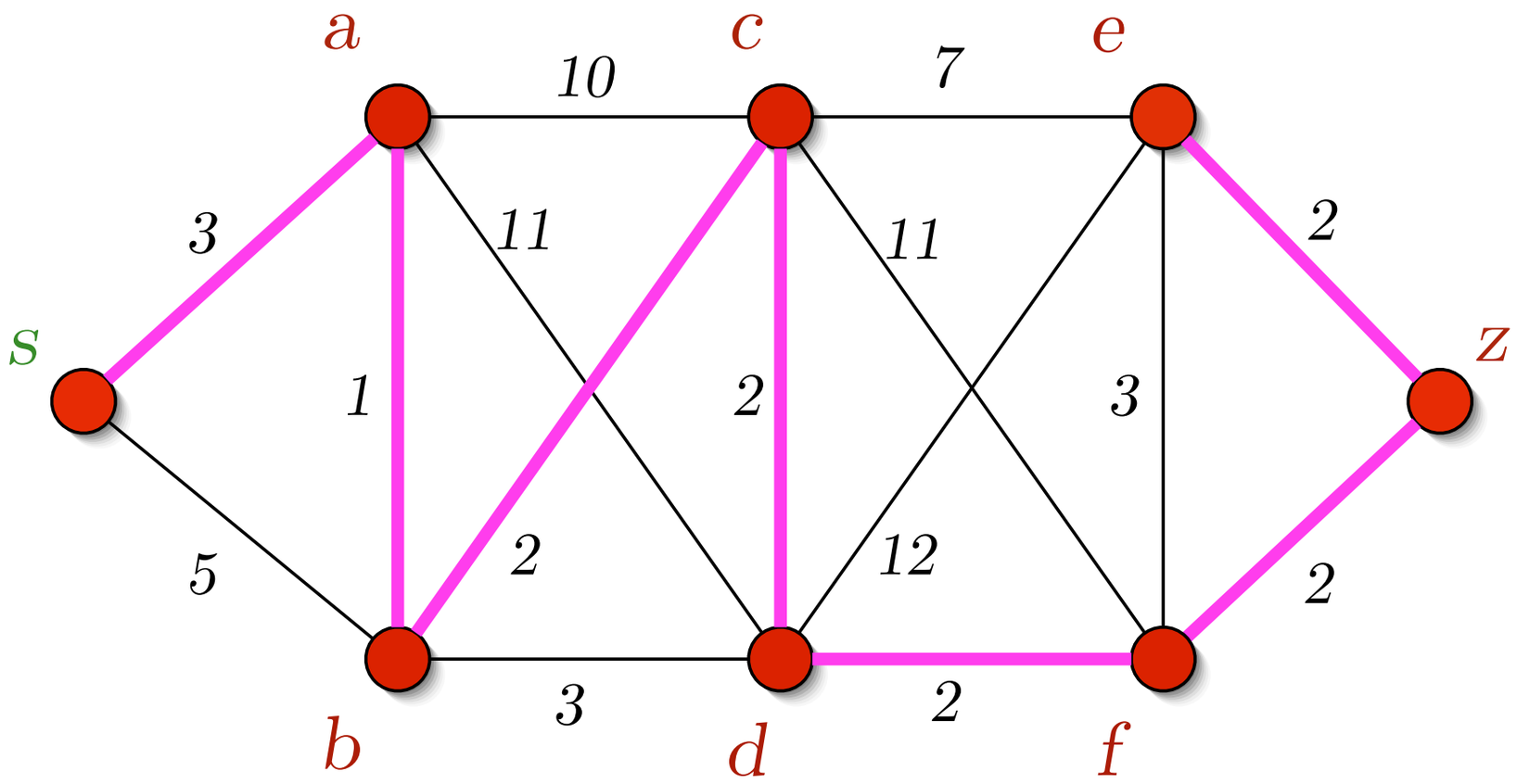
Ein besseres Gerüst:



Spannbaumgewicht $= 15$

Aber es geht mit noch kleinerem Gewicht!

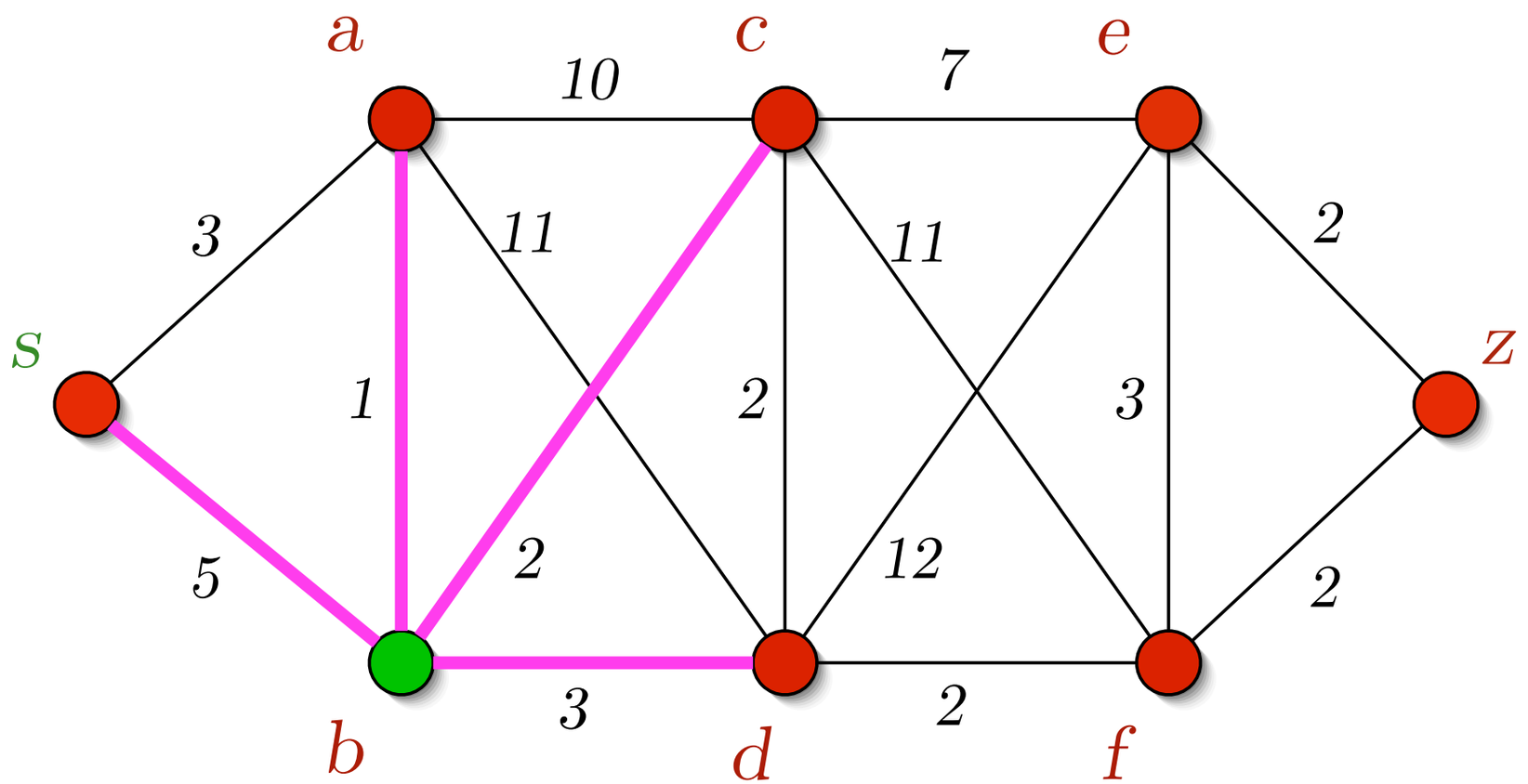
Zum Beispiel auf diese Weise, und das ist minimal:



Spannbaumgewicht = 14, Gerüst ist minimal!

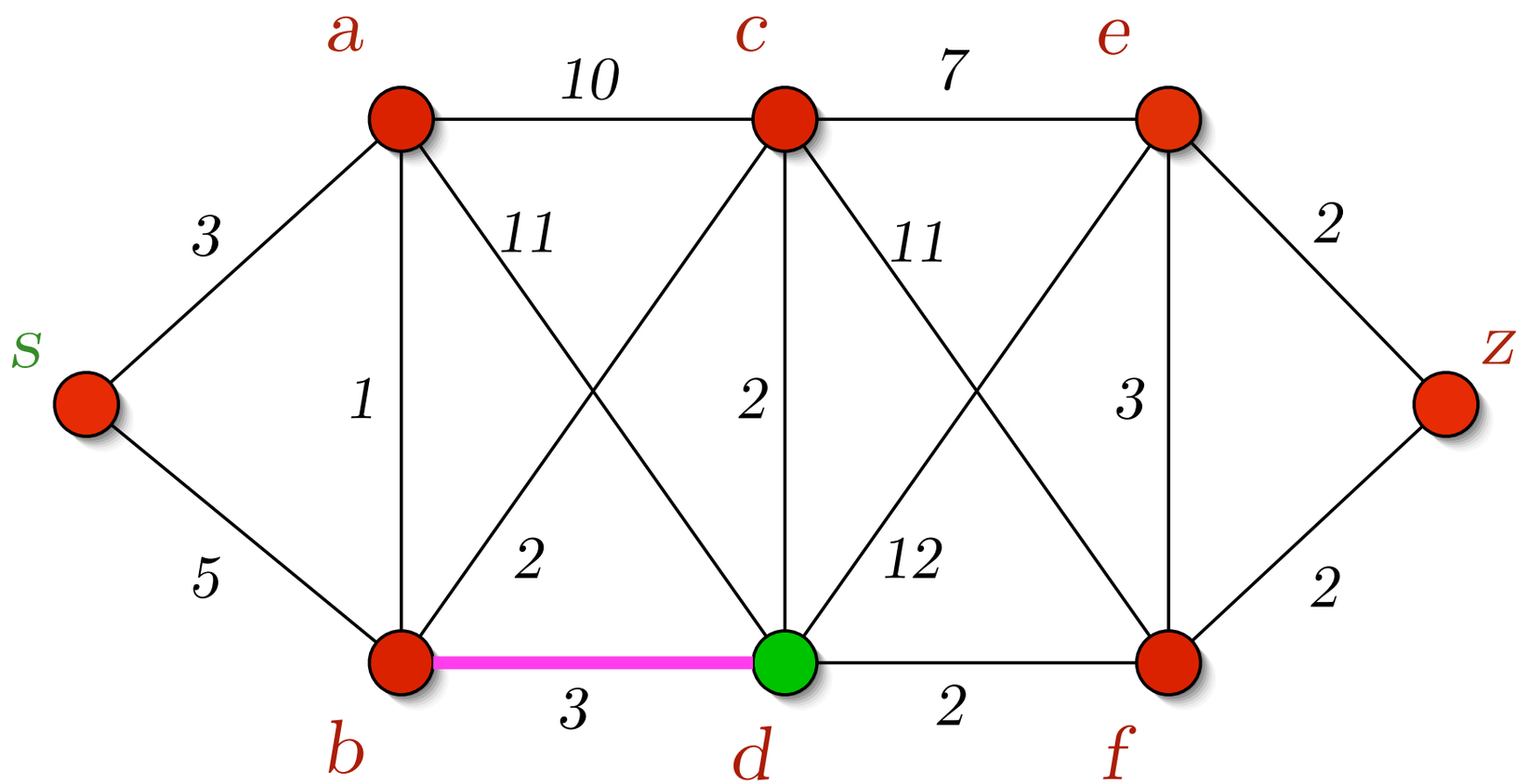
Kann ein minimales Gerüst immer als Kürzeste-Wege-Baum mit dem Algorithmus von Dijkstra gefunden werden?

Nein, ein minimales Gerüst taucht nicht immer als Kürzeste-Wege-Baum auf.



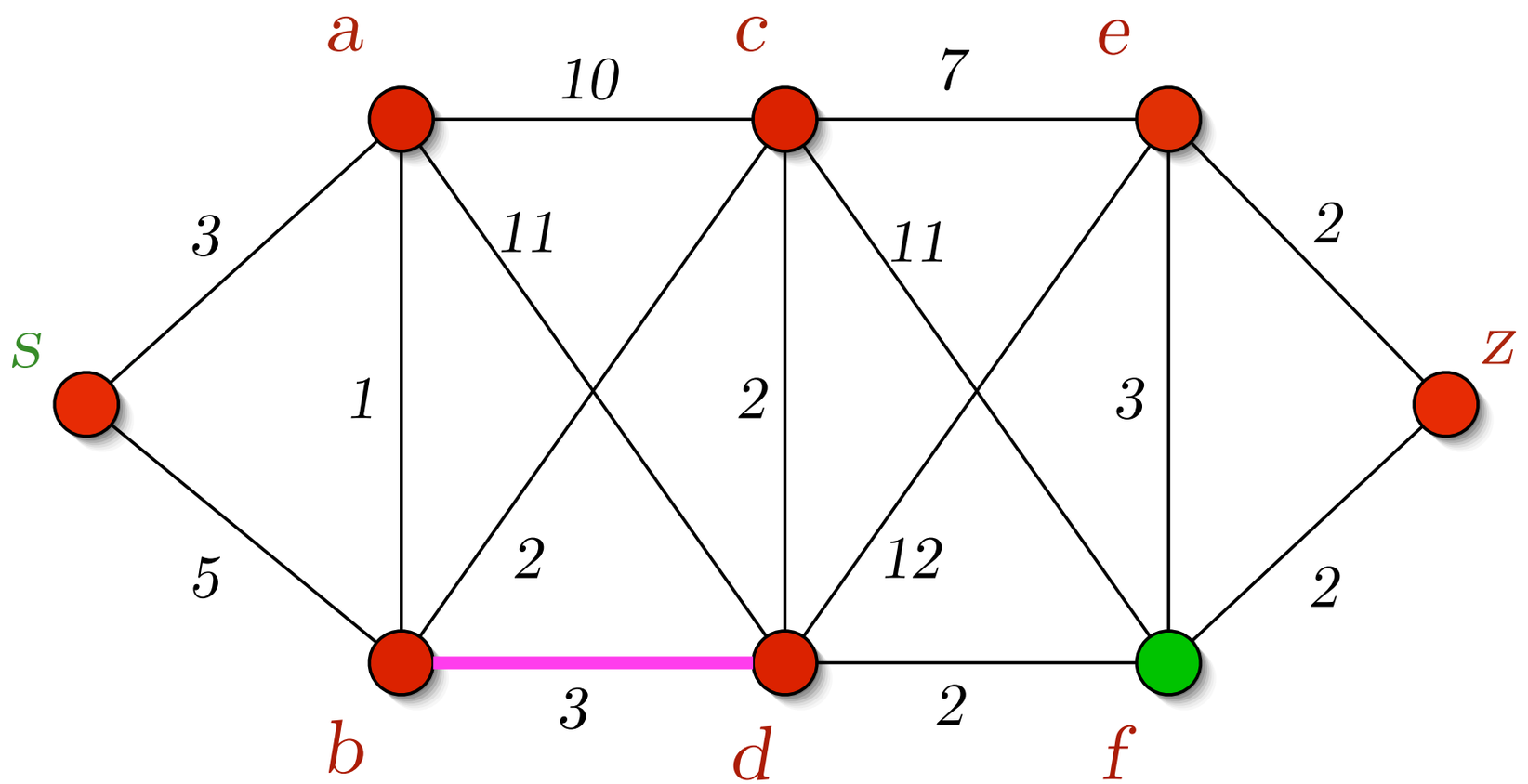
Kante b-d im kürzeste Wege Baum, nicht im minimalen Gerüst!

Nein, ein minimales Gerüst taucht nicht immer als Kürzeste-Wege-Baum auf.



Kante b-d im kürzeste Wege Baum, nicht im minimalen Gerüst!

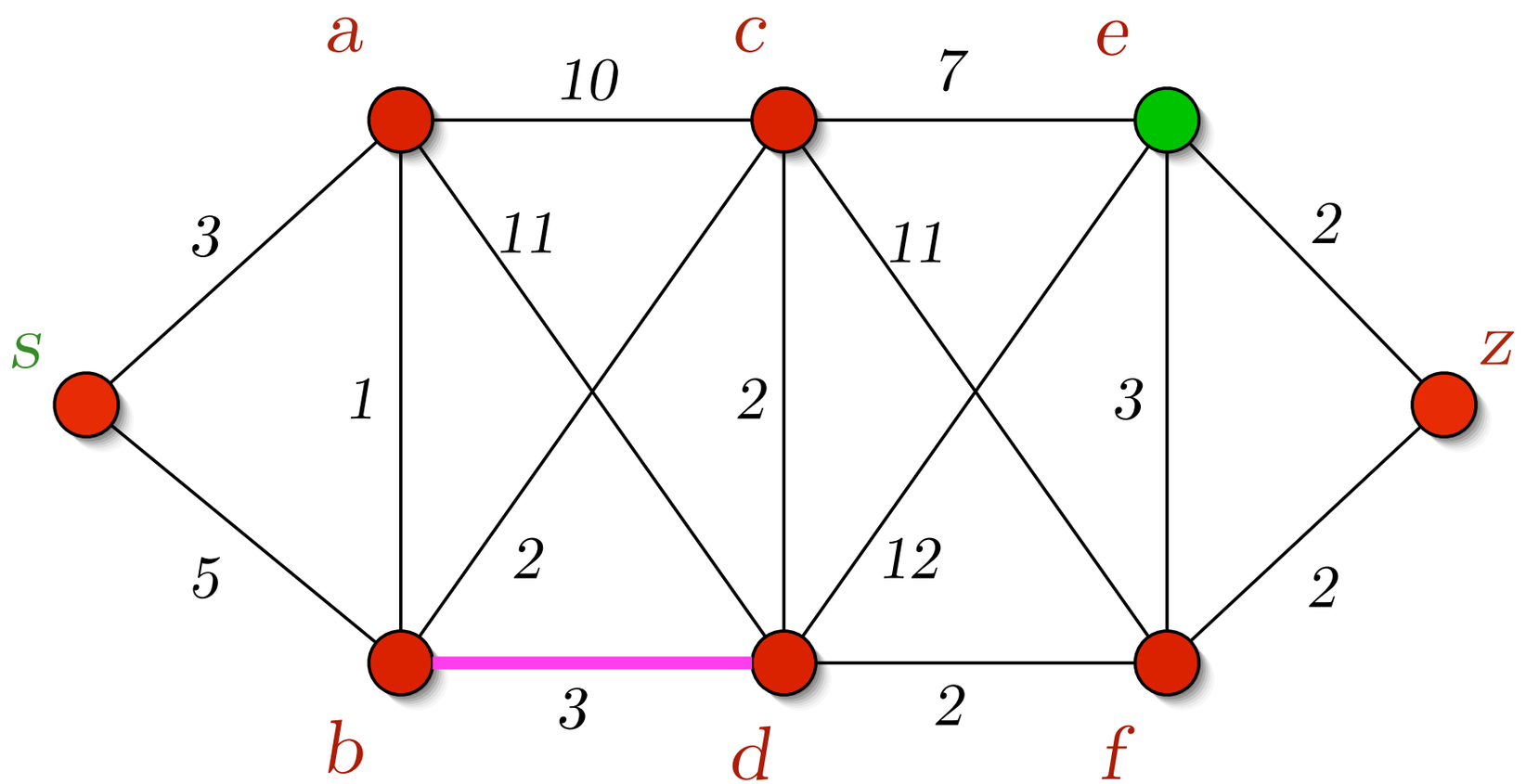
Nein, ein minimales Gerüst taucht nicht immer als Kürzeste-Wege-Baum auf.



Kante b-d im kürzeste Wege Baum, nicht im minimalen Gerüst!

Mit f als Startpunkt, wird immer auch Kante b-d gewählt!

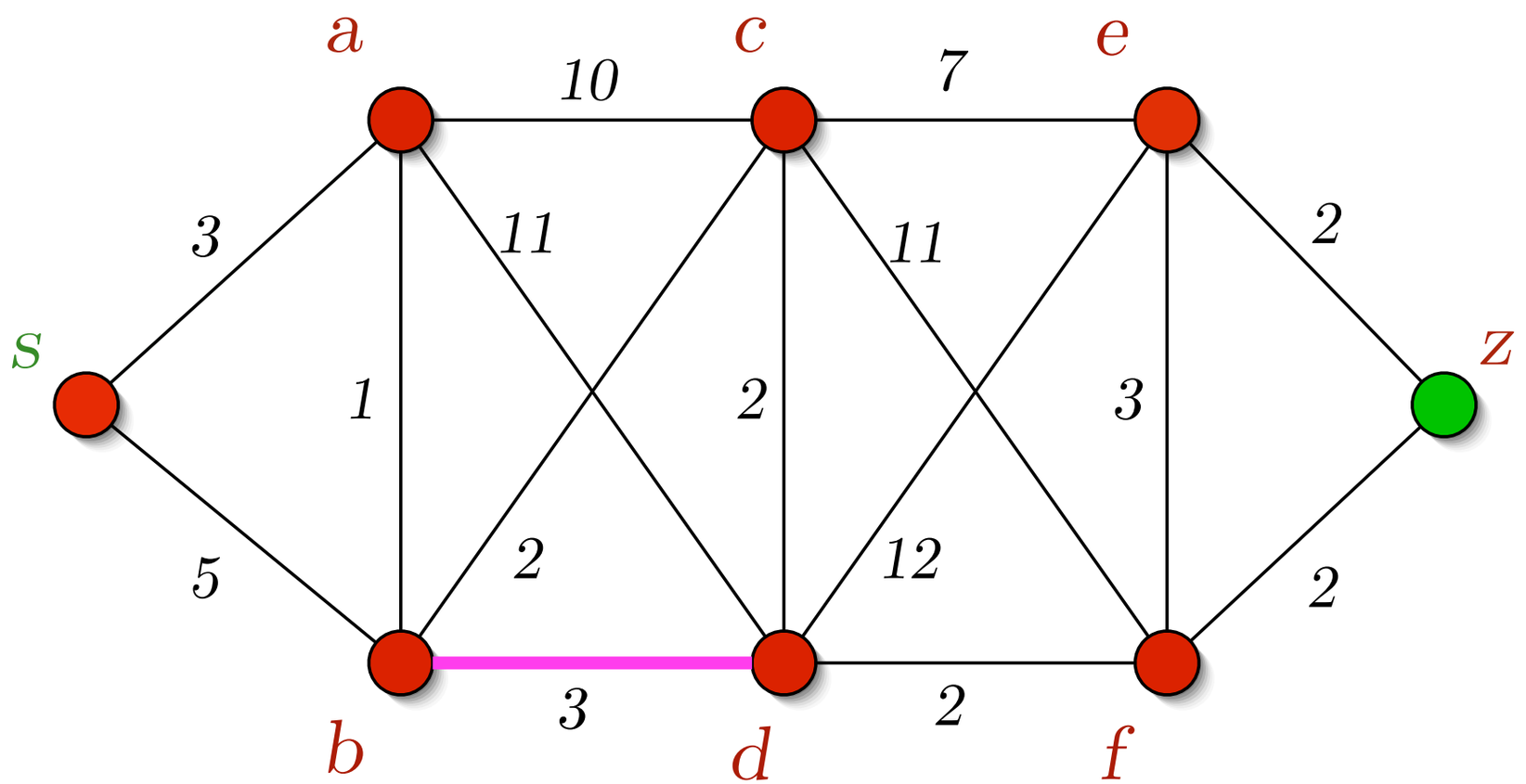
Nein, ein minimales Gerüst taucht nicht immer als Kürzeste-Wege-Baum auf.



Kante b-d im kürzeste Wege Baum, nicht im minimalen Gerüst!

Mit *e* als Startpunkt, wird immer auch Kante *b*-*d* gewählt!

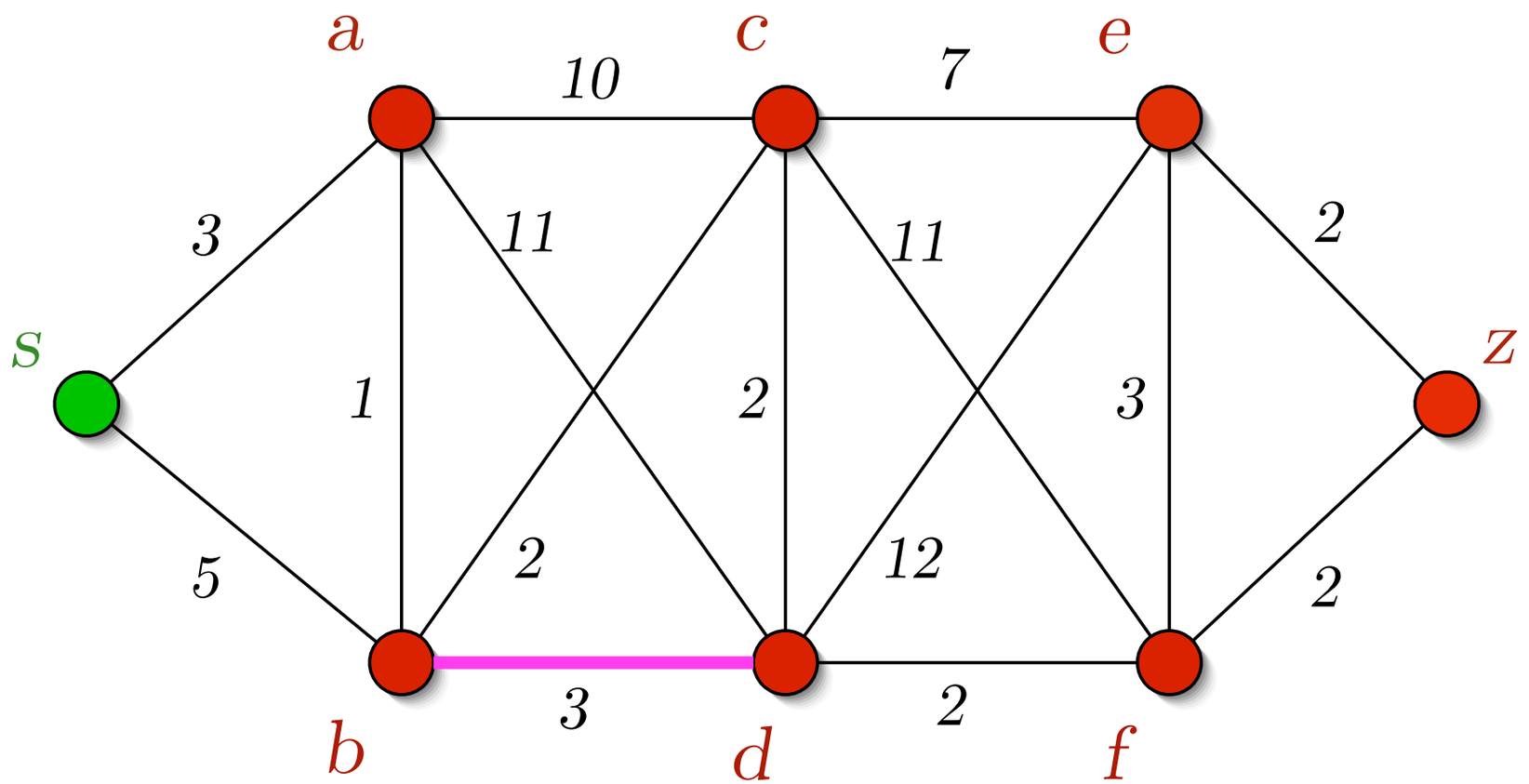
Nein, ein minimales Gerüst taucht nicht immer als Kürzeste-Wege-Baum auf.



Kante b-d im kürzeste Wege Baum, nicht im minimalen Gerüst!

Mit z als Startpunkt, wird immer auch Kante b-d gewählt!

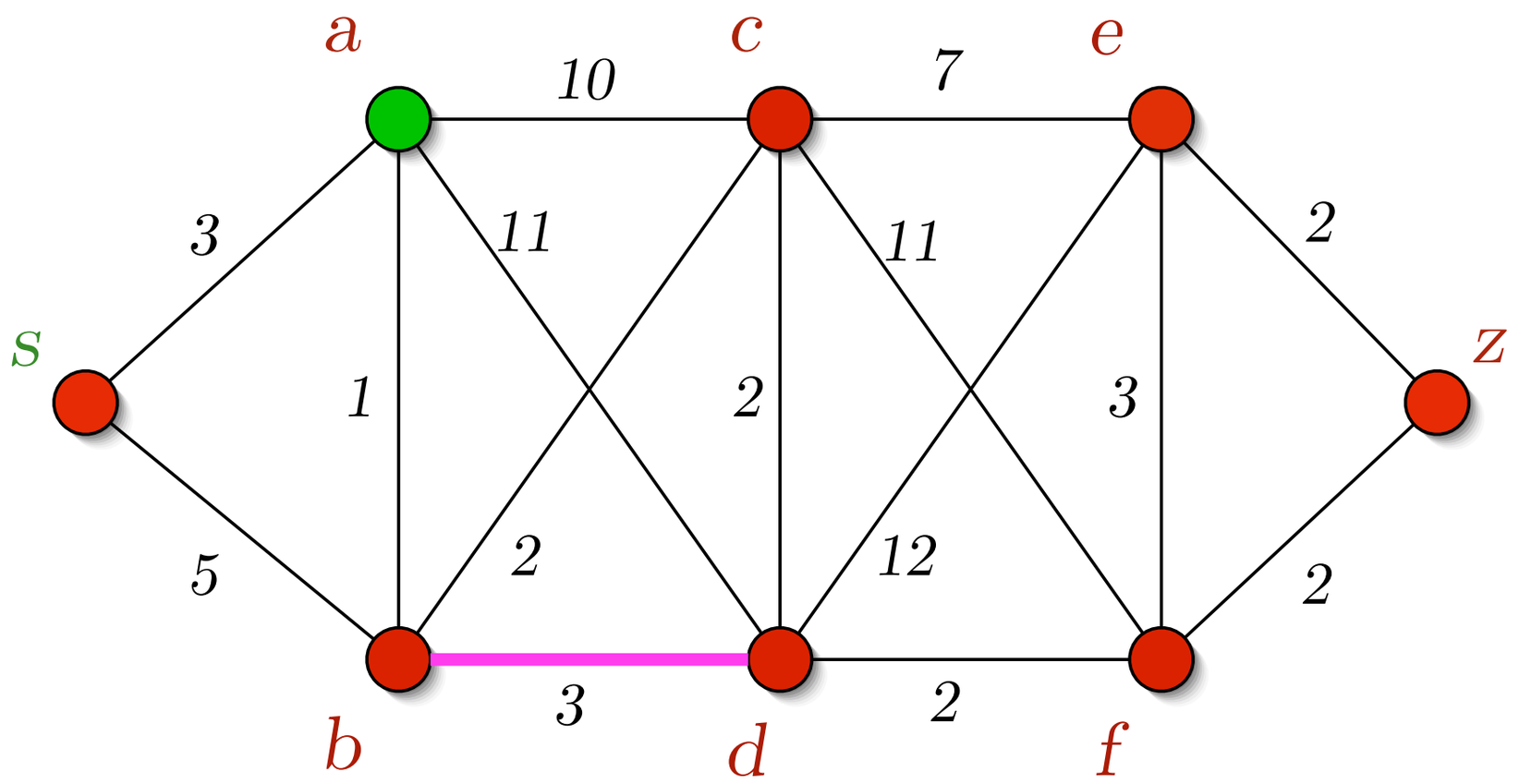
Nein, ein minimales Gerüst taucht nicht immer als Kürzeste-Wege-Baum auf.



Kante b-d im kürzeste Wege Baum, nicht im minimalen Gerüst!

Mit s als Startpunkt, wurde auch Kante b-d gewählt!

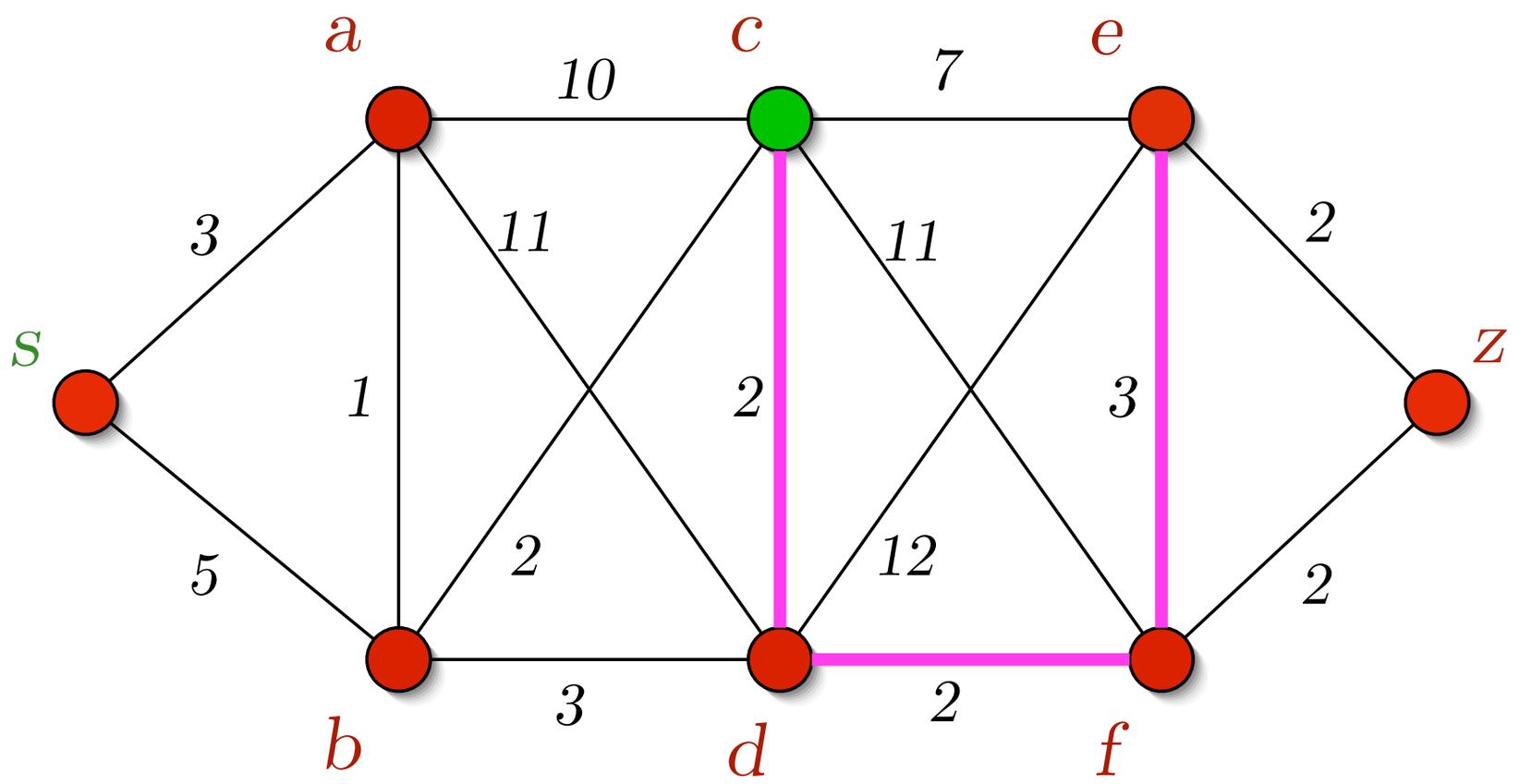
Nein, ein minimales Gerüst taucht nicht immer als Kürzeste-Wege-Baum auf.



Kante b-d im kürzeste Wege Baum, nicht im minimalen Gerüst!

Auch mit a als Startpunkt, wird Kante b-d gewählt!

Nein, ein minimales Gerüst taucht nicht immer als Kürzeste-Wege-Baum auf.



c--e Weg über c-d-f-$e = 7$, oder direkt $= 7$, aber keine der Kanten c-e oder f-e gehören zum minimalen Gerüst!

Es bleibt c als Startpunkt, und dann wird c-e oder f-e als Kante gewählt!

Minimale Gerüste findet man mit anderen Verfahren!

Der Algorithmus von **Kruskal** hat eine Komplexität von $O(m{\cdot}\ell ogn\ +\ n^2)$ und bei besserer Implementation mit einer Halde (*Heap*) eine von: $O(m{\cdot}\ell ogn)$, wobei $m{:=}|E|$ und $n{:=}|V|$ ist. Im schlechtesten Fall gilt $\Theta(m{\cdot}\ell ogm)$.

Das Verfahren von **Prim** ist eine direkte Abwandlung des Dijkstra-Algorithmus für kürzeste Wege. Bei einfachster Implementation hat es eine ähnliche Komplextät von $O(n^2)$.

Bei Benutzung einer Priority-Queue in einer Implementierung als Fibonacci-Heap kann man $O(m\ +\ n{\cdot}\ell ogn)$ erreichen. Im schlechtesten Fall gilt hier $\Theta(n^2)$.

Bei dünnen Graphen (m viel kleiner als n^2) empfielt sich **Kruskal**, bei dichten Graphen (m nahe an n^2) empfielt sich eher **Prim**.

Letzteren sehen wir uns nur in der einfachen Variante an:

Minimales Gerüst, Algorithmus von R.C. Prim (1957)

$input$: ungerichteter Graph $G{:=}(V,\ E),\ s{,}z \in V$, Gewichtung $w{:}E \to \mathbb{N}$.

$output$: ein minimales Gerüst und dessen Gesamtgewicht $g(z)$.

$begin\ g(s) \leftarrow 0$

$\quad for\ all\ v{\in}V \setminus \{s\}\ do$

$\quad\quad g(v) \leftarrow \infty;$

$\quad\quad T \leftarrow V$ % T sind die Knoten, die *noch nicht* zum Gerüst gehören

$\quad end\ for$

$\quad while\ T{\neq}\varnothing\ do$

$\quad\quad begin$

$\quad\quad$ ☺ wähle $v^* \in T$ mit $g(v^*) = min\{g(v) \mid v \in T\};$

$\quad\quad T \leftarrow T \setminus \{v^*\}$

$\quad\quad\quad if\ w(\{v^*{,}v\})\} < g(v)$ % Abstand zum Rand ist Kriterium

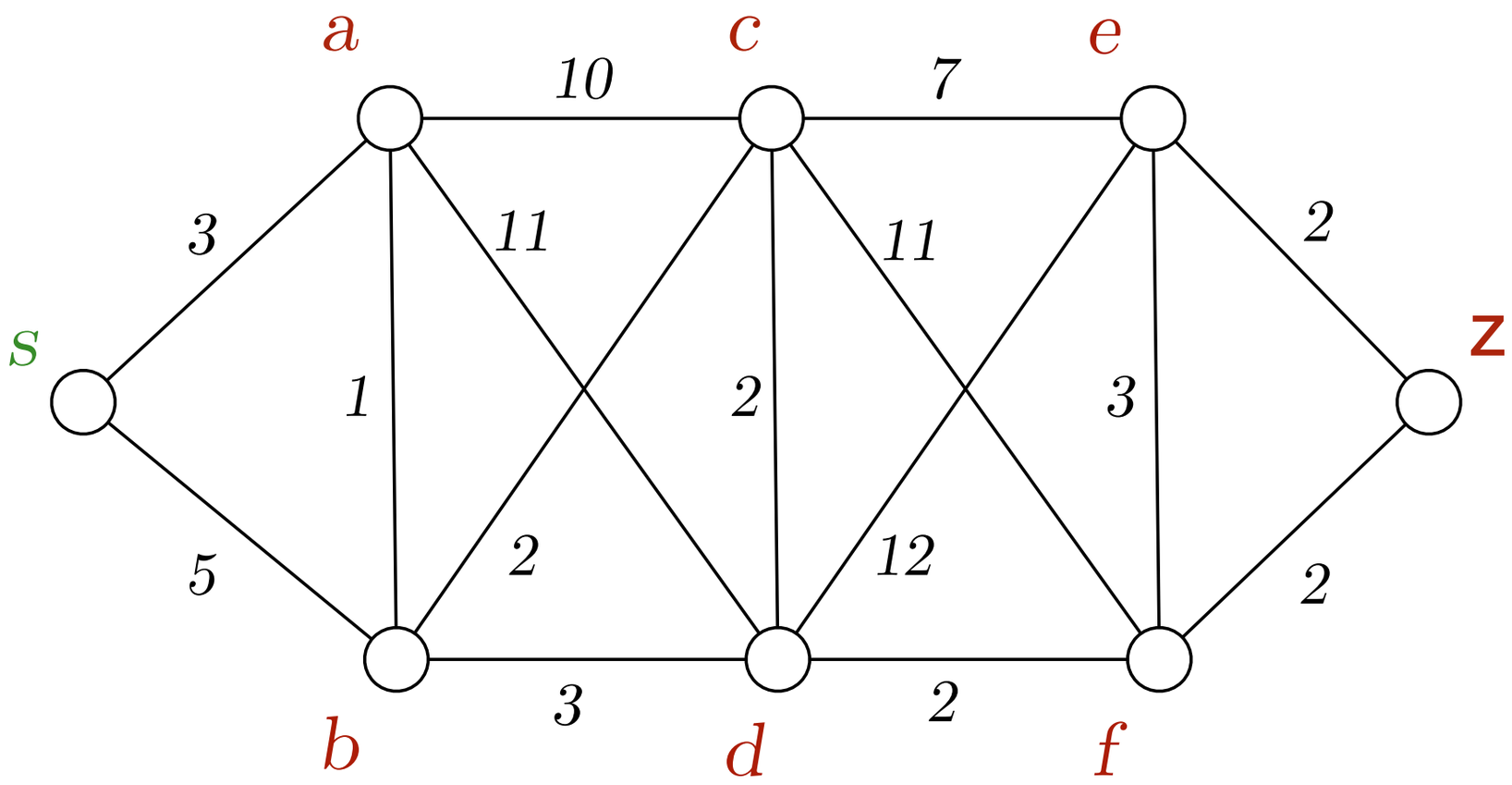
$\quad\quad\quad then\ g(v) \leftarrow w(\{v^*{,}v\});\ pre\{v\} \leftarrow v^*$ % für neue Bewertungen

$\quad\quad\quad end\ if$

$\quad\quad end$

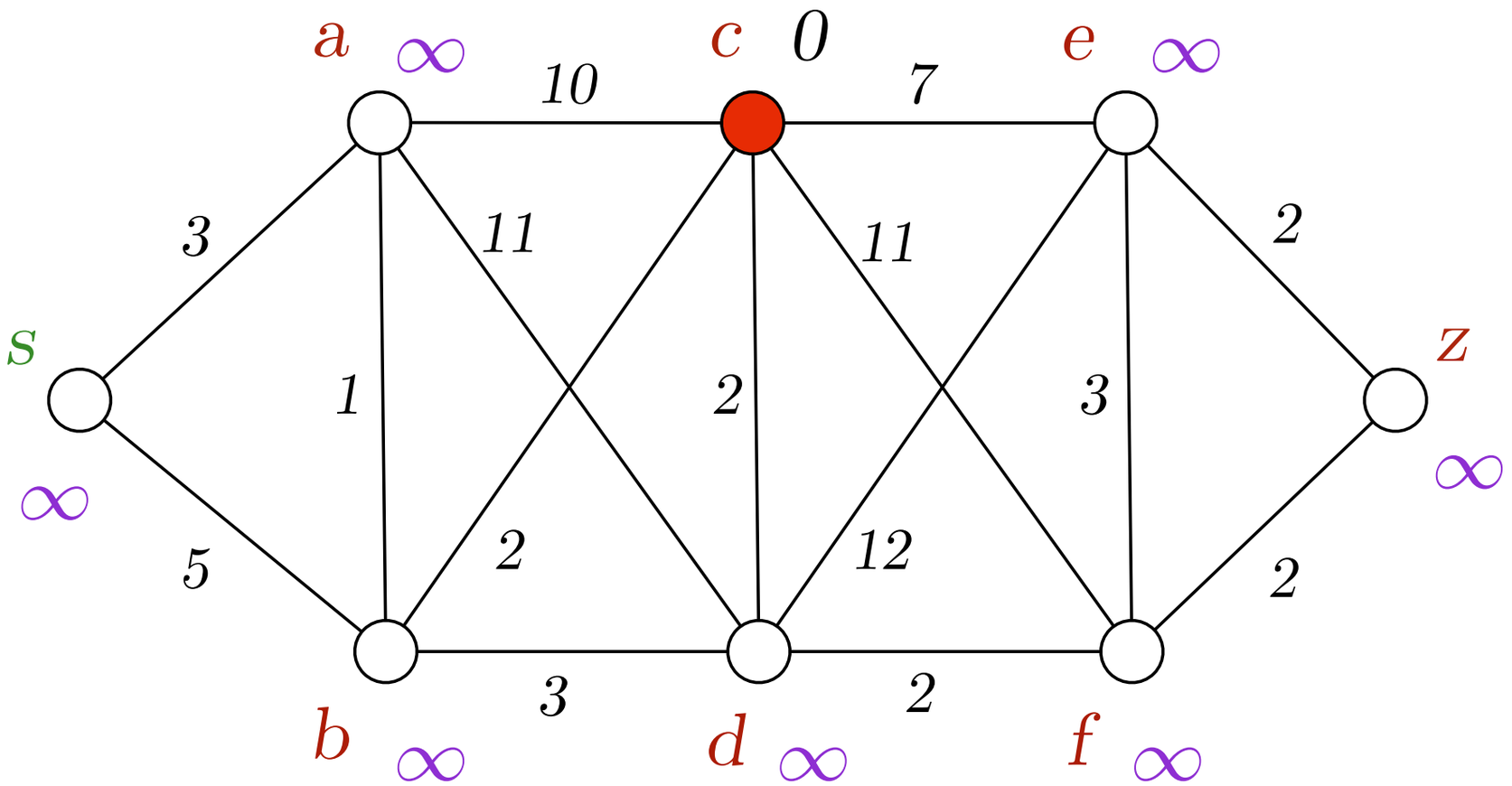
end

Anwendung am Beispiel



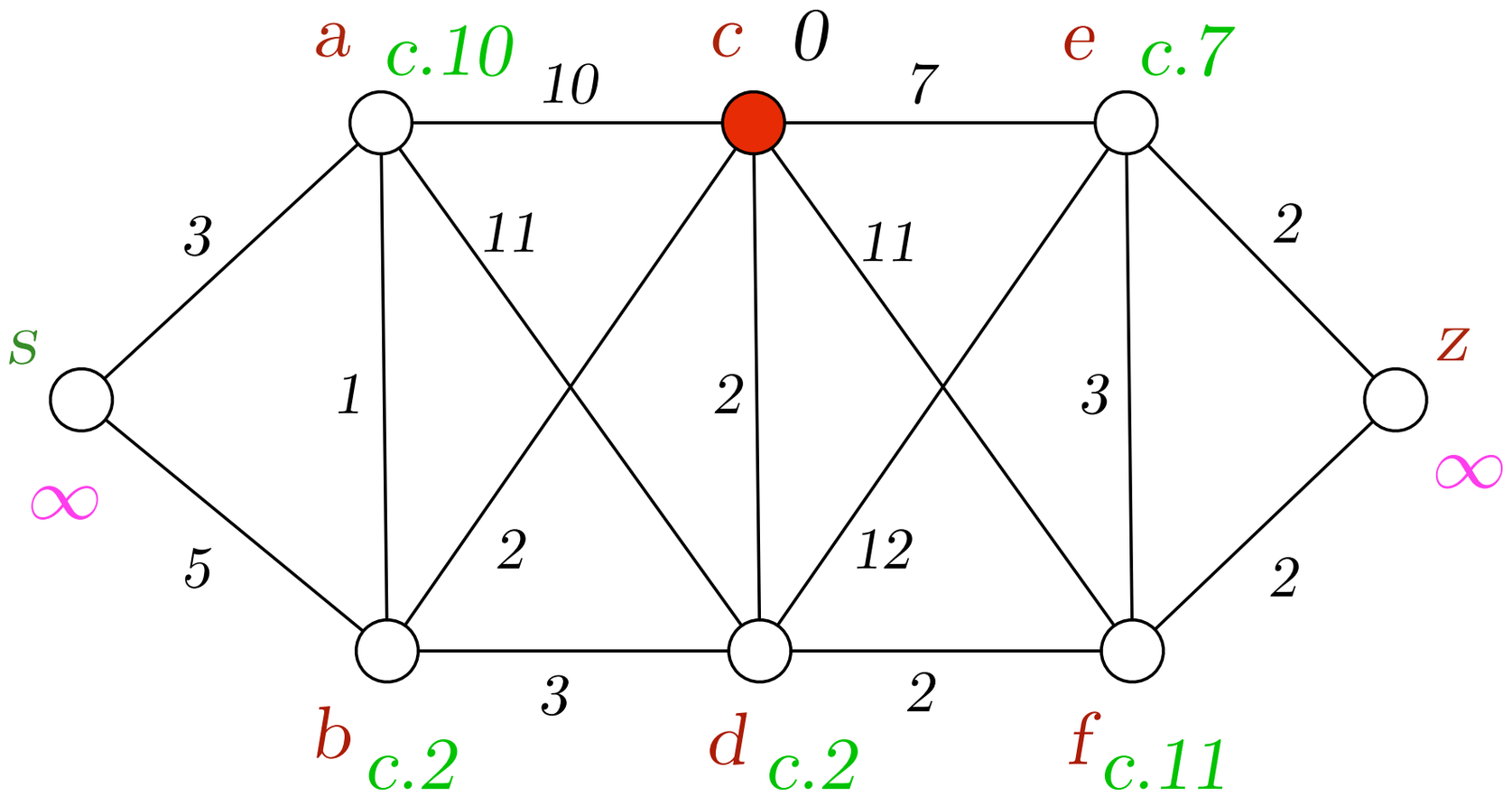
Startknoten beliebig, z.B.: c

Anwendung am Beispiel



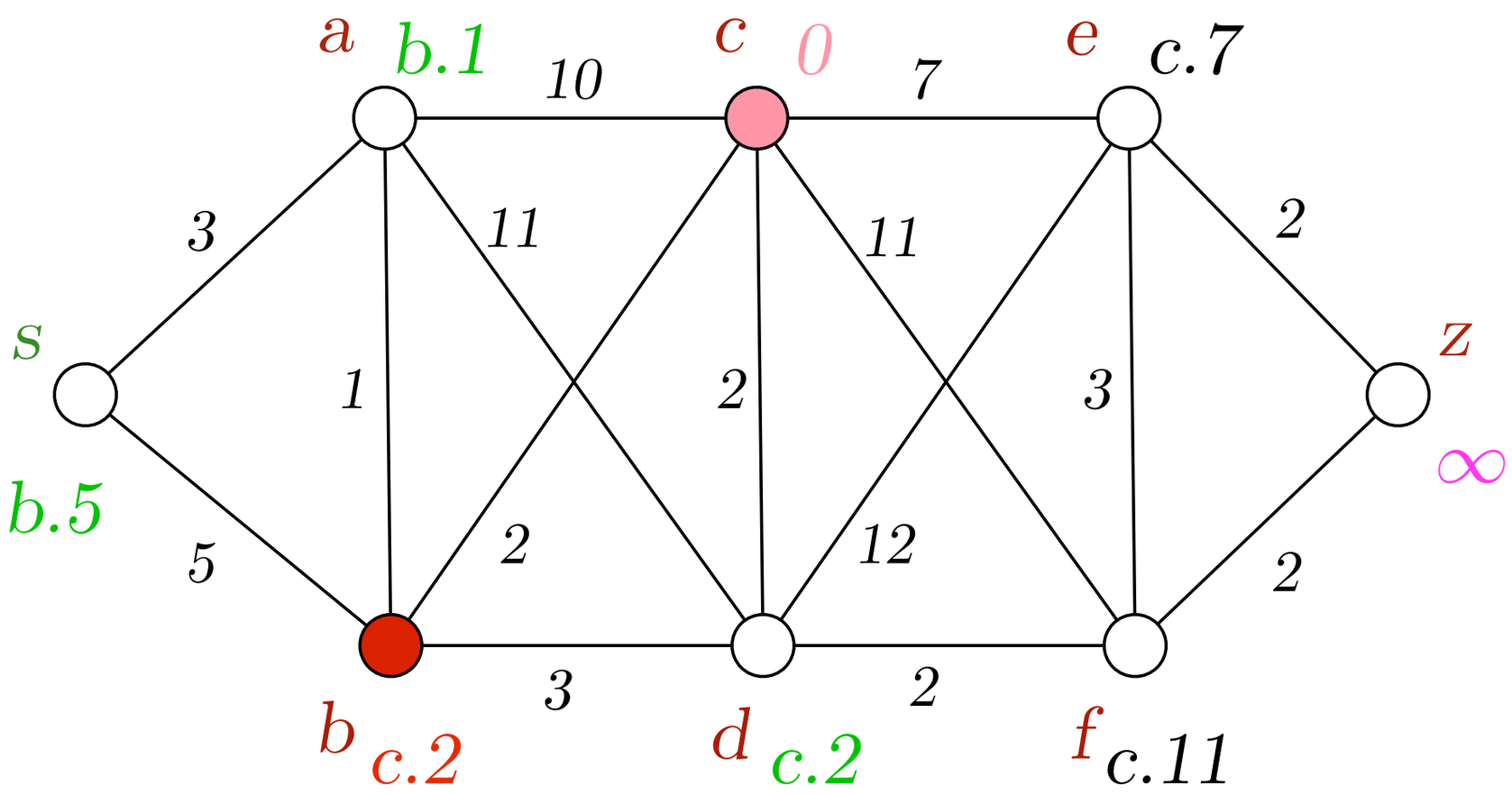
Initalisierung der Gewichte und Auswahl: c wird v^*

Anwendung am Beispiel



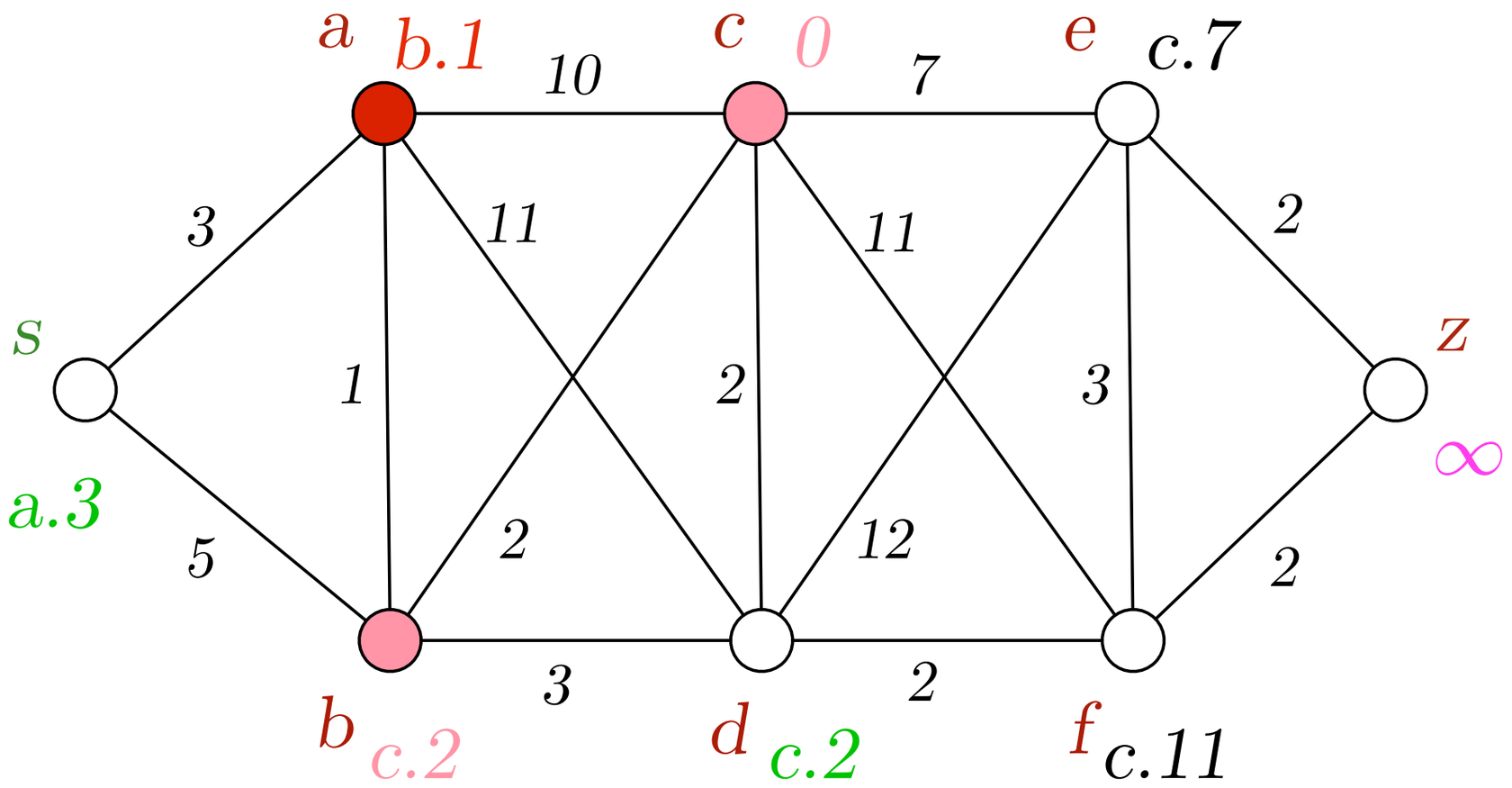
neue Gewichte und Vorgänger bestimmen

Anwendung am Beispiel



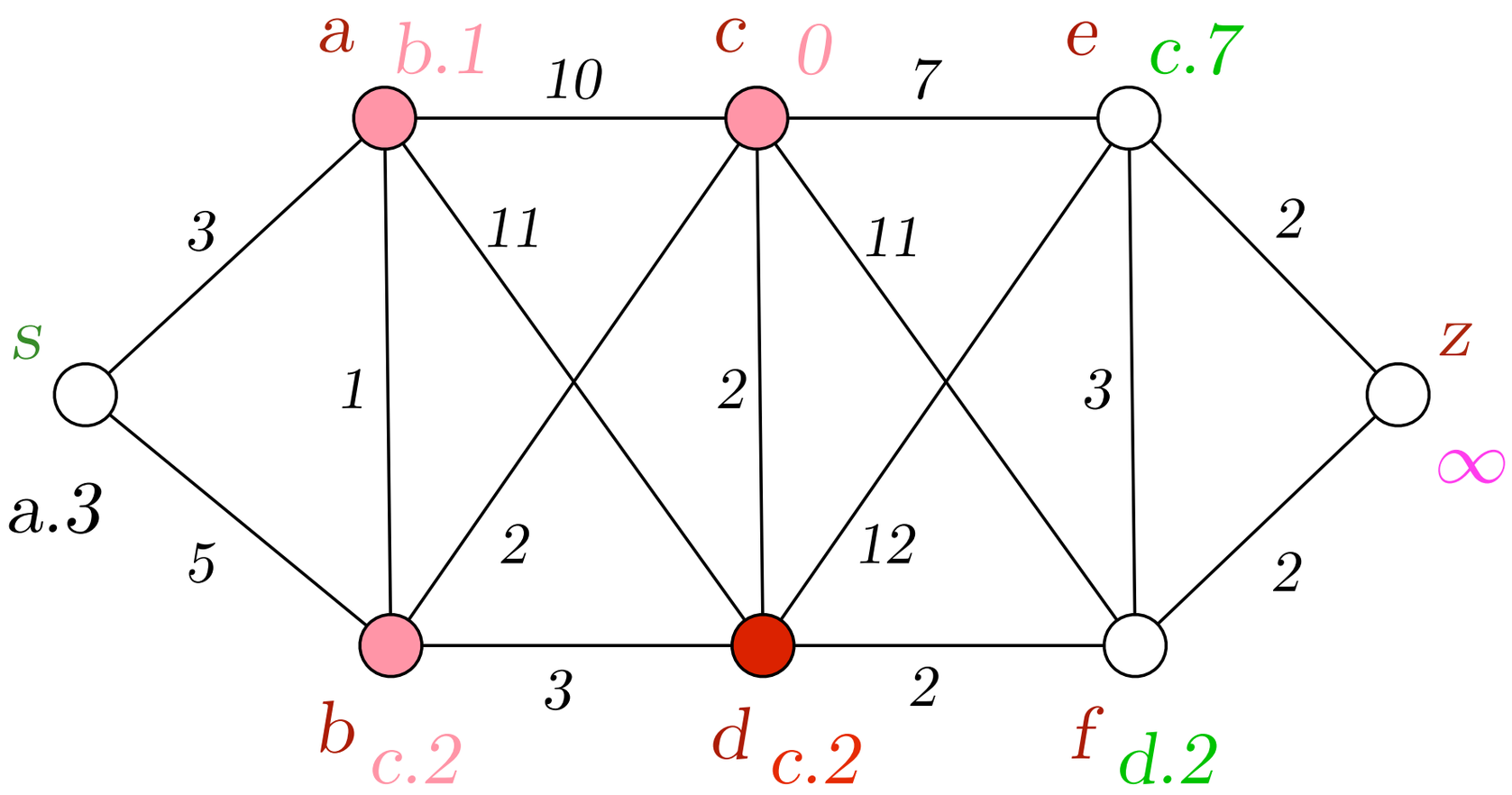
b oder c kann v^* werden, hier z.B. b!

Anwendung am Beispiel



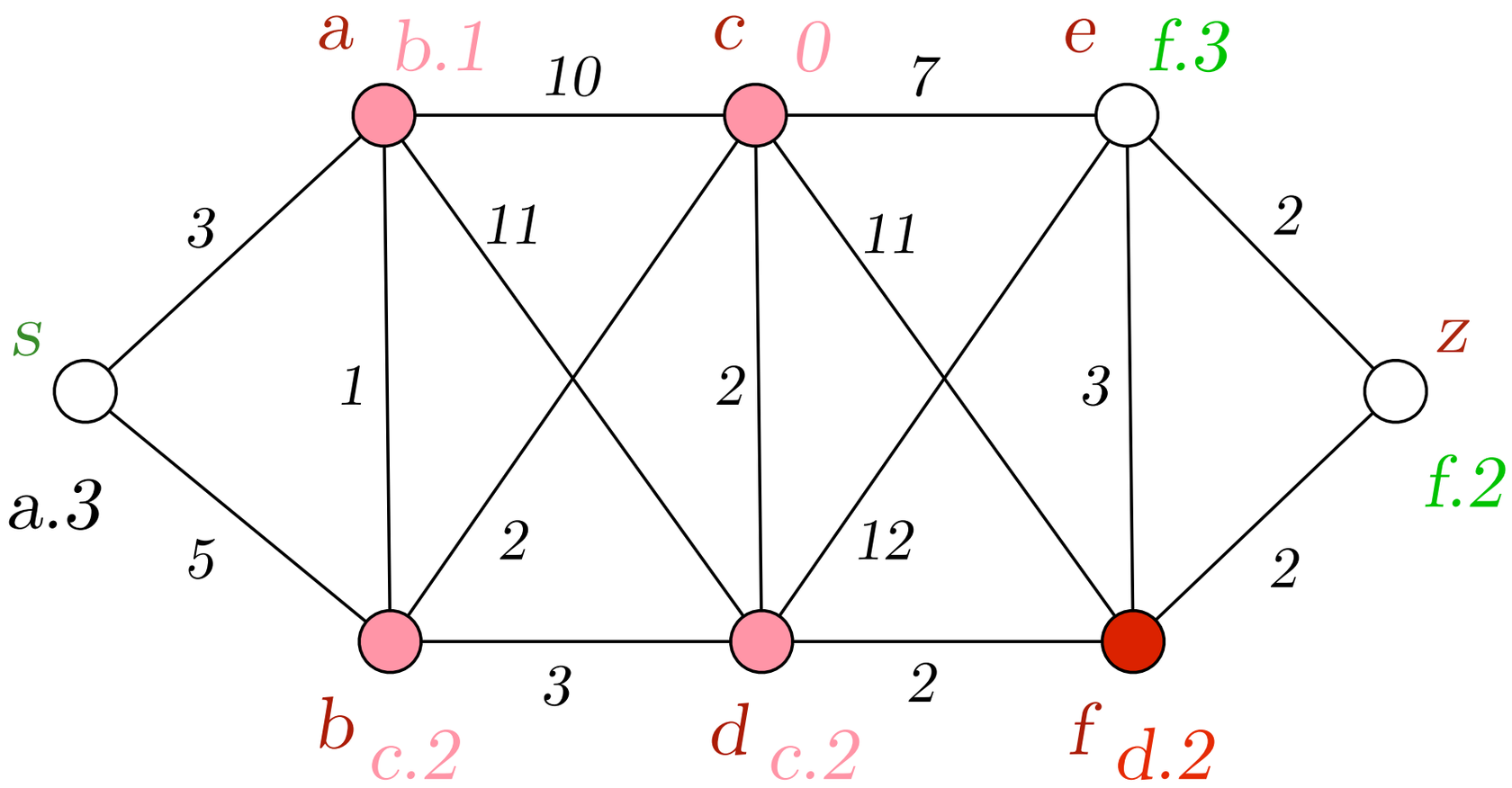
a wird v^*

Anwendung am Beispiel



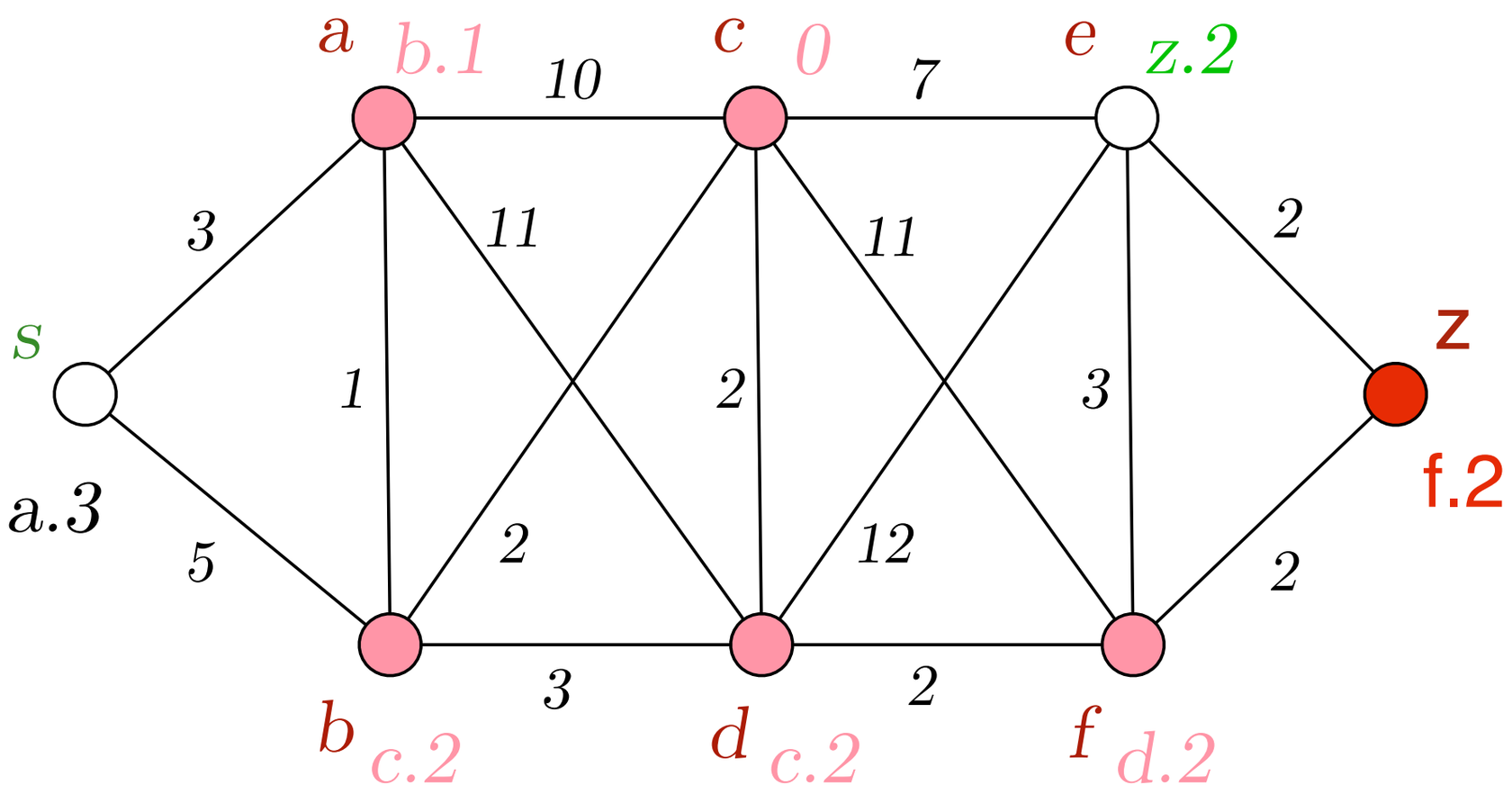
d wird v^*

Anwendung am Beispiel



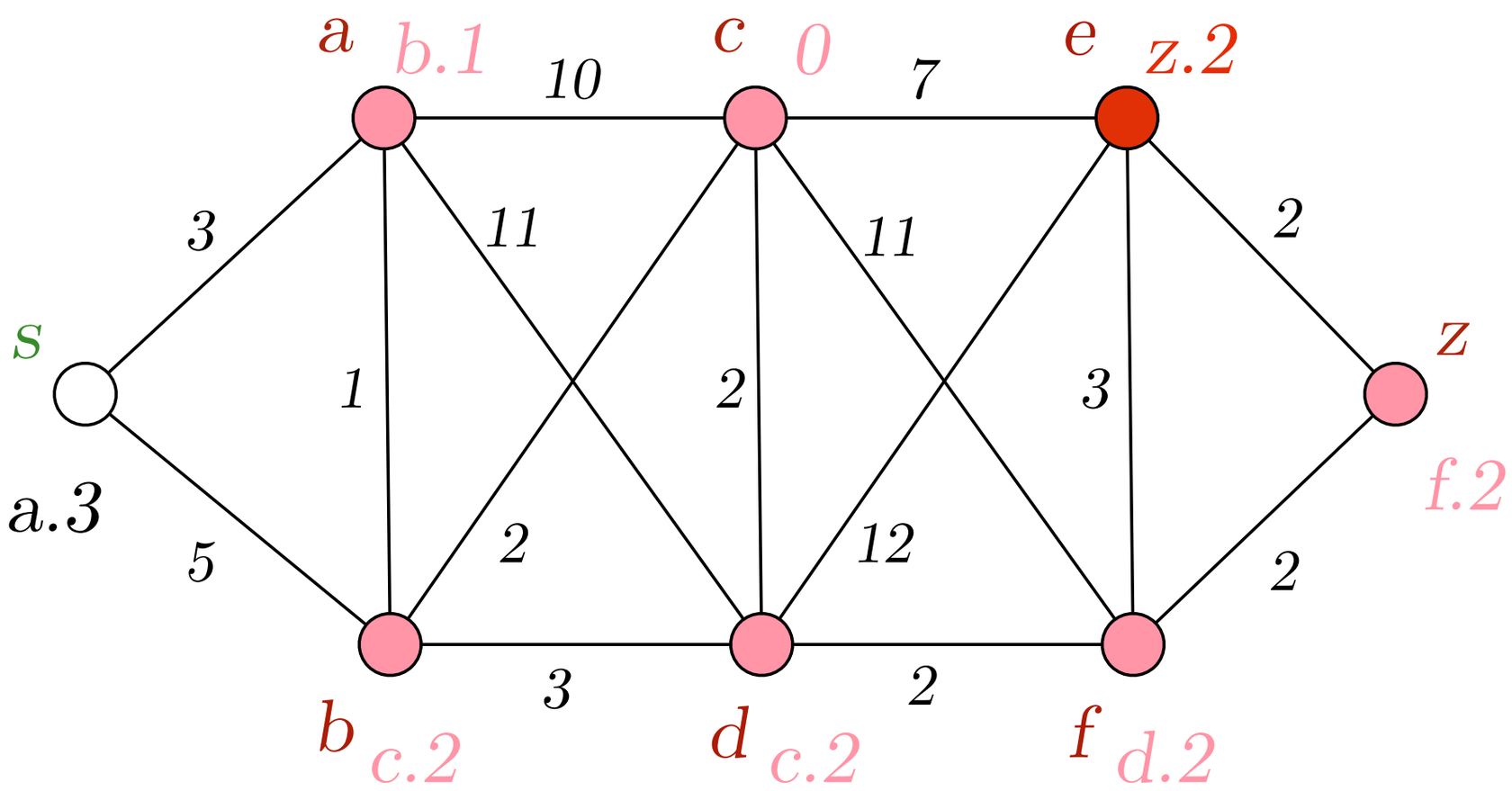
f wird v^*

Anwendung am Beispiel



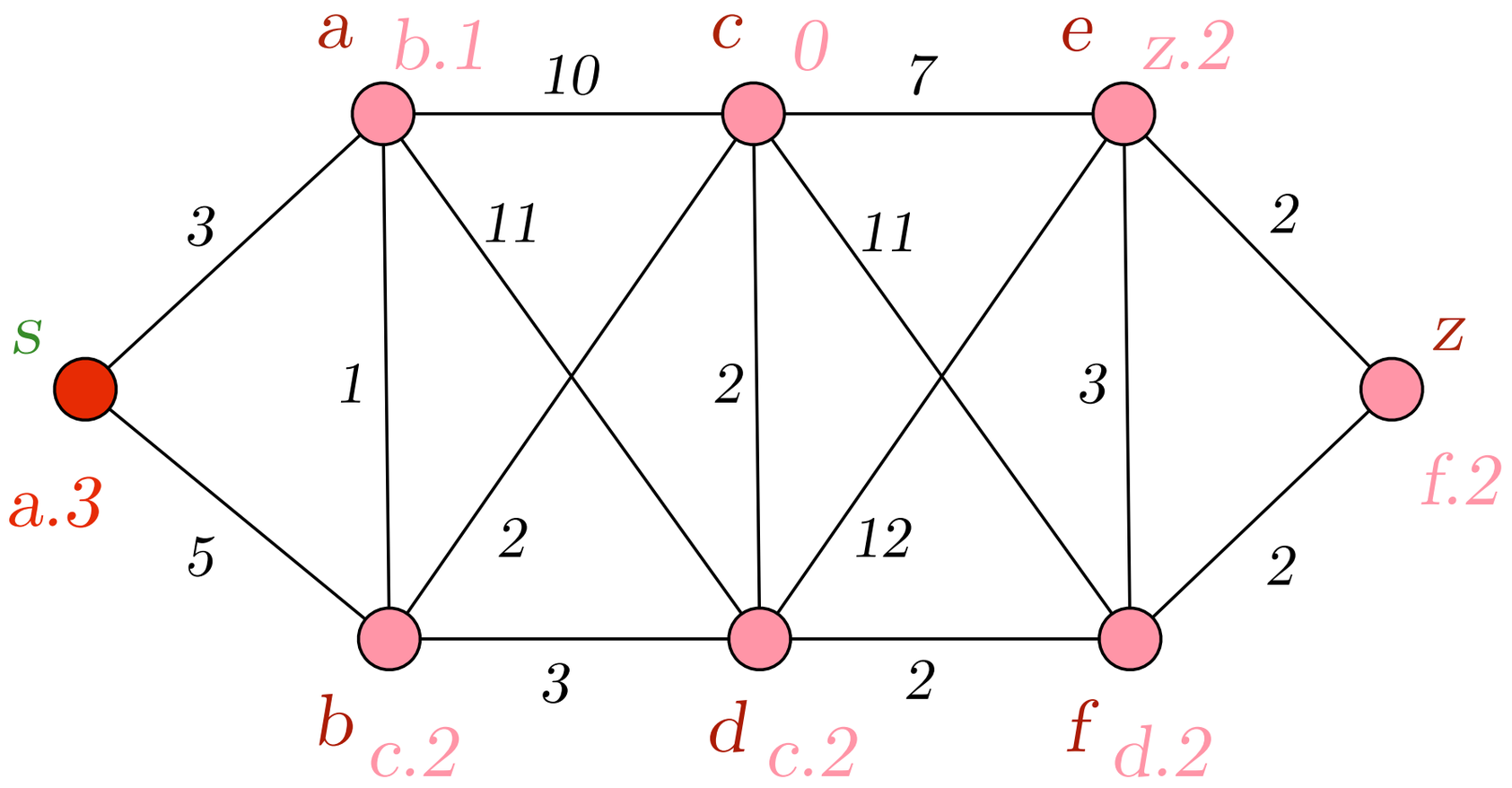
f wird v^*

Anwendung am Beispiel



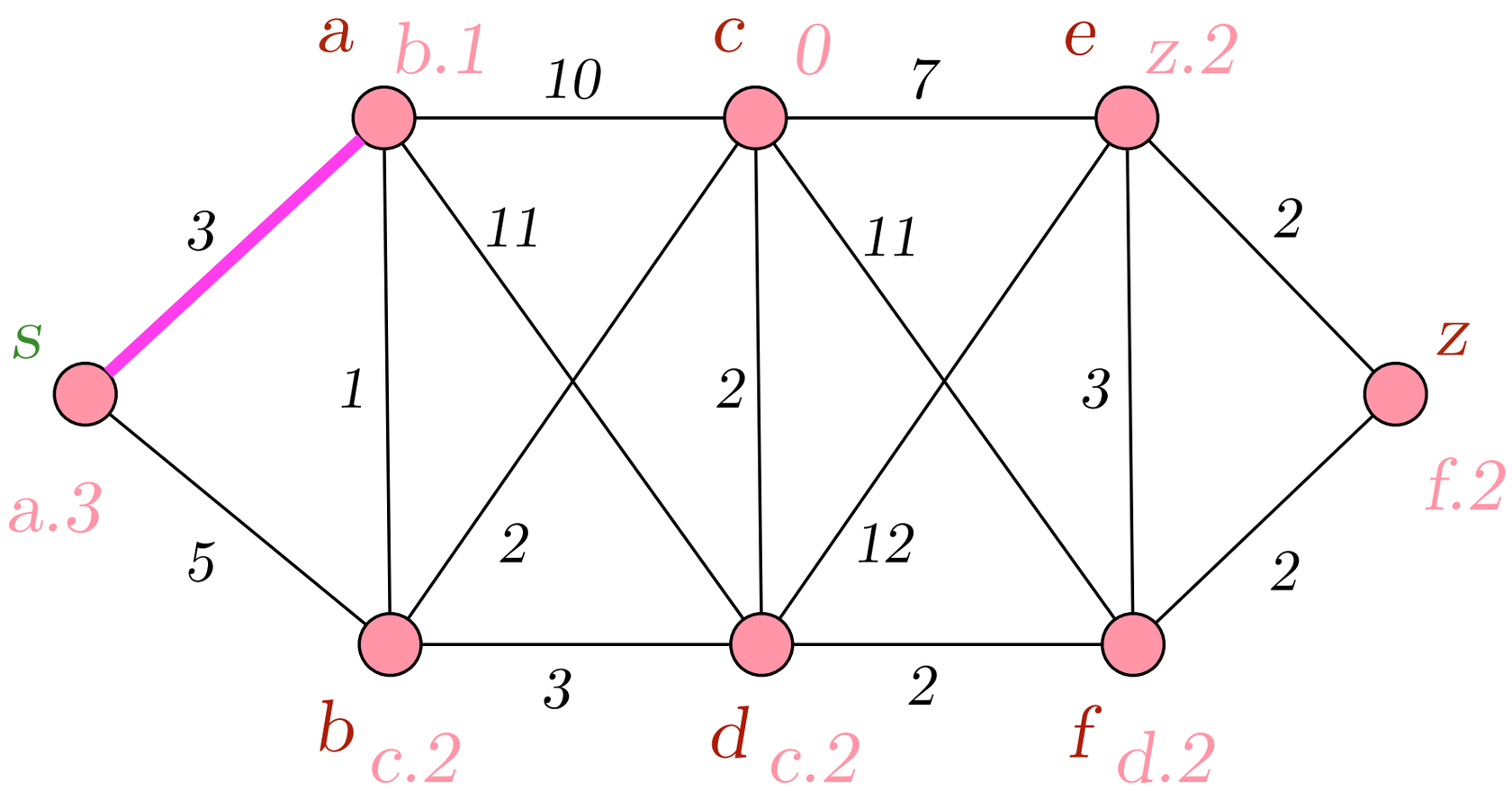
z wird v^*

Anwendung am Beispiel



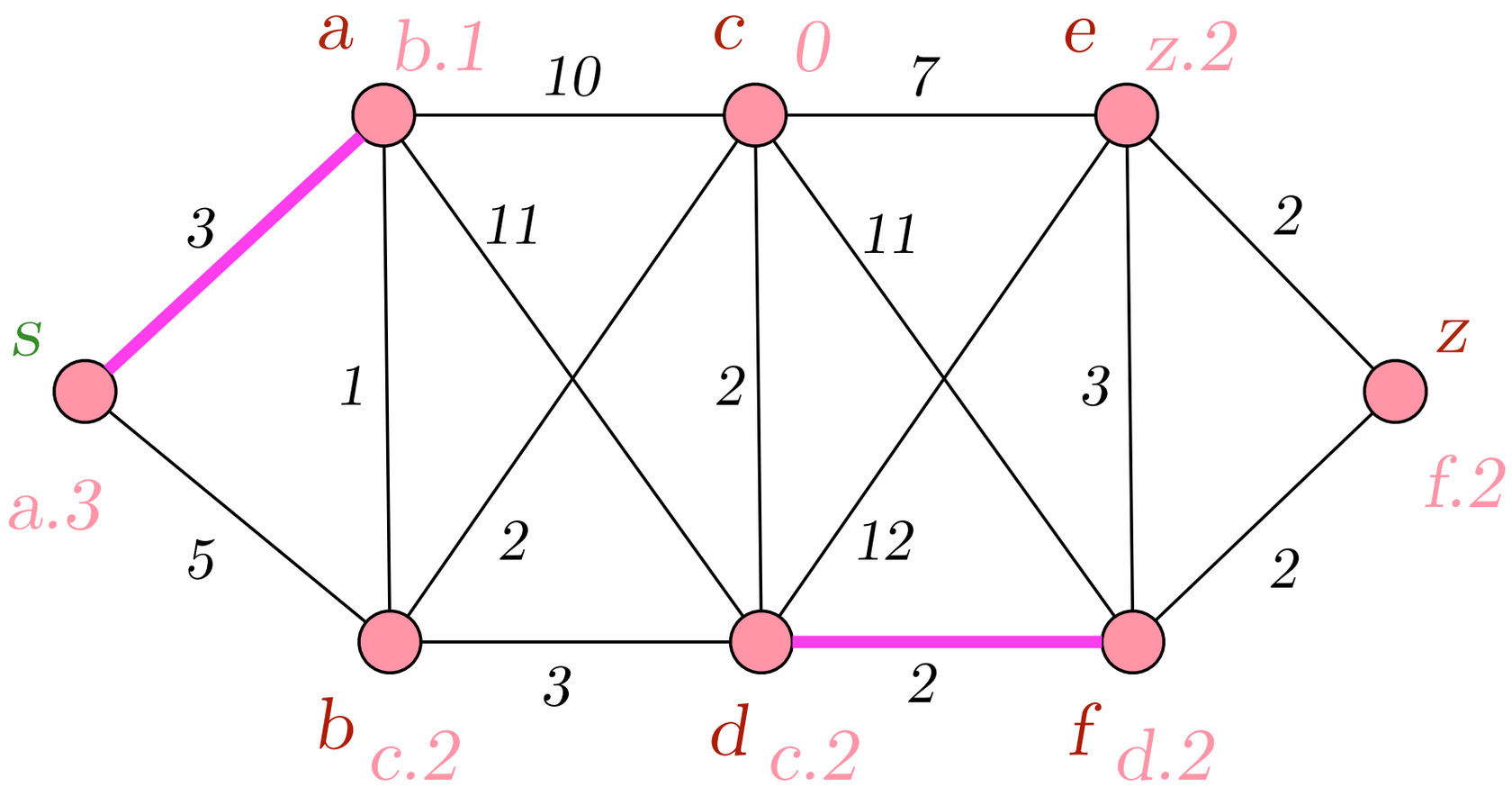
a wird v^*

Gerüst aus Vorgängerinformation erzeugen!



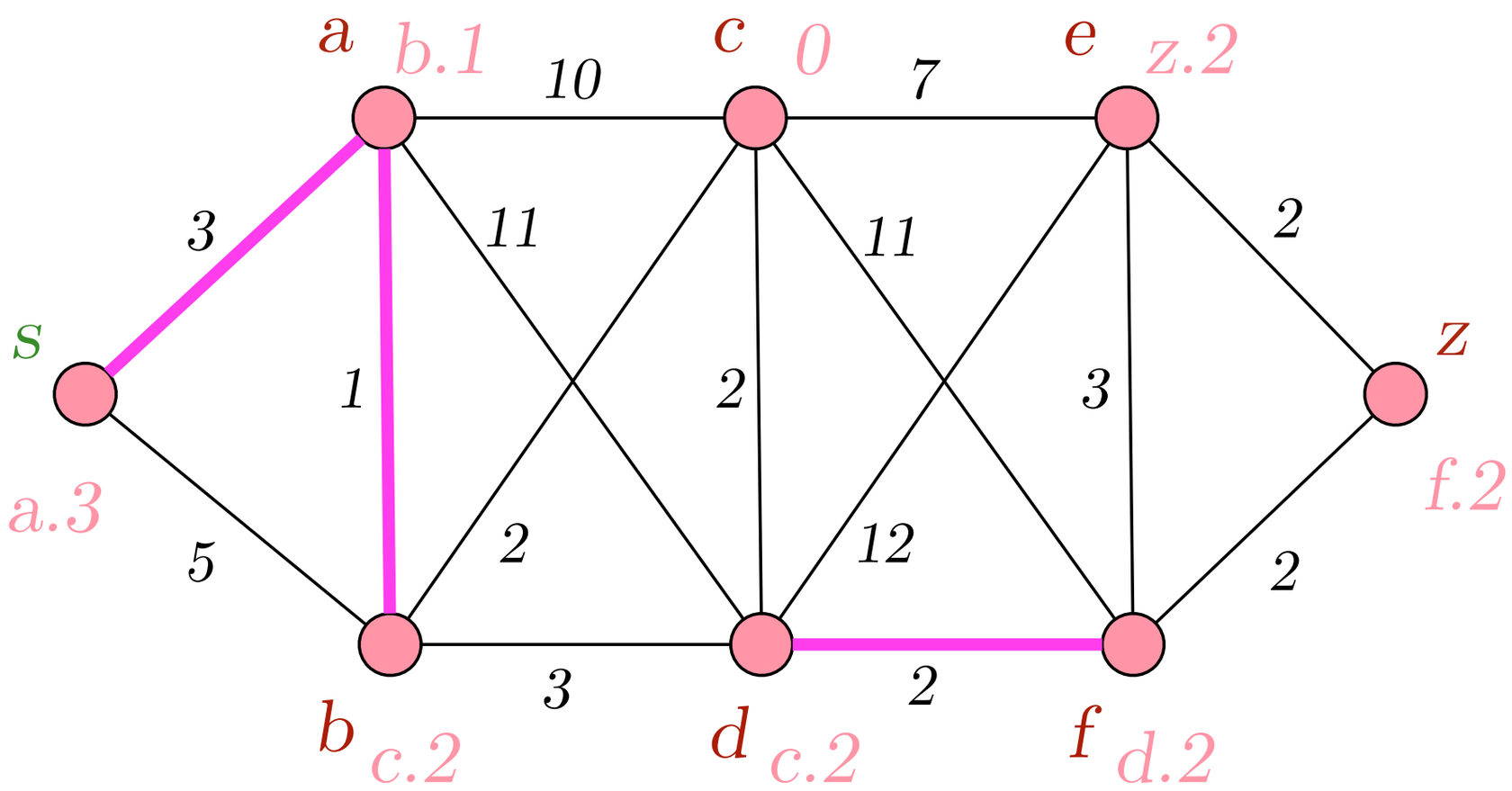
Gerüst-Kanten aus pre-Info erzeugen, $G := G+w(e) = 3$.

Gerüst aus Vorgängerinformation erzeugen!



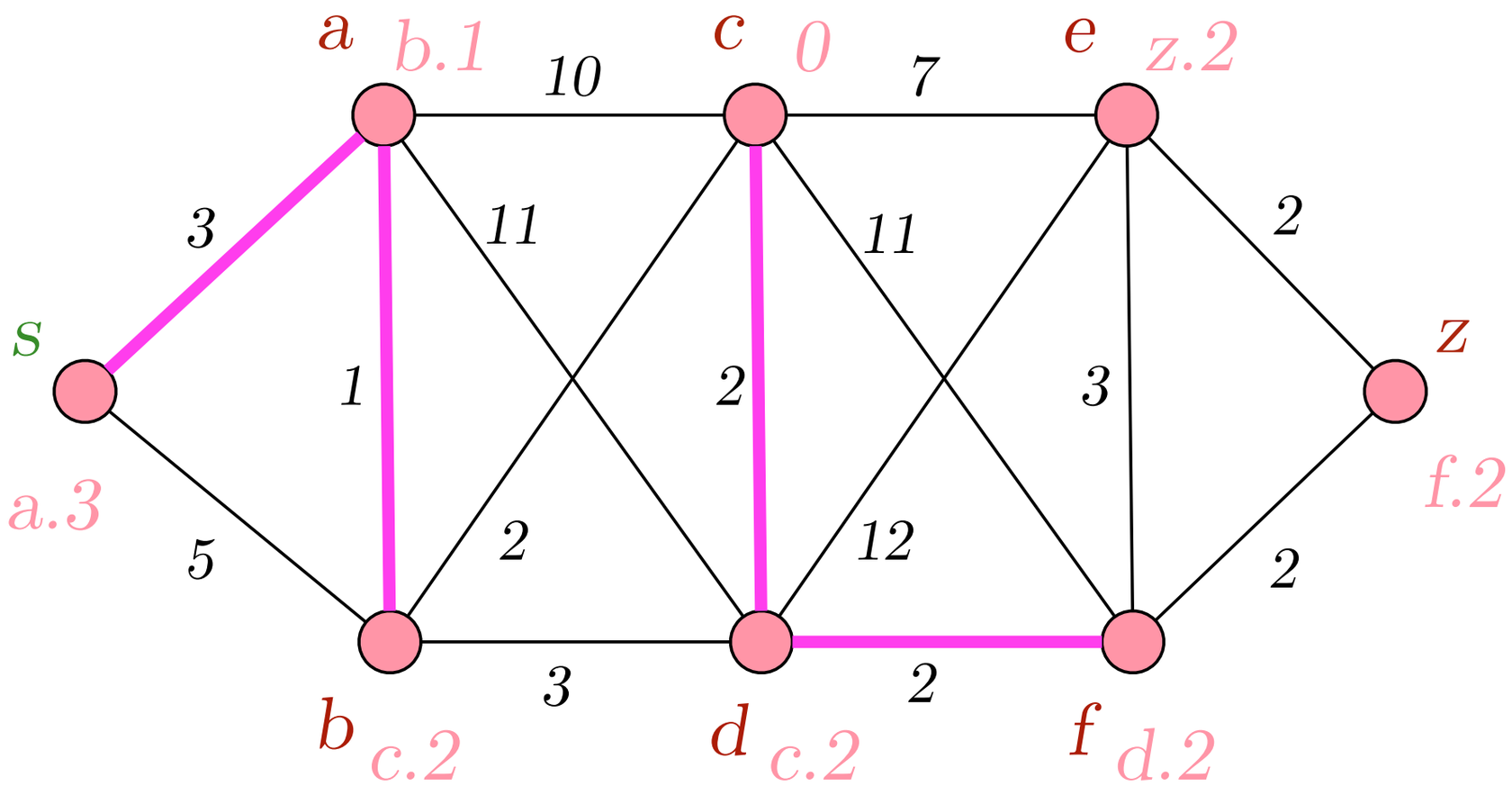
Reihenfolge beliebig, $G = 5$.

Gerüst aus Vorgängerinformation erzeugen!



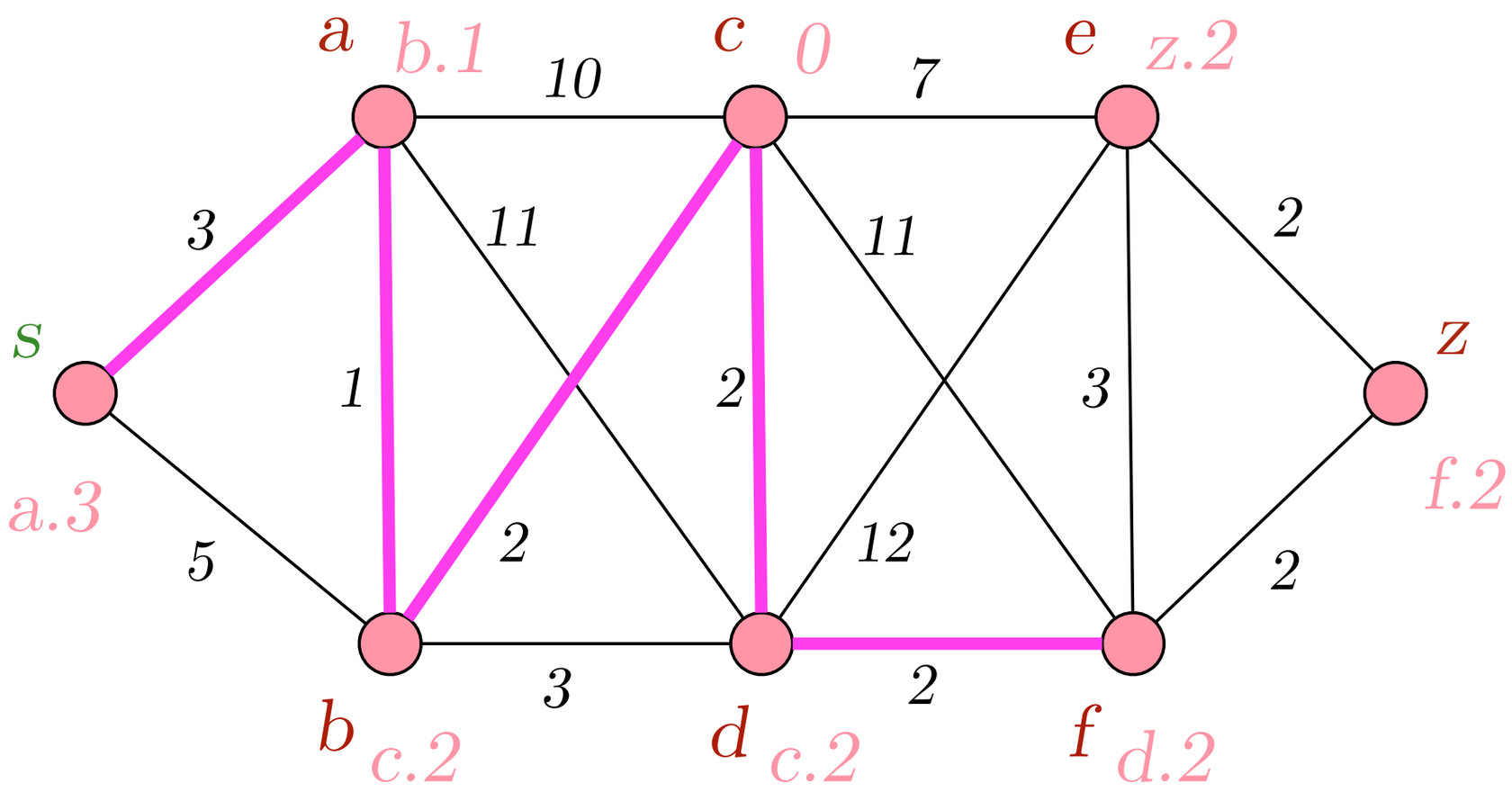
Reihenfolge beliebig, $G = 6$.

Gerüst aus Vorgängerinformation erzeugen!



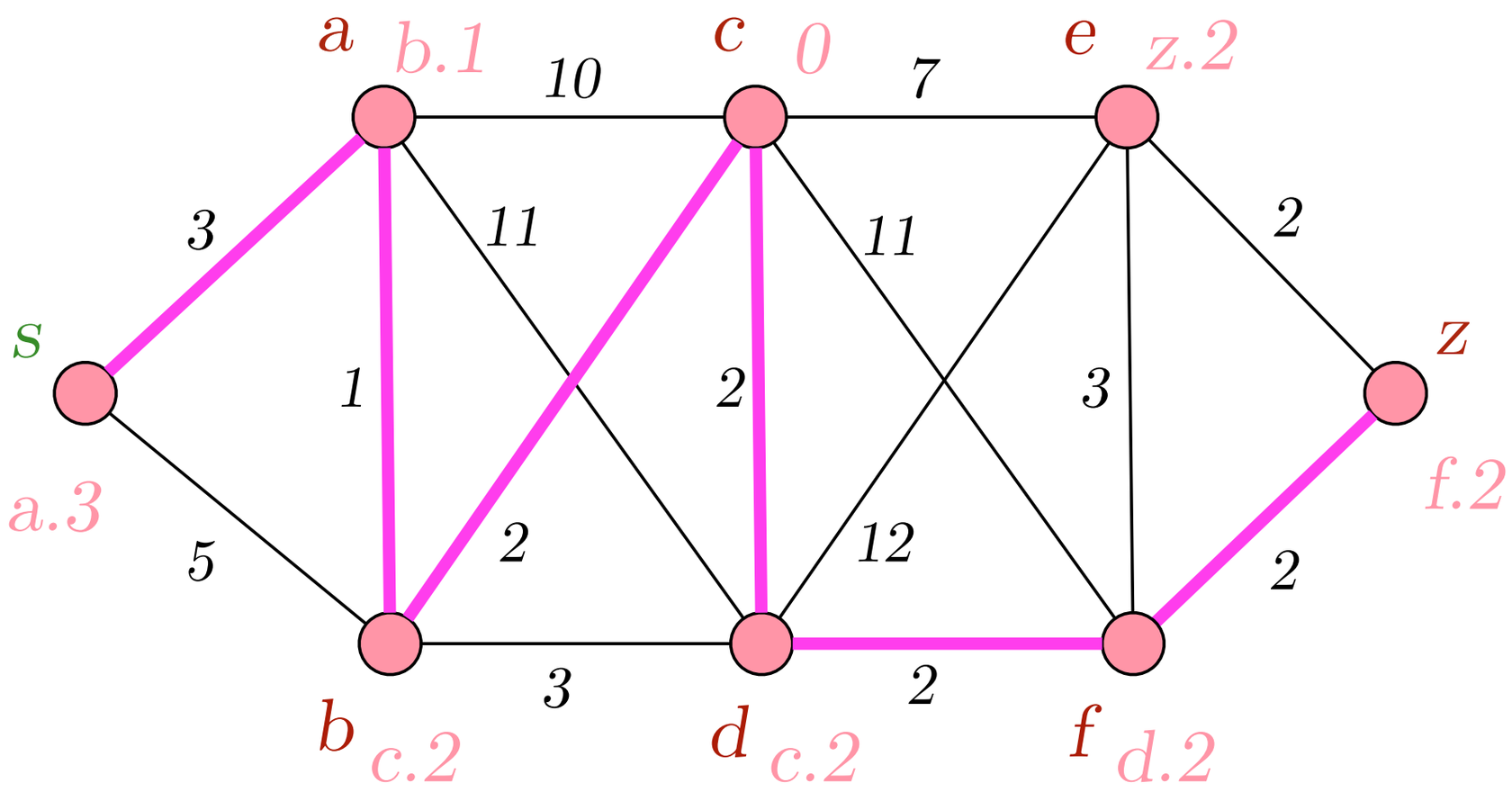
Reihenfolge beliebig, $G = 8$.

Gerüst aus Vorgängerinformation erzeugen!



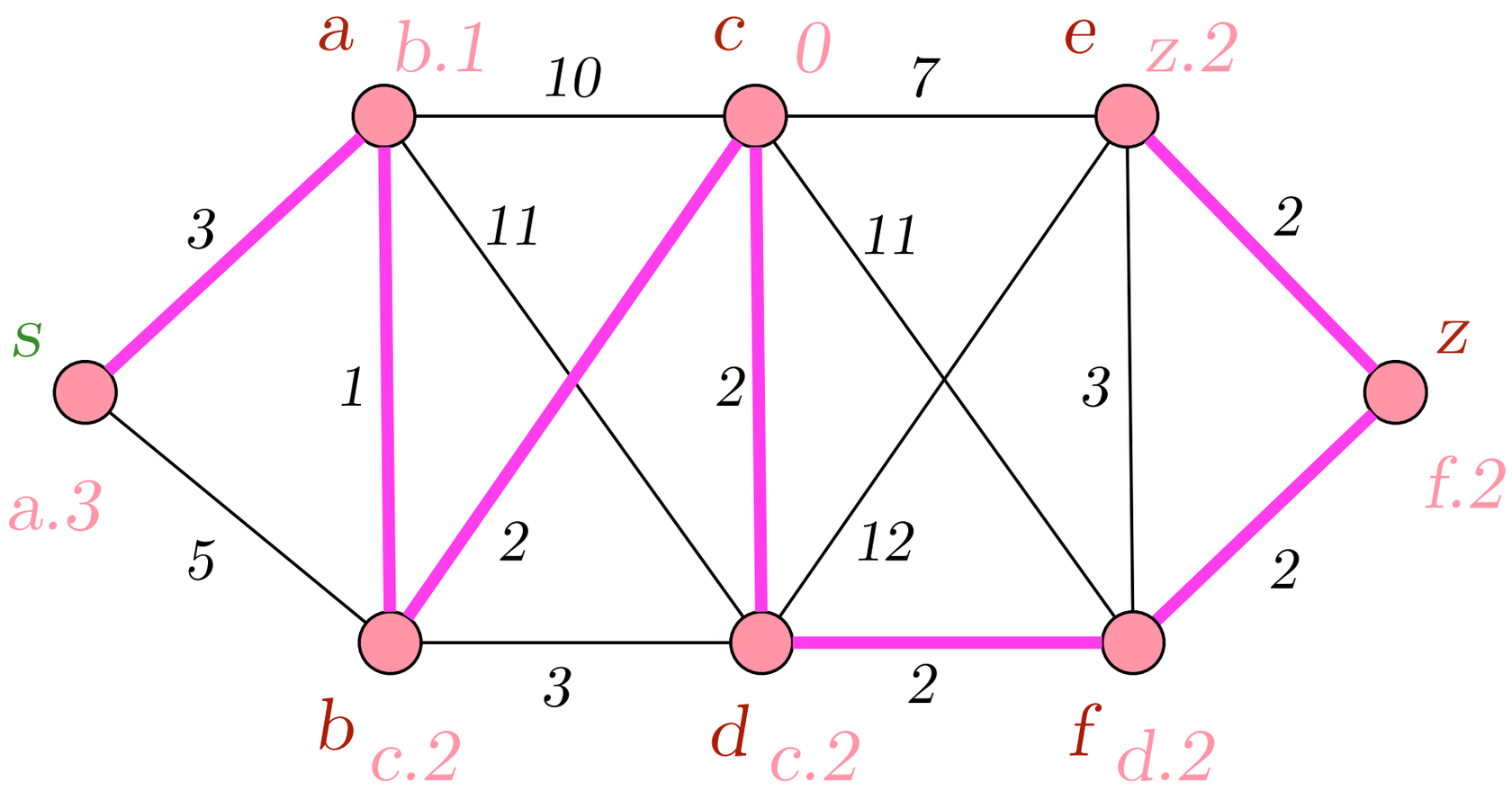
Reihenfolge beliebig, $G = 10$.

Gerüst aus Vorgängerinformation erzeugen!



Reihenfolge beliebig, $G = 12$.

Abschluss des Anwendungsbeispiels zum Prim-Algorithmus

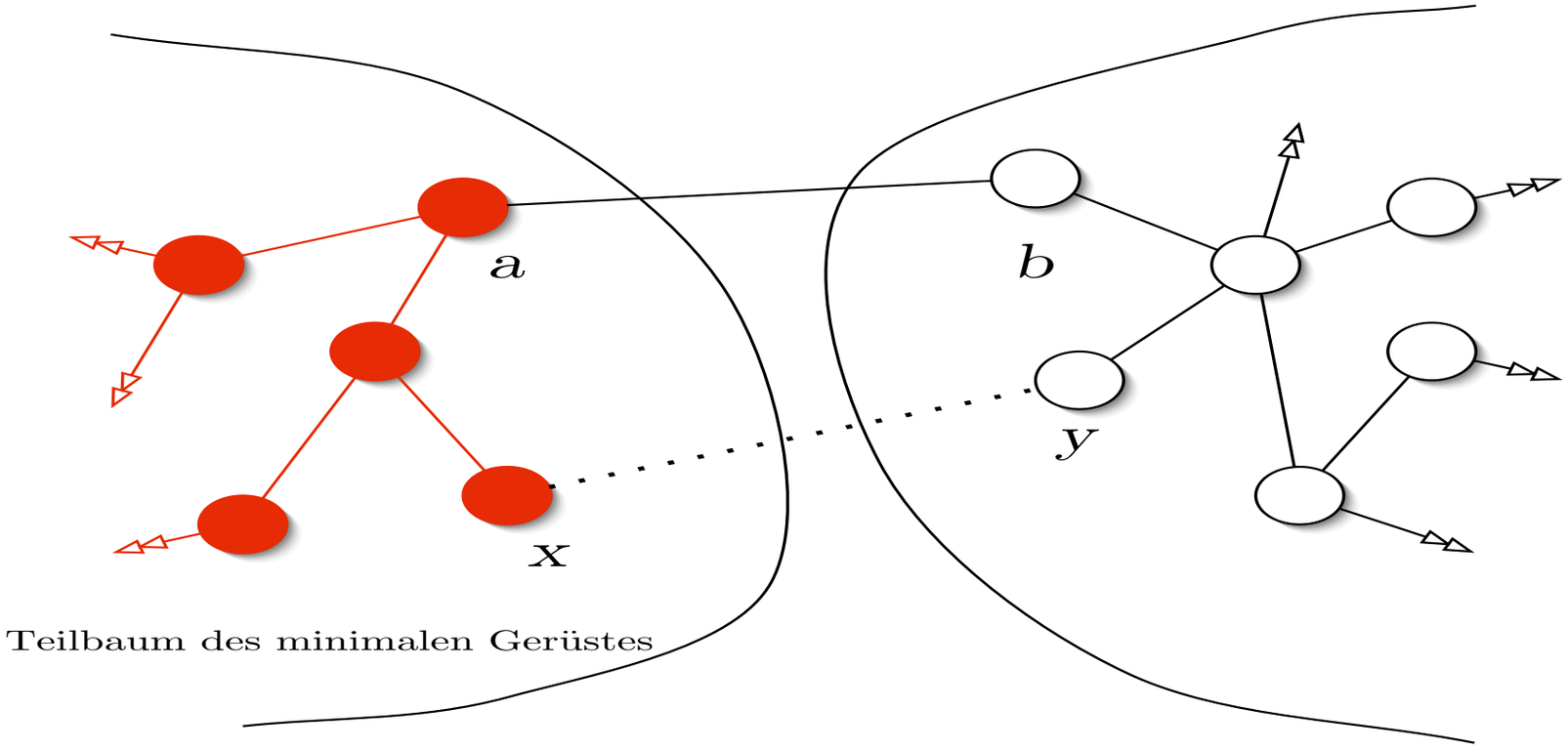


Minimales Gerüst mit Gewicht $G = 14$.

Zum Beweis des Prim-Algorithmus

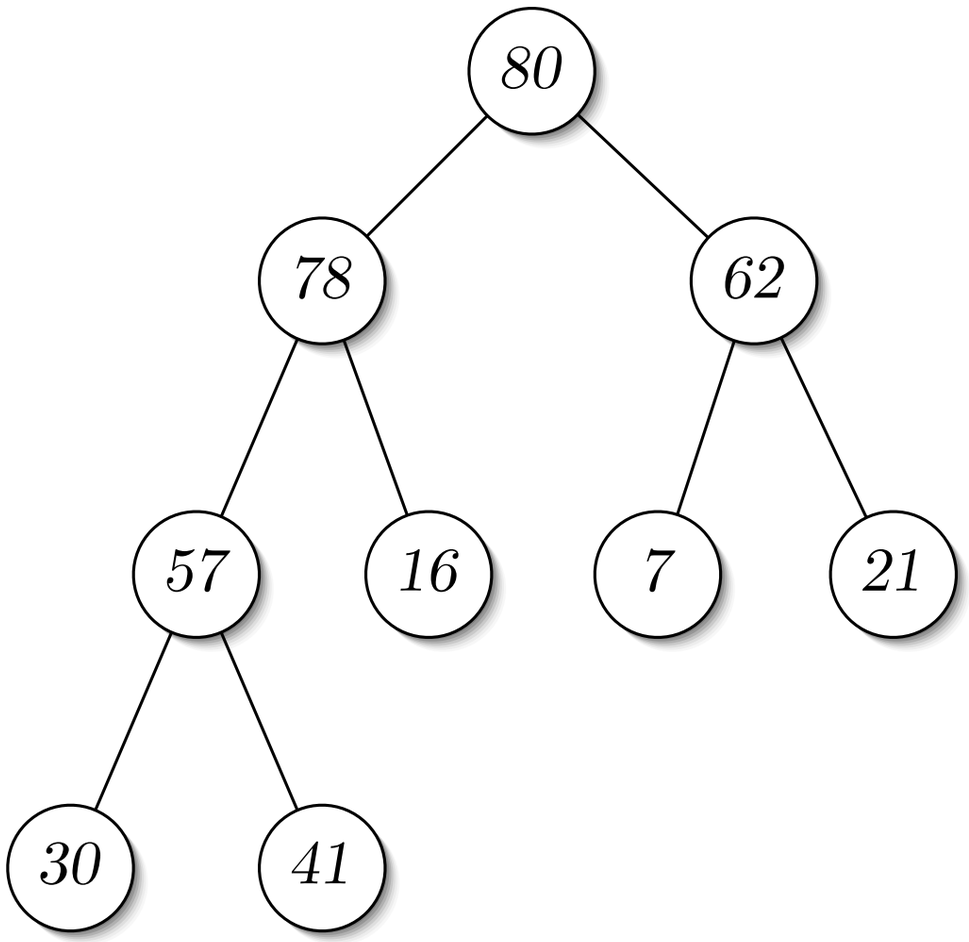
Widerspruchsbeweis:

Angenommen, eine Kante (hier ab) vom bisher richtigen Teil des min. Gerüstes wird zum ersten Mal falsch gewählt, obwohl sie nicht zum minimalen Gerüst gehört. Demnach gäbe es eine Kante xy, die stattdessen richtig wäre. Also: $g(xy) < g(ab)$. Diese Kante hätte das Verfahren jedoch GARANTIERT gewählt, d.h., $g(xy) = g(ab)$ folgt zwangsläufig. Damit ist die *andere* Wahl aber kein Fehler, denn beide Kanten führen zu Gerüst mit gleichem Gewicht!



Was ist ein Heap?

Ein Heap ist eine Datenstruktur für „Schlüssel“ (i.A. natürliche Zahlen).
In Binärbaum-Darstellung befindet sich sets das größte (bzw. kleinste) Element jedes Teilbaums an der Wurzel!



In linearer Darstellung als Feld ist dies:

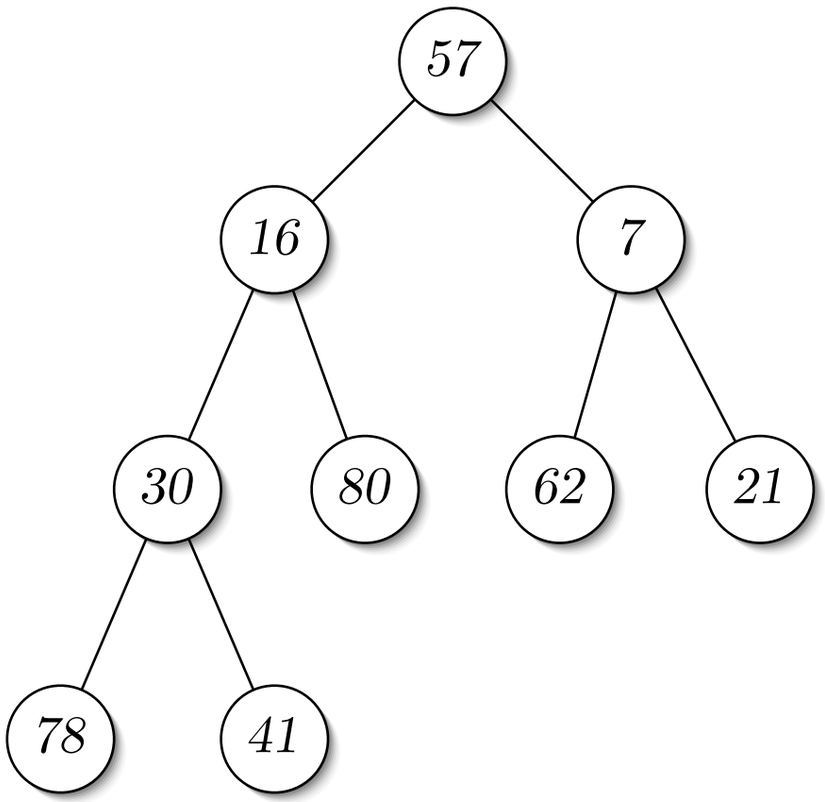
$$A := [80,\ 78,\ 62,\ 57,\ 16,\ 7,\ 21,\ 30,\ 41]$$

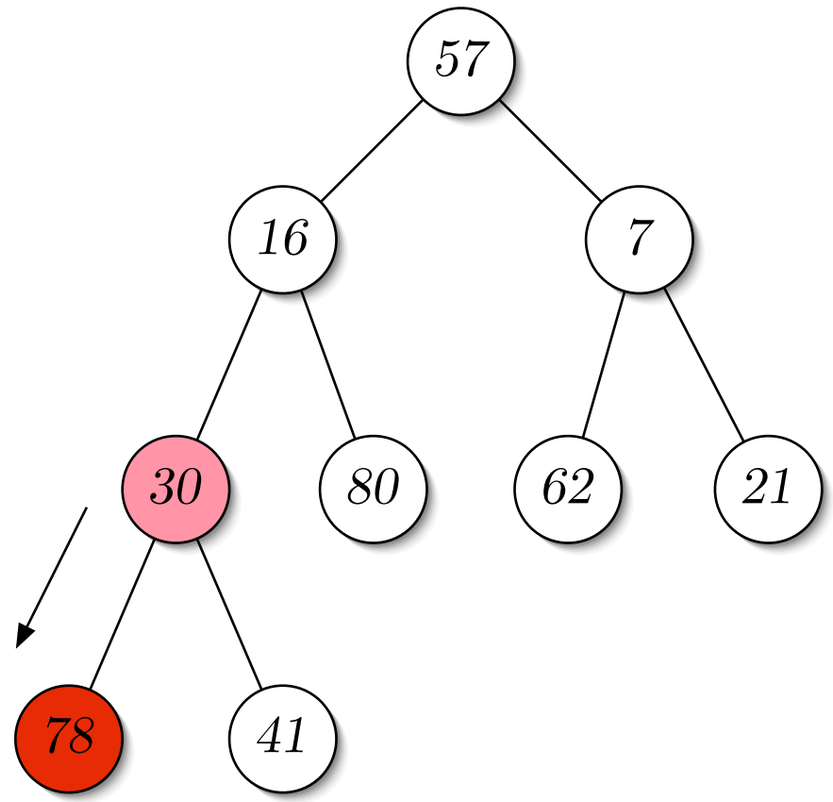
Es gilt $A[i \textbf{ div } 2] \geq A[i]$, mit $i \textbf{ div } 2 := \lfloor i/2 \rfloor$.

Wie erhält man aus einer beliebigen Zahlenfolge einen Heap, und wie kann dieser zum Sortieren (Heap-Sort) so genutzt werden, dass Heap-Sort auch im schlechtesten Fall mit Zeitbedarf $O(n \cdot \ell ogn)$ aus-kommt?

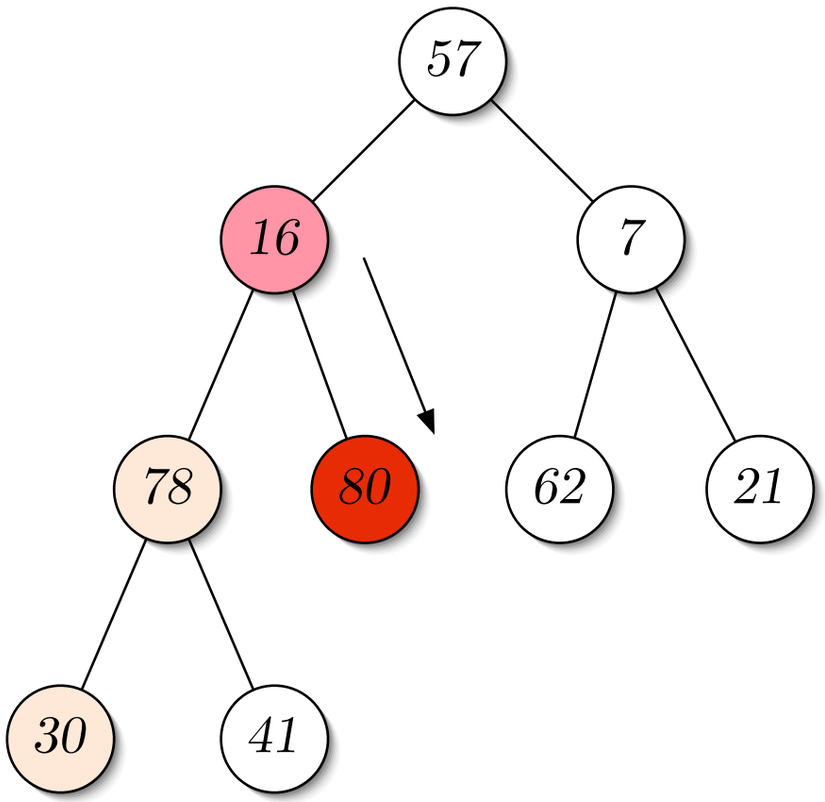
Bottom up Heapsort ist ab n ≥ 400 besser als Quicksort!

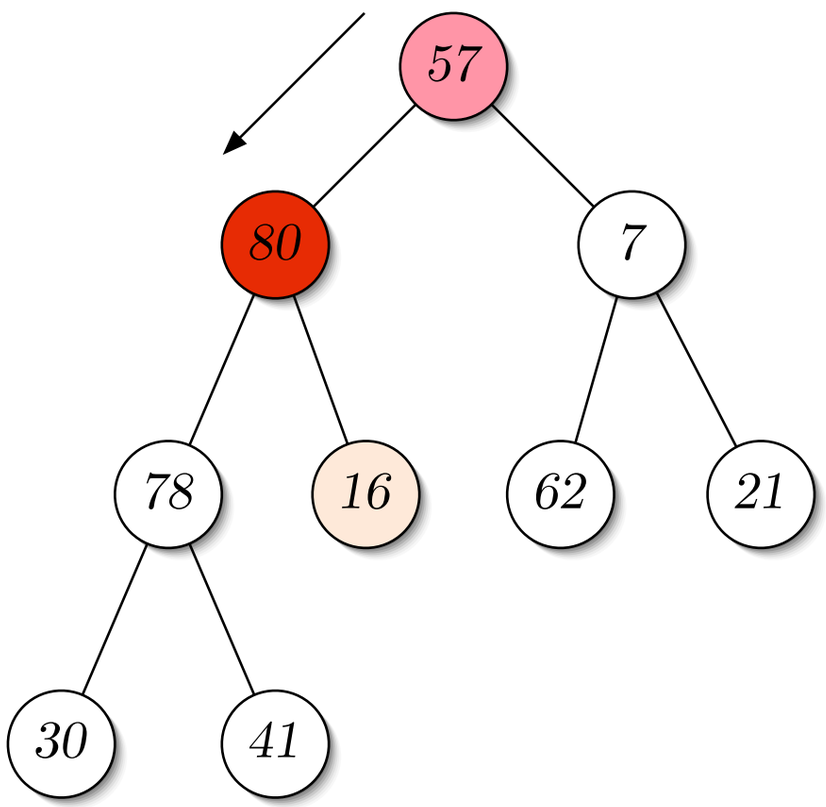
Das Absinken im Baum: kleinere Wurzel mit größtem Sohn tauschen!



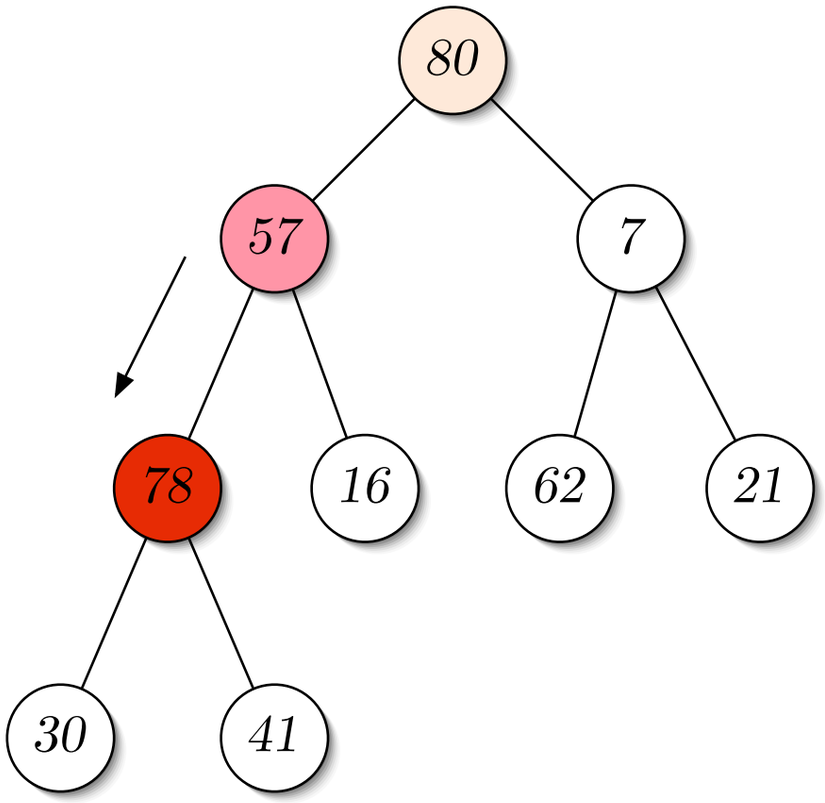


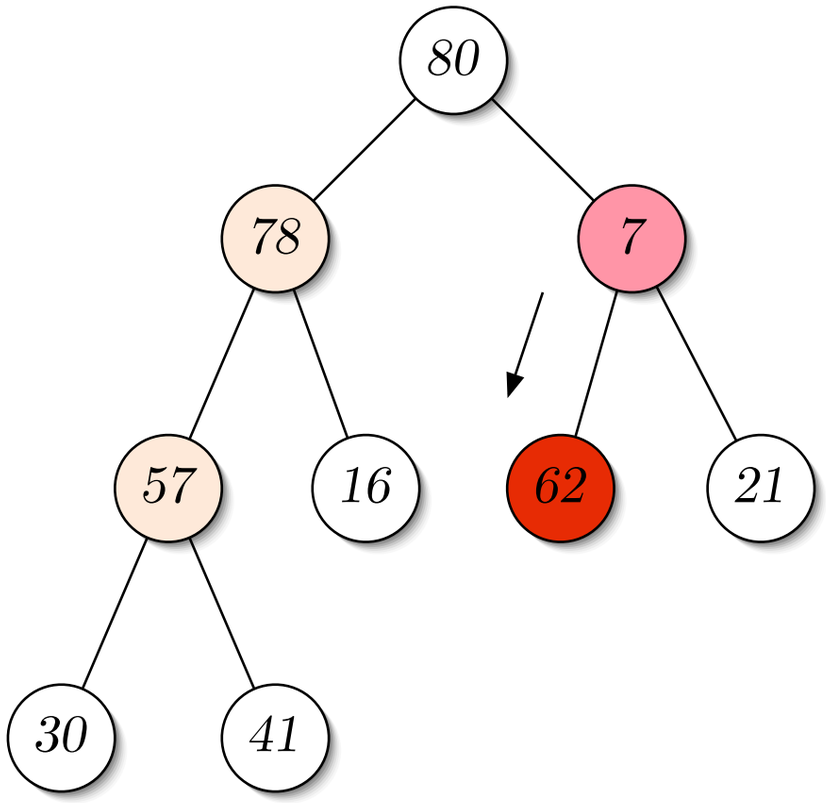
Das Absinken im Baum:



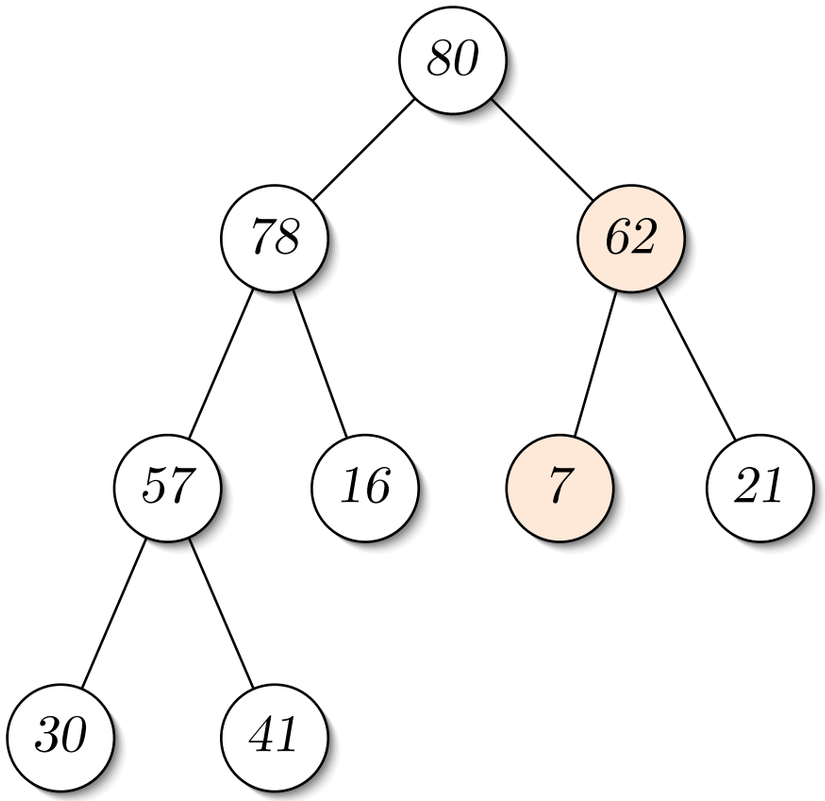


Das Absinken im Baum:

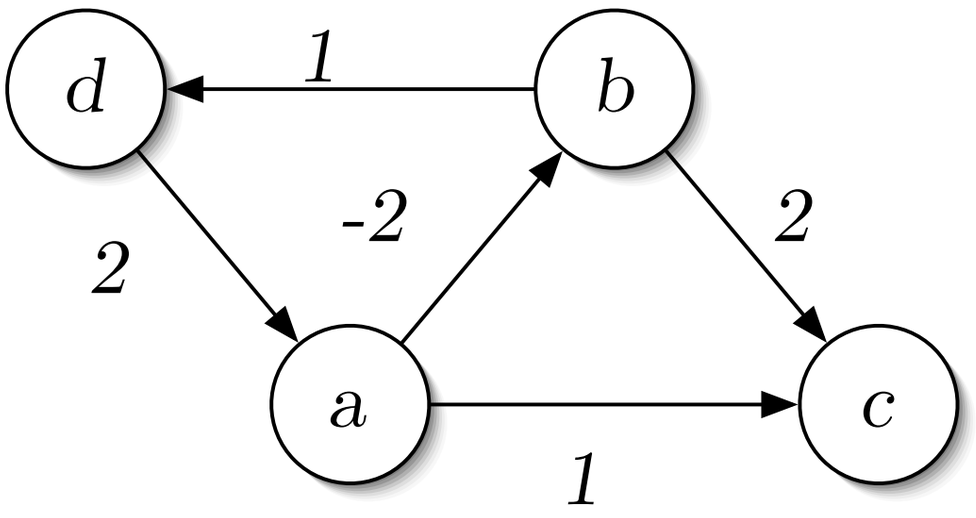




Das Absinken im Baum, bis der Heap entstanden ist:



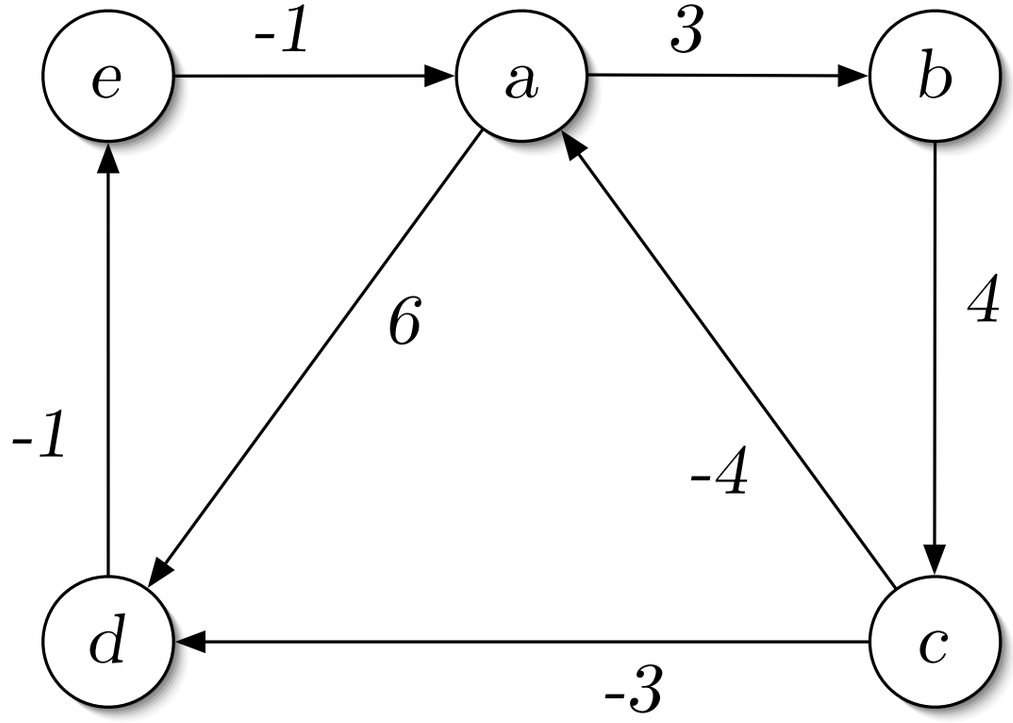
Was tun bei Digraphen mit Bewertungen aus $\mathbb{Z}$?



Überall um den Betrag der kleinsten Bewertung erhöhen und dann kürzesten Pfad suchen? Das führt zu Fehlern!

Das führt zu Fehlern!

Auch Dijkstras Algorithmus ist nicht mehr nutzbar: Der Algorithmus findet einen kürzesten Weg von a nach d der Länge 6, aber $a \to b \to c \to d$ hat nur Länge 4!



Finden eines kürzesten Pfades im $\mathbb{Z}$-Digraphen ist NP-schwer!

Sei G ein ungerichteter Graph dessen Kanten alle mit -1 bewertet sind. In G gibt es von einem Knoten zu einem Nachbarknoten genau dann einen minimalen Pfad des Gewichts $-n$, wenn es in G einen Hamiltonkreis gibt, was ein NP-vollständiges Problem ist.

Beachte: Ein Weg ist eine verbundene Kantenfolge.
Ein Pfad ist eine Weg, in dem jeder Knoten nur einmal besucht wird!

Es kann in polynomieller Zeit getest werden, ob ein $\mathbb{Z}$-Digraph die Eigenschaft $(*)$ besitzt, d.h. ob G einen Kreis mit negativem Gewicht besitzt.

Das Verfahren ist Nebenresultat des Algorithmus von Bellmann-Ford-Moore, der in einem $\mathbb{Z}$-Digraphen einen kürzesten Pfad von einem Start- zu allen Zielknoten findet, wenn G ***keinen*** Kreis mit negativem Gewicht besitzt!

Gutes Buch zu Digraphen:
Bang-Jensen/Gutin: *Digraphs: Theory, Algorithms and Applications (Springer, 2001),*
speziell die Frage negativer Zyklen, siehe: Seite 549 ff , sowie auch:
Cherkassy/Goldberg: *Negative-cycle detection algorithms (Mathem. Programming,* ***85****:277–311, 1999).*
Algorithmen zum download: *http://elib.zib.de/ftp/pub/packages/mathprog/*

Bellmann-Ford-Moore Algorithmus für alle kürzeste Pfade im $\mathbb{Z}$-Digraphen

Sei $\delta^m(s,v)$ die Länge eines kürzesten Pfades von s nach v mit höchstens m Kanten, dann gilt:

$$\delta^{m+1}(s,v) = min\{\delta^m(s,v) \ , \ min\{\delta^m(s,u) + w(u,v) |\ (u,v) \in E\ \}\},$$

was leicht mit Induktion über m zu beweisen ist.

Der BFM-Algorithmus:

1. $let\ \forall v \in V : w(v,v) := 0, \forall v \in V \setminus \{s\} : \delta^1(s,v) := \infty, and\ w(u,v) := \infty, if\ (u,v) \notin E.$

2. $for\ i = 1\ to\ n-1\ do$
   $for\ each\ (u,v) \in E\ do$
   $\delta^{i+1}(s,v) = min\{\delta^i(s,v),\ min\{\delta^i(s,u) + w(u,v) |\ (u,v) \in E\ \}\}$
   $end\ for$
   $end\ for$

% Letzter Teil bei 3. nur, um negativen Zyklus zu finden!

3. $for\ each\ (u,v) \in E\ do$
   $if\ \delta^n(s,v) > \{\delta^{n-1}(s,u) + w(u,v),\ then\ return\ G$ ***hat negativen Zyklus!***

Komplexität des Bellmann-Ford-Moore Algorithmus:

Für alle kürzesten Pfade von s zu jedem Knoten ist das $O(|E|\cdot|V\,|) = O(m\cdot n)$.
Für alle kürzesten Pfade zwischen allen Paaren von Knoten sind das $O(m\cdot n^2)$.

Letztere Frage geht aber besser mit dem Verfahren von **Floyd-Warshall**:

(Hierzu sei $V{:=}\{v_1, v_2, ..., v_n\}$)

$$
\begin{array}{l}
for\ k{=}1\ to\ n\ do \\
\quad for\ i{=}1\ to\ n\ do \\
\quad\quad for\ j{=}1\ to\ n\ do \\
\quad\quad\quad \delta^k(v_i, v_j) := min(\ \delta^{k-1}(v_i, v_j),\ \ \delta^{k-1}(v_i, v_k)\ +\ \delta^{k-1}(v_k, v_j)\ ) \\
\quad\quad end\ for \\
\quad end\ for \\
end\ for
\end{array}
$$

Die Komplexität des **FW**-Verfahrens ist $O(n^3)$, und oftmals besser als **BFM** oder **Dijkstra**. Es wird auch nur die $n{\times}n$-Eingabe-Matrix als Speicher benötigt!

Grundelemente der (algorithmischen) Geometrie

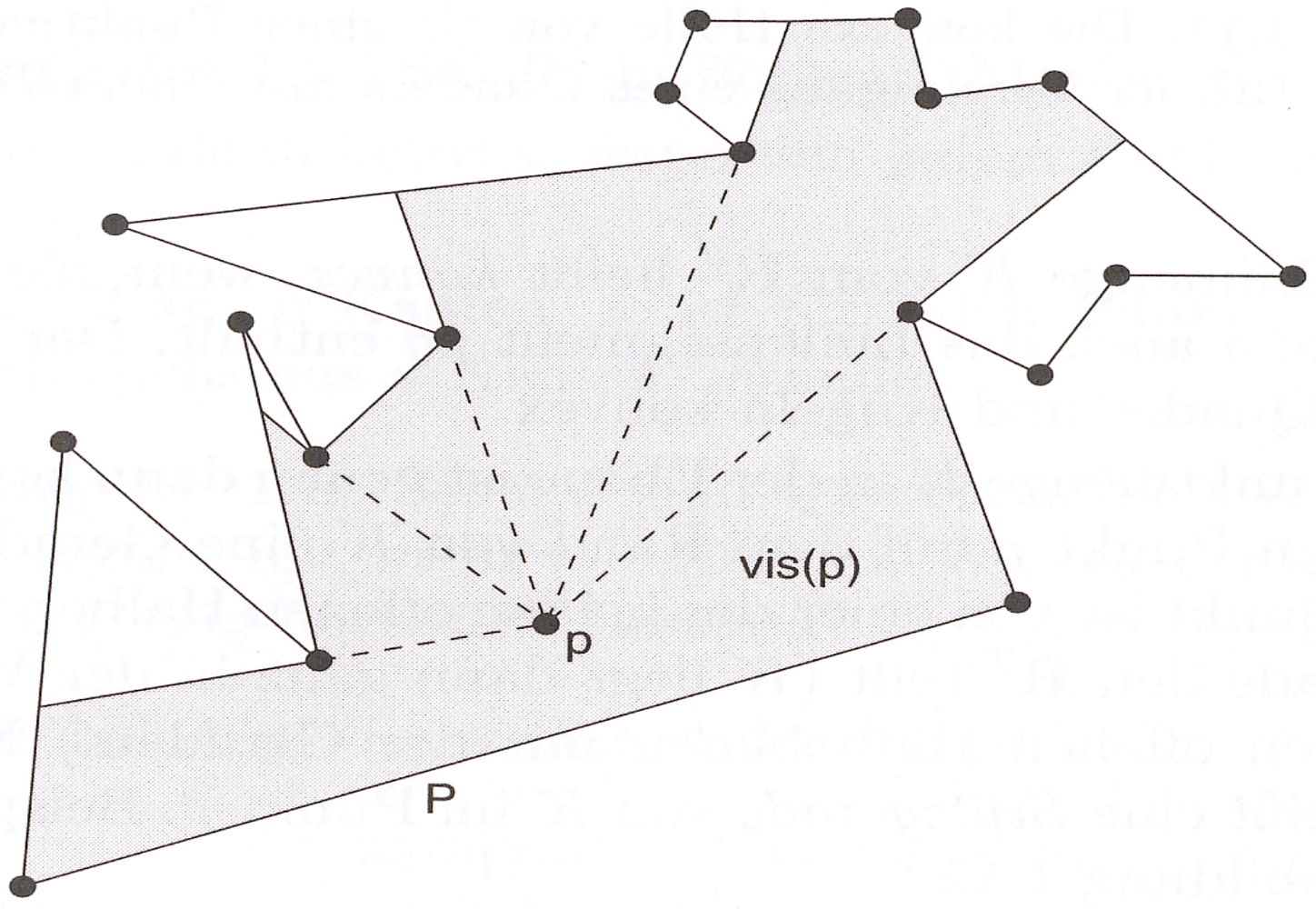
Einige Grundelemente

- Eulersche Formel für kreuzungsfreie Graphen in der Ebene: $v-e+f=c+1$.

$\Rightarrow$ Knotenzahl: v, Kantenzahl: e, Flächenzahl: f, Komponentenzahl: c.

- Punkte im $p,q\in\mathbb{R}^d$ werden als Vektoren, oft transponiert als Tupel, notiert.
- Liniensegmente $pq:=\{p+a(q-p)\,|\,0\leq a\leq 1\ \}=\{ap+bq|\ a+b=1,\ 0\leq a,b\}$
- $P\subseteq\mathbb{R}^2$ heißt einfaches Polygon, wenn begrenzt von geschlossener Kette □
  verbundener Liniensegmente ohne Kreuzungen oder inneren Aussparungen.
- $P\subseteq\mathbb{R}^3$ heißt Polyeder, wenn Ränder Polygone sind.
- Für $p,q\in\mathbb{R}^d$, $P\subseteq\mathbb{R}^d$, heißt q sichtbar von p in P, wenn pq ganz in P liegt.
- $vis_P(q)\subseteq\mathbb{R}^d$ ist Sichtbarkeits-Gebiet (-Polygon, falls $d=2$, -Polyeder, falls $d=3$)
- $K\subseteq\mathbb{R}^d$ ist konvex, wenn zu $p,q\in K$ immer auch $pq\subseteq K$ gilt.

Sichtbarkeitspolygon



Erste einfache Fragen und eine Beweisaufgabe:

1. Ist der K_5 **(a)** in der Ebene, **(b)** auf dem Torus (Rettungsring) einbettbar?

Antworten: **1. (a)** NEIN:

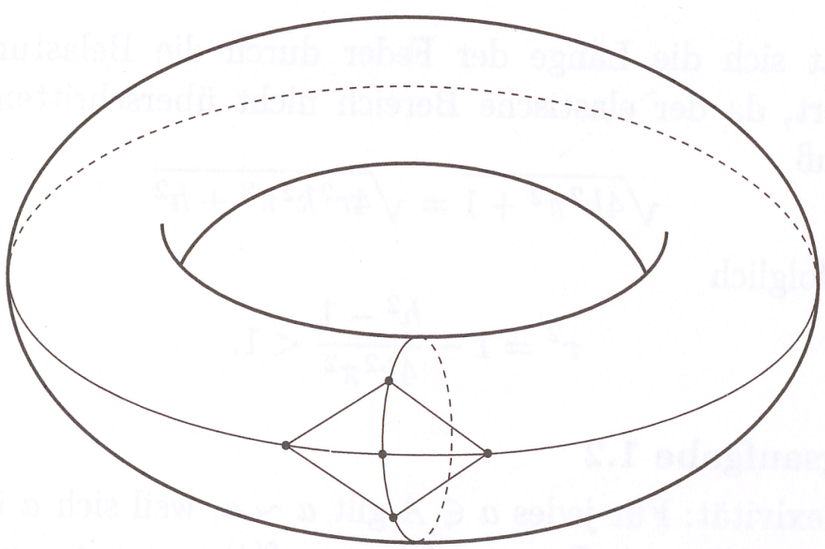
Der K_5 hat $v=5$ Knoten, und jeder hat den Grad $g=4$, also $e=g\cdot 5/2=10$, und $c=1$, woraus für einen (hypothetisch) plättbaren K_5 nach Eulerscher Formel $f=7$ folgt. $(v-e+f=c+1)$

Sei m_i die Anzahl der Kanten auf dem Rand der i-ten Fläche, so ist $3\leq m_i$, denn der K_5 hat weder doppelte Kanten noch Schleifen (er ist schlicht, *simple*).

Also: $3\cdot f \leq \sum m_i \leq 2\cdot e$ (jede Kante wurde 2-mal gezählt), was ein Widerspruch ist:

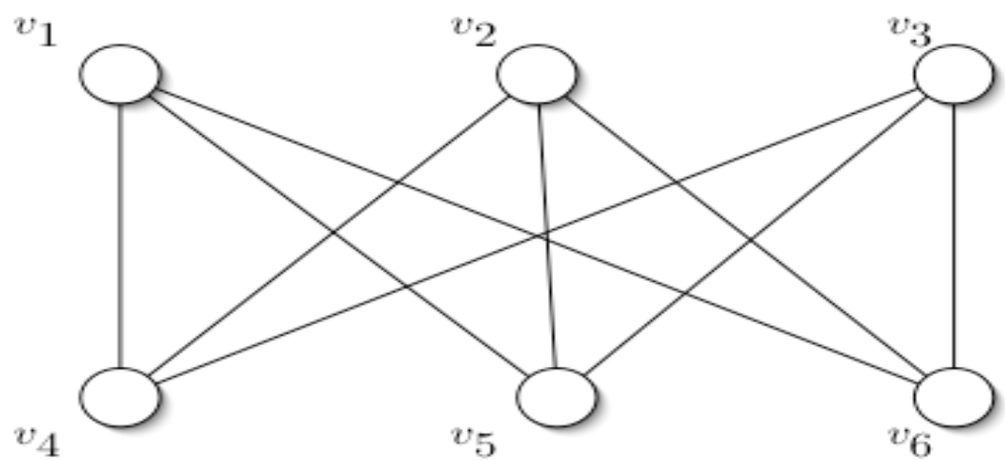
$3\cdot f=21>2\cdot e=20$.

1.(b) JA:



Zeigen Sie, dass auch der $K_{3,3}$ nicht plättbar ist!

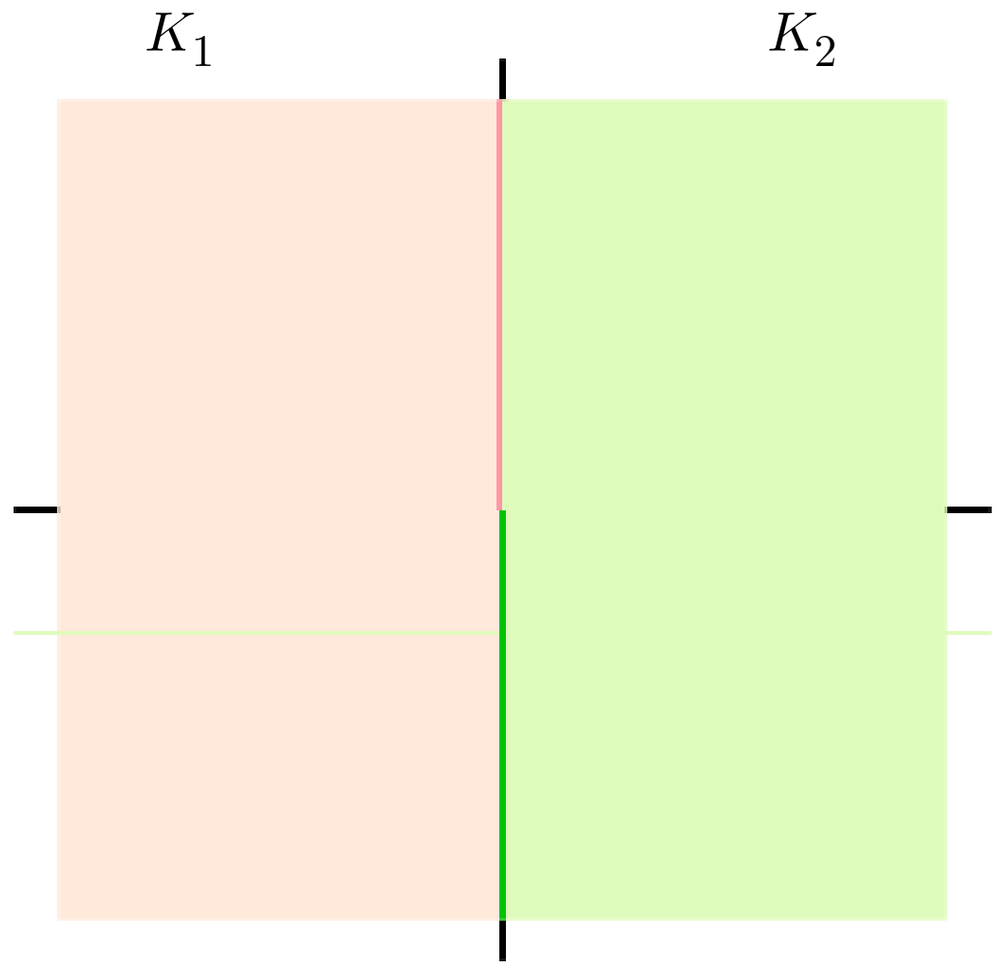
$K_{3,3}$:



Eine zweite Frage:

2. Kann man zu zwei disjunkten, konvexen Teilmengen $K_1, K_2 \subseteq \mathbb{R}^2$ immer eine Gerade finden, die die Mengen trennt und mindestens eine von Ihnen nicht berührt?

Antwort: **NEIN**



Sei:

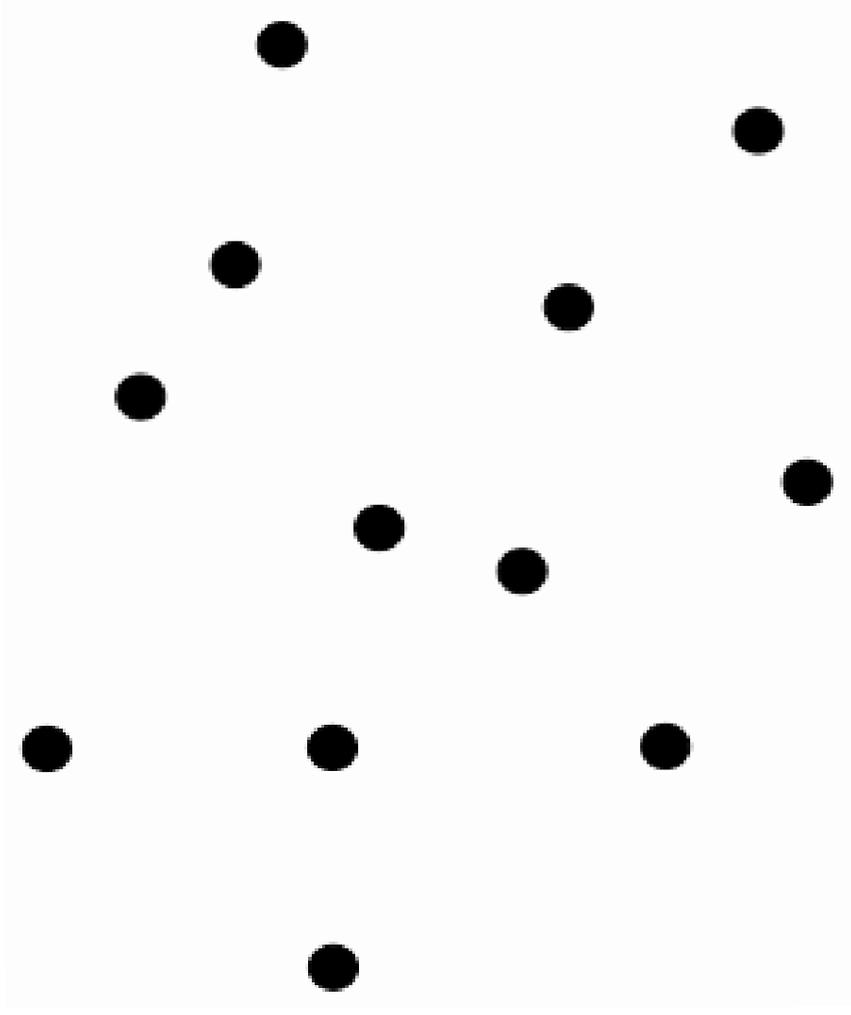
$K_1 := \{(x,y) \in \mathbb{R}^2 \mid x < 0 \text{ oder } (x = 0 \text{ und } y > 0)\}$

$K_2 := \{(x,y) \in \mathbb{R}^2 \mid x > 0 \text{ oder } (x = 0 \text{ und } y < 0)\}$

Dichteste Punkte in $\mathbb{Z}$, $\mathbb{Z}^2$, $\mathbb{R}^d$, im metrischen Raum M.

Probleme:

1. Welche Codierung der Zahlenwerte ist [illegible]raktikabel?
2. Wie speichert man die Menge der Punkte, so dass Anfrageneffizient beantworbar sind?
3. Wie kann Datenbank der Punktmenge leicht verändert werden?
4. Was stellen die „Punkte“ dar?
5. Welche Metrik ist dann vernünftig?
6. Welches Verfahren ist effizient bei der Frage nach den dichtesten Punkten?
7. Gibt es eine untere Schranke für die Komplexität dieser Anfrage?



Codierung der Zahlenwerte

Reelle Zahlen sind nur bis zu einer festen Genauigkeit darstellbar. Zudem wird der betrachtete Zahlenraum auf das abgeschlossene Intervall $[0,1]\in\mathbb{R}$ normiert und die Werte als n-stellige Zeichenkette gespeichert.

Nötig ist jedoch in der Regel eine Grundmenge M mit einer Metrik $\rho{:}M\times M\to\mathbb{R}$, d.h.:
$\forall x,y\in M$:

$$\rho(x,y)\geq 0,\ \rho(x,y)=0 \text{ gdw. } x=y,\ \rho(x,z)\leq\rho(x,y)+\rho(y,z), \text{ und } \rho(x,y)=\rho(y,x).$$

Die übliche Metrik im $\mathbb{R}^2$ ist die bekannte Euklidische Metrik, meist mit ℓ_2 bezeichnet.

In der allgemeineren ℓ_p-Metrik, $p\geq 2$, des $\mathbb{R}^2$ gilt:

$$\rho_p(x,y) := \sqrt[p]{(x_1-y_1)^p+(x_2-y_2)^p}$$

In der sogenannten Manhatten-Metrik, meist als ℓ_1-Metrik bezeichnet, gilt:

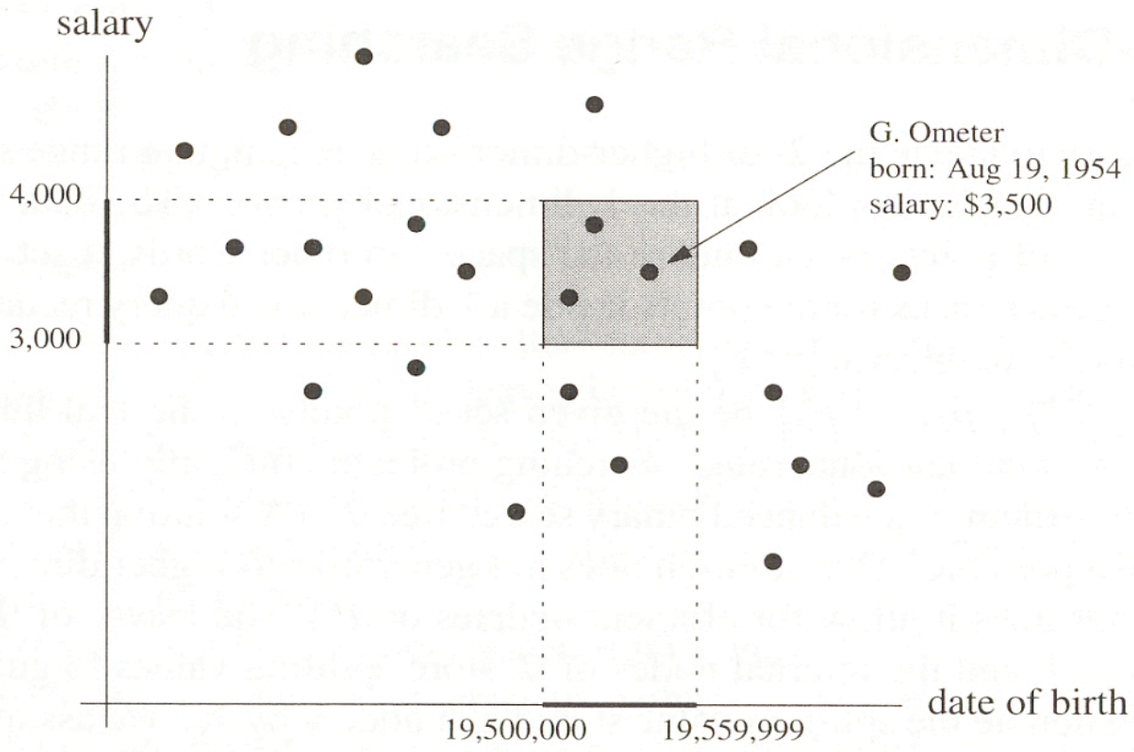
$$\rho(x,y) := |x_1 - x_2| + |y_1 - y_2|.$$

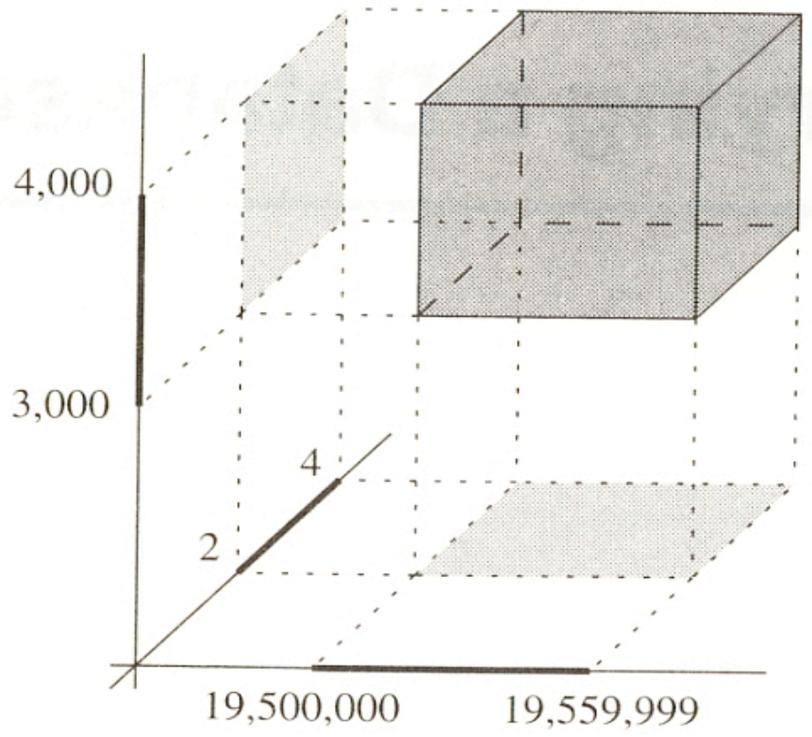
Elemente eines n-dimensinalen Raumes, z.B. in relationalen Datenbanken.

Die Fragen an die Datenbank betreffen sowohl die Existenz eines bestimmten Elementes (Suche), wie auch Fragen nach den Element innerhalb bestimmter Grenzen, sog. Orthogonale Bereichs Anfrage (*rectangular* oder *orthogonal range query*).

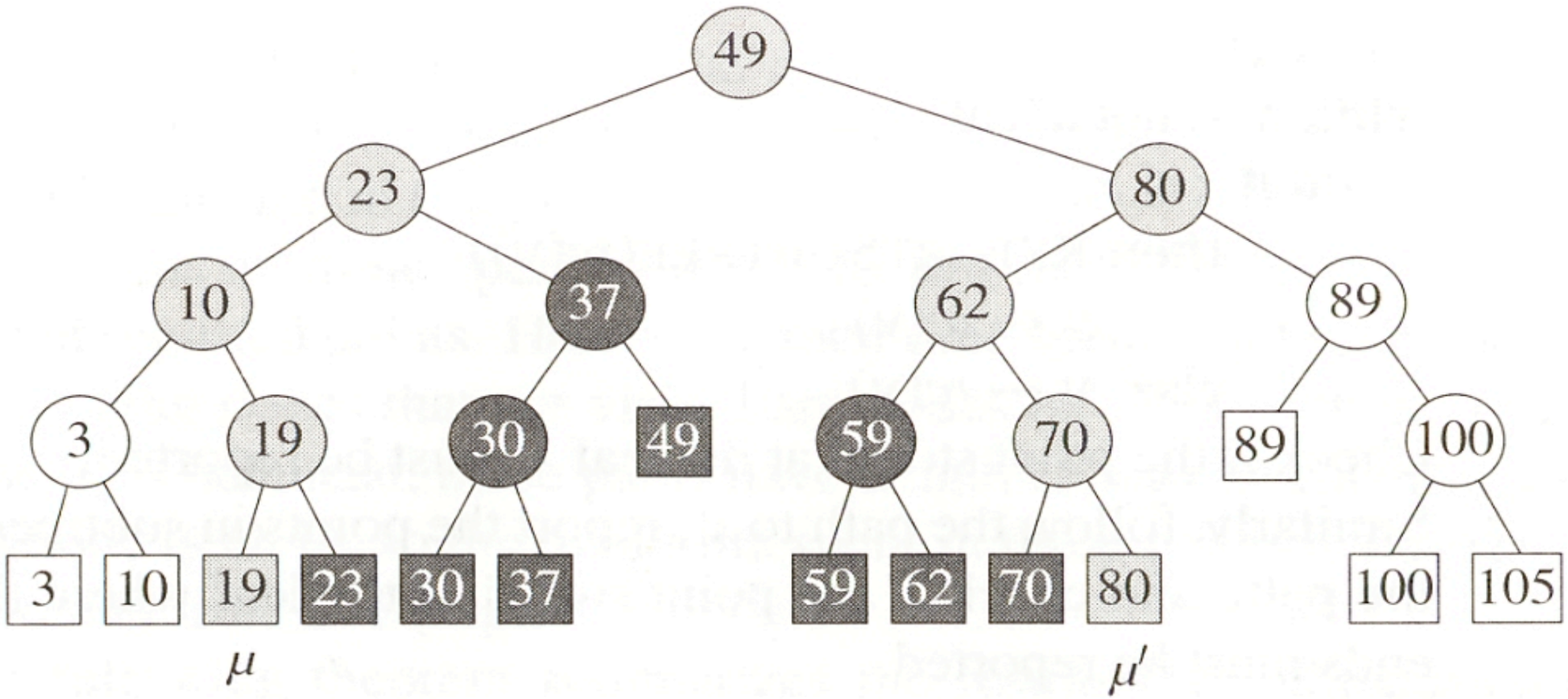
In eindimensionalen Suchräumen bieten sich Suchbäume (*heaps*) an, mit $n \cdot \ell ogn$ Suchzeiten (inkl. Aufbau des *heaps*).

Wie ist das im Mehrdimensionalen und bei Bereichsanfragen (*range queries*)?





Range Tree

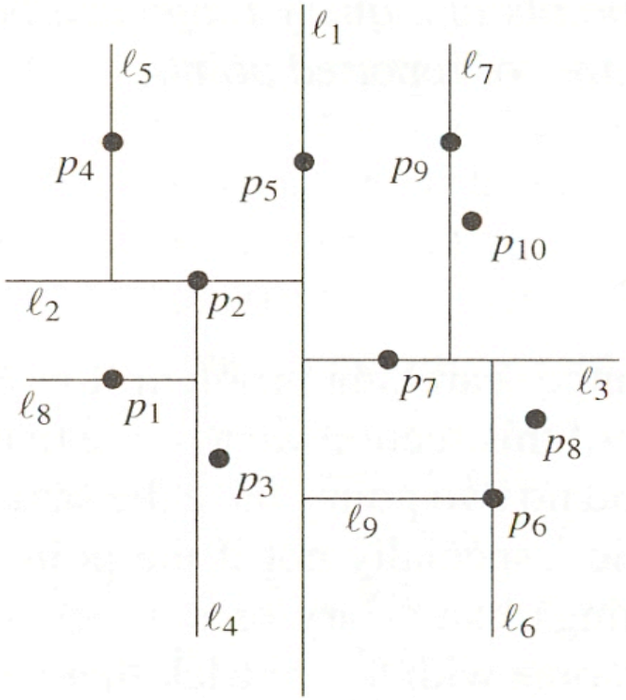

49
23
80
10
37
62
89
3
19
30
49
59
70
89
100
3
10
19
23
30
37
59
62
70
80
100
105
μ
μ′

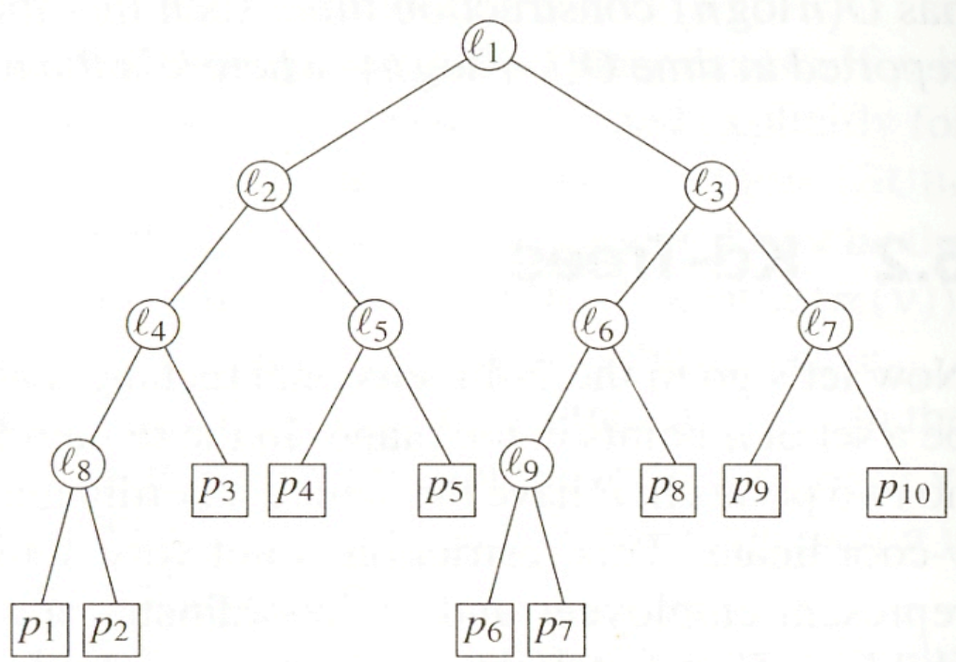
kd-Bäume und Range-Trees

In de Berg et al. findet man das Ergebnis:

Sei P eine Menge von Punkten im **eindimensionalen** Suchraum, dann kann P mit $O(n)$ Platzbedarf in balanziertem Suchbaum in $O(n \cdot \ell ogn)$ Zeit gespeichert werden, und die k Elemente einer Bereichsanfrage können in $O(k + \ell ogn)$ Zeit angegeben werden.

In mehr-dimension-alen Suchräumen können kd-Bäume (2-dimensional im Bild) oder auch Range-Trees genutzt werden.





Kd-Bäume

Theorem:

Ein kd-Baum für eine Menge P von n Punkten im 2-dimensionalen Suchraum kann in $O(n \cdot \ell ogn)$ Zeit aufgebaut werden und benötigt $O(n)$ Speicherplatz. Eine orthogonale Bereichsanfrage benötigt $O(\sqrt{n}+k)$ Zeit um k Ergebniselemente zu ermitteln.

Bei einer Menge P von n Punkten im d-dimensionalen Suchraum benötigt man nur $O(d \cdot n)$ Speicherplatz und $O(d \cdot n \cdot \ell ogn)$ Aufbauzeit. Die Anfragezeit um k Elemente in einem orthogonalen Bereich zu ermitteln ist in $O(n^{1-1/d}+k)$.

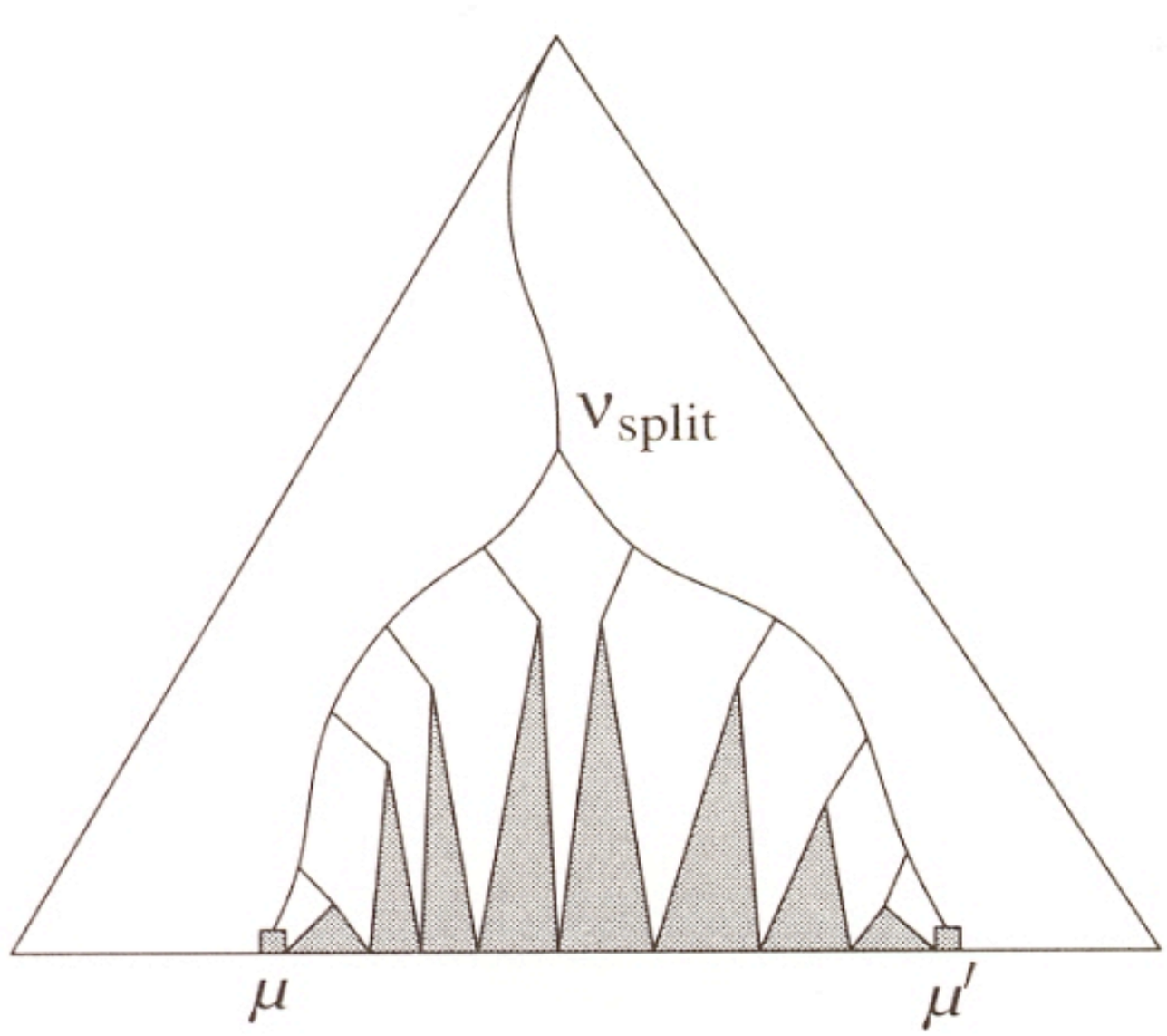
Die Bereichsanfragen sind bei kd-Bäumen relativ groß. Sie sind bei den *Range-Trees* um Größenordnungen besser!

Range-Tree x-Koordinate

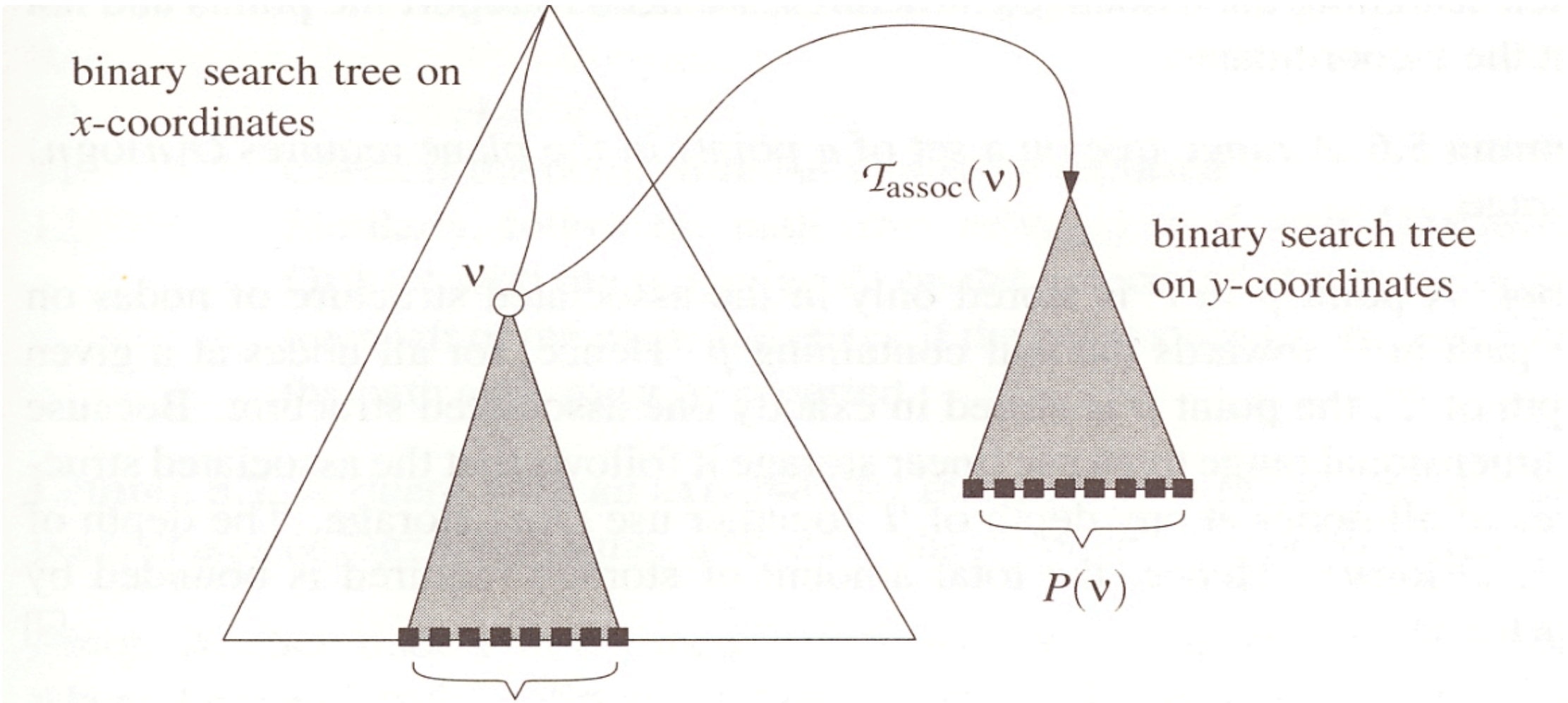
Theorem:

Ein *Range-Tree* für eine Menge P von n Punkten im 2-dimensionalen Suchraum kann in $O(n \cdot \ell og n)$ Zeit aufgebaut werden und benötigt $O(n \cdot \ell og n)$ Speicherplatz. Eine orthogonale Bereichsanfrage benötigt $O(\ell og^2 n + k)$ Zeit um k Ergebniselemente zu ermitteln.

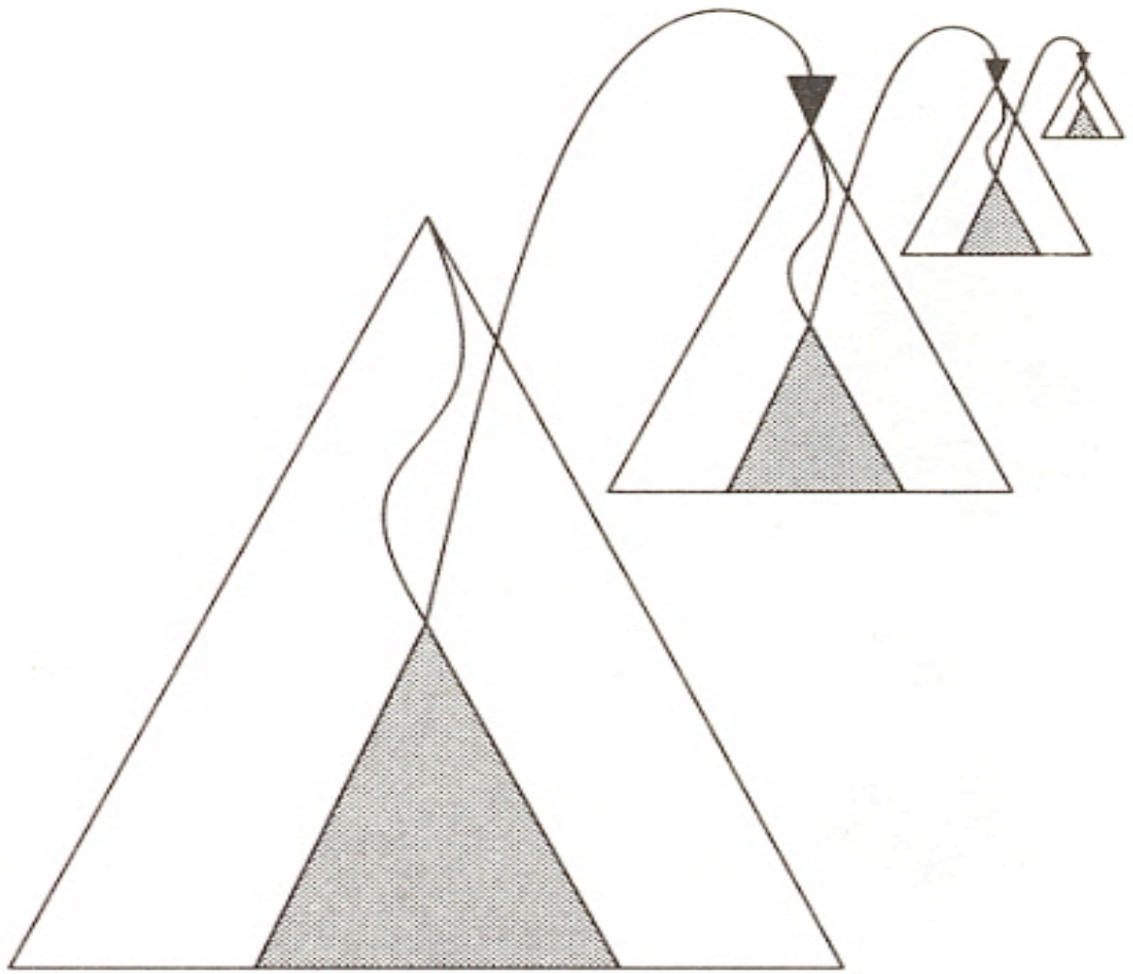
Bei einer Menge P von n Punkten im d-dimensionalen Suchraum, $d \geq 2$, benötigt man $O(n \cdot \ell og^{d-1} n)$ Speicherplatz und $O(n \cdot \ell og^{d-1} n)$ Aufbauzeit. Die Anfragezeit um k Elemente in einem orthogonalen Bereich zu ermitteln ist in $O(\ell og^d n + k)$.



Range-Tree y-Koordinate



Range-Tree d-dimensional



Problem: identische x_i-Koordinaten bei nicht identischen Daten

Verschiedene Punkte können gleiche Werte in einzelnen Koordinaten aufweisen!

Um in jeder Koordinate eine eindimensionale Bereichssuche durchzuführen, müssen die Werte jedes Randes unterschiedlich sein!

Dazu im 2-dimensionalen: statt des Paares (x, y) wähle $(x|y,\ y|x)$ mit der Ordnung $x_1|y_1 < x_2|y_2$ genau dann, wenn $x_1 < x_2$ oder ($x_1 = x_2$ und $y_1 < y_2$).

Der Bereich $[x_1 : x_2] \times [y_1 : y_2]$

wird nun zu $[x_1|-\infty\ :\ x_2\ |+\infty] \times [y_1|-\infty\ :\ y_2|+\infty]$

Minimale Distanz von Punkten im $\mathbb{R}^d$ entspricht in Datenbanken einer Ähnlichkeit der Einträge.

Daher auch hier das Problem minimaler Distanz wichtig und die Verfahren und Erkenntnisse vom $\mathbb{R}^d$ sind übertragbar.