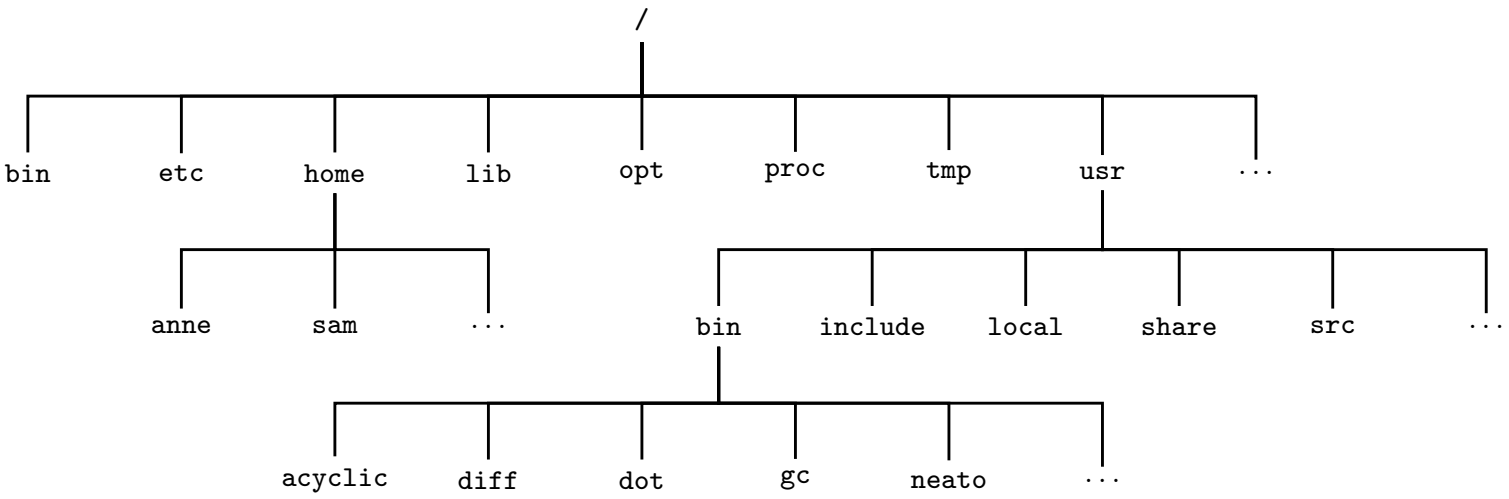
Algorithmische Graphentheorie

Kapitel 4: Suchalgorithmen II

Babeş-Bolyai Universität, Fachbereich Informatik, Klausenburg

Wiederholung: Bäume

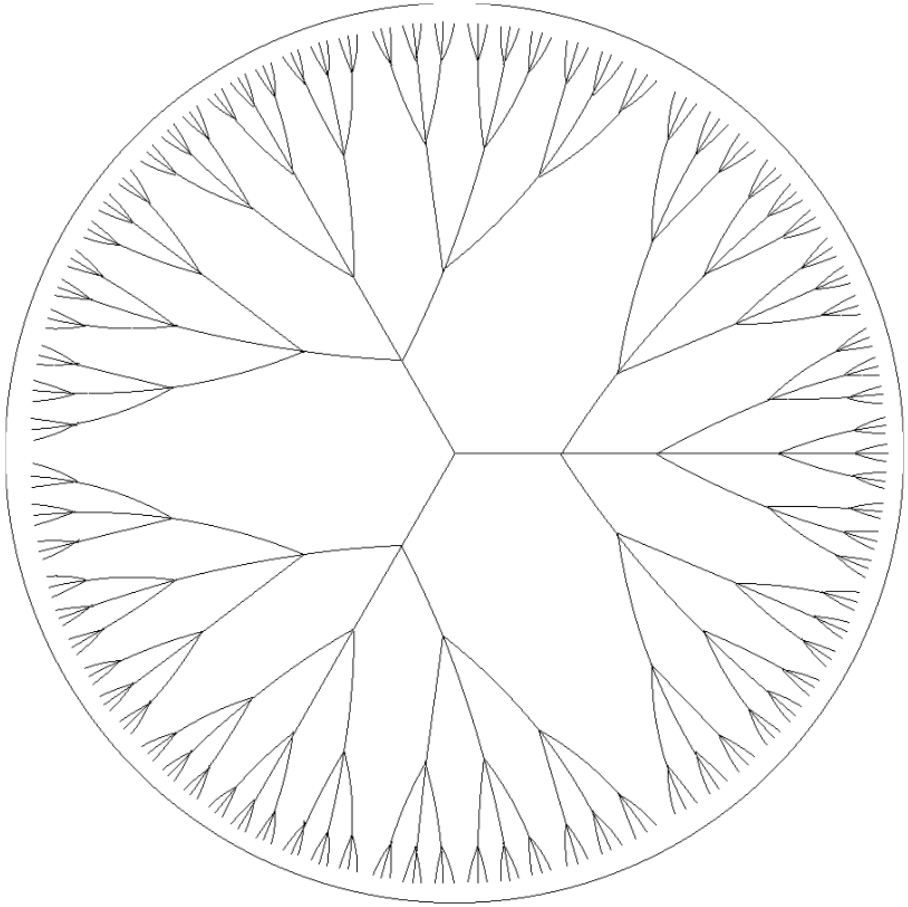


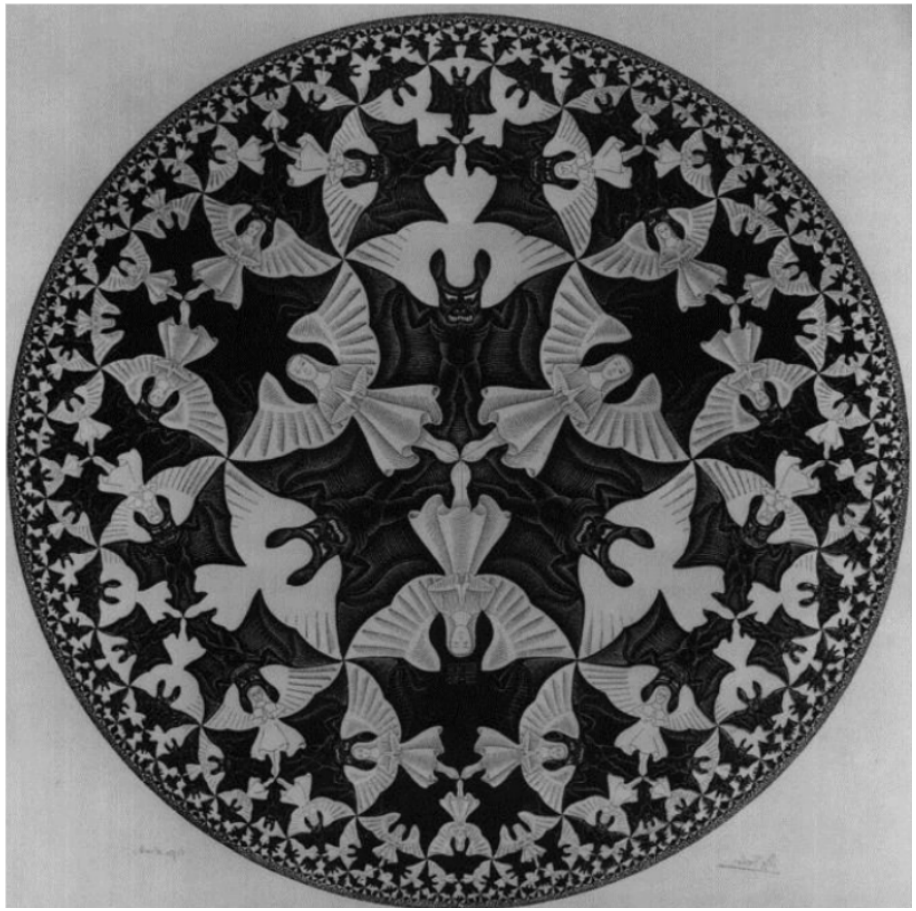
Die Hierarchie des Linux Dateisystems.

Begriffe

- Gerichtete Bäume vs. ungerichtete Bäume
- Wurzel, Vorgänger, Nachgänger oder Kind und Eltern
- Tiefe – Entfernung von der Wurzel und Höhe – Länge des längsten Weges von der Wurzel aus startend
- Verzweigungsfaktor (= Anzahl Kinder)

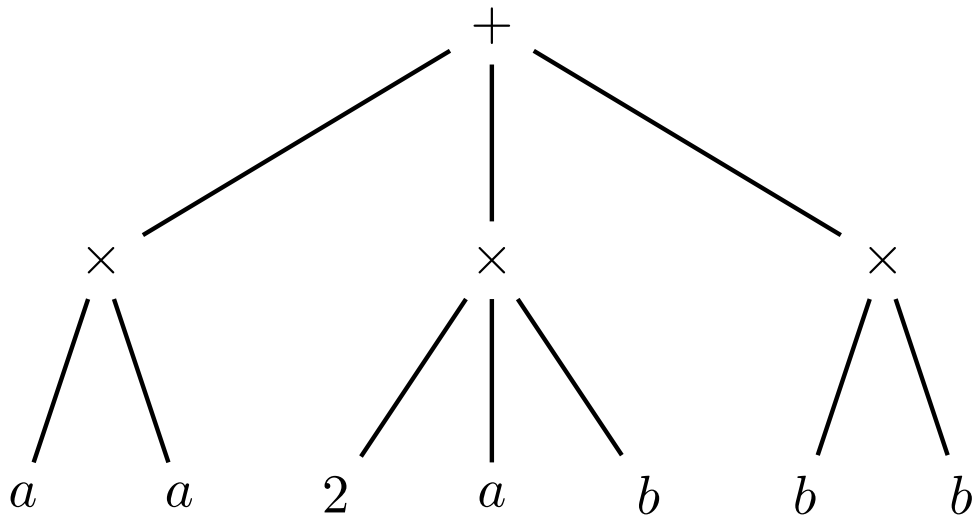
Beispiel 1





M. C. Escher, Circle Limit IV (Heaven and Hell), 1960.
Der dem Bild zugrundeliegende Baum hat Tiefe 5 und Verzweigungsfaktor 3.

Beispiel 2

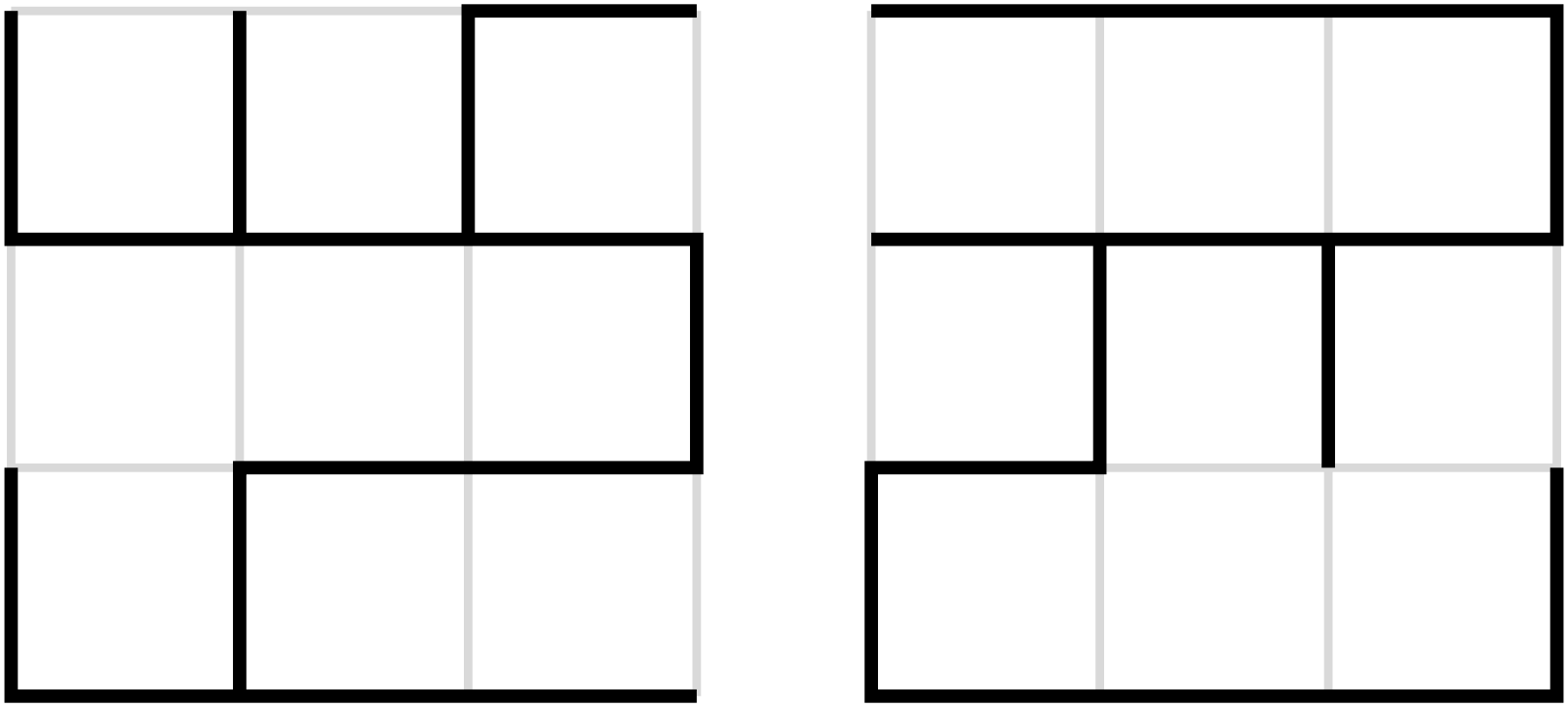


$$a^2 + 2ab + b^2.$$

Wiederholung: Bäume

Es sei $G = (V, E)$ ein ungerichteter Graph und $T = (V', E')$ ein Teilgraph von G. Dann heißt T *aufspannender Baum* oder *Spannbaum* (*spanning tree*) von G, wenn $V' = V$ gilt und T ein Baum ist, also kreisfrei und zusammenhängend.

Beispiel



(a) (b)

Zwei aufspannende Bäume für den 4×4 Grid-Graphen.

Ein Graph besitzt im Allgemeinen mehr als einen Spannbaum, manchmal sogar sehr viele. Um dies zu illustrieren, betrachten wir Graphen mit möglichst vielen Kanten.

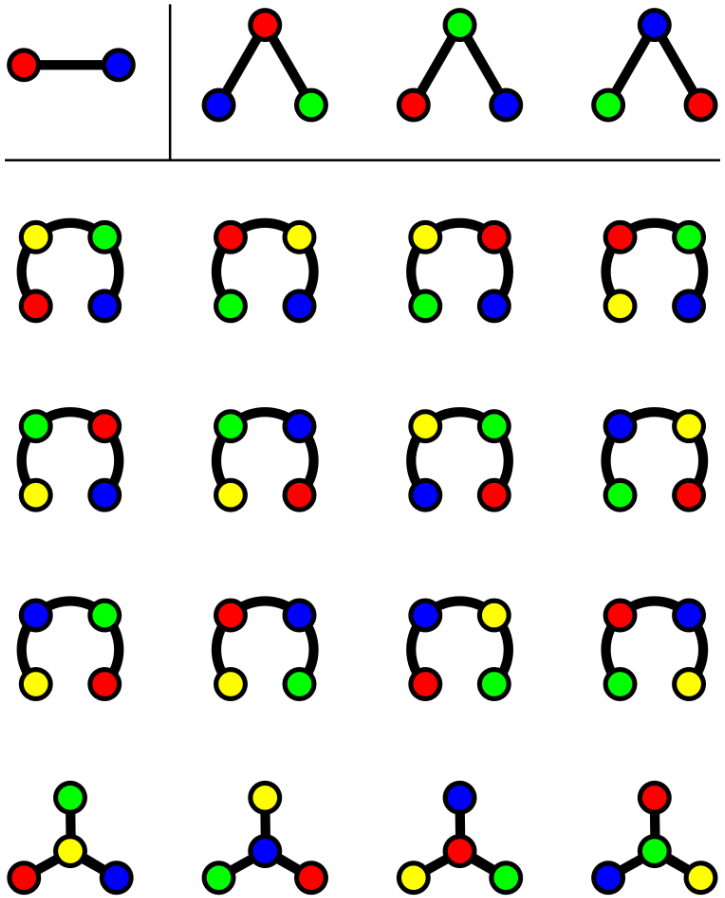
Wiederholung: Ein ungerichteter Graph $G = (V, E)$ heißt *vollständig*, wenn für je zwei Knoten $v, w \in V$ gilt, dass $\{v, w\} \in E$. Einen ungerichteten, einfachen und vollständigen Graphen mit n Knoten bezeichnet man mit K_n.

Aufspannende Bäume zählen

Wir wollen nun untersuchen, *wie viele* Spannbäume ein vollständiger Graph enthält.

Satz (Cayley, 1889). *Für $n \geq 2$ gibt es n^{n-2} verschiedene Spannbäume in einem vollständigen Graphen K_n mit eindeutig markierten Knoten.*

Beispiel



Der Satz von Cayley besagt, dass es $1 = 2^{2-2}$ Spannbaum in K_2 gibt, $3 = 3^{3-2}$ Spannbäume in K_3 und $16 = 4^{4-2}$ Spannbäume in K_4.

Rekursiver Aufbau von Bäumen

Ein isolierter Knoten ist ein Baum. Dieser Knoten ist die Wurzel des Baumes. Gegeben eine Liste $T_1, \ldots, T_n$ von $n > 0$ Bäumen, erzeuge einen neuen Baum wie folgt:

1. Es sei T der Baum mit genau einem Knoten v, der Wurzel von T.
2. Es sei v_i die Wurzel des Baumes T_i.
3. Für $i = 1, 2, \ldots, n$, füge die Kanten vv_i in T ein und füge T_i zu T. Jedes v_i ist ein Kind von v.

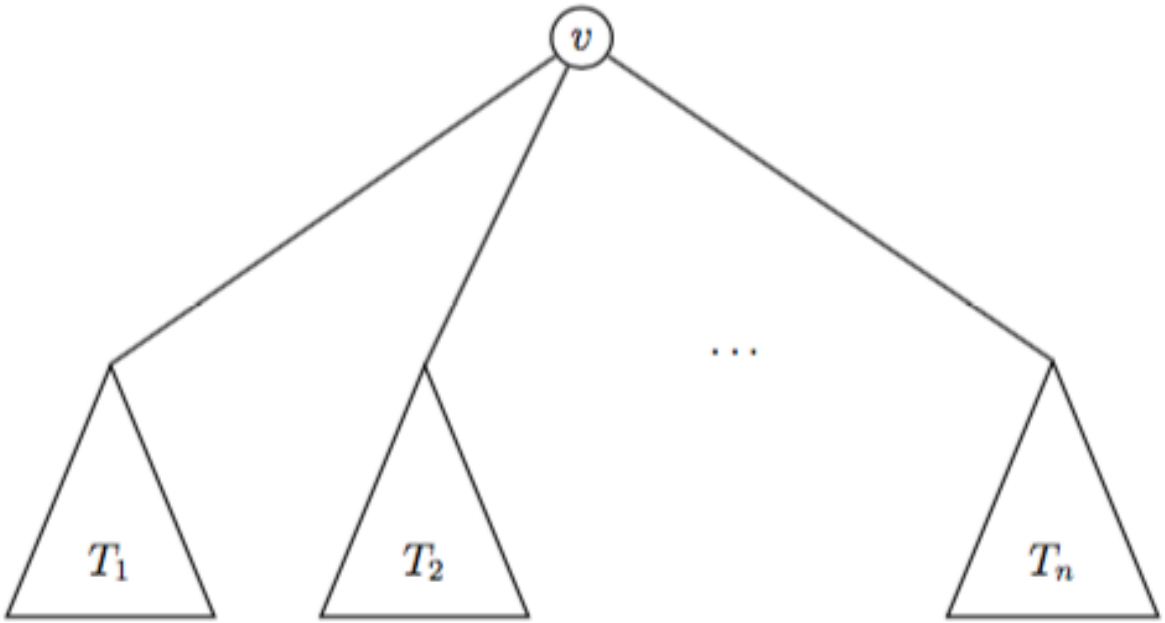
Das Ergebnis ist ein Baum T mit Wurzel v, Knotenmenge

$$V(T) = \{v\} \cup \left(\bigcup_i V(T_i) \right)$$

und Kantenmenge

$$E(T) = \bigcup_i \left(\{vv_i\} \cup E(T_i) \right).$$

Rekursiver Aufbau von Bäumen



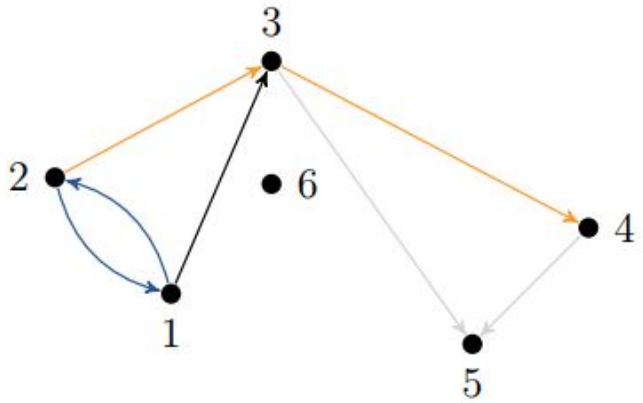
Bäume in gerichteten Graphen

Ist $G = (V, A)$ ein gerichteter Graph, so heißt G *stark zusammenhängend*, wenn für je zwei Knoten $v, w \in V$ ein gerichteter Weg P mit Anfangsknoten v und Endknoten w existiert, und *zusammenhängend* (oder auch: *schwach zusammenhängend*), wenn für je zwei Knoten $v, w \in V$ ein **un**gerichteter Weg P mit Anfangsknoten v und Endknoten w existiert.

Ein Knoten $s \in V$ heißt *Wurzel* von G, wenn für alle Knoten $v \in V$ ein gerichteter Weg P mit Anfangsknoten s und Endknoten v existiert.

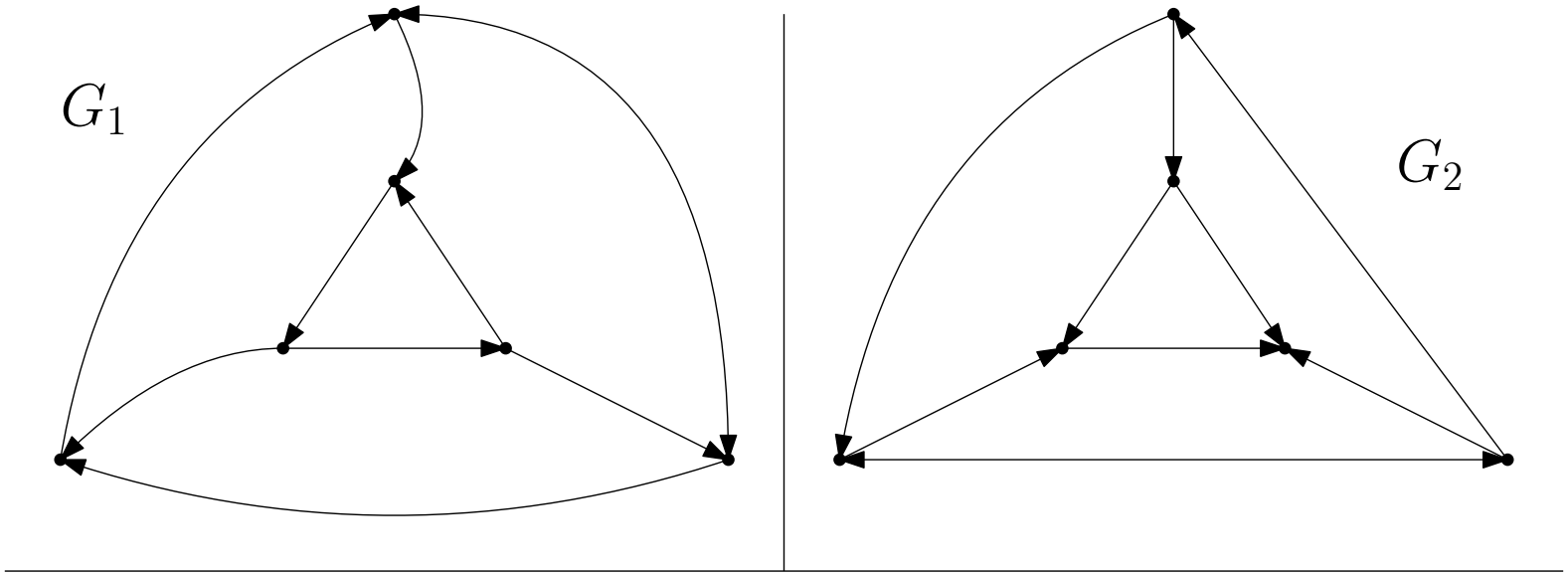
Bemerkung

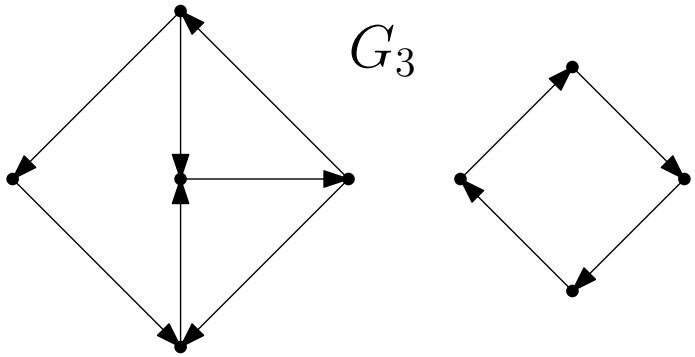
Ein gerichteter Graph, der eine Wurzel besitzt, ist immer schwach zusammenhängend, aber nicht jeder schwach zusammenhängende Graph besitzt eine Wurzel. Ein gerichteter Graph ist stark zusammenhängend genau dann, wenn jeder Knoten eine Wurzel ist.



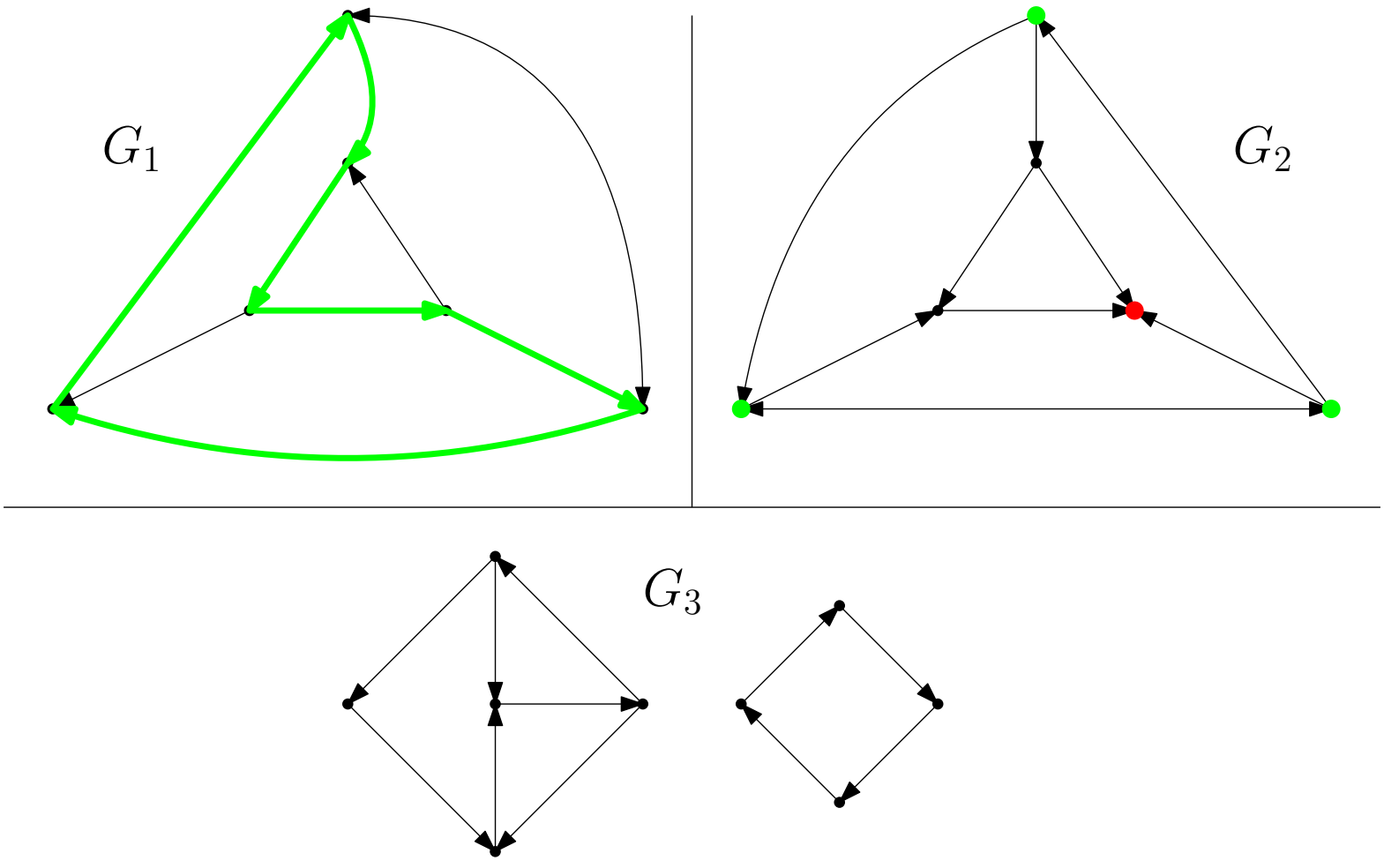
Die Abbildung zeigt einen gerichteten Graphen, der weder stark noch schwach zusammenhängend ist. Würde der Knoten 6 nicht zum Graphen gehören, so wären die Knoten 1 und 2 Wurzeln und der Graph wäre schwach, jedoch nicht stark zusammenhängend.

Aufgabe. Bestimmen Sie für die folgenden drei Digraphen, ob diese stark oder schwach zusammenhängend sind (möglicherweise weder noch). Bestimmen Sie auch alle Knoten, die Wurzeln sind.





G_1 ist stark zshg. (also ist jeder Knoten eine Wurzel) aufgrund des grünen aufspannenden Kreises. G_2 ist nicht stark zshg. wegen des roten Knotens, jedoch schwach zshg. (Wurzeln in grün). G_3 ist nicht schwach zshg. (und daher auch nicht stark zshg.; keine Wurzeln).



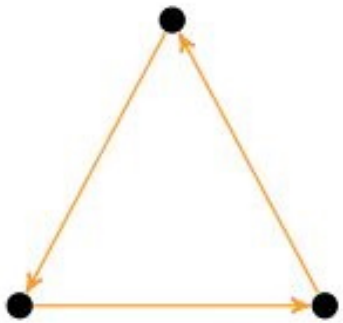
Bäume in gerichteten Graphen

Ein ungerichteter Graph ist ein Baum genau dann, wenn er zusammenhängend und kreisfrei ist.

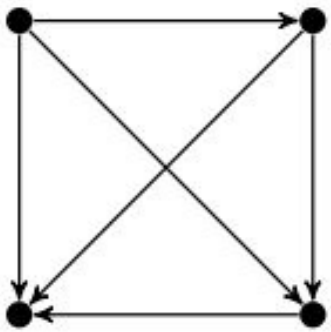
Übertragung auf gerichtete Graphen:

- Ist ein gerichteter Baum stark oder schwach zusammenhängend?
- Darf er keine gerichteten Kreise enthalten oder auch keine ungerichteten Kreise?

Zusammenhängende, gerichtete Graphen



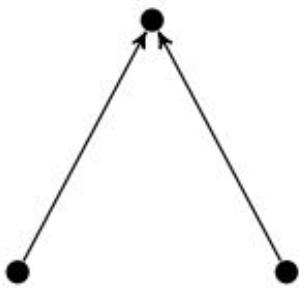
(a) stark zusammenhängend, mit gerichtetem Kreis



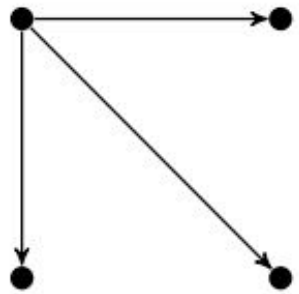
(b) schwach zusammenhängend, ohne gerichtete Kreise

Bäume in gerichteten Graphen

- Es sei $T = (V, A)$ ein gerichteter Graph. T heißt *Baum*, wenn der zugeordnete ungerichtete Graph ein Baum ist.
- Bäume in gerichteten Graphen sind schwach zusammenhängend und kreisfrei, sie enthalten also keine ungerichteten Kreise – ausnahmsweise kann ein Kreis hier auch Länge 2 haben, denn diese möchten wir ebenso ausschließen. D.h. falls $(v, w) \in A$, so gilt $(w, v) \notin A$.
- Der Rückzug auf schwachen Zusammenhang hat allerdings den Nachteil, dass in einem gerichteten Graphen $T = (V, A)$ für zwei beliebige Knoten $v, w \in V$ nicht garantiert ist, dass es in T einen gerichteten Weg von v nach w gibt.
- Es sei $T = (V, A)$ ein gerichteter Graph. T heißt *Wurzelbaum* mit Wurzel s, wenn T ein Baum ist, in dem s eine Wurzel ist.



(a) Baum



(b) Wurzelbaum

Gerichtete Bäume

Wiederholung: Der *Eingangsgrad* (*in-degree*) $d^-(v)$ eines Knotens v ist die Anzahl der Vorgänger von v, also die Anzahl der Kanten, deren Endknoten v ist. Wir können auch schreiben:

$$d^-(v) := |\{(u, v) : u \in V\}|.$$

Der *Ausgangsgrad* (*out-degree*) $d^+(v)$ von v ist die Anzahl der Nachfolger von v, also die Anzahl der Kanten deren Anfangsknoten v ist. Wir können auch schreiben:

$$d^+(v) := |\{(v, w) : w \in V\}|.$$

Satz. *Es sei $T = (V, A)$ ein gerichteter Graph und $s \in V$. Dann sind äquivalent:*

(a) T ist ein Wurzelbaum mit Wurzel s.

(b) T ist ein Baum, $d^-(s) = 0$ und $d^-(v) = 1$ für alle $v \in V \setminus \{s\}$.

(c) s ist eine Wurzel in T, $d^-(s) = 0$ und $d^-(v) \leq 1$ für alle $v \in V \setminus \{s\}$.

(a) $\Longrightarrow$ (b)

- Jeder Knoten $v \in V \setminus \{s\}$ ist von s aus durch einen gerichteten Weg erreichbar, es gilt also $d^-(v) \geq 1$.
- Der zu T gehörende ungerichtete Graph mit Kantenmenge E ist ein Baum, also gilt (Kapitel 2):

$$|V| - 1 = |E| = d^-(s) + \underbrace{\sum_{v \in V \setminus \{s\}} d^-(v)}_{\geq |V|-1} .$$

- Aus $d^-(v) \geq 1$ für alle $v \neq s$ folgt durch obige Gleichungen $d^-(s) = 0$.
- $\sum_{v \in V \setminus \{s\}} d^-(v)$ besitzt $|V| - 1$ Summanden, wovon jeder ≥ 1 ist. Die Summe ergibt jedoch genau $|V| - 1$. Somit gilt $d^-(v) = 1$ für alle $v \in V \setminus \{s\}$.

(b) $\Longrightarrow$ (c)

- Es reicht zu zeigen, dass s eine Wurzel von T ist.
- Es sei $v \in V$ beliebig.
- Z.z.: v ist durch einen gerichteten Weg von s aus erreichbar.
- Für $v = s$ trivialerweise erfüllt, es sei also $v \neq s$.
- Da T schwach zshg. ist, gibt es einen (möglicherweise ungerichteten) Weg P definiert durch $s = v_0 \overset{e_1}{—} v_1 \overset{e_2}{—} \ldots \overset{e_\ell}{—} v_\ell = v$.
- Wegen $d^-(s) = 0$ folgt $e_1 = v_0 v_1$, die Kante e_1 ist also für unsere Zwecke richtig orientiert.
- Wegen $d^-(v_1) = 1$ und $v_2 \neq v_0$ muss dann aber auch $e_2 = v_1 v_2$ gelten.
- Iterativ zeigt man so, dass P ein gerichteter Weg sein muss.

(c) $\Longrightarrow$ (a)

- Z.z.: T ist ein Baum.
- Nach Voraussetzung ist s Wurzel von T, daher ist T schwach zusammenhängend und $|A| \geq |V| - 1$.
- Andererseits gilt

$$|A| = d^-(s) + \sum_{v \in V \setminus \{s\}} d^-(v) \leq |V| - 1$$

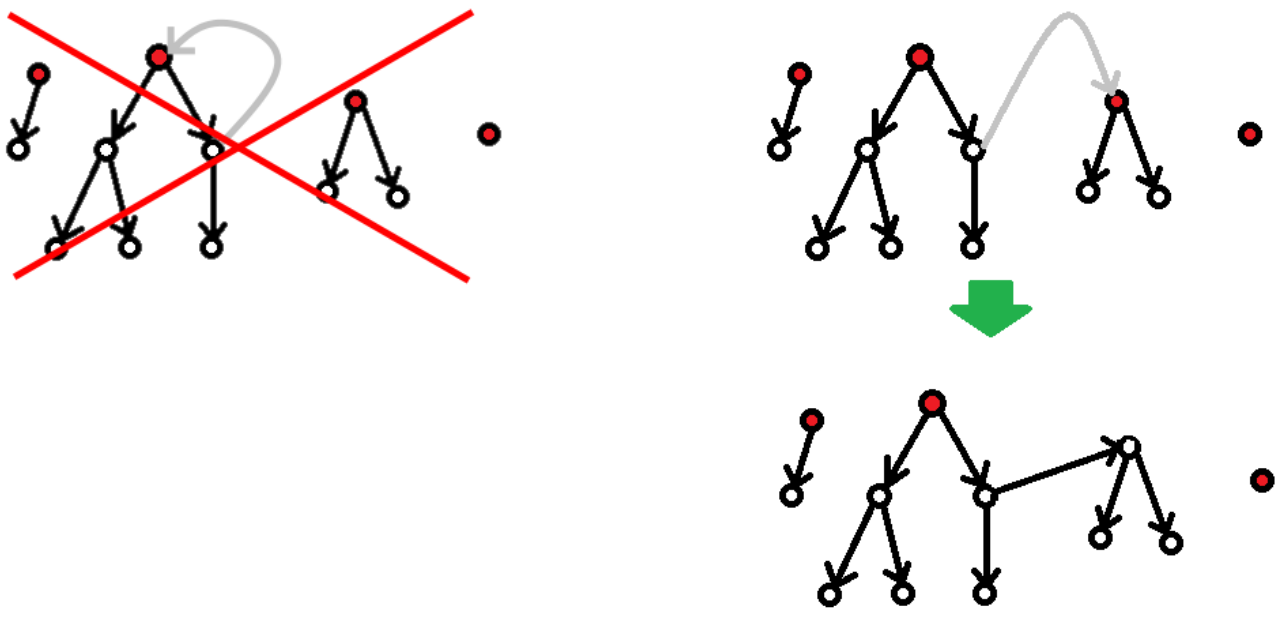
  wegen $d^-(s) = 0$ und $d^-(v) \leq 1$ für alle $v \neq s$.
- Somit gilt $|A| = |V| - 1$ und T muss ein Baum sein.

Beweis (via *double counting*) von J. Pitman der cayleyschen Formel von Folie 9 (dieser und weitere Beweise zu finden in: M. Aigner und G. M. Ziegler, Proofs from THE BOOK, 1998, Springer Verlag):

- Wir zählen auf zwei Weisen die Anzahl der verschiedenen Folgen gerichteter Kanten, die einem kantenfreien Graphen mit n Knoten hinzugefügt werden können, um daraus einen Wurzelbaum zu bilden.
- Eine Möglichkeit, eine solche Folge zu bilden, besteht darin, mit einem der t_n möglichen Bäume ohne Wurzel zu beginnen, einen seiner n Knoten als Wurzel zu wählen und eine der $(n-1)!$ möglichen Folgen zu wählen, in der seine $n-1$ (gerichteten) Kanten hinzugefügt werden.
- Die Gesamtzahl der auf diese Weise möglichen Folgen ist also $t_n n(n-1)! = t_n n!$.

- Weitere Möglichkeit, diese Kantenfolgen zu zählen: Kanten nacheinander zu einem kantenfreien Graphen hinzufügen und die Anzahl der bei jedem Schritt verfügbaren Auswahlmöglichkeiten zu zählen.

- Wenn man bereits $n - k$ Kanten hinzugefügt hat, sodass der durch diese Kanten gebildete Graph ein Wurzelwald (:= Menge disjunkter Wurzelbäume) mit k Bäumen ist, gibt es $n(k-1)$ Auswahlmöglichkeiten für die nächste hinzuzufügende Kante: ihr Anfangspunkt kann jeder der n Knoten des Graphen sein, und ihr Endpunkt kann jede der $k - 1$ Wurzeln sein, außer der Wurzel des Baumes, die den Anfangsknoten enthält.



Wenn man also die Anzahl der Auswahlmöglichkeiten aus dem ersten Schritt, dem zweiten Schritt usw. miteinander multipliziert, ergibt sich die Gesamtzahl der Auswahlmöglichkeiten

$$\prod_{k=2}^{n} n(k-1) = n^{n-1}(n-1)! = n^{n-2}n!,$$

also

$$t_n n! = n^{n-2}n!$$

und somit

$$t_n = n^{n-2}$$

was zu zeigen war.

Minimale aufspannende Bäume

In vielen Anwendungsproblemen geht es darum, gewisse Knoten mit möglichst wenigen Kanten so zu verbinden, dass zwischen allen Paaren von Knoten ein Weg existiert.

Magnetschwebebahn

Städte werden als Knoten modelliert und die dazwischen technisch möglichen Verbindungen als Kanten. Wenn wir davon ausgehen, dass die Bahn immer in beide Richtungen fährt, erhalten wir so einen ungerichteten Graphen. Aus Kostengründen werden nur möglichst wenige Strecken tatsächlich gebaut, d.h. wir suchen einen Spannbaum. Wir wissen schon, dass wir mindestens *Anzahl der Städte - 1* Strecken realisieren müssen. Nun kosten die möglichen Strecken aber nicht alle gleich viel. Welchen aufspannenden Baum sollen wir also wählen?

Minimale aufspannende Bäume

Broadcasting

Wir betrachten ein Netzwerk von Sendemasten unterschiedlicher Reichweiten, in dem eine Information kostengünstig von einem zentralen Mast an alle anderen weitergegeben werden soll. So erhalten wir einen gerichteten Graphen, in dem wir einen aufspannenden Wurzelbaum suchen.

Gewichtete Graphen und minimale aufspannende Bäume

Es sei $G = (V, E)$ ein ungerichteter oder gerichteter Graph. G heißt *gewichtet*, wenn jeder Kante $e \in E$ ein Gewicht $w(e) \in \mathbb{R}$ zugeordnet wird.

Es sei $G = (V, E)$ ein gewichteter, ungerichteter Graph. Ein Teilgraph $T = (V, E_T)$ von G heißt *minimaler Spannbaum* (*minimum spanning tree; MST*) von G, wenn sein Gewicht

$$w(T) := \sum_{e \in E_T} w(e)$$

minimal unter allen Spannbäumen von G ist.

Charakterisierungssatz minimaler Spannbäume

Satz. *Es sei $G = (V, E)$ ein ungerichteter, gewichteter und zusammenhängender Graph mit $n = |V|$ Knoten und Kantengewichten $w(e)$ für alle $e \in E$ und $T = (V, E_T)$ ein aufspannender Baum. Dann sind äquivalent:*

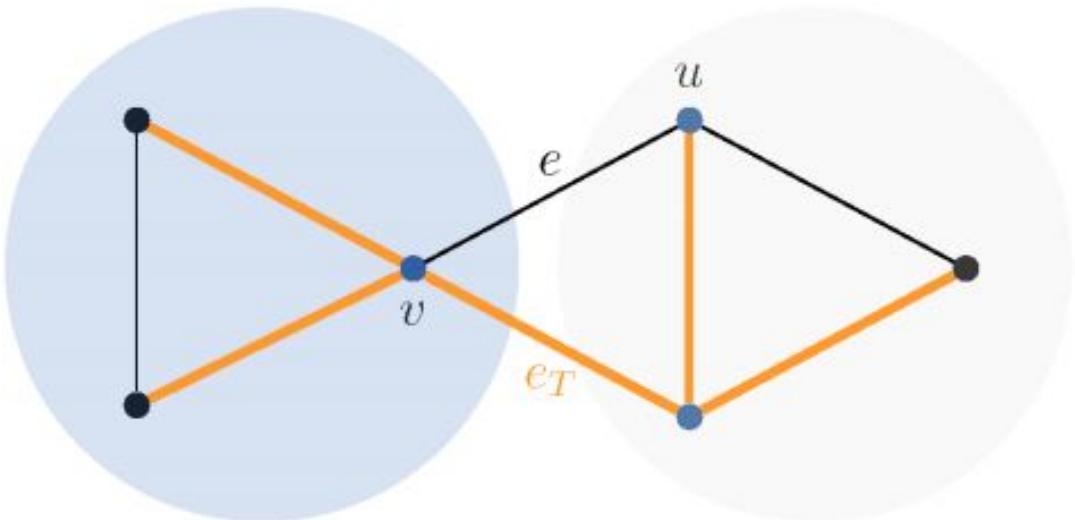
1. *T ist ein minimaler aufspannender Baum.*
2. *Für jede Kante $e \in E \setminus E_T$ gilt: e ist eine Kante mit maximalem Gewicht in dem eindeutigen Kreis, der entsteht, wenn man e zu T hinzufügt.*
3. *Für jede Kante $e_T \in E_T$ gilt: Entfernt man e_T aus T, so zerfällt T in zwei Zusammenhangskomponenten Z_1 und Z_2. Die Kante e_T hat in beiden Zusammenhangskomponenten jeweils einen Endknoten und minimales Gewicht unter allen solchen Kanten.*

Beweis (1/5): (1) $\Rightarrow$ (2)

- Es sei T ein minimaler Spannbaum und $e = \{v, u\} \in E \setminus E_T$ beliebig.
- Da die Knoten v, u im Baum T durch einen eindeutigen Weg P verbunden sind, entsteht durch das Hinzufügen der Kante e zu T ein eindeutiger Kreis.
- Angenommen, es gibt eine Kante $e_T \in E(P)$ mit $w(e_T) > w(e)$.

Beweis (2/5): (1) $\Rightarrow$ (2)

- Dann betrachten wir den Graphen $T' = (V, (E_T \setminus \{e_T\}) \cup \{e\})$.
- Es gilt offensichtlich $w(T') < w(T)$.
- Außerdem hat T' wieder $n - 1$ Kanten und ist zusammenhängend.
- Folglich ist T' ein Spannbaum mit einem kleineren Gewicht als T; Widerspruch.



Entferne e_T aus T und füge e hinzu.

Beweis (3/5): (2) $\Rightarrow$ (3)

- Es sei T ein Spannbaum mit Eigenschaft (2).
- Angenommen es gibt eine Kante $e_T \in E_T$, für die (3) verletzt ist, d.h. es gibt eine Kante e mit Endknoten in Z_1 und Z_2, für die gilt $w(e) < w(e_T)$.
- Die Kanten e und e_T verbinden die gleichen Zusammenhangskomponenten, folglich gilt $e \notin E_T$.
- Fügen wir die Kante e zu T hinzu, so entsteht ein eindeutiger Kreis, auf dem auch e_T liegen muss, da beide Kanten die Zusammenhangskomponenten Z_1 und Z_2 verbinden.
- Widerspruch zu (2), da e nicht die Kante mit maximalem Gewicht auf diesem Kreis ist.

Beweis (4/5): (3) $\Rightarrow$ (1)

- Es sei T ein Spannbaum mit Eigenschaft (3) und $T^* = (V, E^*)$ ein minimaler aufspannender Baum, der maximal viele Kanten mit T gemeinsam hat.
- T und T^* sind Spannbäume, d.h. sie haben gleich viele Kanten. Gilt $T \neq T^*$, so muss es also mindestens eine Kante $e_T \in E_T \setminus E^*$ geben.
- Entfernen wir e_T aus T, so zerfällt T in zwei Zusammenhangskomponenten Z_1 und Z_2.
- Da T^* ein aufspannender Baum ist, muss es eine Kante $e^* \in E^*$ geben, die Endknoten in Z_1 und Z_2 hat.
- Wegen $e_T \notin E^*$ folgt $e^* \neq e_T$ und daher $w(e^*) \geq w(e_T)$ wegen Eigenschaft (3).

Beweis (5/5): (3) $\Rightarrow$ (1)

- Betrachten wir nun den Teilgraphen $T' = (V, (E^* \setminus \{e^*\}) \cup \{e_T\})$, so gilt $w(T') \leq w(T^*)$ und T' ist ebenfalls ein Spannbaum (vergleiche $(1) \Rightarrow (2)$).
- T^* ist ein minimaler Spannbaum, es muss also $w(T') = w(T^*)$ gelten.
- Wir haben also einen minimalen Spannbaum T' konstruiert, der eine Kante mehr mit T gemeinsam hat als T^*, Widerspruch!
- Folglich muss $T = T^*$ gelten, d.h. T ist ein minimaler Spannbaum.

Algorithmus von Prim

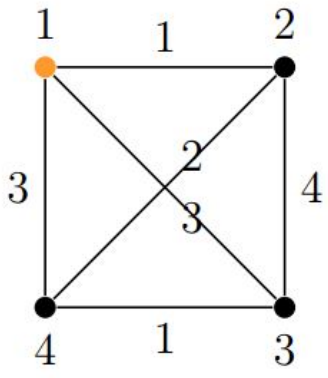
Wie findet man einen minimalen Spannbaum mit kleinstem Gewicht?

Der folgende Algorithmus wurde 1930 von Vojtěch Jarník entwickelt. 1957 wurde er zunächst von Robert C. Prim und dann 1959 von Edsger W. Dijkstra wiederentdeckt.

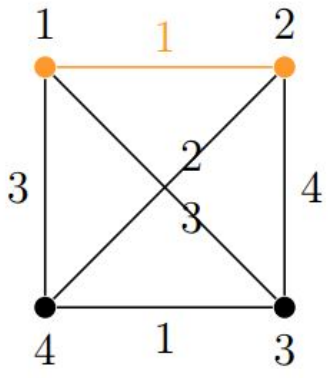
Idee

Starte mit einem beliebigen Knoten. Der Baum ist zusammenhängend, also muss von diesem Knoten eine Kante ausgehen. Wähle hierfür diejenige mit kleinstem Gewicht. Nun hat man einen Teilgraphen mit zwei Knoten und einer Kante. Von diesem Teilgraphen muss wiederum eine Kante ausgehen zu einem noch nicht enthaltenen Knoten. Wähle hierfür wieder die mit kleinstem Gewicht und setze das Verfahren fort, bis alle Knoten verbunden sind.

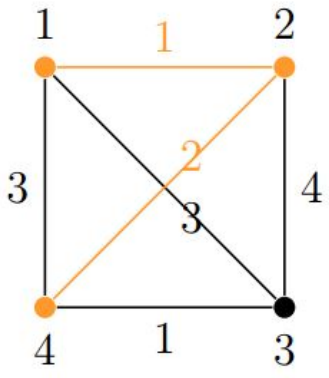
Algorithmus von Prim: Beispiel



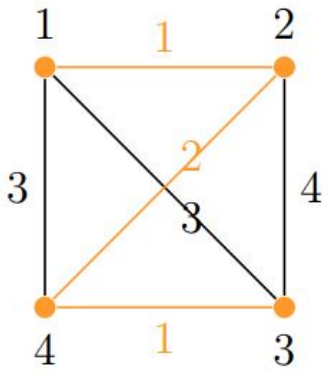
(a) Start



(b) 1. Iteration



(c) 2. Iteration

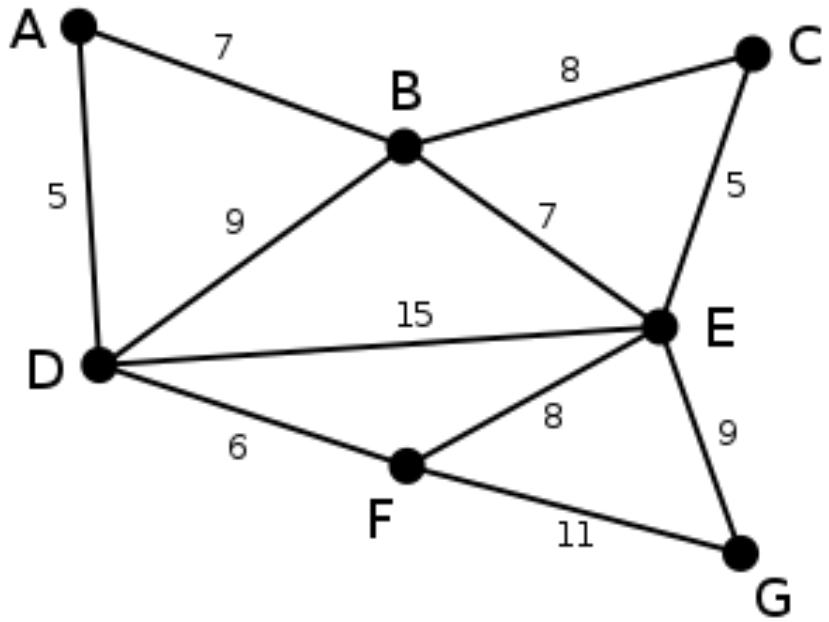


(d) 3. Iteration

Algorithmus von Prim: Pseudocode

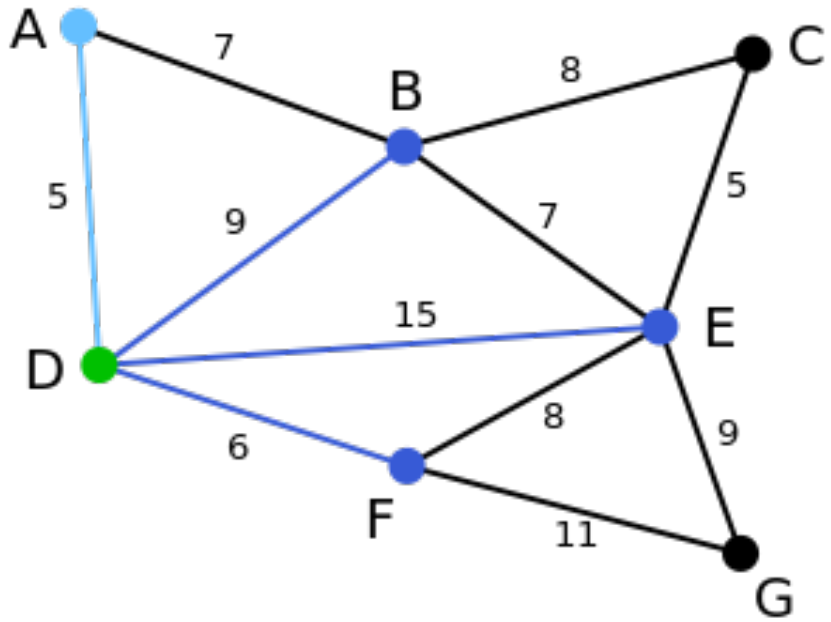
```
procedure prim(G : Graph; var B : Baum);
      U : set of Integer;
      u, v : Integer;
begin
      B.initBaum(G);
      U := {1};
      while U.anzahl ≠ n do begin
          Sei (v,u) die Kante aus G mit der kleinsten Bewertung,
              so daß u ∈ U und v ∈ V\U
          B.einfügen(v,u);
          U.einfügen(v);
      end
end
```

Beispiel (1/8)



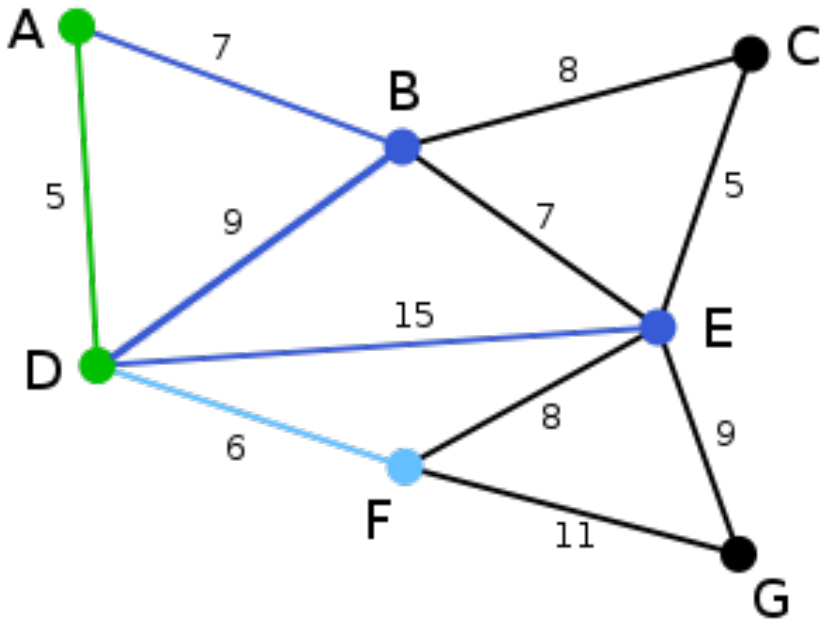
Aufgabe. Es sei obiger ungerichteter Graph mit Kantengewichten gegeben. Führen Sie den Algorithmus von Prim aus, mit Knoten *D* startend.

BEISPIEL (2/8)

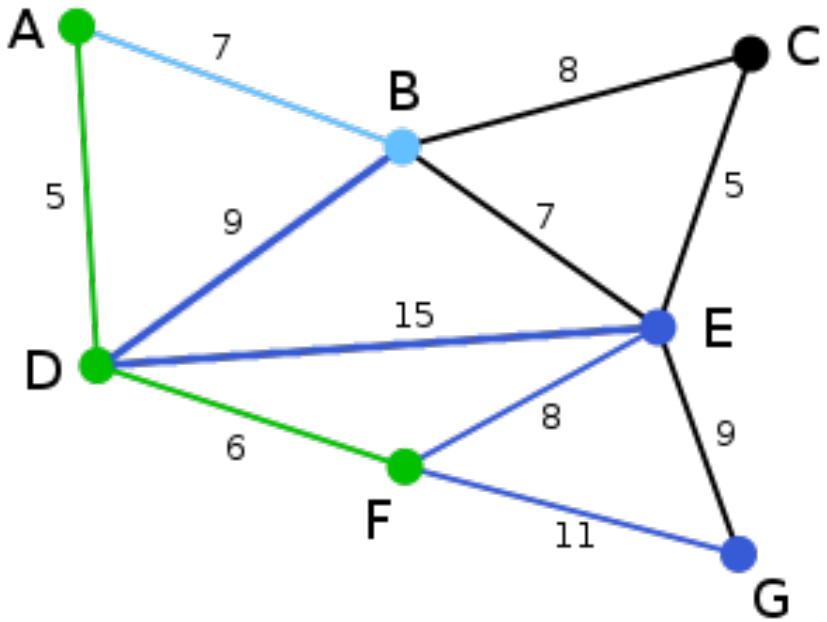


Wir prüfen alle zu D inzidenten Kanten (lila) und wählen die günstigste (hellblau). Wir fügen diese Kante und ihren Endknoten $\neq D$, also A, zum Baum hinzu.

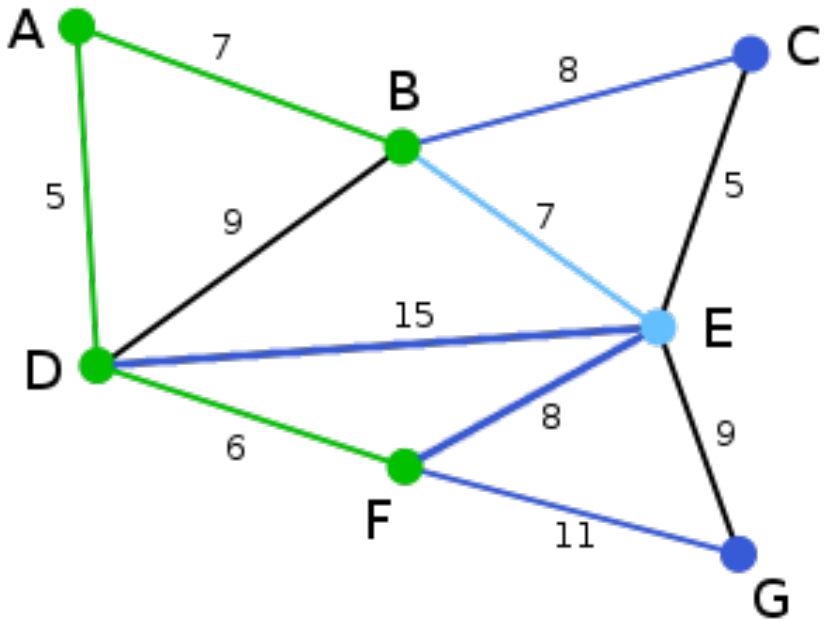
Beispiel (3/8)



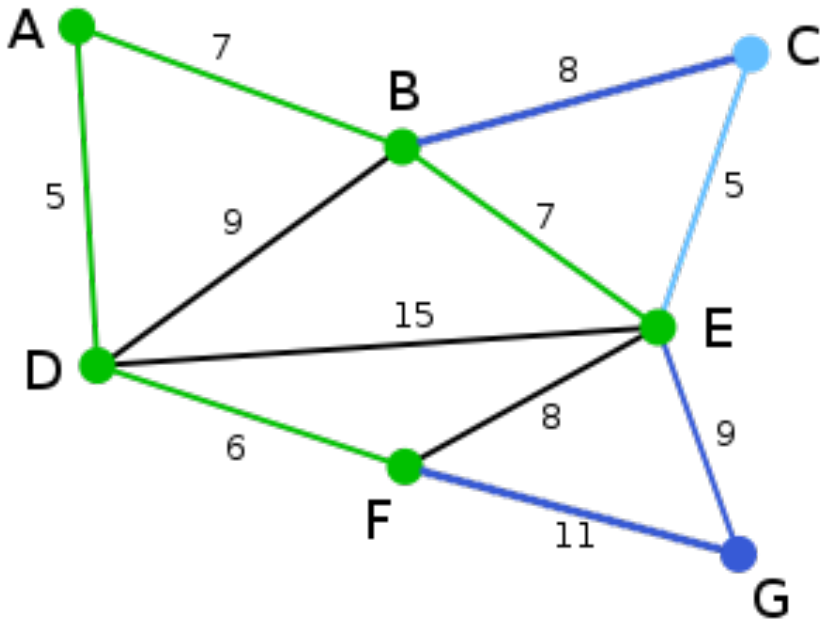
Beispiel (4/8)



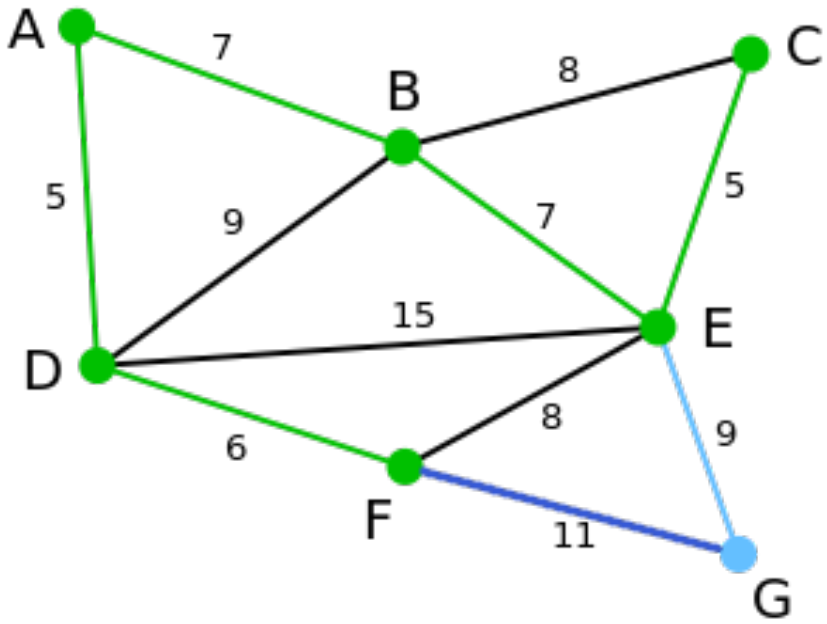
Beispiel (5/8)



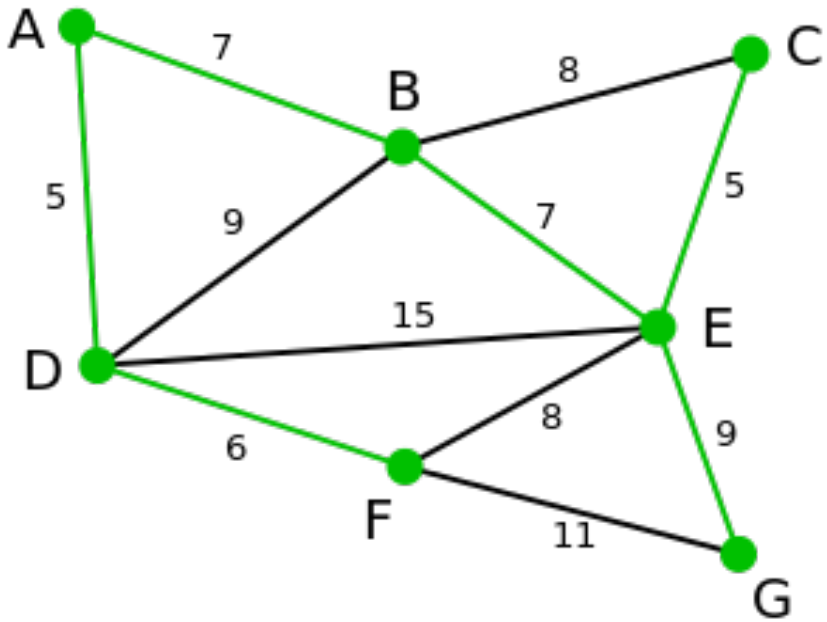
Beispiel (6/8)



Beispiel (7/8)

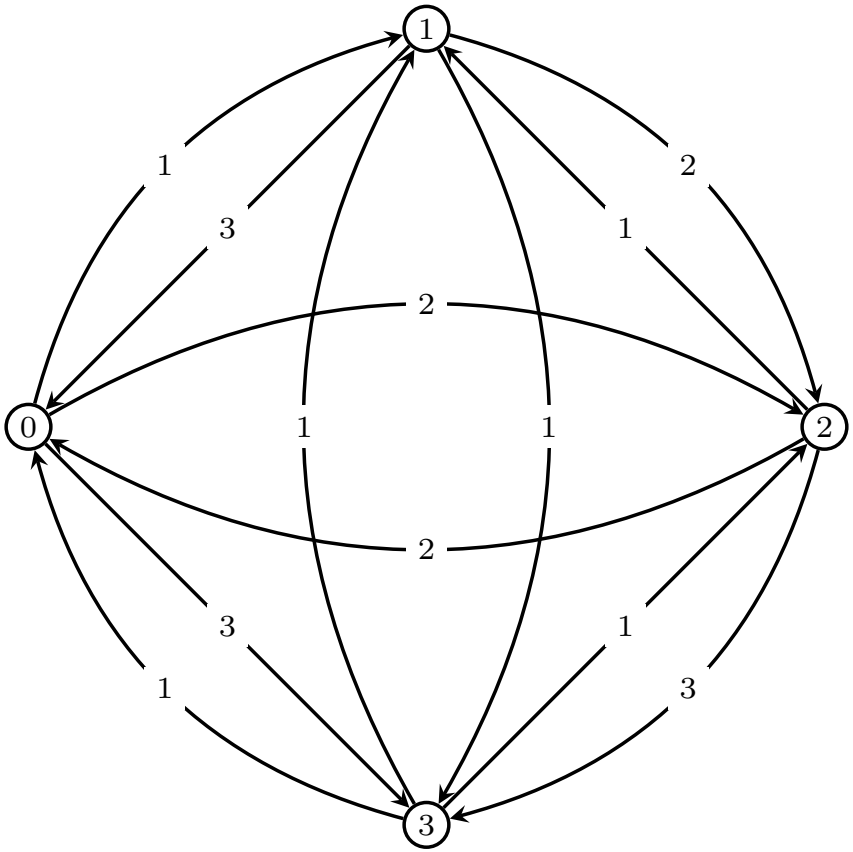


Beispiel (8/8)



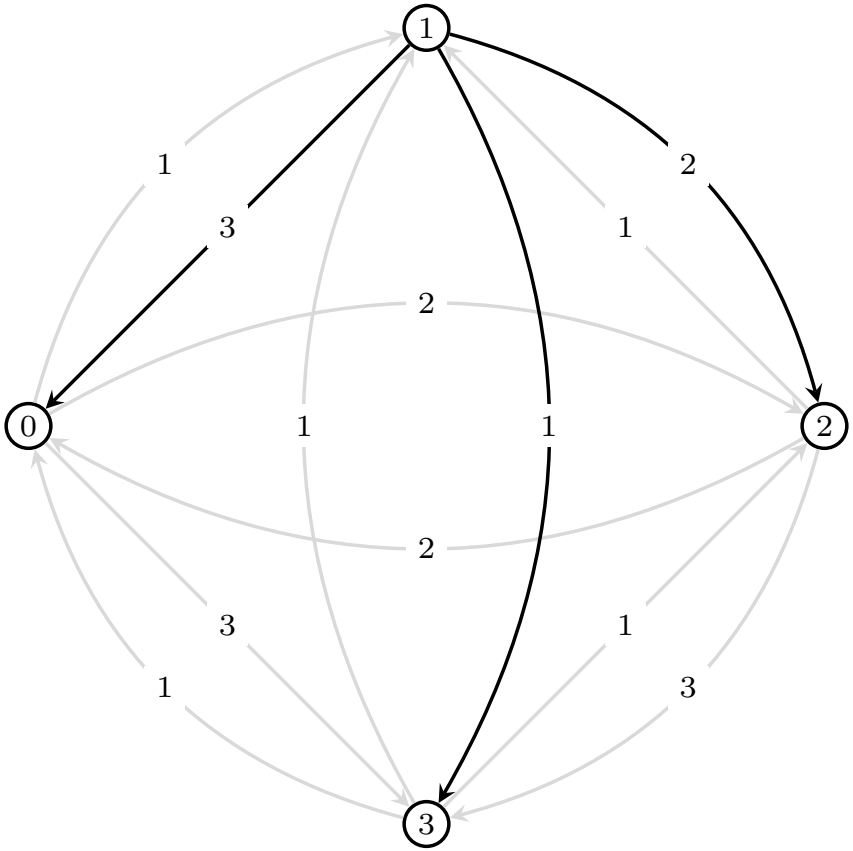
Den Algorithmus von Prim kann man auch für bestimmte Digraphen benutzen, und zwar stark zusammenhängende. Wir geben ein Beispiel, formalisieren den allgemeinen Sachverhalt hier jedoch nicht.

Beispiel (1/4)



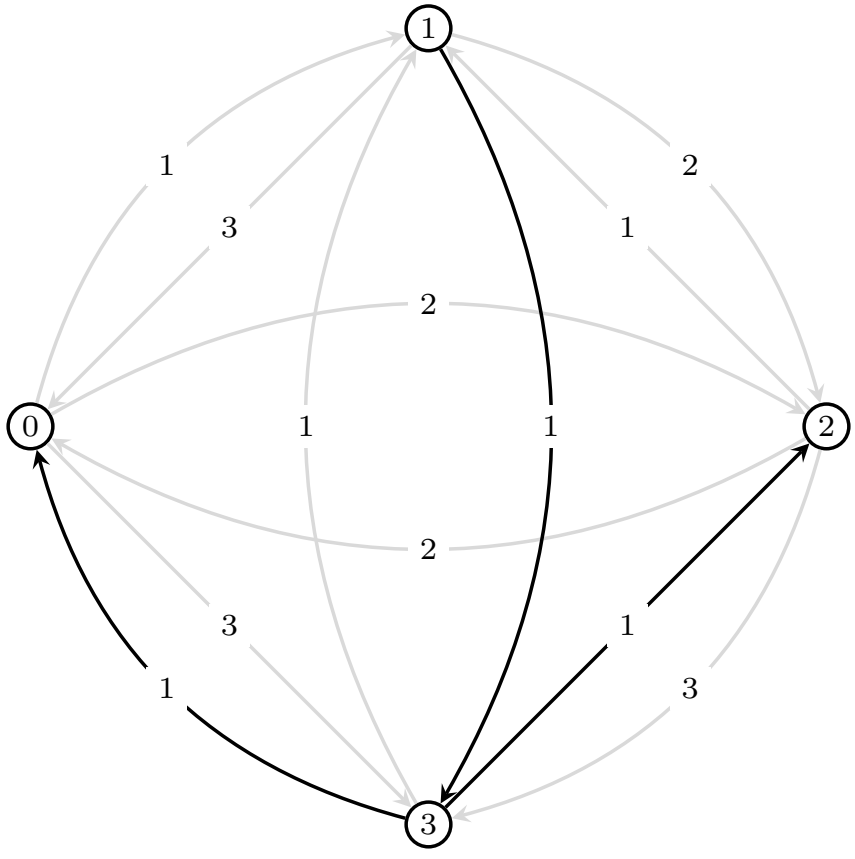
Stark zusammenhängender, gerichteter Graph mit Kantengewichten.
Startknoten = 1.

Beispiel (2/4)



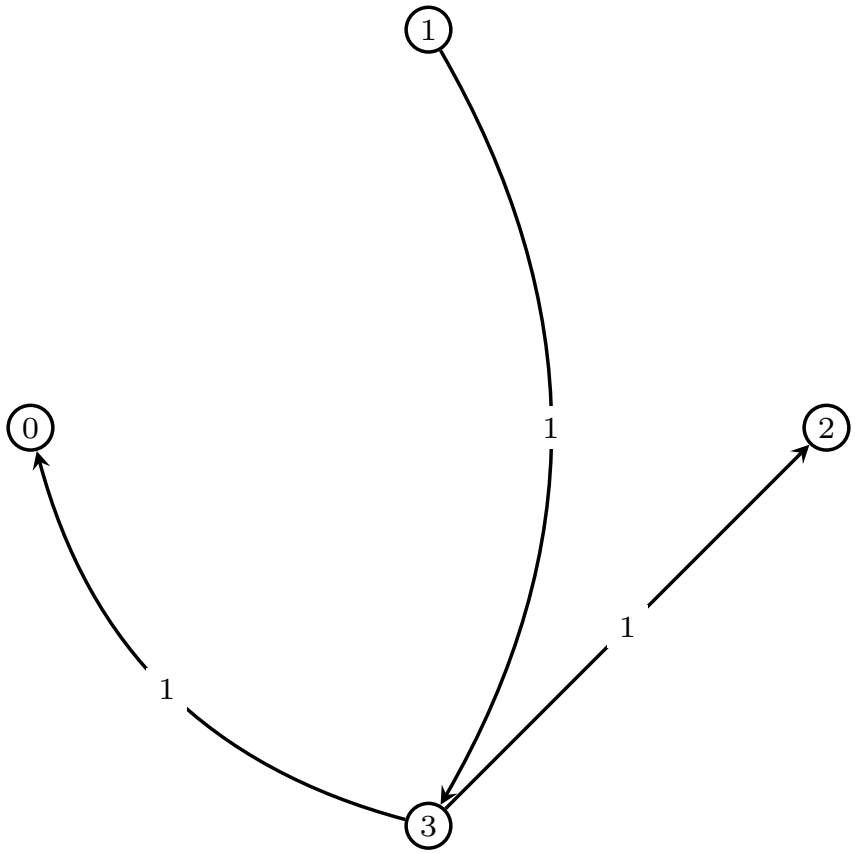
Wir prüfen die Kanten, die von Knoten 1 ausgehen. Diese haben hier Kosten 1, 2 und 3. Wir wählen die gewichtsminimale Kante aus und erhalten den Teilgraphen T, der aus den Knoten 1 und 3 besteht und die Kante $(1, 3)$ enthält. T ist ein Baum, aber noch nicht aufspannend.

Beispiel (3/4)



Wir betrachten von $V(T) = \{1, 3\}$ aus die günstigsten Kanten, die zu den noch nicht besuchten Knoten (0 und 2) führen. Diese fügen wir zum Baum hinzu.

Beispiel (4/4)



Ein minimaler aufspannender Baum.
Dies ist ein Wurzelbaum mit Wurzel 1.

Wohldefiniertheit

Satz. *Es sei $G = (V, E)$ ein ungerichteter, gewichteter und zusammenhängender Graph mit $n := |V|$ Knoten und Kantengewichten $w(e)$ für alle $e \in E$. Dann ist der Algorithmus von Prim wohldefiniert und der erzeugte Teilgraph ein minimaler Spannbaum von G.*

Den Beweis führt man ähnlich zum Wohldefiniertheitsbeweis des Algorithmus von Kruskal.

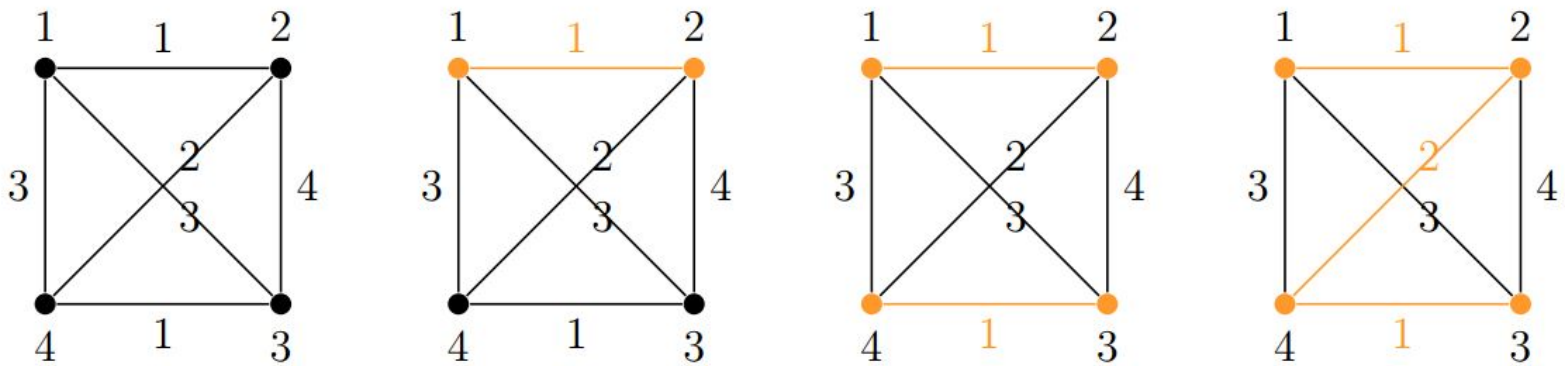
Bemerkungen

Der Algorithmus von Prim erzeugt ausgehend von einem Knoten eine immer größere Zusammenhangskomponente mit möglichst kleinem Gewicht. Bei den entstehenden Teilgraphen wird also der Fokus darauf gelegt, dass sie zusammenhängend sind. Dass sie gleichzeitig auch kreisfrei sind, folgt daraus, dass man die Zusammenhangskomponente in jedem Schritt vergrößern will.

Für einen Graphen $G = (V, E)$ ist die Laufzeit $O(|V|^2)$ bei einer naiven Implementierung. Die Effizienz des Algorithmus hängt von der Implementierung der Warteschlange ab. Bei Verwendung spezieller Datenstrukuren (in diesem Fall sogenannte *Fibonacci-Heaps*) ergibt sich eine optimale Laufzeit von $O(|E| + |V| \log |V|)$. Weitere Details im Buch von Krumke und Noltemeier.

Algorithmus von Kruskal (Wiederholung)

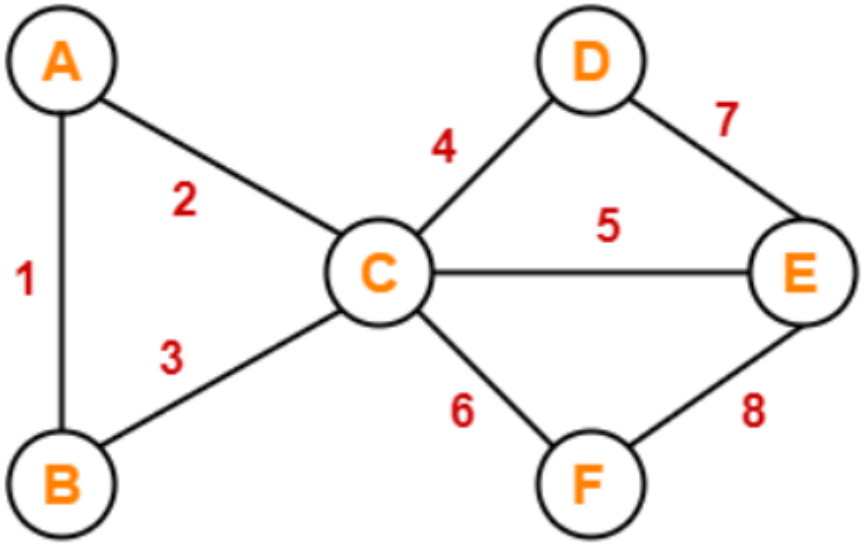
Man kann die Prioritäten auch umkehren: Kann man stattdessen eine Folge von Teilgraphen erzeugen, bei denen man immer eine Kante mit möglichst kleinem Gewicht dazu nimmt, solange der Graph kreisfrei bleibt? Die zwischendurch entstehenden Teilgraphen wären dann nicht unbedingt zusammenhängend. Der letzte jedoch muss zusammenhängend sein, da man sonst noch eine weitere Kante hinzunehmen könnte.



(a) Start (b) 1. Iteration (c) 2. Iteration (d) 3. Iteration

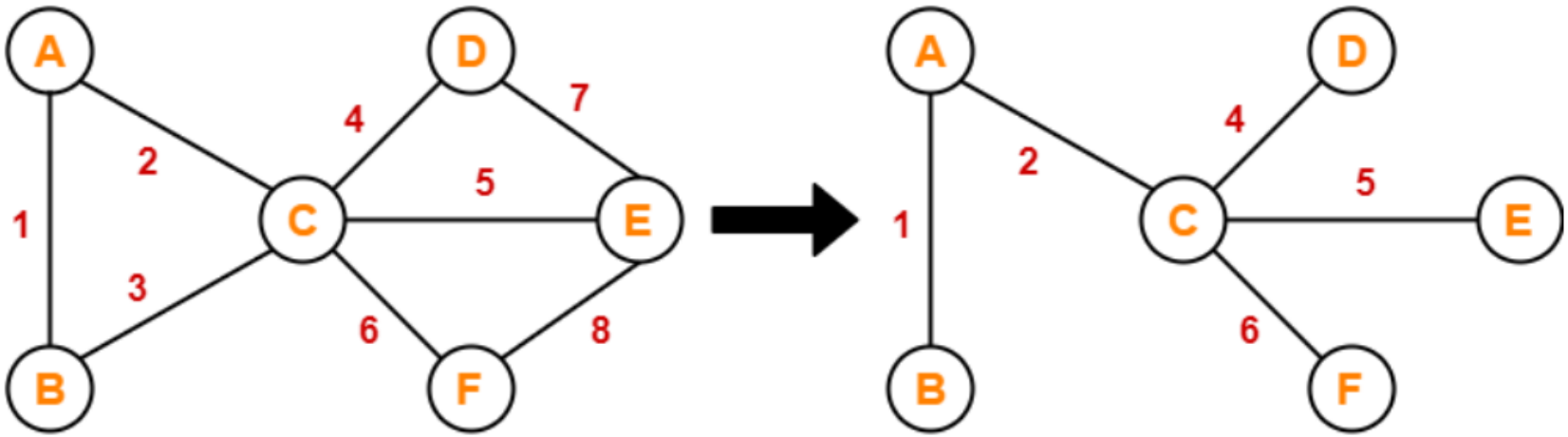
KRUSKAL VS. PRIM (1/3)

Aufgabe. Führen Sie den Algorithmus von Kruskal (Kap. 2) und den Algorithmus von Prim (beliebiger Startknoten) auf dem folgenden Graphen durch. Was fällt auf?



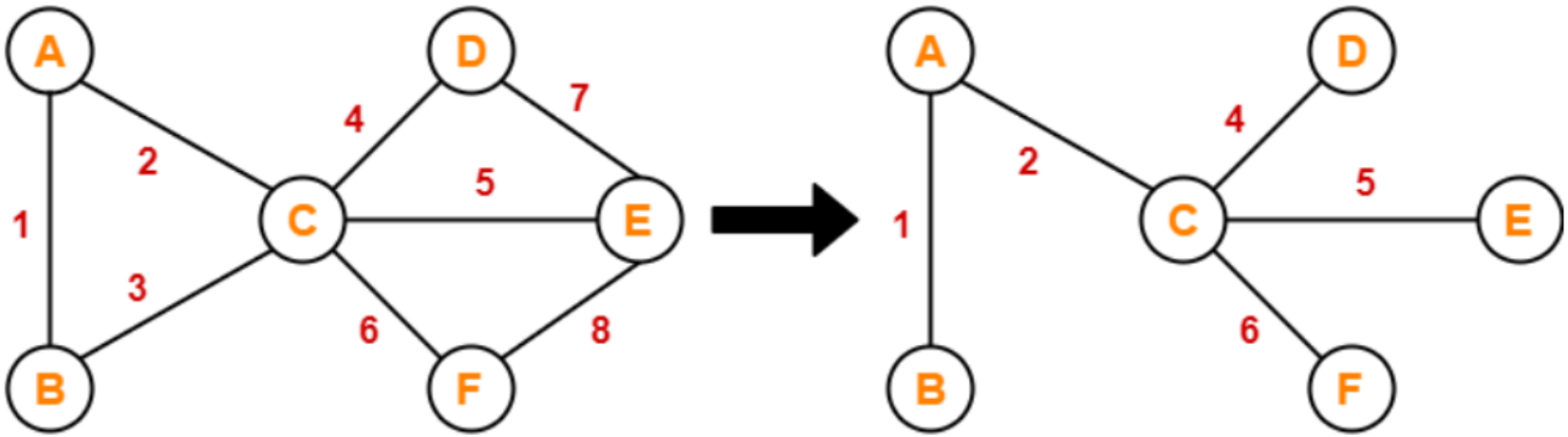
KRUSKAL VS. PRIM (2/3)

Sind die Kantengewichte paarweise verschieden, so liefern die Algorithmen von Kruskal und Prim denselben minimalen Spannbaum. **Frage:** Wieso?



KRUSKAL VS. PRIM (2/3)

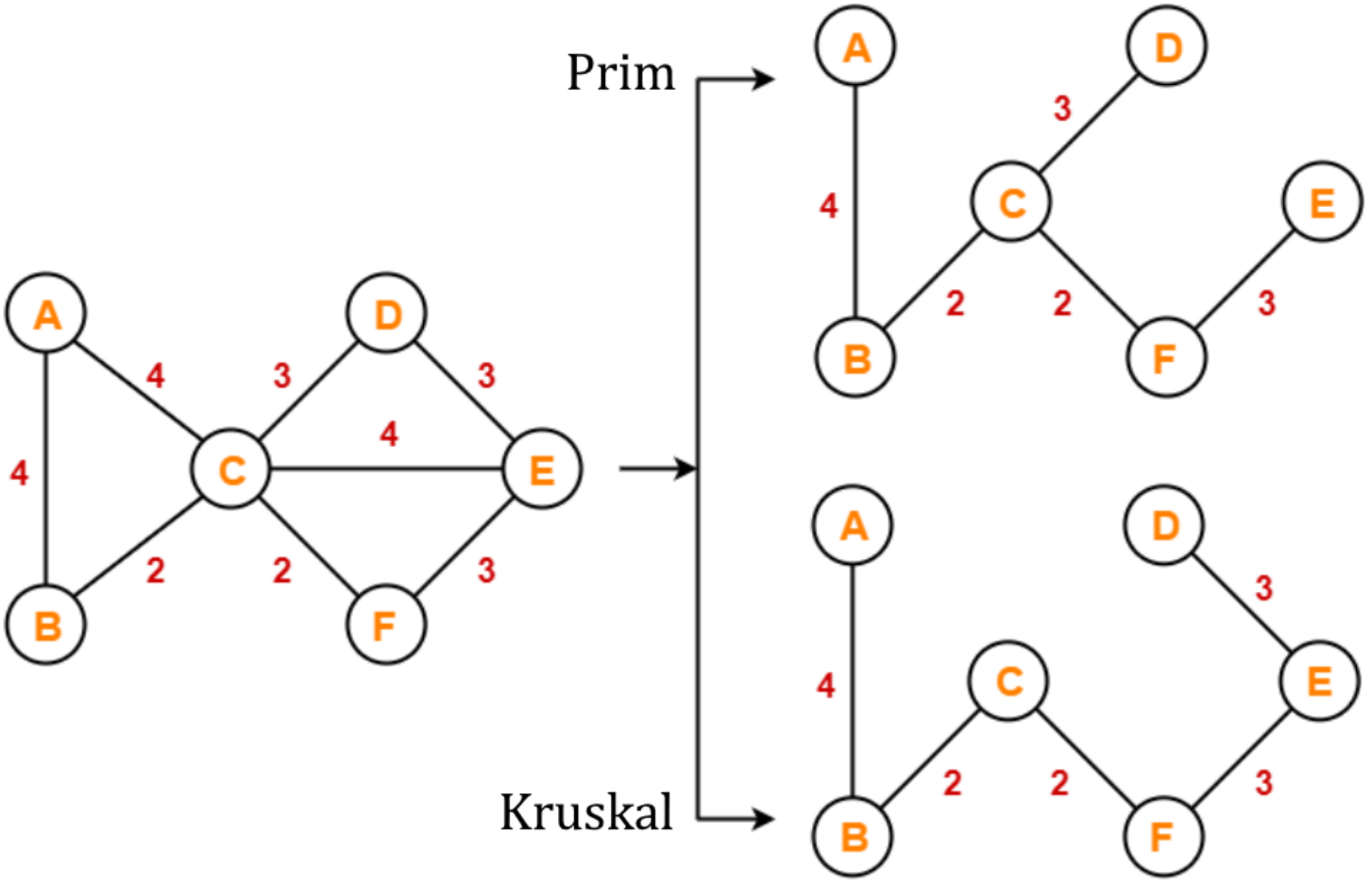
Sind die Kantengewichte paarweise verschieden, so liefern die Algorithmen von Kruskal und Prim denselben minimalen Spannbaum. **Frage:** Wieso?



Nehme an, die Alg. liefern versch. Spannbäume T, T'. Unter allen Kanten, die in genau einem dieser Bäume liegen, sei e_1 jene mit min. Gew.; o.B.d.A. $e_1 \in E(T) \setminus E(T')$. T' Baum $\Longrightarrow T' + e_1$ enthält Kreis C. T Baum $\Longrightarrow \exists e_2 \in E(C) \setminus E(T)$. $w(e_1) < w(e_2) \Longrightarrow w(T' - e_2 + e_1) < w(T')$, Widerspruch.

Kruskal vs. Prim (3/3)

Gibt es identische Kantengewichte, so liefern die Algorithmen von Kruskal und Prim (Startknoten: *A*) nicht zwingend denselben minimalen Spannbaum (die *Gewichte* dieser minimalen Spannbäume sind natürlich trotzdem identisch):



Algorithmus von Borůvka

- Findet einen minimalen Spannbaum für gewichtete, zusammenhängende Graphen $G = (V, E)$ deren **Kantengewichte paarweise verschieden** sind.
- Gilt als erster Algorithmus zum Auffinden minimaler Spannbäume in ungerichteten Graphen.
- Wurde 1926 von Otakar Borůvka beschrieben.
- Kann für einen Graphen $G = (V, E)$ in $O(|E| \log |V|)$ Zeit implementiert werden.
- In der Initialisierungsphase wird ein aufspannender Wald erzeugt, der alle Knoten enthält und keine Kanten.
- Für jede Zusammenhangskomponente wird die Kante mit minimalem Gewicht gewählt, um den Baum zu erzeugen.

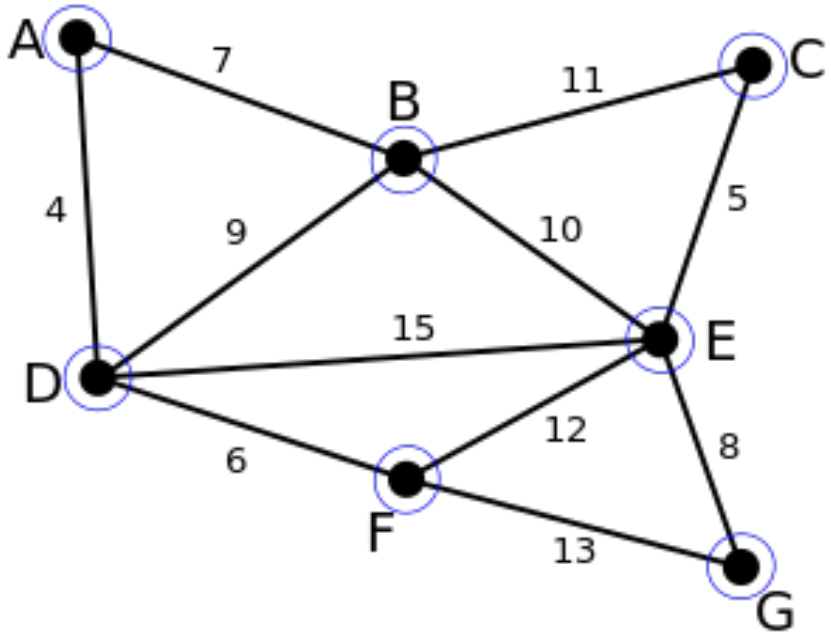
Algorithmus von Borůvka

Input: A weighted connected graph $G = (V, E)$ with weight function w. All the edge weights of G are distinct.

Output: A minimum spanning tree T of G.

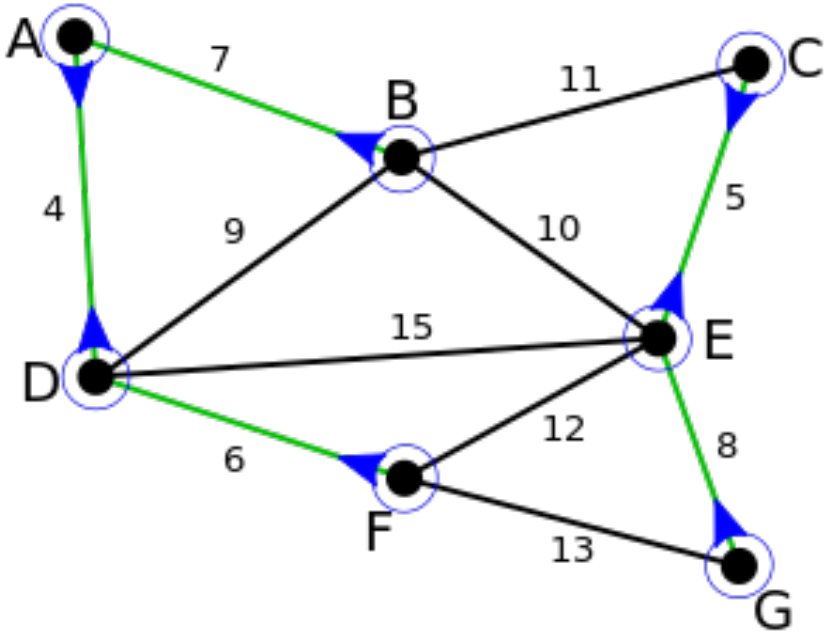
```
1 n ← |V|
2 T ← K̄_n
3 while |E(T)| < n − 1 do
4     for each component T′ of T do
5         e′ ← edge of minimum weight that leaves T′
6         E(T) ← E(T) ∪ e′
7 return T
```

Beispiel 1 (1/3)



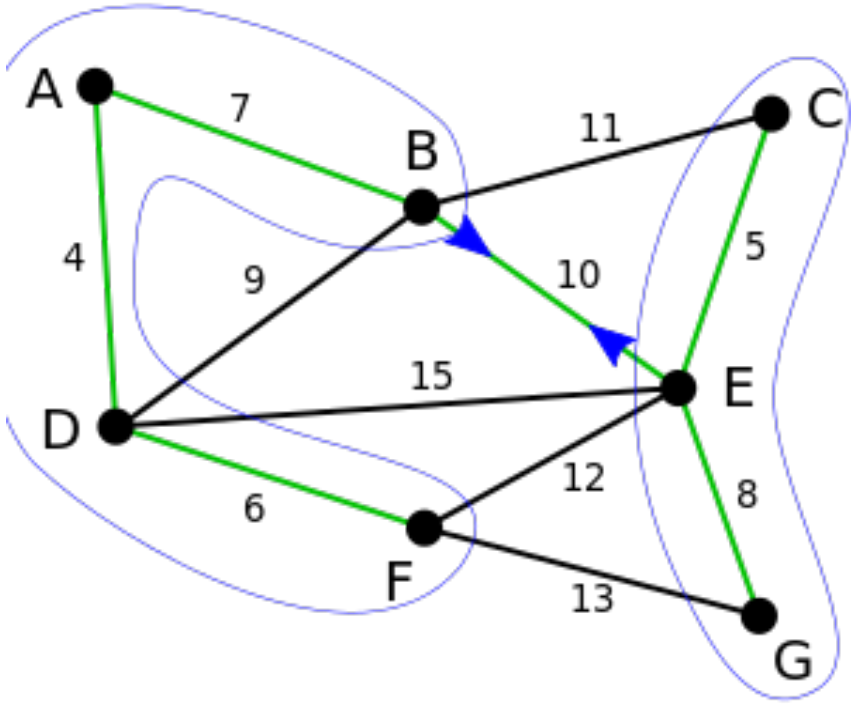
Aufgabe. Wenden Sie den Algorithmus von Borůvka auf diesen Graphen an.

Beispiel 1 (2/3)



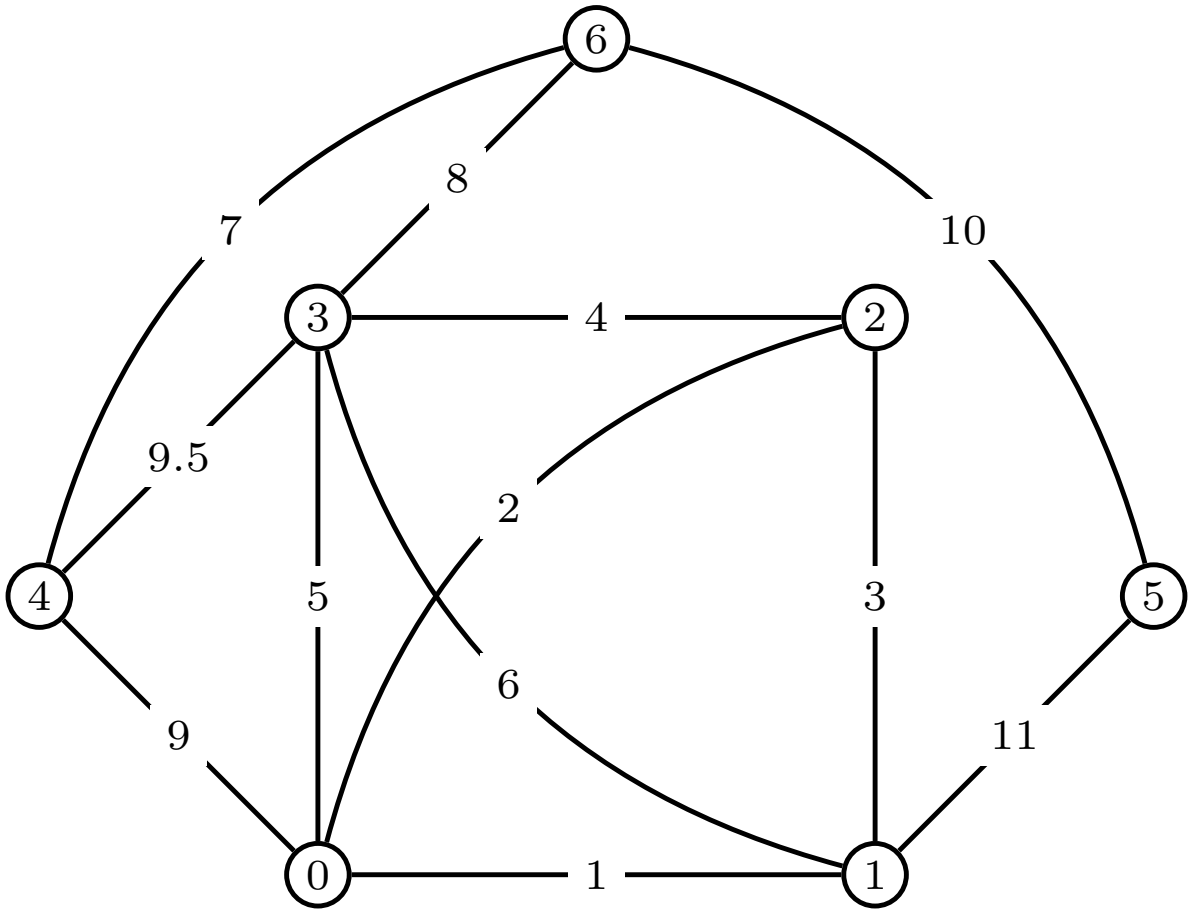
Kante minimalen Gewichts aus jeder Zusammenhangskomponente wird hinzugefügt. Manche Kanten kommen zweimal vor (AD, CE). Es bleiben zwei Komponenten übrig.

Beispiel 1 (3/3)

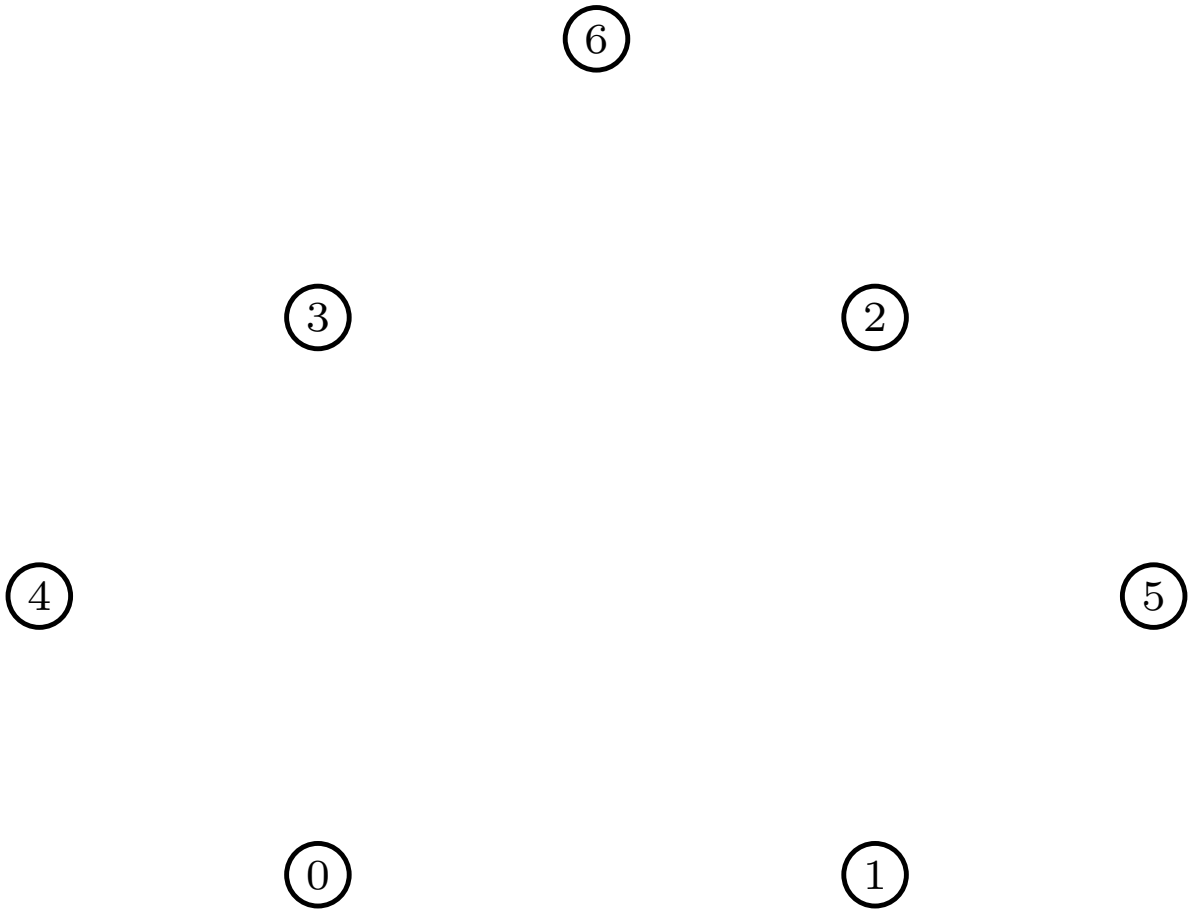


Kante minimalen Gewichts aus jeder der zwei verbleibenden Zusammenhangskomponenten wird hinzugefügt. Dies ist hier die Kante BE. Es bleibt eine Komponente übrig und wir sind fertig. (Die Kante BD wird nicht betrachtet, da ihre Endpunkte in derselben Komponente liegen.)

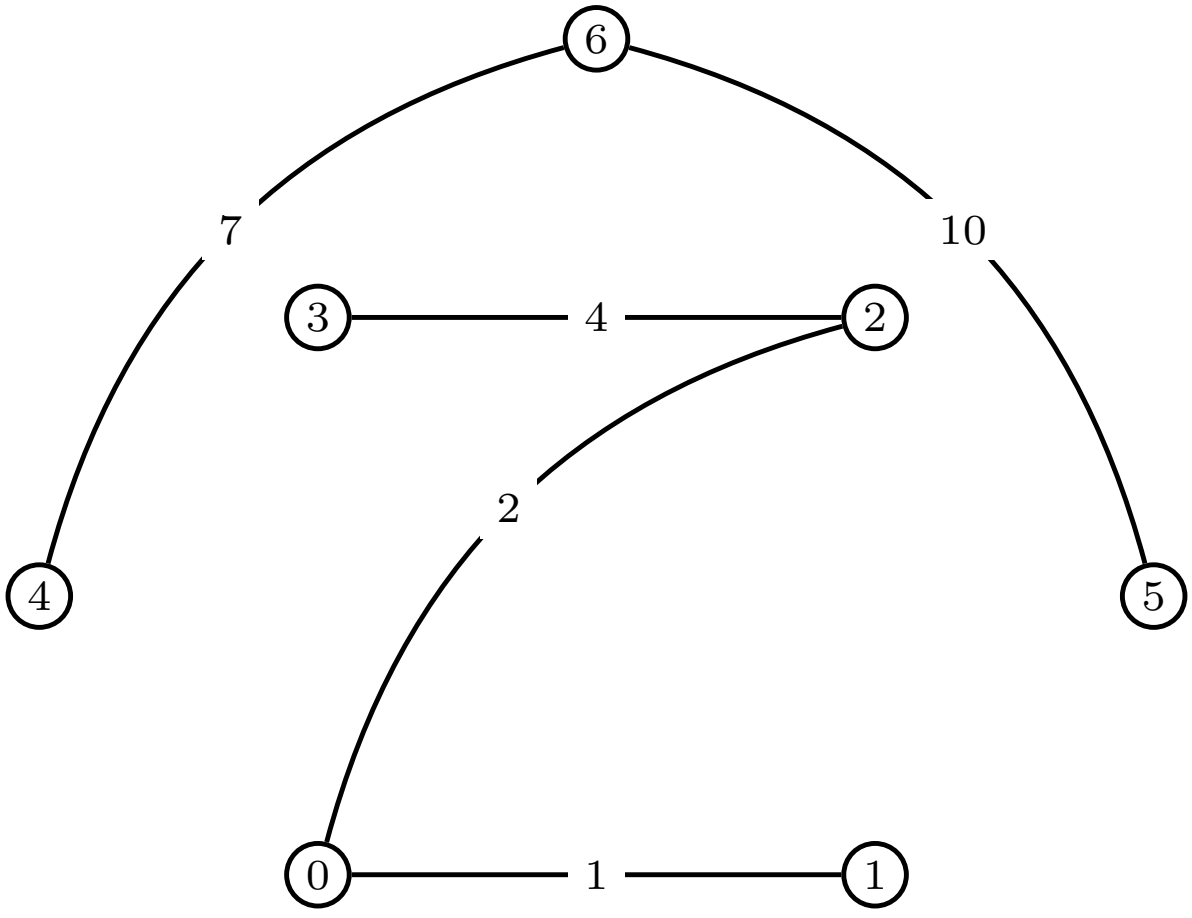
Beispiel 2 (1 / 4)



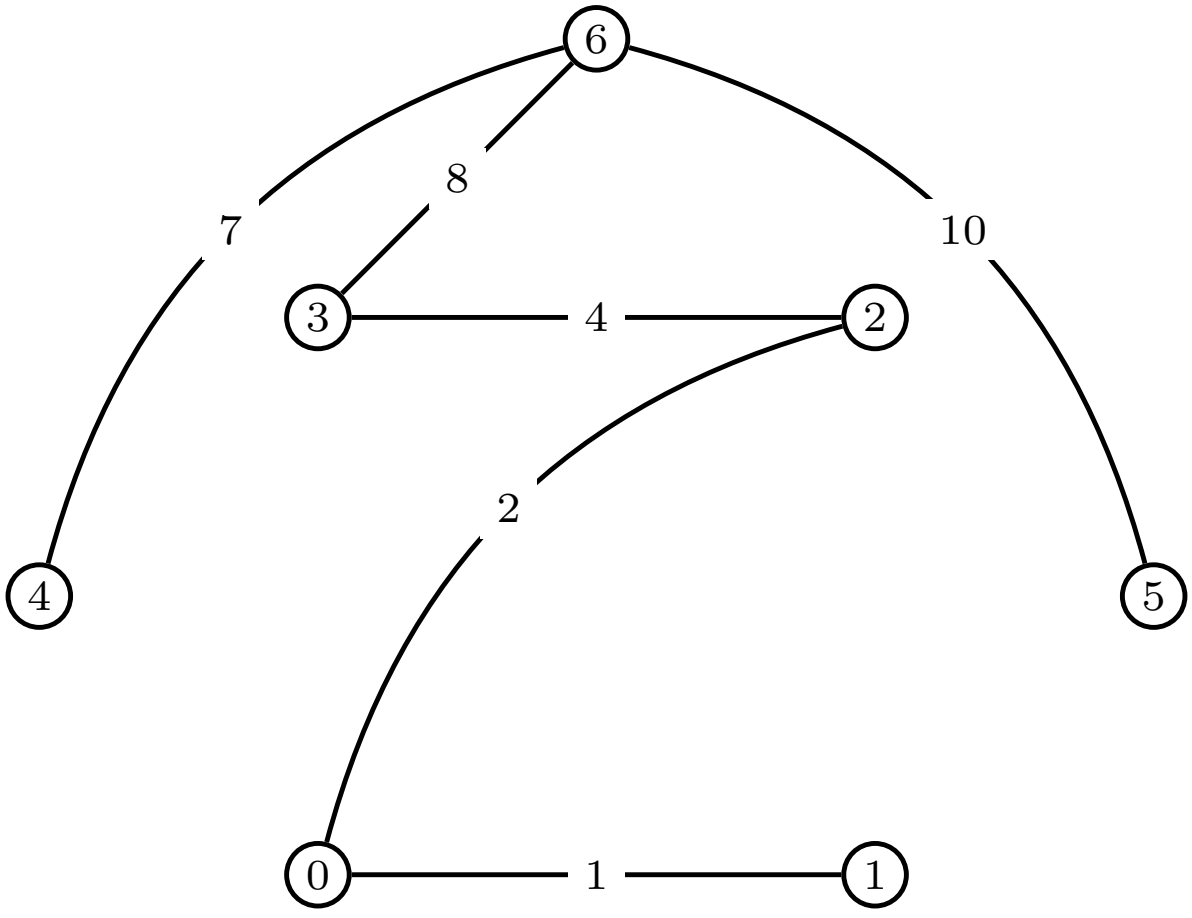
Beispiel 2 (2/4)



Beispiel 2 (3/4)



Beispiel 2 (4/4)



Zusammenfassung

- Die Algorithmen von Prim, Kruskal und Borůvka sind sogenannte Greedy-Verfahren.
- Als Greedy-Verfahren bezeichnet man einen Algorithmus, der versucht, ein Problem zu lösen, indem er in jeder Iteration das tut, was gerade (lokal – zeitlich gesehen) am besten erscheint.
- Dies muss global gesehen nicht die beste Entscheidung sein und führt auch nicht immer zum Erfolg, wie man an dem folgenden Beispiel sieht:

- Ein Greedy-Verfahren, um von einem Berggipfel herunter zu finden, wäre zum Beispiel immer in die Richtung des steilsten Abstiegs zu gehen.
- Wenn man Glück hat, kommt man so am Fuße des Berges an. Hat man hingegen Pech, so landet man in einer Senke, aus der man mit diesem Ansatz nicht mehr herausfindet.
- Dass Greedy-Verfahren zur Bestimmung minimaler Spannbäume funktionieren, liegt, etwas vereinfacht, daran, dass man einen minimalen Spannbaum eines Graphen aus minimalen Spannbäumen von Teilgraphen zusammenbauen kann.

Berechnung kürzester Wege

- Es sei $G = (V, E)$ ein Graph mit gewichteten Kanten $c(e) \geq 0$ für alle $e \in E$.
- Für einen Weg P von $u \in V$ nach $v \in V$ ist dessen *Länge* $c(P)$ die Summe aller Gewichte aller Kanten von P, sprich

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

- Es sei $d(u, v)$ der kleinste Wert $c(P)$ unter allen Wegen P von u nach v.
- Ein Weg P mit $c(P) = d(u, v)$ heißt *kürzester Weg* von u nach v.
- Diese Definitionen können auch für gewichtete Graphen mit negativen Gewichten übertragen werden.
- Vorsicht: in anderen Kontexten kann auch $|E(P)|$ die Länge eines Weges bezeichnen. Dies ist dasselbe wie die obige Definition, wenn man auf jeder Kante Gewicht 1 annimmt.

Algorithmus von Dijkstra

- Findet den kürzesten Weg in positiv gewichteten Graphen.
- Verallgemeinerung der Breitensuche.
- Ist ein Greedy Algorithmus.
- Hat Ähnlichkeit zum Algorithmus von Prim.
- Wir schauen uns nun den Algorithmus von Dijkstra für gerichtete Graphen an.

- Es sei $G = (V, A)$ ein gerichteter Graph mit nichtnegativen Kantengewichten und $s \in V$.
- Bei diesem Algorithmus bauen wir den sogenannten *Vorgängergraphen* $G_\pi = (V_\pi, A_\pi)$ auf, der wie folgt definiert ist:

$$V_\pi := \{v \in V : \pi[v] \neq \texttt{nil}\} \cup \{s\}$$

und

$$A_\pi := \{(\pi[v], v) \in A : v \in V_\pi \ \wedge \ v \neq s\},$$

wobei die Funktion π zu einem Knoten den entsprechenden Vorgängerknoten speichert und `nil` für einen leeren Eintrag steht.

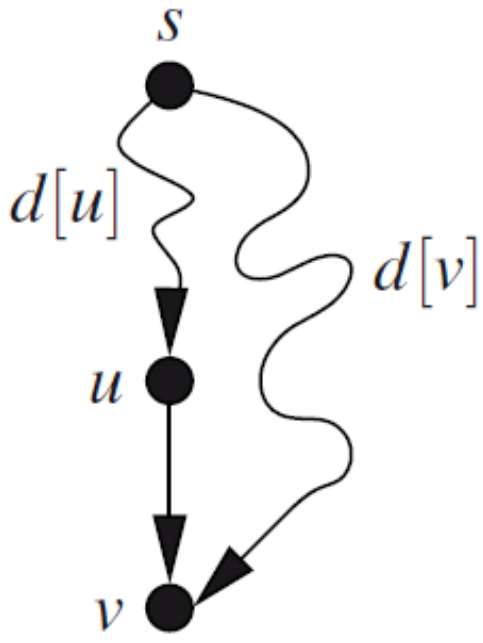
- G_π ist ein Baum kürzester Wege bezüglich s.
- (Vorgängergraph = predecessor graph)

INIT(G, s)

```
1 for all v ∈ V do
2     d[v] := +∞
3     π[v] := nil
4 d[s] := 0
```

TEST(u, v)

```
1 if d[v] > d[u] + c(u, v) then
2     d[v] := d[u] + c(u, v)
3     π[v] := u
```



Testschritt TEST(u, v)

Initialisierung und Test: Algorithmus, der für eine gerichtete Kante $(u, v) \in A$ prüft, ob diese benutzt werden kann, um einen kürzeren Weg von s nach v zu finden als der bisher bekannte mit Länge $d[v]$.

- Der Algorithmus von Dijkstra arbeitet mit einer *Wellenfront*-Strategie:
- im Verfahren halten wir eine Menge PERM $\subset V$ von **permanent markierten** Knoten, d.h. Knoten v, für die bereits $d[v] = \text{dist}_c(s, v)$ gilt, wobei $\text{dist}_c(s, v)$ der kürzeste Abstand zwischen s und v bezüglich der Gewichtsfunktion c ist.
- Die Werte $d[v]$, wobei stets $v \in V$, sind obere Schranken für $\text{dist}_c(s, v)$, die im Laufe des Algorithmus angepasst werden.
- Anfangs ist PERM $= \emptyset$.
- In jeder Iteration entfernen wir einen Knoten u aus $Q := V \setminus$ PERM mit minimalem Schlüsselwert $d[u]$ und fügen u PERM hinzu.
- Anschließend testen wir alle gerichteten Kanten $(u, v) \in A$ mit TEST(u, v).

Algorithmus von Dijkstra

Dijkstra(G, c, s)

Input: Ein gerichteter Graph $G = (V, A)$ in Adjazenzlistendarstellung, eine nichtnegative Gewichtsfunktion $c\colon A \to \mathbb{R}_+$ und ein Knoten $s \in V$ mit allen Knoten aus s erreichbar

Output: Für alle $v \in V$ die Distanz $d[v] = \mathrm{dist}_c(s, v)$ sowie ein Baum G_π kürzester Wege von s aus

```
1 Init(G, s)
2 PERM := ∅        { PERM ist die Menge der »permanent markierten« Knoten }
3 while PERM ≠ V do
4    Wähle u ∈ Q := V \ PERM mit minimalem Schlüsselwert d[u].
5    PERM := PERM ∪ {u}
6    for all v ∈ Adj[u] \ PERM do
7       Test(u, v)
8 return d[] und G_π    { G_π wird durch die Vorgängerzeiger π »aufgespannt« }
```

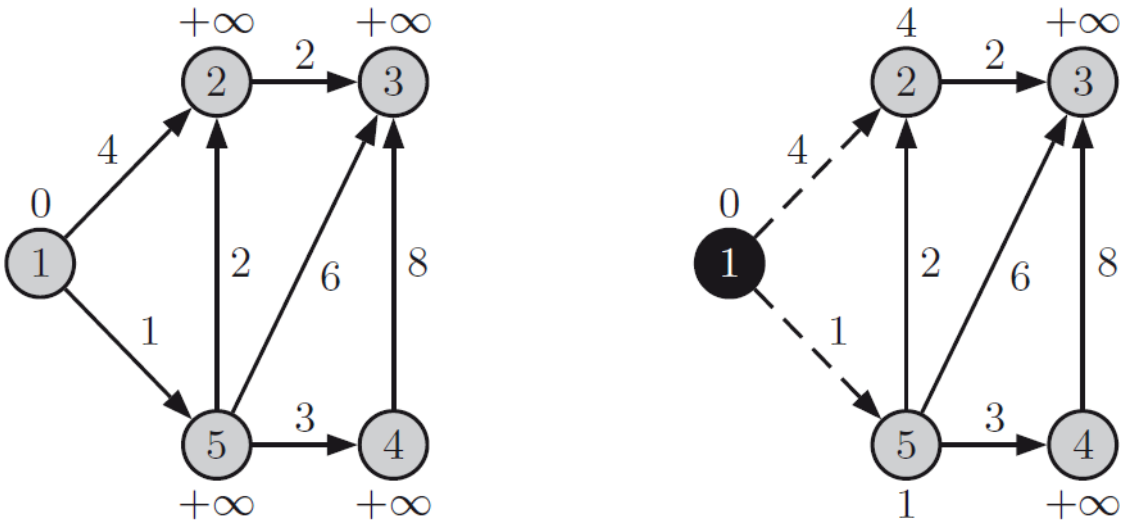
Man beachte: Gewichte dürfen nicht negativ sein!

Satz. *Bei Abbruch des Algorithmus von Dijkstra gilt* $d[v] = \mathrm{dist}_c(s, v)$ *für alle* $v \in V$. *Der Graph* G_π *ist ein Baum kürzester Wege bezüglich* s.

Wir beschäftigen uns im nächsten Kapitel mit dem Beweis.

Beispiel 1 (1/3)

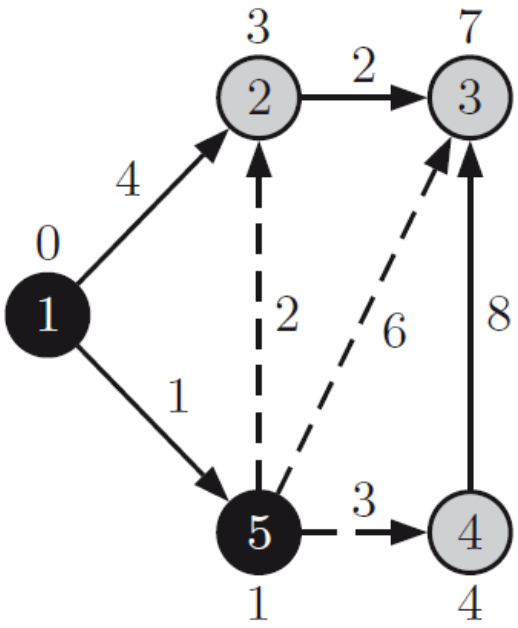
Die Zahlen neben den Knoten bezeichnen die Distanzmarken d, schwarze Knoten sind die permanent markierten Knoten PERM. Die in der aktuellen Iteration mittels TEST überprüften Kanten sind gestrichelt.

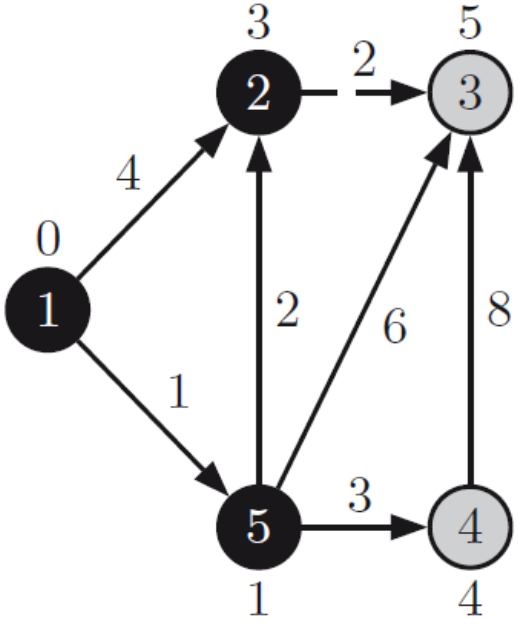


Links: Initialisierung. Startknoten = 1. PERM $= \emptyset$.

Rechts: Knoten 1 wird als Minimum aus der Prioritätsschlange entfernt, mit der Q ($:= V \setminus$ PERM) verwaltet wird. Für alle Nachfolger werden die Distanzmarken d mittels TEST korrigiert.

Beispiel 1 (2/3)

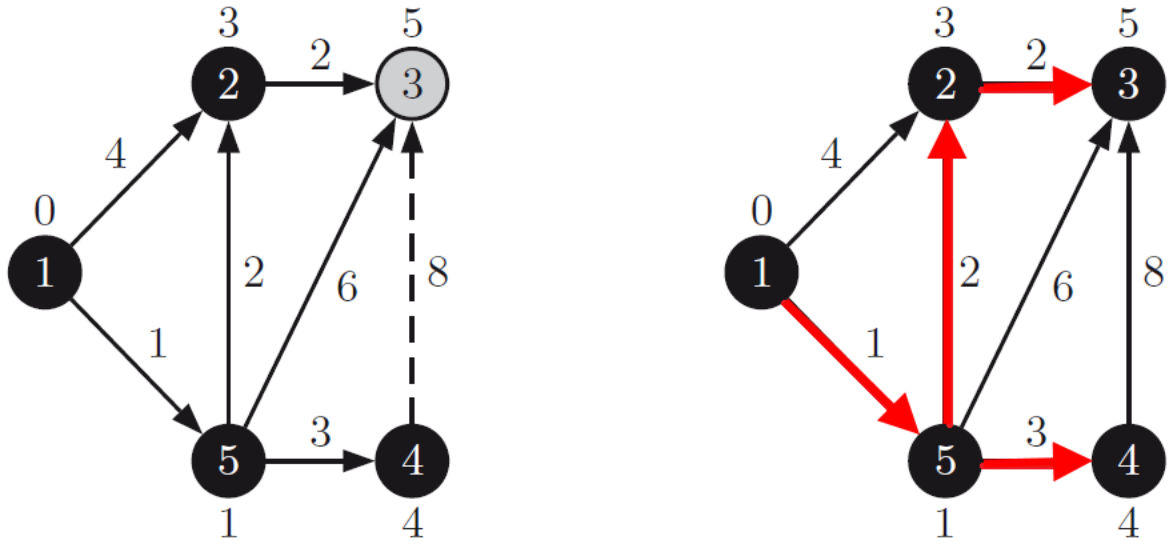




Links: Knoten 5 wird als Minimum aus der Prioritätsschlange entfernt und PERM hinzugefügt. Dabei wird unter anderem die Distanzmarke des Knotens 2 von 4 auf 3 verringert. Es gilt $\pi(5) = 1$.

Rechts: Knoten 2 wird aus der Schlange entfernt. Es gilt $\pi(2) = 5$.

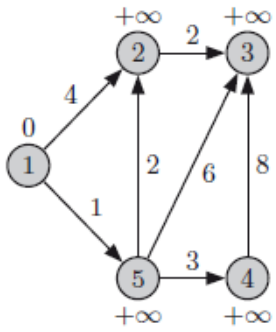
Beispiel 1 (3/3)



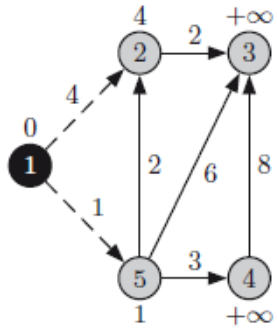
Links: Knoten 4 wird aus der Schlange entfernt. Distanzmarke des Knotens 3 wird dabei nicht verringert, da $4 + 8 > 5$. Es gilt $\pi(4) = 5$.

Rechts: Knoten 3 wird als Minimum aus der Prioritätsschlange entfernt. Danach terminiert der Algorithmus, da PERM $= V$ (und die Schlange leer ist). Es gilt $\pi(3) = 2$. Vorgängergraph G_π in rot.

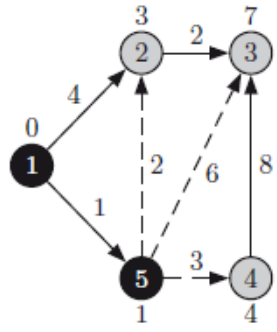
Beispiel 1 stammt aus dem Buch von Krumke und Noltemeier, Seite 178 (siehe nächste Folie). Man betrachte diese Seite 178 genauer: es hat sich ein (mathematischer) Tippfehler eingeschlichen. Können Sie diesen finden?



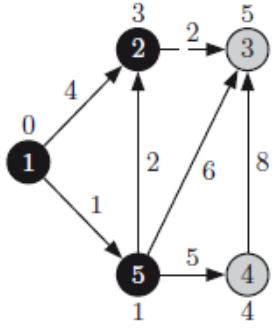
a: Initialisierung, die Startecke ist die Ecke 1. Wir haben PERM = ∅.



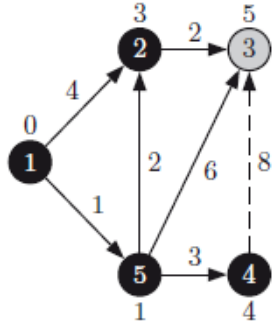
b: Die Ecke 1 wird als Minimum aus der Prioritätsschlange entfernt, mit der $Q := V \setminus \text{PERM}$ verwaltet wird. Für alle Nachfolger werden die Distanzmarken d mittels TEST korrigiert.



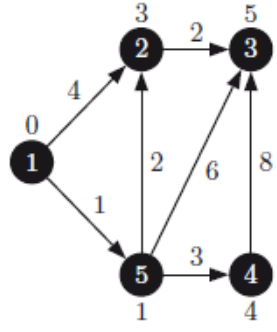
c: Die Ecke 5 wird als Minimum aus der Prioritätsschlange entfernt und PERM hinzugefügt. Dabei wird unter anderem die Distanzmarke der Ecke 2 von 4 auf 3 verringert.



d: Die Ecke 2 wird aus der Schlange entfernt.

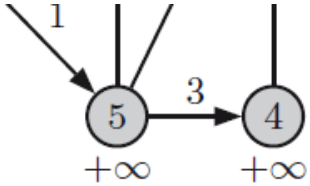


e: Die Ecke 4 wird aus der Schlange entfernt. Die Distanzmarke der Ecke 3 wird dabei nicht verringert, da $4 + 8 > 5$ gilt.

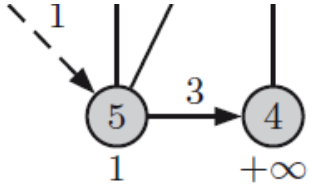


f: Die Ecke 3 wird als Minimum aus der Prioritätsschlange entfernt. Danach terminiert der Algorithmus, da PERM $= V$ (und die Schlange leer) ist.

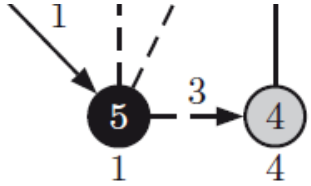
Bild 8.1: Arbeitsweise des Dijkstra-Algorithmus. Die Zahlen an den Ecken bezeichnen die Distanzmarken d, die schwarz gefärbten Ecken sind die permanent markierten Ecken PERM. Die in der aktuellen Iteration mittels TEST überprüften Pfeile sind gestrichelt hervorgehoben.



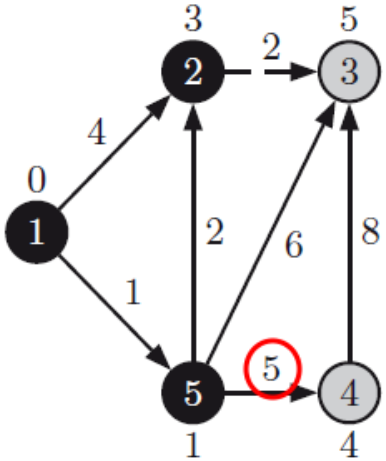
a: Initialisierung, die Startecke ist die Ecke 1. Wir haben PERM = ∅.



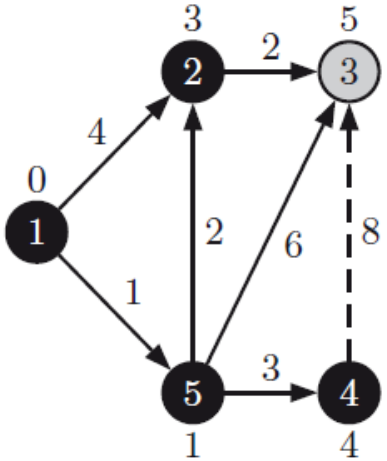
b: Die Ecke 1 wird als Minimum aus der Prioritätsschlange entfernt, mit der $Q := V \setminus \text{PERM}$ verwaltet wird. Für alle Nachfolger werden die Distanzmarken d mittels TEST korrigiert.



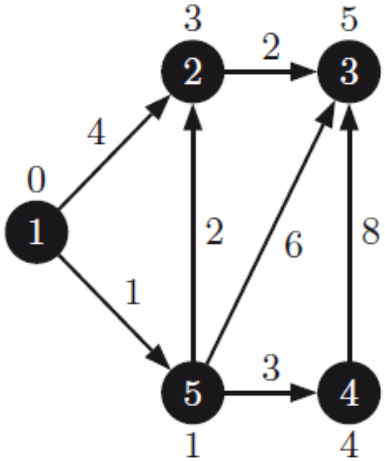
c: Die Ecke 5 wird als Minimum aus der Prioritätsschlange entfernt und PERM hinzugefügt. Dabei wird unter anderem die Distanzmarke der Ecke 2 von 4 auf 3 verringert.



d: Die Ecke 2 wird aus der Schlange entfernt.



e: Die Ecke 4 wird aus der Schlange entfernt. Die Distanzmarke der Ecke 3 wird dabei nicht verringert, da $4 + 8 > 5$ gilt.

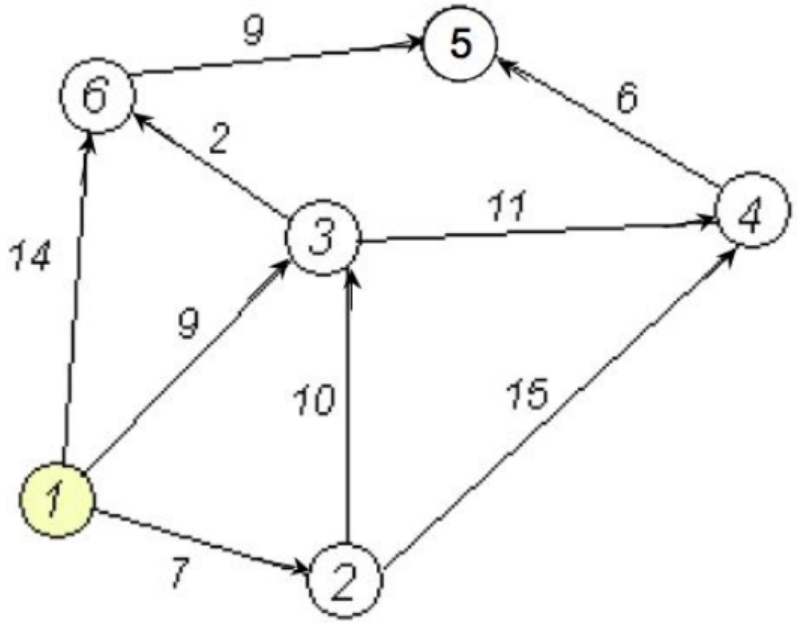


f: Die Ecke 3 wird als Minimum aus der Prioritätsschlange entfernt. Danach terminiert der Algorithmus, da PERM = V und

Frage. In welchem Maße hat das Gewicht der Kante $(5, 4)$ Einfluss auf die Struktur des Graphen G_π bei Anwendung des Algorithmus von Dijkstra?

Keinerlei Einfluss: Es gibt keine andere Art und Weise vom Knoten 0 aus den Knoten 4 zu erreichen. Egal welches Gewicht die Kante $(5, 4)$ hätte, sie würde in G_π liegen.

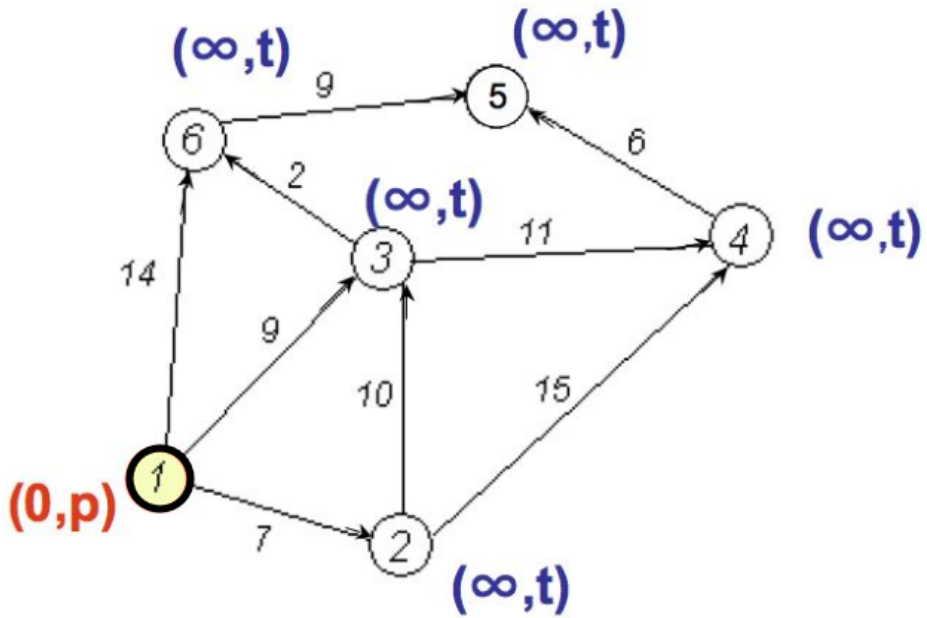
BEISPIEL 2 (1/7)



Wir starten in Knoten 1 und suchen den Baum T mit Wurzel 1 und den kürzesten Wegen zu allen anderen Knoten im Graphen.

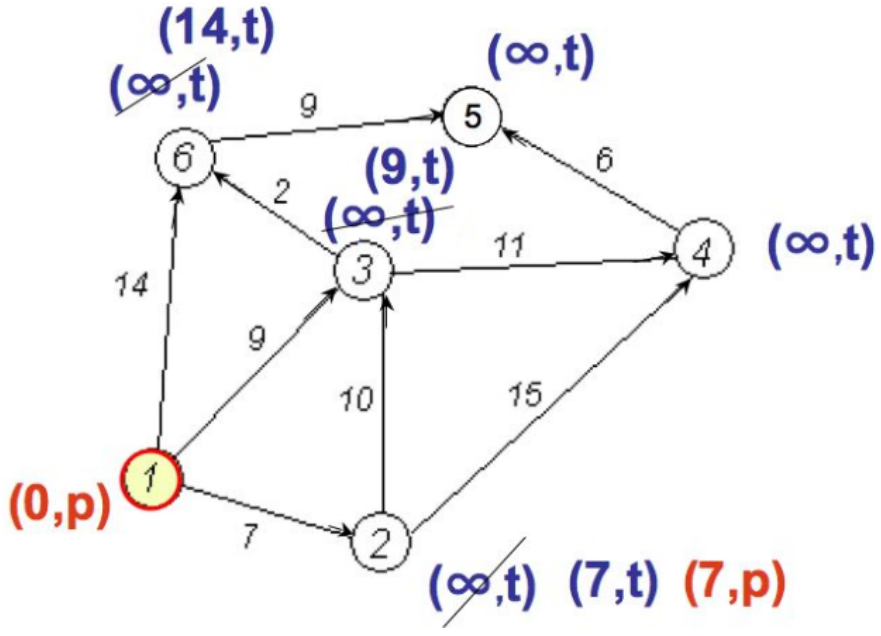
Aufgabe. Finden Sie T mithilfe des Dijkstra-Algorithmus.

Beispiel 2 (2/7)



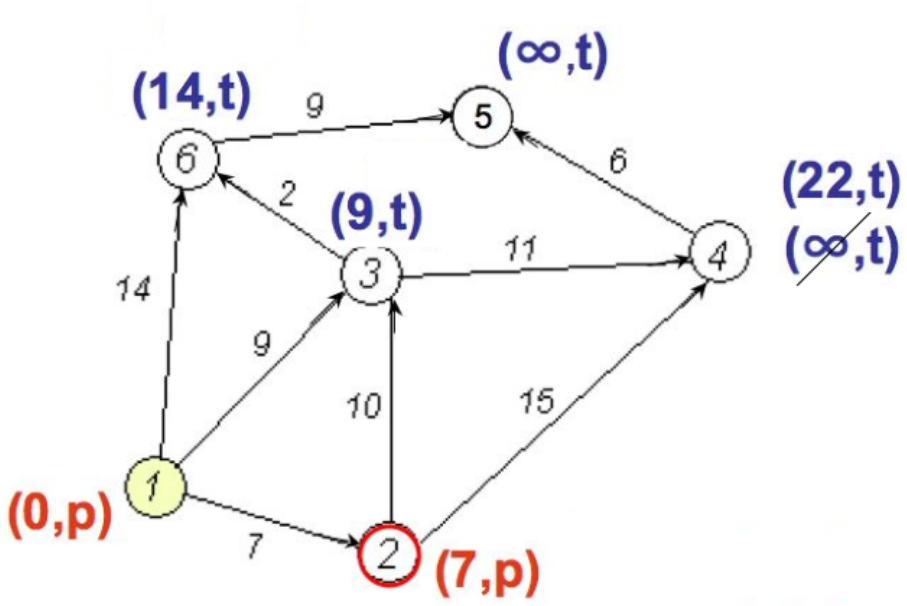
Zu Beginn sind die Marken aller Knoten bis auf die Wurzel (gelb unterlegt) gleich ∞. **p** steht für permanent (diese Marke wird nicht mehr geändert) und **t** für temporär.

BEISPIEL 2 (3/7)



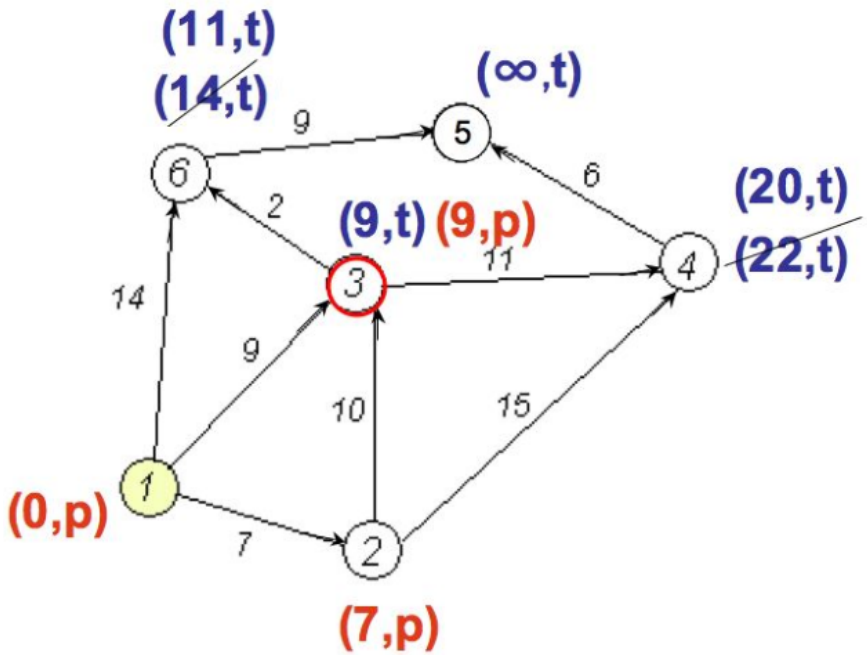
Wir umranden in rot den aktuell betrachteten Knoten (hier Knoten 1). In diesem Schritt aktualisieren wir die Entfernungen zu den Nachbarn des Startknotens 1. Knoten 3 und 6 bleiben temporär markiert, Knoten 2 jedoch wird nun permanent markiert, da es keinen kürzeren Weg von 1 zu 2 geben kann.

Beispiel 2 (4/7)



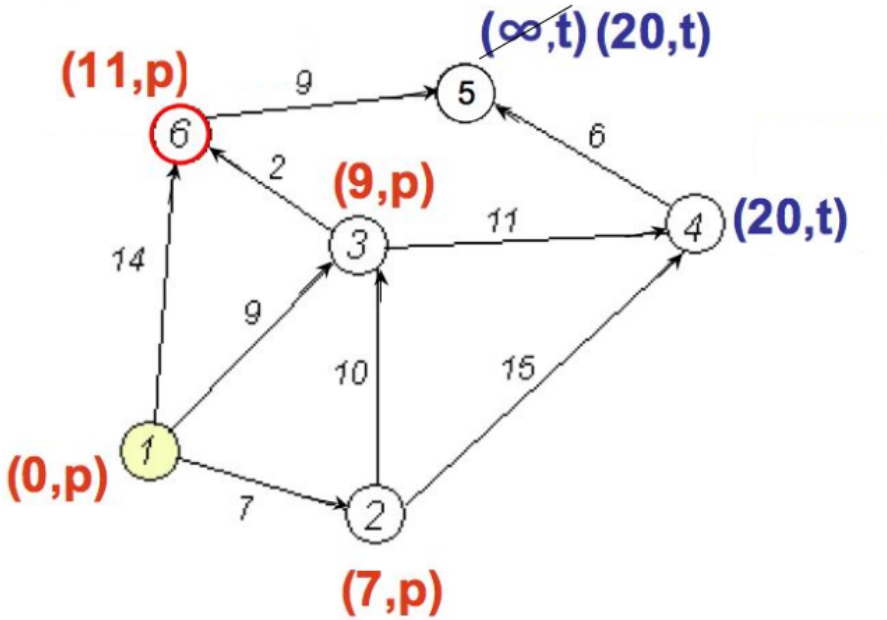
In diesem Schritt aktualisieren wir die Entfernungen zu den Nachbarn des Knotens 2. Dies liefert einen kürzeren Weg zu Knoten 4.

Beispiel 2 (5/7)

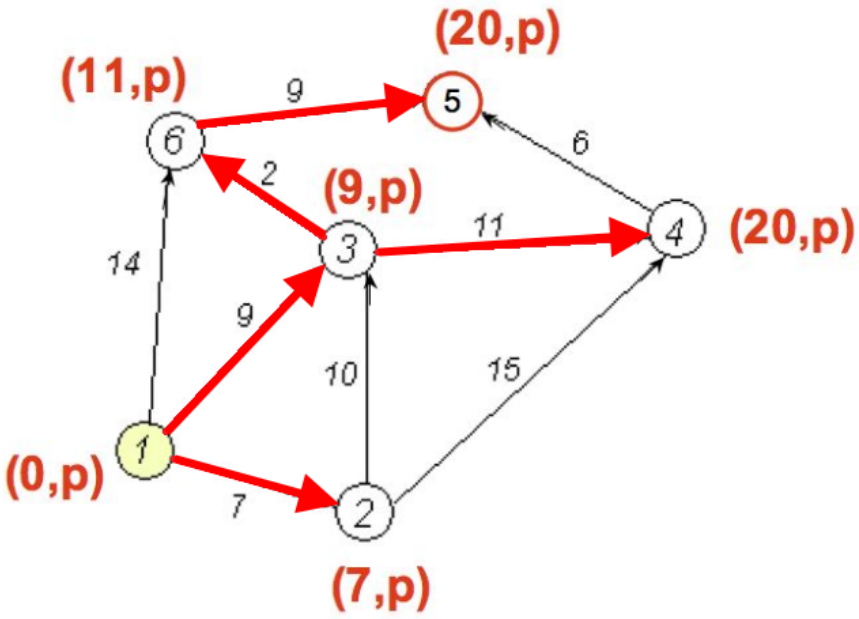


Betrachtung des Knotens 3 liefert einen noch kürzeren Weg zu Knoten 4 und einen kürzeren Weg zu Knoten 6. Knoten 3 wird permanent markiert.

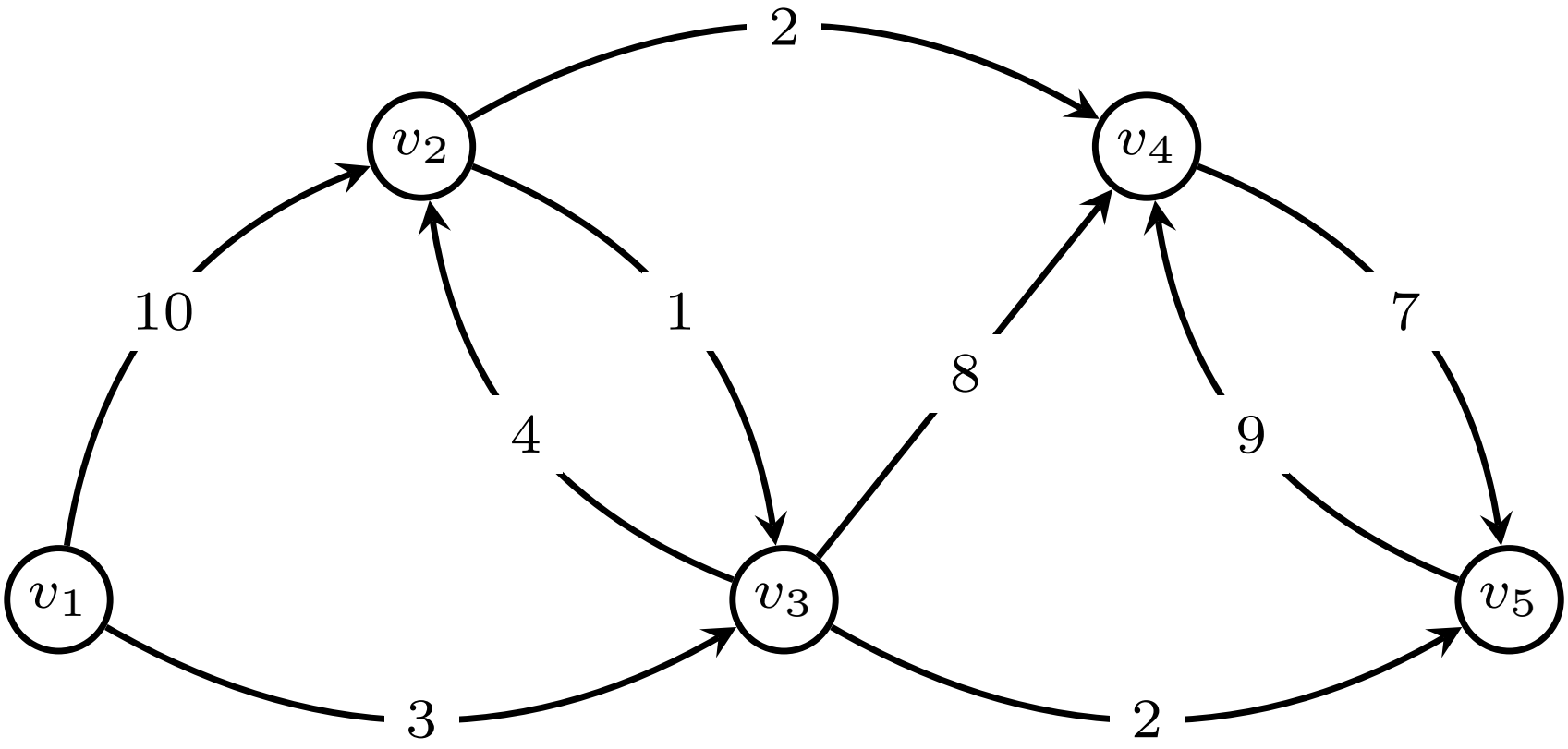
BEISPIEL 2 (6/7)



BEISPIEL 2 (7 / 7)

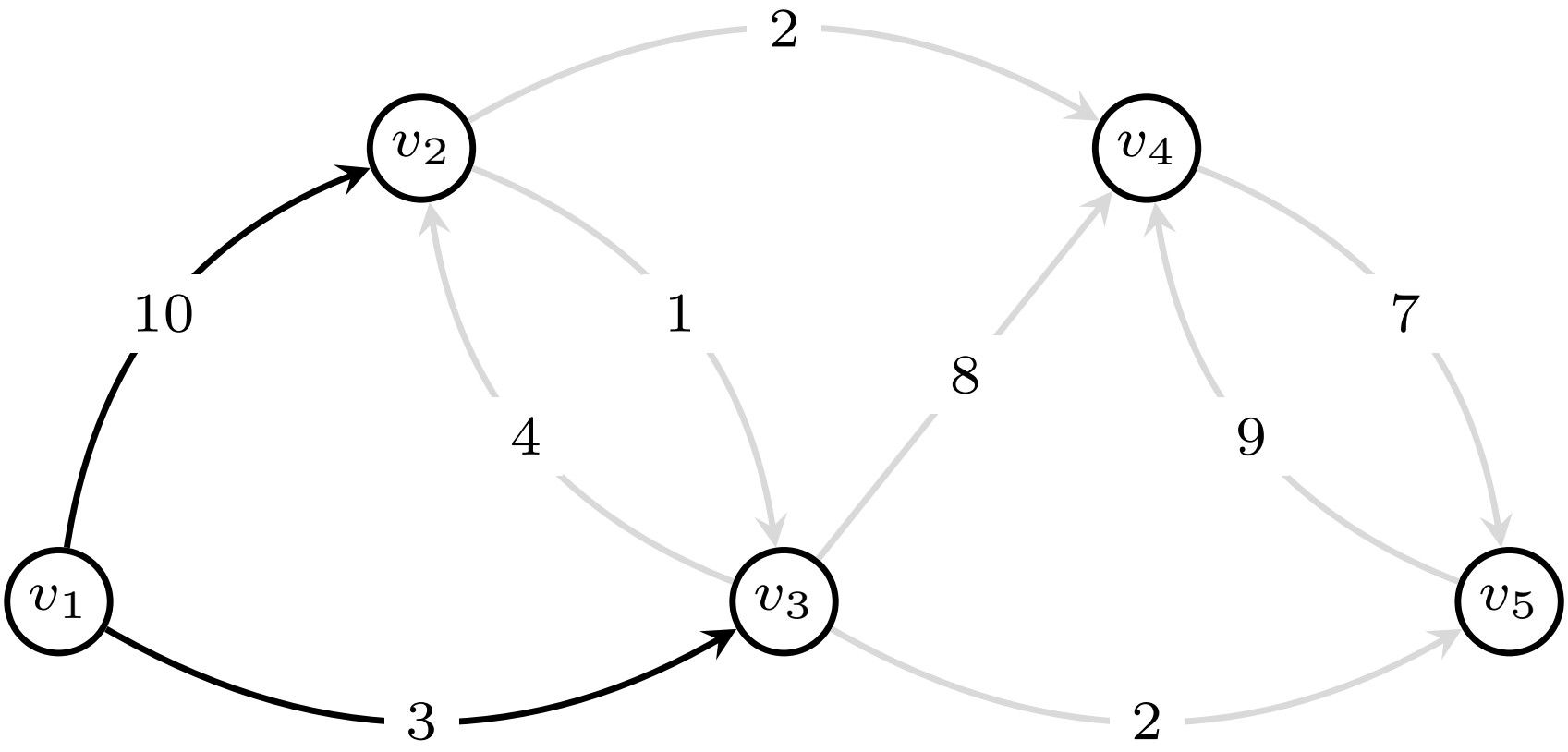


Beispiel 3 (1/6)



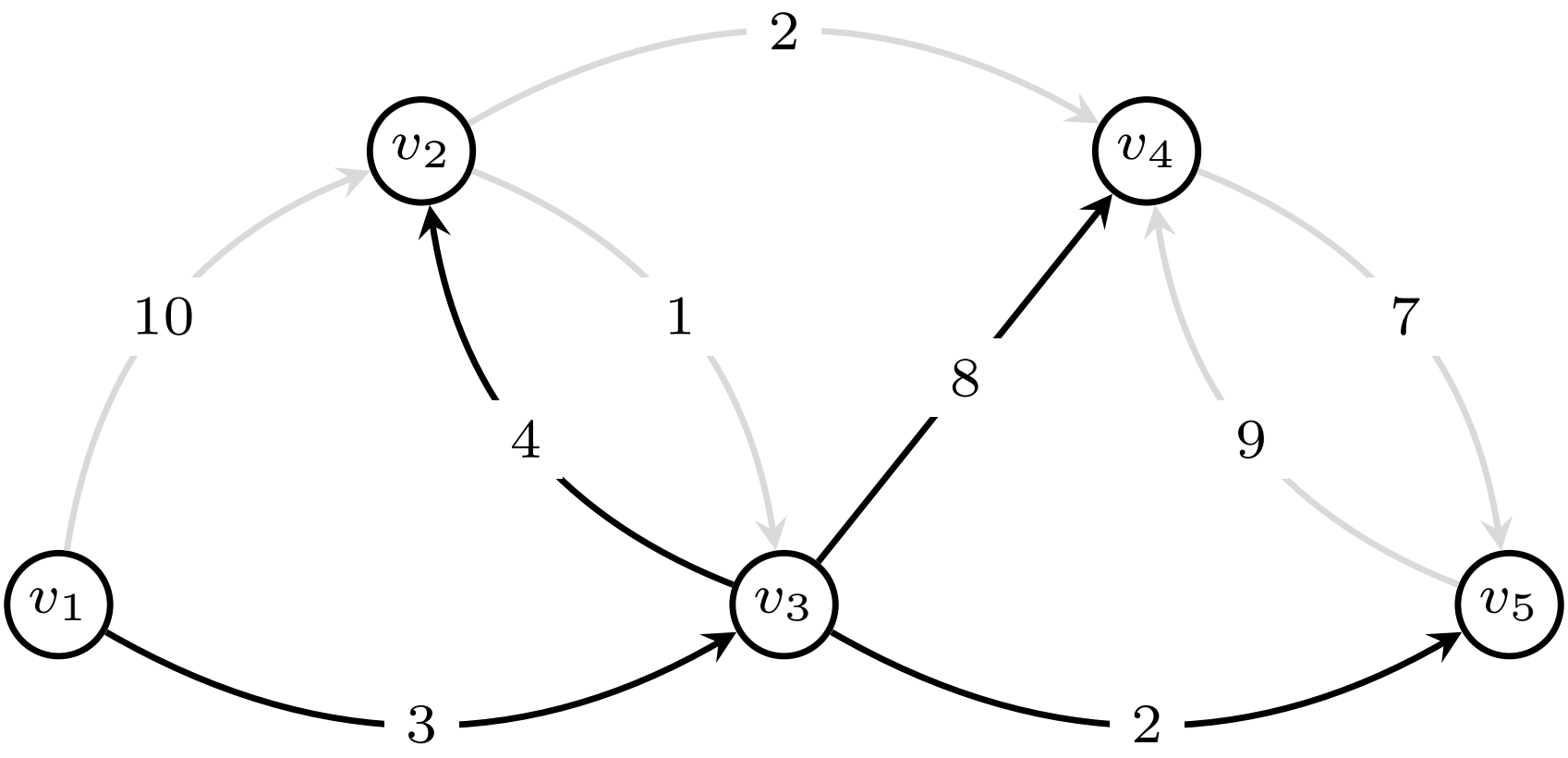
Startknoten v_1

Beispiel 3 (2/6)



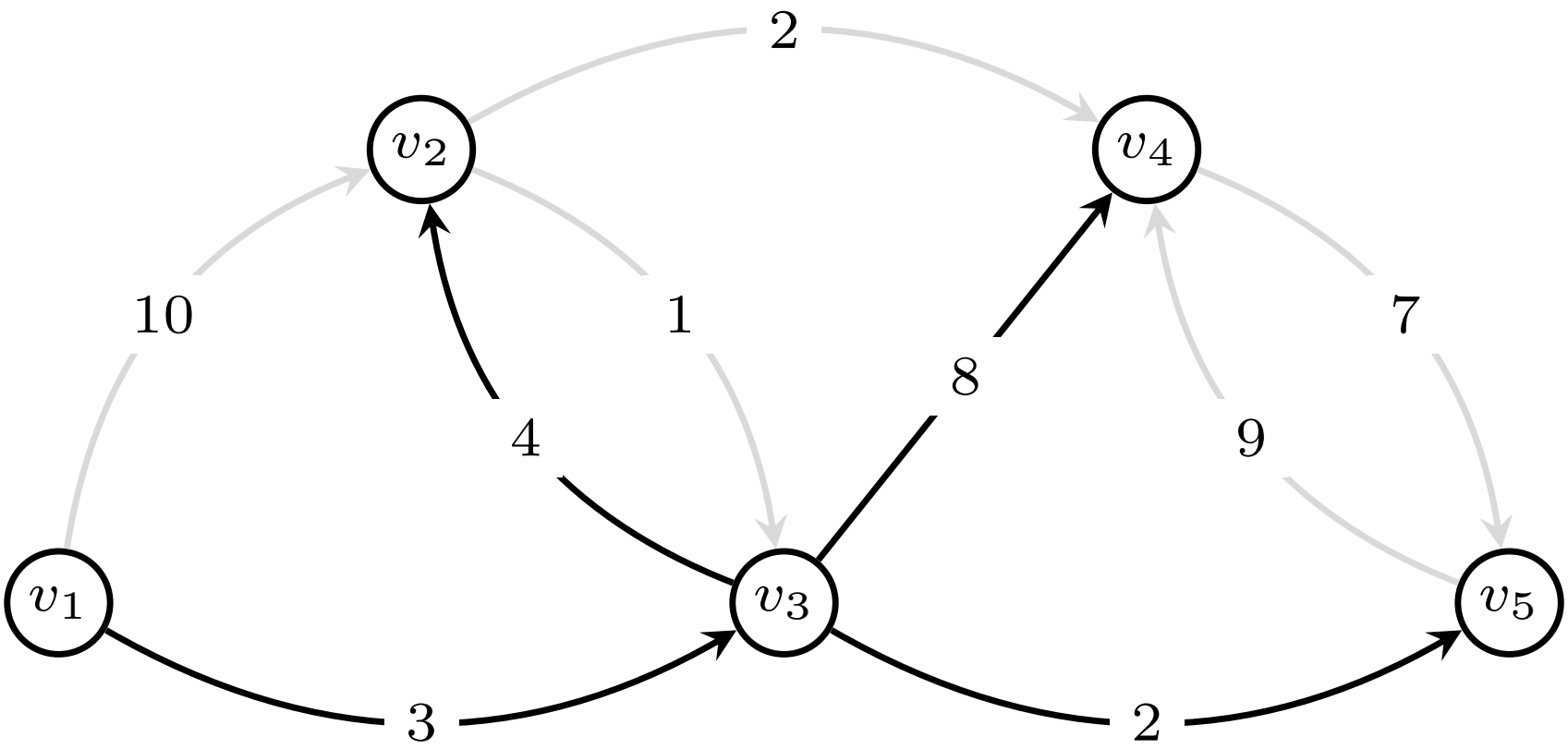
Von v_1 aus sehen wir, dass unter seinen 2 Nachbarn, v_3 günstiger zu erreichen ist. Dieser wird also als nächster betrachtet.

Beispiel 3 (3/6)



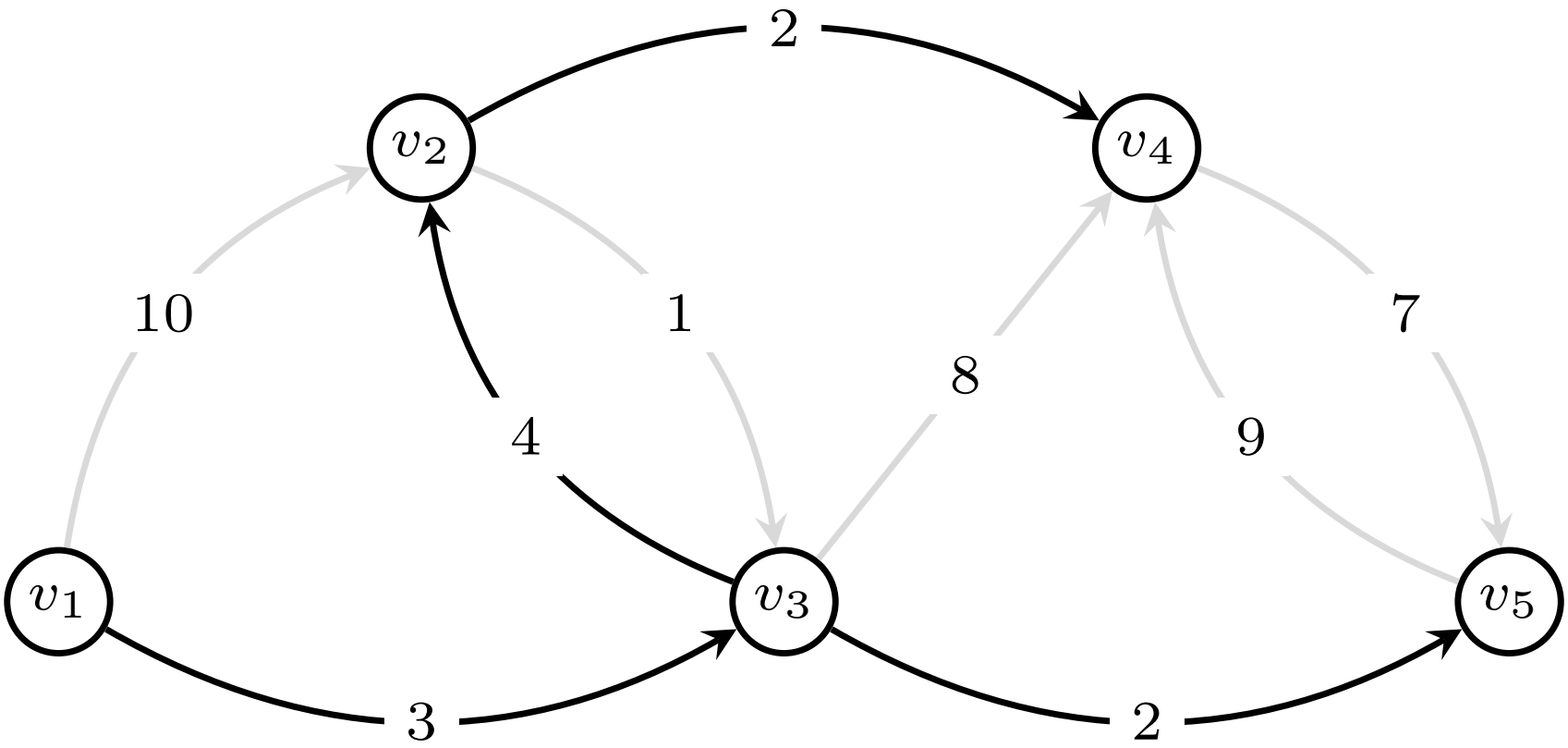
Nun ist v_5 am günstigsten zu erreichen, diesen Knoten betrachten wir also nun.

Beispiel 3 (4/6)



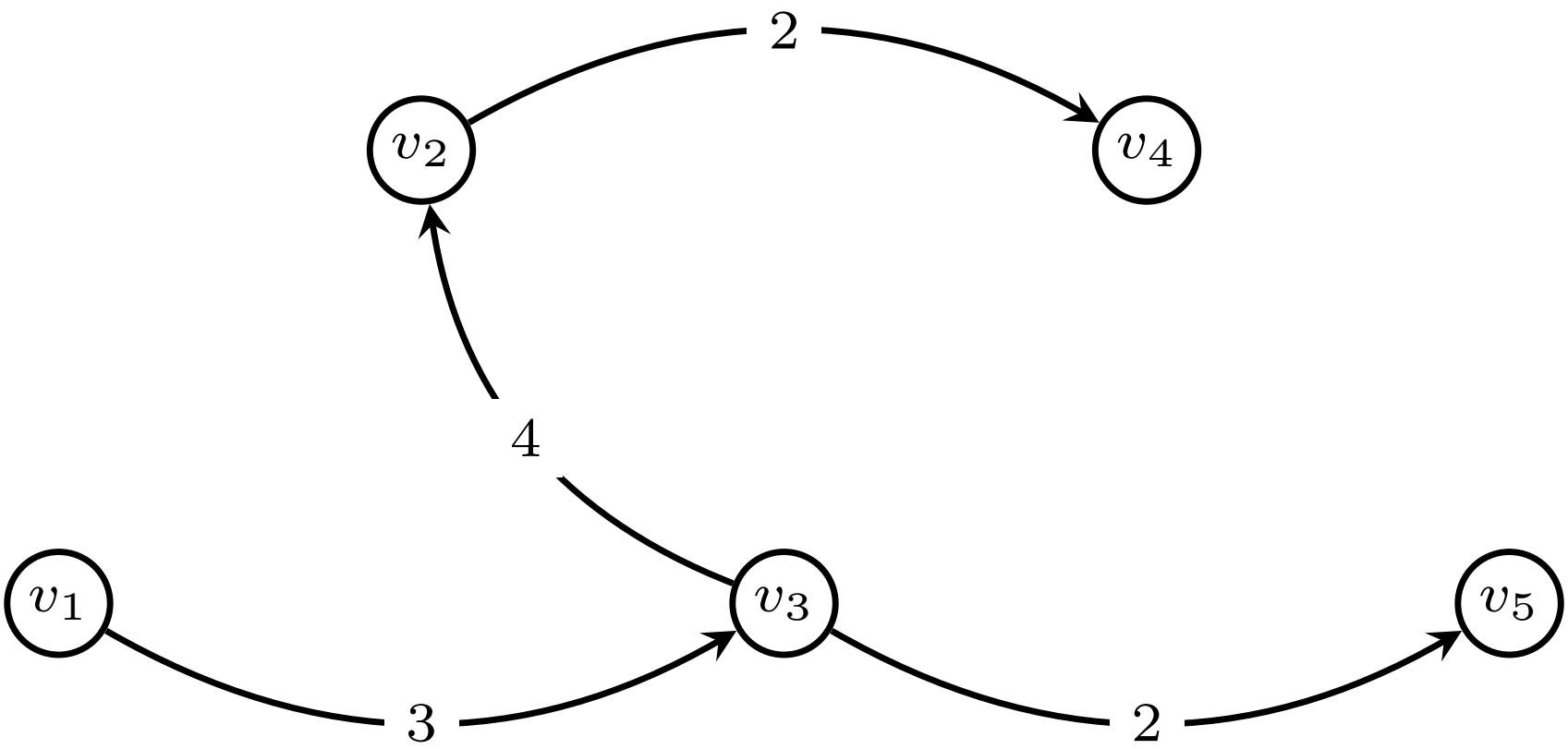
Die Betrachtung von v_5 hat nichts gebracht. Der nächstgünstigere Knoten ist v_2, den wir nun betrachten.

Beispiel 3 (5/6)



Nun sehen wir, dass wir die Distanzmarke von v_4 verbessern können: Der Weg $v_1v_3v_4$ hat Gewicht 11, der Weg $v_1v_3v_2v_4$ jedoch lediglich 9.

Beispiel 3 (6/6)



Beispiel: Algorithmus von Dijkstra

v_1	v_2	v_3	v_4	v_5
$\underline{(0,-)}$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$	$(\infty,-)$
	$(10,v_1)$	$\underline{(3,v_1)}$	$(11,v_3)$	$\underline{(5,v_3)}$
	$\underline{(7,v_3)}$		$\underline{(9,v_2)}$	

Tabelle 1: Zwischenschritte des Dijkstra Algorithmus. Die Einträge der Tabelle sind Paare: an erster Stelle steht das Gewicht eines günstigsten Weges von v_1 zu dem entsprechenden Knoten und an zweiter Stelle der Vorgängerknoten.

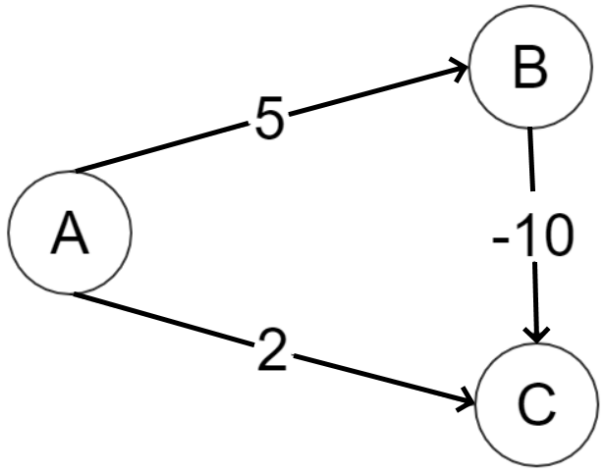
Beispiel: Algorithmus von Dijkstra

Kürzeste Wege und Entfernungen

$$
\begin{aligned}
v_1\text{-}v_2 &: v_1, v_3, v_2 & d(v_1, v_2) &= 7 \\
v_1\text{-}v_3 &: v_1, v_3 & d(v_1, v_3) &= 3 \\
v_1\text{-}v_4 &: v_1, v_3, v_2, v_4 & d(v_1, v_4) &= 9 \\
v_1\text{-}v_5 &: v_1, v_3, v_5 & d(v_1, v_5) &= 5
\end{aligned}
$$

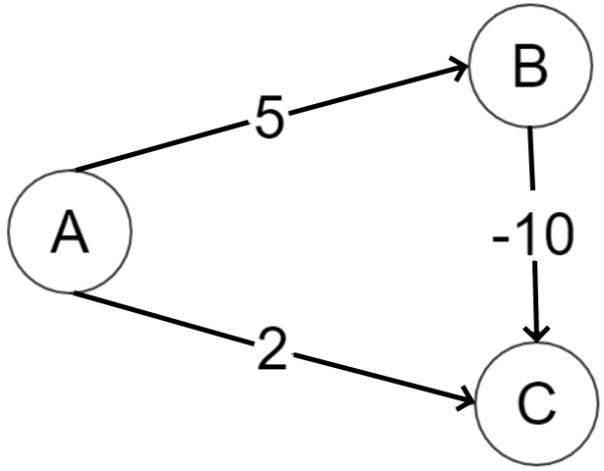
Für Graphen mit negativen Gewichten kann der Algorithmus falsche Resultate liefern.

Aufgabe. Wenden Sie den Algorithmus von Dijkstra auf folgenden Graphen an, mit Startknoten A.



Für Graphen mit negativen Gewichten kann der Algorithmus falsche Resultate liefern.

Aufgabe. Wenden Sie den Algorithmus von Dijkstra auf folgenden Graphen an, mit Startknoten A.



Der Algorithmus durchläuft die Knoten in der Reihenfolge A, C, B – insbesondere wird der Knoten C permanent markiert, nachdem dieser besucht wurde, seine Distanzmarke wird also nicht mehr abgeändert – und liefert als Abstände von A:

$d(AA) = 0$, $d(AB) = 5$ und $d(AC) = 2$, obwohl $d(AC) = -5$.