Algorithmentheorie

Daniel Neuen (Universität Bremen)

WiSe 2023/24

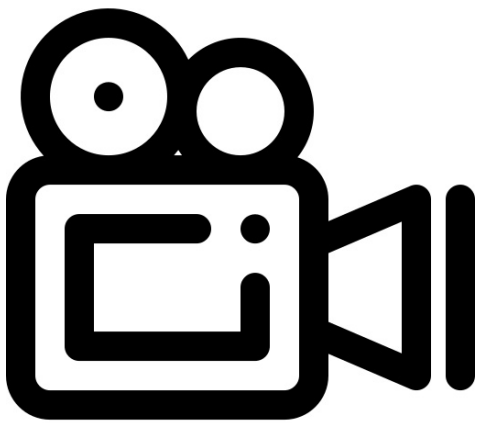
Eulertouren und Minimale aufspannende Bäume

6. Vorlesung

Aufzeichnung der Vorlesung

Diese Vorlesung wird aufgezeichnet und live gestreamt.

- Aufzeichnungen nur der Lehrenden durch sich selbst.
- Bei Rückfragen aus dem Auditorium und Diskussion bitte deutlich anzeigen, falls das Mikro stumm geschaltet werden soll.



Organisatorisches

Klausur:

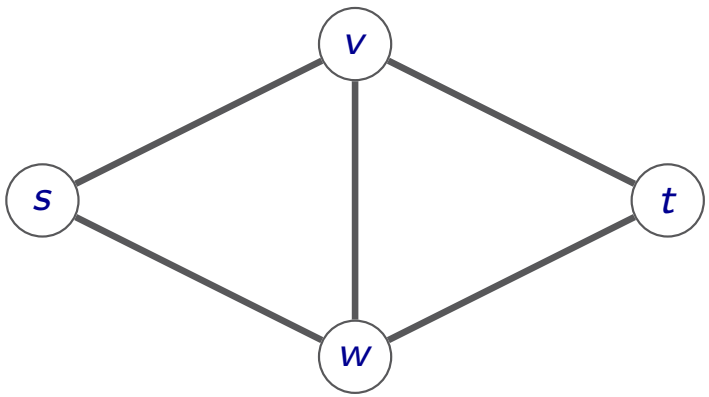
- Klausur findet am Montag, den 19.02.2024, 14:00-16:00 statt

Ungerichtete Graphen

Definition

Ein ungerichteter Graph $G = (V, E)$ ist ein Paar bestehend aus

- einer endlichen Menge V von Knoten (vertices), auch $V(G)$
- und einer Menge E von ungeordneten Paaren von Knoten, genannt Kanten (edges), auch $E(G)$.

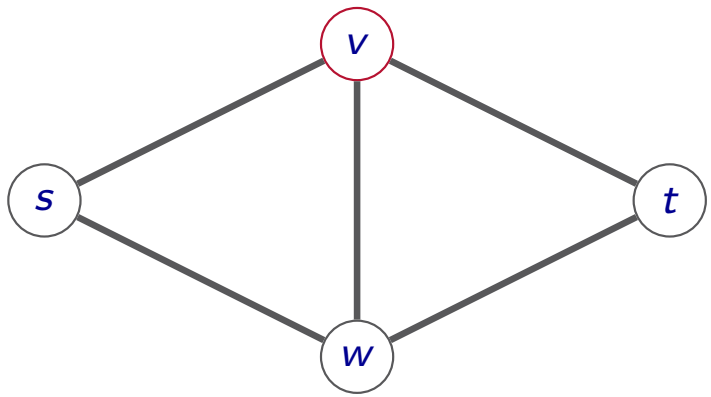


Ungerichtete Graphen

Definition

Ein ungerichteter Graph $G = (V, E)$ ist ein Paar bestehend aus

- einer endlichen Menge V von Knoten (vertices), auch $V(G)$
- und einer Menge E von ungeordneten Paaren von Knoten, genannt Kanten (edges), auch $E(G)$.



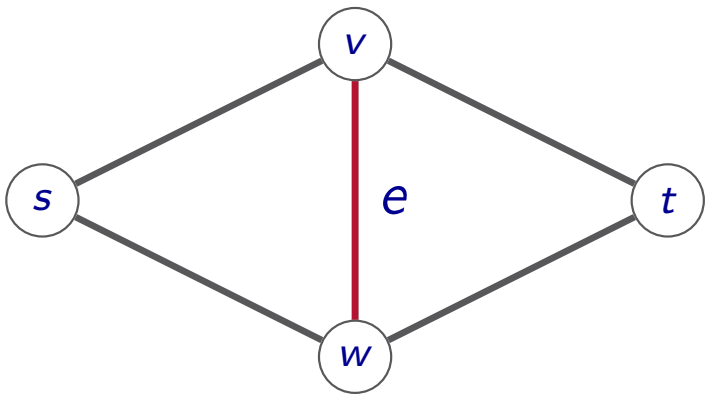
Knoten v

Ungerichtete Graphen

Definition

Ein ungerichteter Graph $G = (V, E)$ ist ein Paar bestehend aus

- einer endlichen Menge V von Knoten (vertices), auch $V(G)$
- und einer Menge E von ungeordneten Paaren von Knoten, genannt Kanten (edges), auch $E(G)$.



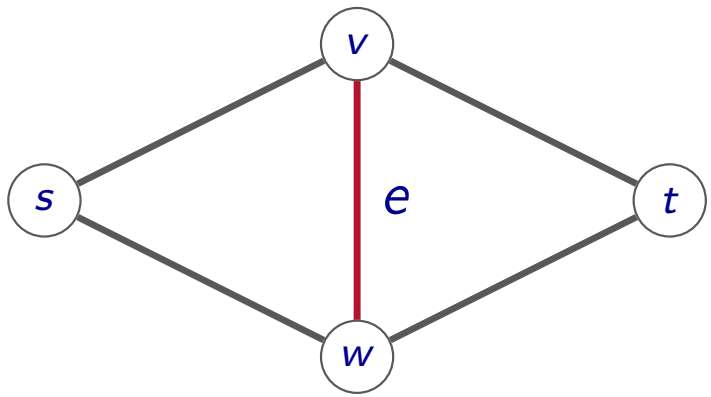
Knoten v und Kante $e = \{v, w\}$

Ungerichtete Graphen

Definition

Ein ungerichteter Graph $G = (V, E)$ ist ein Paar bestehend aus

- einer endlichen Menge V von Knoten (vertices), auch $V(G)$
- und einer Menge E von ungeordneten Paaren von Knoten, genannt Kanten (edges), auch $E(G)$.



Knoten v und Kante $e = \{v, w\}$

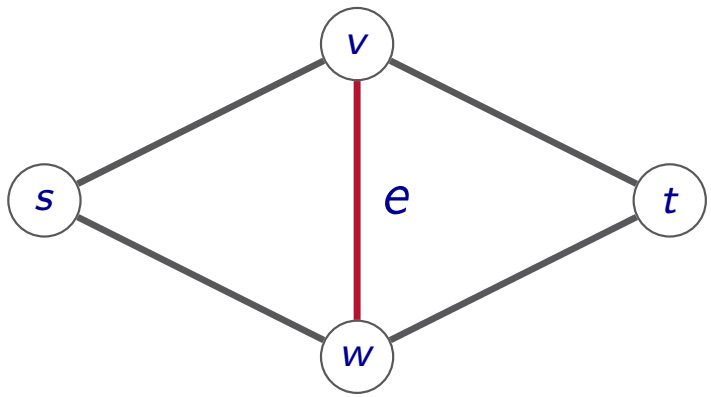
Notation: Kante e auch vw oder (v, w)

Ungerichtete Graphen

Definition

Ein ungerichteter Graph $G = (V, E)$ ist ein Paar bestehend aus

- einer endlichen Menge V von Knoten (vertices), auch $V(G)$
- und einer Menge E von ungeordneten Paaren von Knoten, genannt Kanten (edges), auch $E(G)$.



Knoten v und Kante $e = \{v, w\}$

Notation: Kante e auch vw oder (v, w)

Sprechweise: v, w sind adjazent (benachbart); e ist inzident zu v, w

Weg, Pfad, Kantenzug

Definition

Ein Weg (auch Kantenzug) der Länge k in einem ungerichteten Graphen $G = (V, E)$ von einem Knoten v zu einem Knoten w ist eine endliche Folge von Knoten $v_1, v_2, ..., v_{k+1}$, sodass:

- $\{v_i, v_{i+1}\} \in E$ für $1 \leq i \leq k$ und
- $v = v_1$ und $w = v_{k+1}$.

Weg, Pfad, Kantenzug

Definition

Ein Weg (auch Kantenzug) der Länge k in einem ungerichteten Graphen $G = (V, E)$ von einem Knoten v zu einem Knoten w ist eine endliche Folge von Knoten $v_1, v_2, ..., v_{k+1}$, sodass:

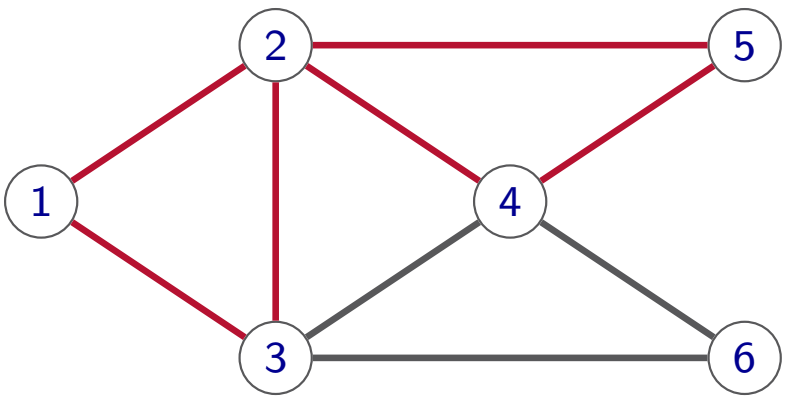
- $\{v_i, v_{i+1}\} \in E$ für $1 \leq i \leq k$ und
- $v = v_1$ und $w = v_{k+1}$.

Ein Weg ist einfach (oder elementar oder ein Pfad), wenn kein Knoten (und damit keine Kante) mehrfach vorkommt.

Kreise

Definition

Ein geschlossener Weg in einem ungerichteten Graphen $G = (V, E)$ ist ein Weg mit gleichem Start- und Endknoten. Ein Kreis ist ein Weg $v_1, \ldots, v_k$ mit $v_1 = v_k$ und $v_i \neq v_j$ für $1 \leq i < j < k$.

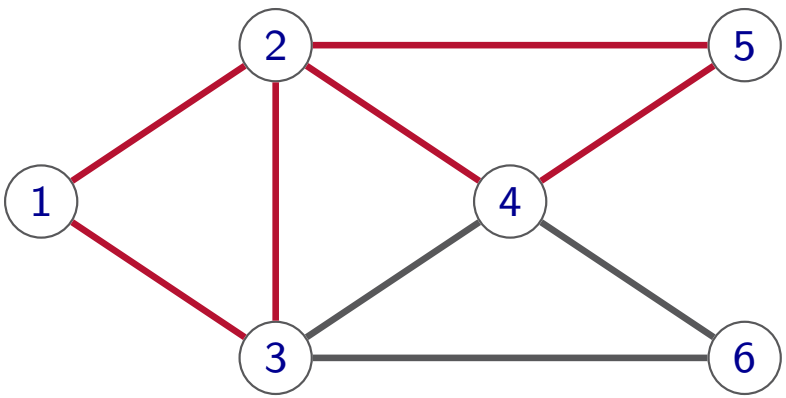


Der Weg (1,2,4,5,2,3,1) ist kein Kreis

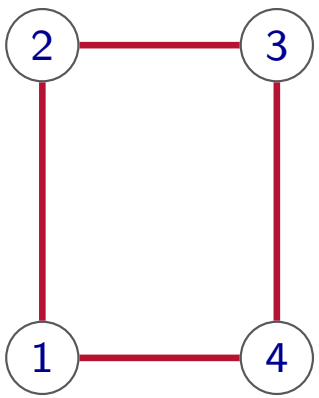
Kreise

Definition

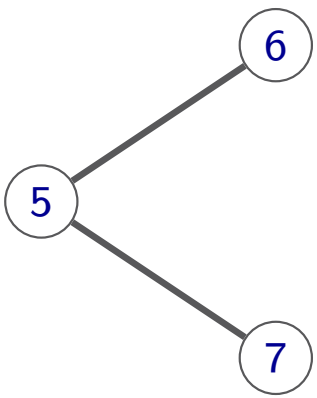
Ein geschlossener Weg in einem ungerichteten Graphen $G = (V, E)$ ist ein Weg mit gleichem Start- und Endknoten. Ein Kreis ist ein Weg $v_1, \ldots, v_k$ mit $v_1 = v_k$ und $v_i \neq v_j$ für $1 \leq i < j < k$.



Der Weg (1,2,4,5,2,3,1) ist kein Kreis



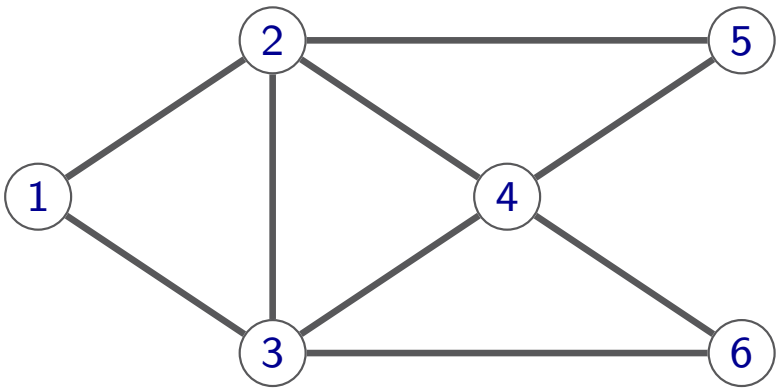
Der Weg (1,2,3,4,1) ist ein Kreis



Zusammenhang (ungerichtete Graphen)

Definition

Ein ungerichteter Graph $G = (V, E)$ ist zusammenhängend („connected“), wenn es in G zwischen je zwei Knoten $u, v \in V$ einen Weg von u nach v gibt.

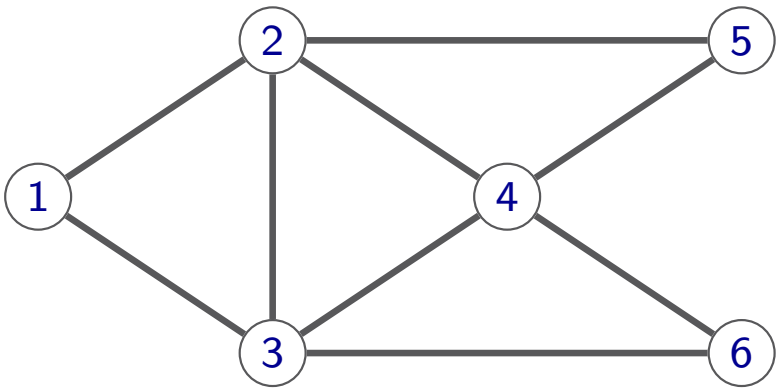


Zusammenhängender Graph

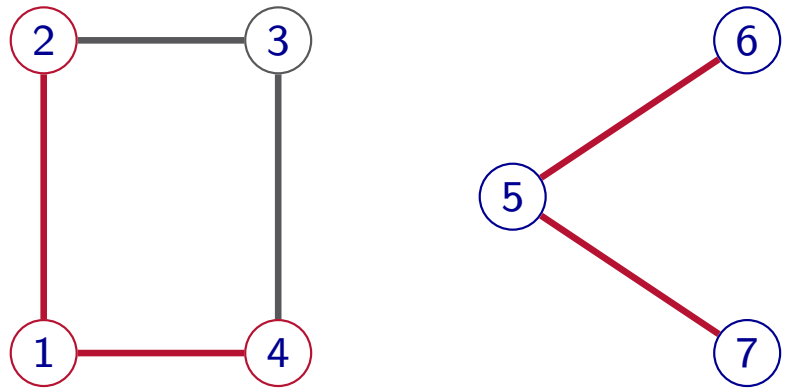
Zusammenhang (ungerichtete Graphen)

Definition

Ein ungerichteter Graph $G = (V, E)$ ist zusammenhängend („connected“), wenn es in G zwischen je zwei Knoten $u, v \in V$ einen Weg von u nach v gibt.



Zusammenhängender Graph

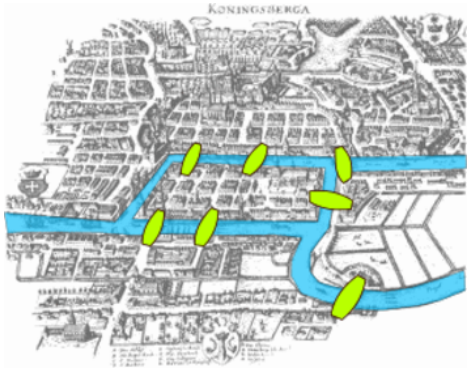


Der blaue Teilgraph ist eine Zusammenhangskomponente, der rote Teilgraph nicht

Eine Zusammenhangskomponente („connected component“) eines ungerichteten Graphen G ist ein maximal zusammenhängender Teilgraph.

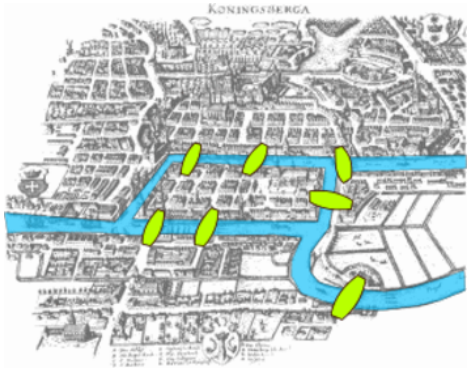
Das Königsberger Brückenproblem

Das Königsberger Brückenproblem



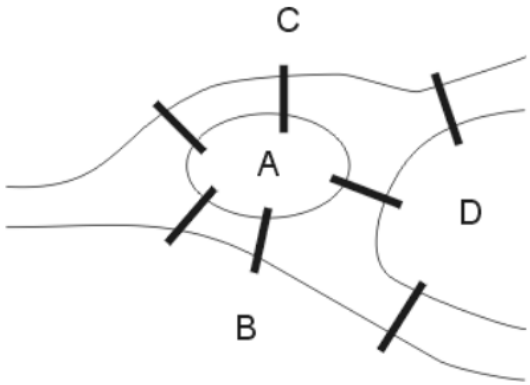
L. Euler: Gibt es einen (Rund)Weg, bei dem man jede der sieben Brücken genau einmal überquert?
$\rightarrow$ Frage nach Eulerweg (-tour) in Graphen

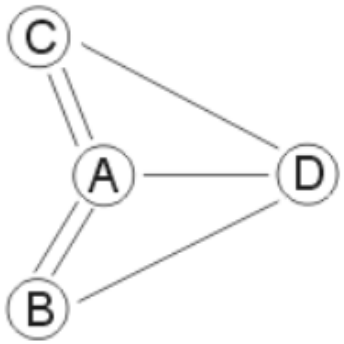
Das Königsberger Brückenproblem



L. Euler: Gibt es einen (Rund)Weg, bei dem man jede der sieben Brücken genau einmal überquert?
→ Frage nach Eulerweg (-tour) in Graphen

Geburtsstunde der Graphentheorie





L. Euler (1707-1783)

Eulersche Graphen

Definition

Sei $G = (V, E)$ ein (un-)gerichteter Graph.

- Ein Eulerweg in G ist ein Weg, der jede Kante in G genau einmal enthält. (Haus des Nikolaus)
- Eine Eulertour (Eulerkreis) ist ein geschlossener Eulerweg.
- Ein Graph heißt Eulersch, wenn er eine Eulertour enthält.

Eulersche Graphen

Definition

Sei $G = (V, E)$ ein (un-)gerichteter Graph.

- Ein Eulerweg in G ist ein Weg, der jede Kante in G genau einmal enthält. (Haus des Nikolaus)
- Eine Eulertour (Eulerkreis) ist ein geschlossener Eulerweg.
- Ein Graph heißt Eulersch, wenn er eine Eulertour enthält.

Satz (Euler 1736, Hierholzer 1873)

Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph. G ist Eulersch genau dann wenn alle Knoten in V geraden Grad haben.

Eulersche Graphen

Definition

Sei $G = (V, E)$ ein (un-)gerichteter Graph.

- Ein Eulerweg in G ist ein Weg, der jede Kante in G genau einmal enthält. (Haus des Nikolaus)
- Eine Eulertour (Eulerkreis) ist ein geschlossener Eulerweg.
- Ein Graph heißt Eulersch, wenn er eine Eulertour enthält.

Satz (Euler 1736, Hierholzer 1873)

Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph. G ist Eulersch genau dann wenn alle Knoten in V geraden Grad haben.

Beweis. → an Tafel

Wie findet man eine Eulertour?

Sei G Eulersch.

Wir haben gezeigt:

- Es existiert ein geschlossener Weg C in G.
- Jede Zusammenhangskomponente hat geraden Knotengrad auch ohne Kanten aus C und mind. einen Knoten mit C gemeinsam.

$\Rightarrow$ **Rekursion auf Zusammenhangskomponenten**

Hierholzer Algorithmus

Input : Ein Graph $G = (V, E)$
Output : Eine Eulertour K, oder false, falls keine existiert.

1 Falls Graph nicht Eulersch (check Knotengrad $\forall v \in V$), **return** false.
2 Wähle $v_0 \in V$ beliebig. Setze $K = \emptyset$.
3 **while** true **do**
4 Konstruiere geschlossenen Weg K' in G, der in v_0 startet und keine Kante doppelt enthält.
5 $K \leftarrow K \cup K'$, d.h. durchlaufe K bis v_0, dann K', dann Rest von K.
6 Setze $E \leftarrow E \setminus E(K)$.
7 **if** $E = \emptyset$ **then**
8 **return** K
9 **else**
10 $v_0 \leftarrow$ erster Knoten in K mit Grad > 0 (bzgl. verbliebener Kanten)

Hierholzer Algorithmus

Input : Ein Graph $G = (V, E)$
Output : Eine Eulertour K, oder false, falls keine existiert.

1 Falls Graph nicht Eulersch (check Knotengrad $\forall v \in V$), **return** false.
2 Wähle $v_0 \in V$ beliebig. Setze $K = \emptyset$.
3 **while** true **do**
4 Konstruiere geschlossenen Weg K' in G, der in v_0 startet und keine Kante doppelt enthält.
5 $K \leftarrow K \cup K'$, d.h. durchlaufe K bis v_0, dann K', dann Rest von K.
6 Setze $E \leftarrow E \setminus E(K)$.
7 **if** $E = \emptyset$ **then**
8 **return** K
9 **else**
10 $v_0 \leftarrow$ erster Knoten in K mit Grad > 0 (bzgl. verbliebener Kanten)

Korrektheit: Folgt aus konstruktivem Beweis des Satzes von Euler/Hierholzer. Insbesondere terminiert der Algorithmus, da in jeder Iteration mindestens eine Kante entfernt wird.

Hierholzer Algorithmus - Laufzeit

Initialisierung (Schritte 1 und 2) in Zeit $\mathcal{O}(n+m)$.

Hierholzer Algorithmus - Laufzeit

Initialisierung (Schritte 1 und 2) in Zeit $\mathcal{O}(n+m)$.

Jede Iteration in der while-Schleife läuft in Zeit $\mathcal{O}(|K'|)$

Hierholzer Algorithmus - Laufzeit

Initialisierung (Schritte 1 und 2) in Zeit $\mathcal{O}(n+m)$.

Jede Iteration in der while-Schleife läuft in Zeit $\mathcal{O}(|K'|)$

- Adjazenzlisten enthalten nur verbliebene Kanten

Hierholzer Algorithmus - Laufzeit

Initialisierung (Schritte 1 und 2) in Zeit $\mathcal{O}(n+m)$.

Jede Iteration in der while-Schleife läuft in Zeit $\mathcal{O}(|K'|)$

- Adjazenzlisten enthalten nur verbliebene Kanten
- Nutze immer die erste Kante in der Liste

Hierholzer Algorithmus - Laufzeit

Initialisierung (Schritte 1 und 2) in Zeit $\mathcal{O}(n+m)$.

Jede Iteration in der while-Schleife läuft in Zeit $\mathcal{O}(|K'|)$

- Adjazenzlisten enthalten nur verbliebene Kanten
- Nutze immer die erste Kante in der Liste
- Doubly Linked List zum Löschen der Kanten in Zeit $\mathcal{O}(1)$

Hierholzer Algorithmus - Laufzeit

Initialisierung (Schritte 1 und 2) in Zeit $\mathcal{O}(n+m)$.

Jede Iteration in der while-Schleife läuft in Zeit $\mathcal{O}(|K'|)$

- Adjazenzlisten enthalten nur verbliebene Kanten
- Nutze immer die erste Kante in der Liste
- Doubly Linked List zum Löschen der Kanten in Zeit $\mathcal{O}(1)$

In jeder Iteration werden $|K'|$ Kanten gelöscht, also terminiert die while-Scheife nach $\mathcal{O}(m)$ Schritten.

Hierholzer Algorithmus - Laufzeit

Initialisierung (Schritte 1 und 2) in Zeit $\mathcal{O}(n+m)$.

Jede Iteration in der while-Schleife läuft in Zeit $\mathcal{O}(|K'|)$

- Adjazenzlisten enthalten nur verbliebene Kanten
- Nutze immer die erste Kante in der Liste
- Doubly Linked List zum Löschen der Kanten in Zeit $\mathcal{O}(1)$

In jeder Iteration werden $|K'|$ Kanten gelöscht, also terminiert die while-Scheife nach $\mathcal{O}(m)$ Schritten.

$\rightarrow \mathcal{O}(n+m)$ Linearzeit

Hamiltonsche Graphen

Definition

Sei $G = (V, E)$ ein ungerichteter Graph.

- Ein Kreis in G ist ein Hamiltonkreis, wenn er jeden Knoten in V genau einmal enthält.
- Ein Graph heisst hamiltonsch, wenn er einen Hamiltonkreis enthält.

Hamiltonsche Graphen

Definition

Sei $G = (V, E)$ ein ungerichteter Graph.

- Ein Kreis in G ist ein Hamiltonkreis, wenn er jeden Knoten in V genau einmal enthält.
- Ein Graph heisst hamiltonsch, wenn er einen Hamiltonkreis enthält.

Frage: Ist ein gegebener Graph hamiltonsch?

Hamiltonsche Graphen

Definition

Sei $G = (V, E)$ ein ungerichteter Graph.

- Ein Kreis in G ist ein Hamiltonkreis, wenn er jeden Knoten in V genau einmal enthält.
- Ein Graph heisst hamiltonsch, wenn er einen Hamiltonkreis enthält.

Frage: Ist ein gegebener Graph hamiltonsch?

- Vermeintlich ähnlich zu Frage, ob Graph Eulersch.
- Andere Komplexitätsklasse $\rightarrow$ Problem ist NP-vollständig
  Man nimmt an, dass es keinen effizienten Test gibt. (später mehr)

Aufspannende Bäume

Bäume und Wälder

Definition

Ein ungerichteter Graph heißt Wald, wenn er kreisfrei ist.

Bäume und Wälder

Definition

Ein ungerichteter Graph heißt Wald, wenn er kreisfrei ist.

Ein ungerichteter Graph G heißt Baum, wenn er zusammenhängend und kreisfrei ist.

Bäume und Wälder

Definition

Ein ungerichteter Graph heißt Wald, wenn er kreisfrei ist.

Ein ungerichteter Graph G heißt Baum, wenn er zusammenhängend und kreisfrei ist.

Ein Knoten $v \in V$ mit $deg(v) = 1$ heißt Blatt (leaf). Alle anderen Knoten heißen innere Knoten.

Bäume und Wälder

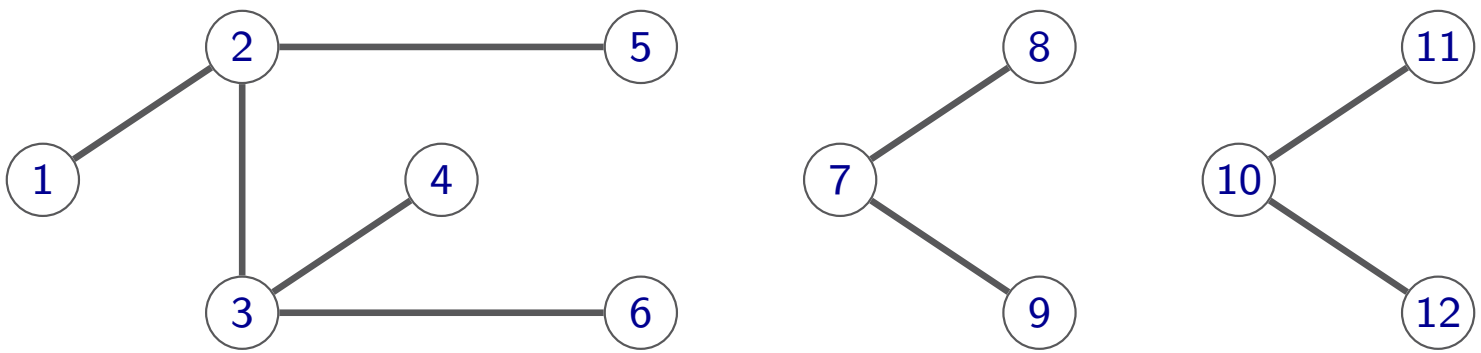
Definition

Ein ungerichteter Graph heißt Wald, wenn er kreisfrei ist.

Ein ungerichteter Graph G heißt Baum, wenn er zusammenhängend und kreisfrei ist.

Ein Knoten $v \in V$ mit $deg(v) = 1$ heißt Blatt (leaf). Alle anderen Knoten heißen innere Knoten.

Beispiel: Wald aus drei Bäumen (Zusammenhangskomponenten)



Aquivalente Charakterisierungen

Satz

Sei $G = (V, E)$ ein ungerichteter Graph. Dann sind folgende Aussagen äquivalent.

(i) G ist ein Baum.

(ii) G ist zusammenhängend und kreisfrei.

Aquivalente Charakterisierungen

Satz

Sei $G = (V, E)$ ein ungerichteter Graph. Dann sind folgende Aussagen äquivalent.

(i) G ist ein Baum.

(ii) G ist zusammenhängend und kreisfrei.

(iii) G ist zusammenhängend und $|E| = n - 1$.

Aquivalente Charakterisierungen

Satz

Sei $G = (V, E)$ ein ungerichteter Graph. Dann sind folgende Aussagen äquivalent.

(i) G ist ein Baum.

(ii) G ist zusammenhängend und kreisfrei.

(iii) G ist zusammenhängend und $|E| = n - 1$.

(iv) G ist kreisfrei und $|E| = n - 1$.

Aquivalente Charakterisierungen

Satz

Sei $G = (V, E)$ ein ungerichteter Graph. Dann sind folgende Aussagen äquivalent.

(i) G ist ein Baum.

(ii) G ist zusammenhängend und kreisfrei.

(iii) G ist zusammenhängend und $|E| = n - 1$.

(iv) G ist kreisfrei und $|E| = n - 1$.

(v) G ist maximal kreisfrei, d.h. $G + e$ enthält Kreis für alle neu hinzugefügten Kanten e.

Aquivalente Charakterisierungen

Satz

Sei $G = (V, E)$ ein ungerichteter Graph. Dann sind folgende Aussagen äquivalent.

(i) G ist ein Baum.

(ii) G ist zusammenhängend und kreisfrei.

(iii) G ist zusammenhängend und $|E| = n - 1$.

(iv) G ist kreisfrei und $|E| = n - 1$.

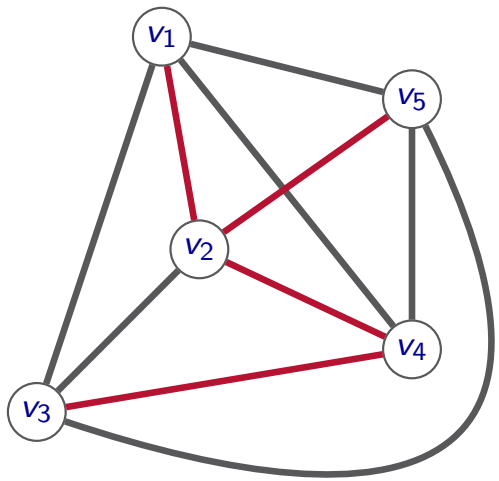
(v) G ist maximal kreisfrei, d.h. $G + e$ enthält Kreis für alle neu hinzugefügten Kanten e.

(vi) Für jedes Paar u, v von Knoten aus V existiert genau ein Pfad in G, der u und v verbindet.

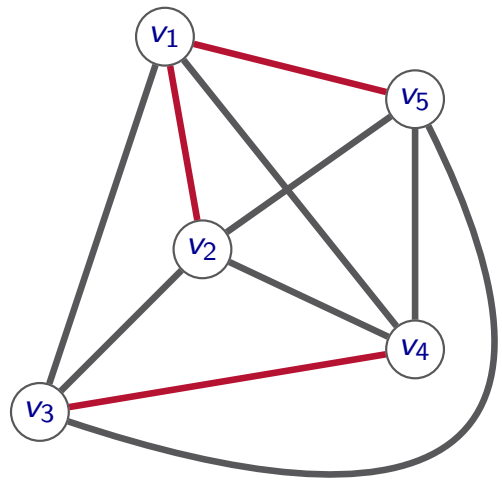
Aufspannender Baum (Spannbaum)

Definition

Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph. Ist der Teilgraph $T = (V, E')$ von G mit $E' \subseteq E$ ein Baum, dann heißt T aufspannender Baum (Spannbaum, spanning tree) von G.



Spannbaum T



Kein Spannbaum G'

Minimaler aufspannender Baum

Minimaler Spannbaum Problem (MST)

Gegeben: Ein ungerichteter, zusammenhängender, gewichteter Graph $G = (V, E, c)$ mit Kantenkosten $c(e) \in \mathbb{R}$ für alle $e \in E$.

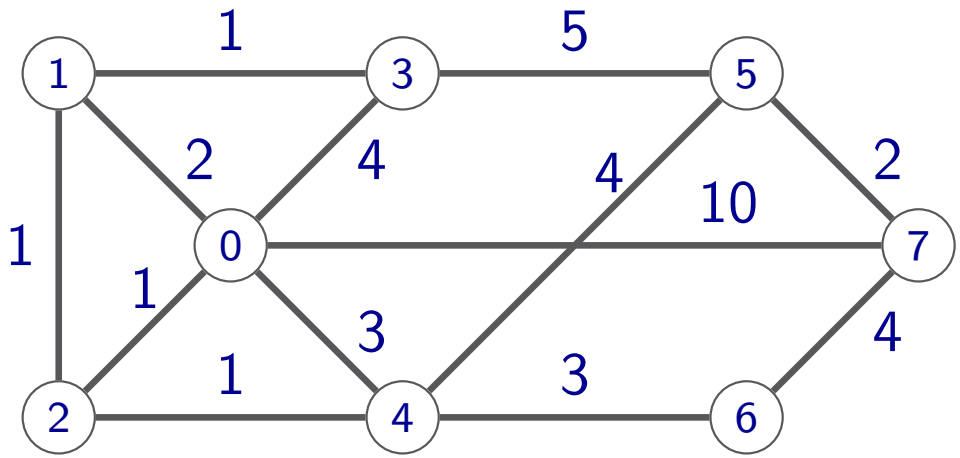
Gesucht: Eine kostenminimale Teilmenge $T \subseteq E$ der Kanten, so dass der Teilgraph $G[T] = (V, T)$ aufspannender Baum ist.

Minimaler aufspannender Baum

Minimaler Spannbaum Problem (MST)

Gegeben: Ein ungerichteter, zusammenhängender, gewichteter Graph $G = (V, E, c)$ mit Kantenkosten $c(e) \in \mathbb{R}$ für alle $e \in E$.

Gesucht: Eine kostenminimale Teilmenge $T \subseteq E$ der Kanten, so dass der Teilgraph $G[T] = (V, T)$ aufspannender Baum ist.

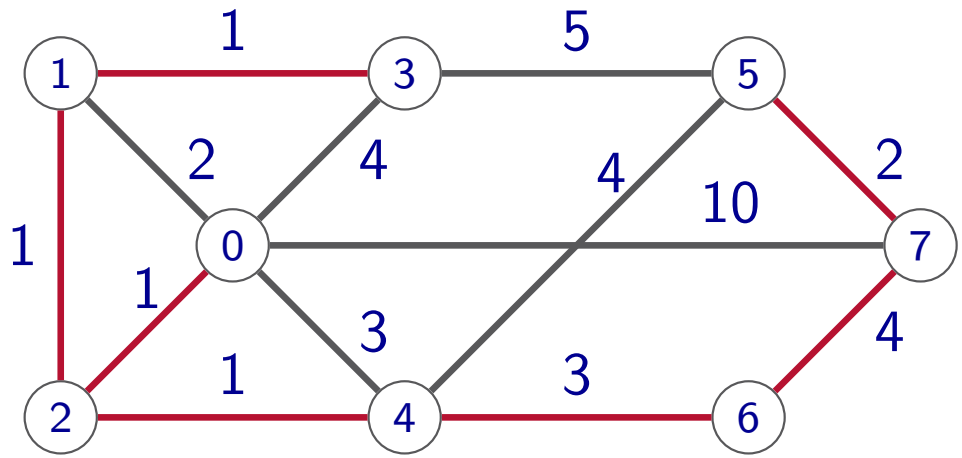


Minimaler aufspannender Baum

Minimaler Spannbaum Problem (MST)

Gegeben: Ein ungerichteter, zusammenhängender, gewichteter Graph $G = (V, E, c)$ mit Kantenkosten $c(e) \in \mathbb{R}$ für alle $e \in E$.

Gesucht: Eine kostenminimale Teilmenge $T \subseteq E$ der Kanten, so dass der Teilgraph $G[T] = (V, T)$ aufspannender Baum ist.



Die Kosten einer Teilmenge $T \subseteq E$ ist die Summe der Kantenkosten, $c(T) = \sum_{e \in T} c(e)$.

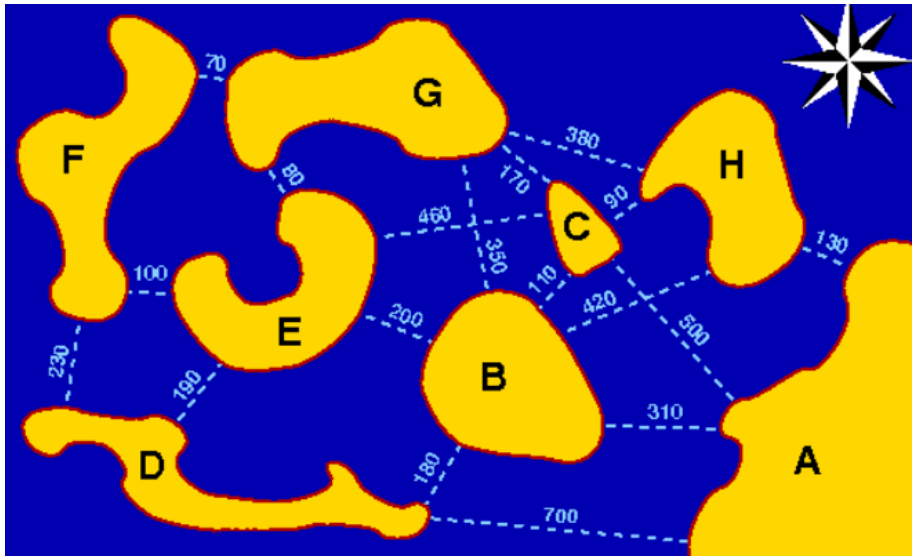
Anwendungen

Vielfältige Anwendungen: Wann immer Knoten (Kunden, Terminals, Computer, ...) möglichst günstig durch ein Netzwerk (Straßen, Leitungen, ...) verbunden werden sollen.

Anwendungen

Vielfältige Anwendungen: Wann immer Knoten (Kunden, Terminals, Computer, . . .) möglichst günstig durch ein Netzwerk (Straßen, Leitungen, . . .) verbunden werden sollen.

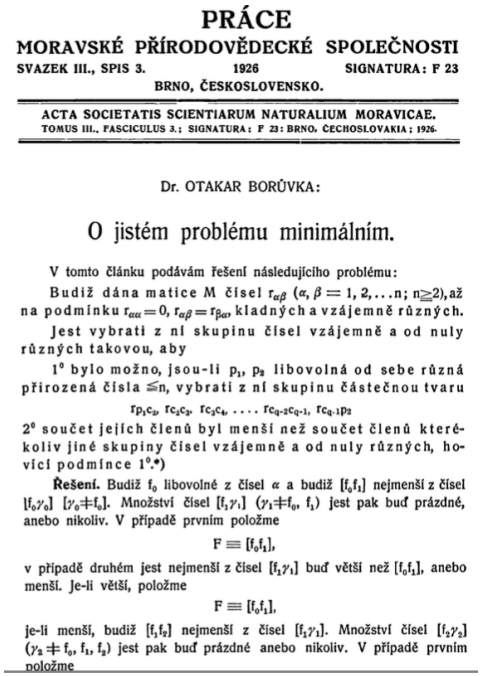
Frage im Inselreich der Algolaner: Welche Fährverbindungen sollten durch Brücken ersetzt werden?



Quelle: “Algorithmus der Woche” (2006)

Historischer Kontext

- Elektrifizierung Süd-West-Mährens 1926: wie können wirmöglichst effizient ein Elektrizitätsnetz bauen, das alle Kunden verbindet?
- Der Tschechische Mathematiker Otokar Boruvka (1899-1959) modellierte das Problem graphentheoretisch als MST-Problem und beschrieb erstmals einen Algorithmus zur Lösung des Problems.



PRÁCE
MORAVSKÉ PŘÍRODOVĚDECKÉ SPOLEČNOSTI
SVAZEK III., SPIS 3. 1926 SIGNATURA: F 23
BRNO, ČESKOSLOVENSKO.

ACTA SOCIETATIS SCIENTIARUM NATURALIUM MORAVICAE.
TOMUS III., FASCICULUS 3.; SIGNATURA: F 23: BRNO, CECHOSLOVAKIA; 1926.

Dr. OTAKAR BORŮVKA:

O jistém problému minimálním.

V tomto článku podávám řešení následujícího problému:

Budiž dána matice M čísel $r_{\alpha\beta}$ ($\alpha, \beta = 1, 2, \ldots n$; $n \geqq 2$), až na podmínku $r_{\alpha\alpha} = 0$, $r_{\alpha\beta} = r_{\beta\alpha}$, kladných a vzájemně různých.

Jest vybrati z ní skupinu čísel vzájemně a od nuly různých takovou, aby

1° bylo možno, jsou-li p_1, p_2 libovolná od sebe různá přirozená čísla $\leqq n$, vybrati z ní skupinu částečnou tvaru

$$r_{p_1 c_2},\ r_{c_2 c_3},\ r_{c_3 c_4},\ \ldots .\ r_{c_{q-2} c_{q-1}},\ r_{c_{q-1} p_2}$$

2° součet jejích členů byl menší než součet členů kterékoliv jiné skupiny čísel vzájemně a od nuly různých, hovící podmínce 1°.*)

Řešení. Budiž f_0 libovolné z čísel α a budiž $[f_0 f_1]$ nejmenší z čísel $[f_0 \gamma_0]$ $[\gamma_0 \neq f_0]$. Množství čísel $[f_1 \gamma_1]$ ($\gamma_1 \neq f_0, f_1$) jest pak buď prázdné, anebo nikoliv. V případě prvním položme

$$F \equiv [f_0 f_1],$$

v případě druhém jest nejmenší z čísel $[f_1 \gamma_1]$ buď větší než $[f_0 f_1]$, anebo menší. Je-li větší, položme

$$F \equiv [f_0 f_1],$$

je-li menší, budiž $[f_1 f_2]$ nejmenší z čísel $[f_1 \gamma_1]$. Množství čísel $[f_2 \gamma_2]$ ($\gamma_2 \neq f_0, f_1, f_2$) jest pak buď prázdné anebo nikoliv. V případě prvním položme

Bäume Enumerieren?

Formel von Cayley (Arthur Cayley 1821-1895)

In einem ungerichteten vollständigen Graphen mit n Knoten gibt es n^{n-2} unterschiedliche aufspannende Bäume.

$\rightarrow$ Viele (sehr schöne) Beweise; siehe Buch der Beweise (Aigner, Ziegler).

Bäume Enumerieren?

Formel von Cayley (Arthur Cayley 1821-1895)

In einem ungerichteten vollständigen Graphen mit n Knoten gibt es n^{n-2} unterschiedliche aufspannende Bäume.

$\rightarrow$ Viele (sehr schöne) Beweise; siehe Buch der Beweise (Aigner, Ziegler).

Folgerung: Brute-Force Ansatz für MST Problem, alle aufspannenden Bäume enumerieren und günstigsten auswählen, dauert viel zu lange!

Bäume Enumerieren?

Formel von Cayley (Arthur Cayley 1821-1895)

In einem ungerichteten vollständigen Graphen mit n Knoten gibt es n^{n-2} unterschiedliche aufspannende Bäume.

$\rightarrow$ Viele (sehr schöne) Beweise; siehe Buch der Beweise (Aigner, Ziegler).

Folgerung: Brute-Force Ansatz für MST Problem, alle aufspannenden Bäume enumerieren und günstigsten auswählen, dauert viel zu lange!

Beispiel: $n = 30$

- $30^{28} = 2,3 \cdot 10^{41}$ aufspannende Bäume
- Schnellster Supercomputer: 112 PetaFLOPS $= 112 \cdot 10^{15}$ FLOPS (floating point operations per second)
- Enumerieren aller aufspannenden Bäume dauert etwa mindestens $2,05 \cdot 10^{24}$ Sekunden $= 6,4 \cdot 10^{16}$ Jahre
- Geschätztes Alter des Universums: $\approx 1,37 \cdot 10^{10}$ Jahre

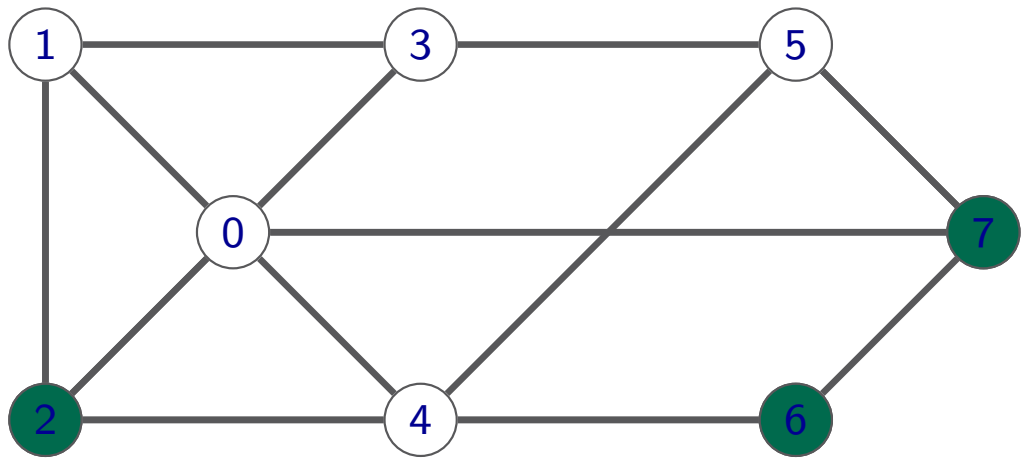
Schnitt- und Kreiseigenschaft minimaler Spannbäume

Schnitte in Graphen

Definition

Sei $G = (V, E)$ ein ungerichteter Graph und $S \subseteq V$ eine Knotenmenge.

Beispiel: $S = \{2, 6, 7\}$

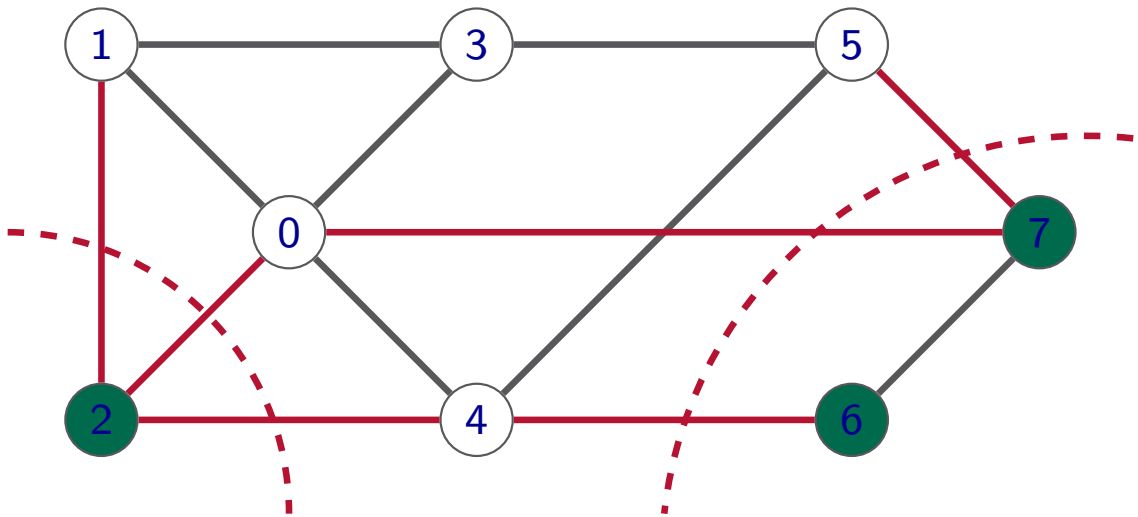


Schnitte in Graphen

Definition

Sei $G = (V, E)$ ein ungerichteter Graph und $S \subseteq V$ eine Knotenmenge. Der Schnitt von S ist die Kantenmenge $\delta(S) \subseteq E$, die genau einen Endknoten in S besitzen.

Beispiel: $S = \{2, 6, 7\}$, Schnitt $\delta(S) =$ rote Kanten

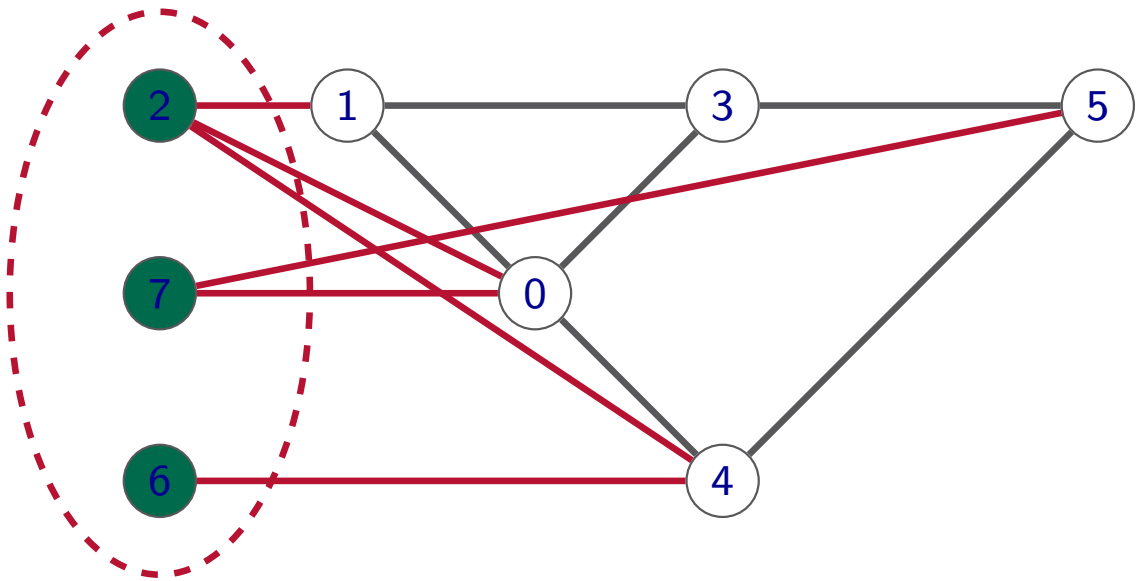


Schnitte in Graphen

Definition

Sei $G = (V, E)$ ein ungerichteter Graph und $S \subseteq V$ eine Knotenmenge. Der Schnitt von S ist die Kantenmenge $\delta(S) \subseteq E$, die genau einen Endknoten in S besitzen.

Beispiel: $S = \{2, 6, 7\}$, Schnitt $\delta(S) =$ rote Kanten

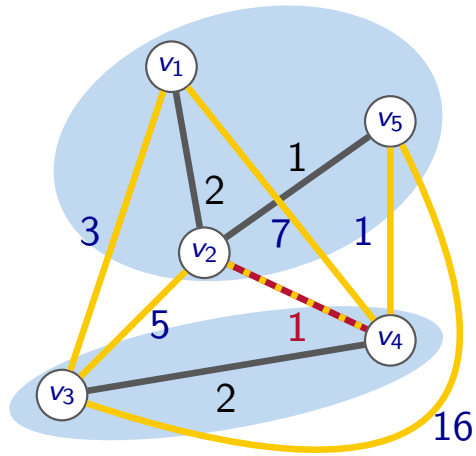


Schnitteigenschaft

Sei $G = (V, E, c)$ ein einfacher zusammenhängender Graph.

Satz (Schnitteigenschaft)

Sei $S \subset V$ und e eine Kante im Schnitt $\delta(S)$ mit minimalen Kantenkosten $c(e)$.
(1) Dann existiert ein MST, der e enthält.
(2) Sei T' eine Kantenmenge, die in einem MST enthalten ist und keine Kante aus $\delta(S)$ enthält. Dann existiert ein MST der $T' \cup \{e\}$ enthält.

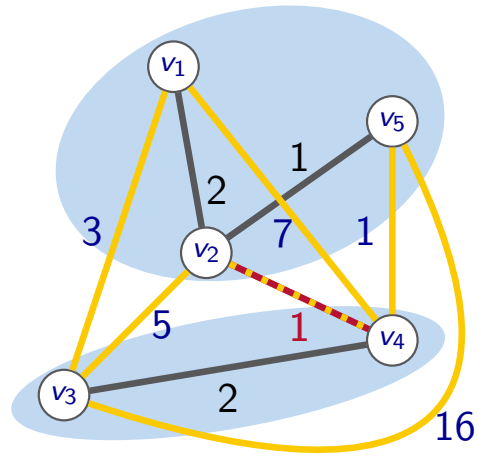


Schnitteigenschaft

Sei $G = (V, E, c)$ ein einfacher zusammenhängender Graph.

Satz (Schnitteigenschaft)

Sei $S \subset V$ und e eine Kante im Schnitt $\delta(S)$ mit minimalen Kantenkosten $c(e)$.
(1) Dann existiert ein MST, der e enthält.
(2) Sei T' eine Kantenmenge, die in einem MST enthalten ist und keine Kante aus $\delta(S)$ enthält. Dann existiert ein MST der $T' \cup \{e\}$ enthält.



Wenn $e \in \delta(S)$ eindeutig, also $c(e) < c(e'), \forall e' \in \delta(S) \setminus \{e\}$, dann liegt e in jedem MST.

Schnitteigenschaft

Beweis. Wir beweisen (2); dann folgt (1) für $T' = \emptyset$.

Schnitteigenschaft

Beweis. Wir beweisen (2); dann folgt (1) für $T' = \emptyset$.
Sei T ein MST in G mit $T' \subseteq T$, wie oben und angenommen $e \notin T$, wobei e minimale Kantenkosten im Schnitt $\delta(S)$ hat.
– Hinzufügen von e zu T schließt einen Kreis C.

Schnitteigenschaft

Beweis. Wir beweisen (2); dann folgt (1) für $T' = \emptyset$.
Sei T ein MST in G mit $T' \subseteq T$, wie oben und angenommen $e \notin T$, wobei e minimale Kantenkosten im Schnitt $\delta(S)$ hat.

- Hinzufügen von e zu T schließt einen Kreis C.
- Es existiert eine weitere Kante $f \neq e$ in T, die im Schnitt $\delta(S)$ und auf dem Kreis C liegt.

Schnitteigenschaft

Beweis. Wir beweisen (2); dann folgt (1) für $T' = \emptyset$.
Sei T ein MST in G mit $T' \subseteq T$, wie oben und angenommen $e \notin T$, wobei e minimale Kantenkosten im Schnitt $\delta(S)$ hat.

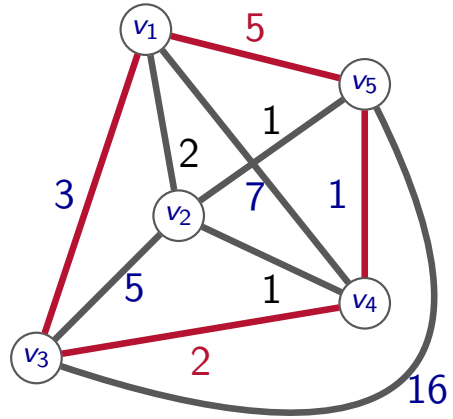
- Hinzufügen von e zu T schließt einen Kreis C.
- Es existiert eine weitere Kante $f \neq e$ in T, die im Schnitt $\delta(S)$ und auf dem Kreis C liegt. Also ist $T'' := T \setminus \{f\} \cup \{e\}$ ein aufspannender Baum. Wegen $c(e) \leq c(f)$ gilt $c(T'') \leq c(T)$. Also ist T' auch ein MST und $e \in T'$. □

Kreiseigenschaft

Sei $G = (V, E, c)$ ein einfacher zusammenhängender Graph.

Satz (Kreiseigenschaft)

Sei C ein Kreis in G und $e \in C$ eine Kante mit maximalen Kantenkosten in C. Dann existiert ein minimaler aufspannender Baum T mit $e \notin T$.



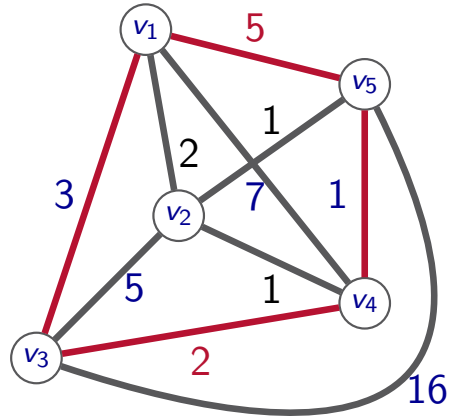
Wenn $e \in C$ eindeutig, also $c(e) > c(e'), \forall e' \in C \setminus \{e\}$, dann ist e in keinem MST enthalten.

Kreiseigenschaft

Sei $G = (V, E, c)$ ein einfacher zusammenhängender Graph.

Satz (Kreiseigenschaft)

Sei C ein Kreis in G und $e \in C$ eine Kante mit maximalen Kantenkosten in C. Dann existiert ein minimaler aufspannender Baum T mit $e \notin T$.

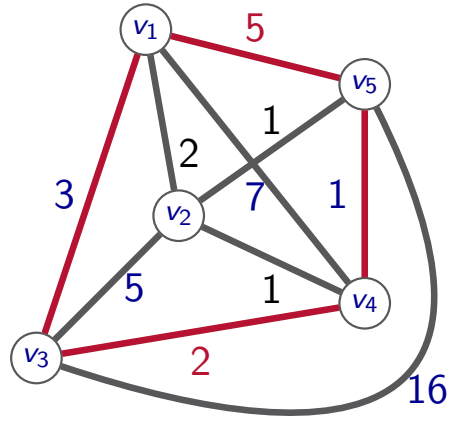


Wenn $e \in C$ eindeutig, also $c(e) > c(e'), \forall e' \in C \setminus \{e\}$, dann ist e in keinem MST enthalten.

Beweis. Sei C ein Kreis in G und $e \in C$ habe maximale Kantenkosten. Sei T ein MST in G. Angenommen $e \in T$.

Kreiseigenschaft

Sei $G = (V, E, c)$ ein einfacher zusammenhängender Graph.



Satz (Kreiseigenschaft)

Sei C ein Kreis in G und $e \in C$ eine Kante mit maximalen Kantenkosten in C. Dann existiert ein minimaler aufspannender Baum T mit $e \notin T$.

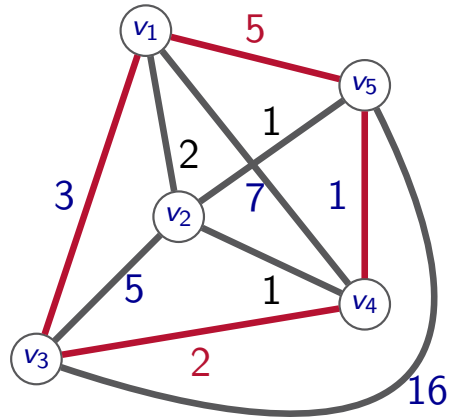
Wenn $e \in C$ eindeutig, also $c(e) > c(e'), \forall e' \in C \setminus \{e\}$, dann ist e in keinem MST enthalten.

Beweis. Sei C ein Kreis in G und $e \in C$ habe maximale Kantenkosten. Sei T ein MST in G. Angenommen $e \in T$.

– Durch Entfernen von e zerfällt T in zwei Zus.- komponenten H_1 und H_2.

Kreiseigenschaft

Sei $G = (V, E, c)$ ein einfacher zusammenhängender Graph.



Satz (Kreiseigenschaft)

Sei C ein Kreis in G und $e \in C$ eine Kante mit maximalen Kantenkosten in C. Dann existiert ein minimaler aufspannender Baum T mit $e \notin T$.

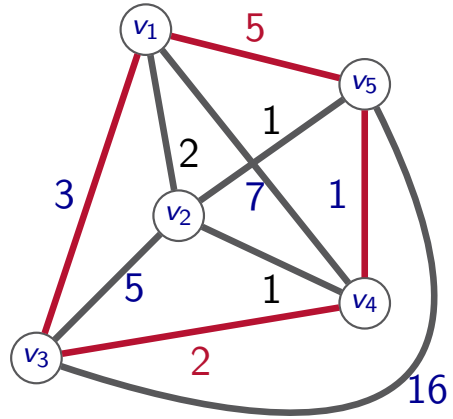
Wenn $e \in C$ eindeutig, also $c(e) > c(e'), \forall e' \in C \setminus \{e\}$, dann ist e in keinem MST enthalten.

Beweis. Sei C ein Kreis in G und $e \in C$ habe maximale Kantenkosten. Sei T ein MST in G. Angenommen $e \in T$.

- Durch Entfernen von e zerfällt T in zwei Zus.- komponenten H_1 und H_2.
- Da $e \in C$, muss eine andere Kante $f \neq e$ existieren, die im Kreis C und im Schnitt $\delta(H_1) = \delta(H_2)$ liegt.

Kreiseigenschaft

Sei $G = (V, E, c)$ ein einfacher zusammenhängender Graph.



Satz (Kreiseigenschaft)

Sei C ein Kreis in G und $e \in C$ eine Kante mit maximalen Kantenkosten in C. Dann existiert ein minimaler aufspannender Baum T mit $e \notin T$.

Wenn $e \in C$ eindeutig, also $c(e) > c(e'), \forall e' \in C \setminus \{e\}$, dann ist e in keinem MST enthalten.

Beweis. Sei C ein Kreis in G und $e \in C$ habe maximale Kantenkosten. Sei T ein MST in G. Angenommen $e \in T$.

- Durch Entfernen von e zerfällt T in zwei Zus.- komponenten H_1 und H_2.
- Da $e \in C$, muss eine andere Kante $f \neq e$ existieren, die im Kreis C und im Schnitt $\delta(H_1) = \delta(H_2)$ liegt.
- Also ist $T' := T \setminus \{e\} \cup \{f\}$ ein aufspannender Baum. Wegen $c(e) \geq c(f)$ gilt $c(T') \leq c(T)$. Also ist T' auch ein MST und $e \notin T'$. □

Greedy Algorithmen

Greedy: Triff in jedem Schritt eine lokal optimale Entscheidung.

Schnitteigenschaft

- Beginne mit leerer Kantenmenge T.
- Solange T nicht zusammenhängend in G: Wähle einen Schnitt, der keine Kante aus T enthält, und füge eine Schnittkante mit minimalen Kosten zu T hinzu.

Greedy Algorithmen

Greedy: Triff in jedem Schritt eine lokal optimale Entscheidung.

Schnitteigenschaft

- Beginne mit leerer Kantenmenge T.
- Solange T nicht zusammenhängend in G: Wähle einen Schnitt, der keine Kante aus T enthält, und füge eine Schnittkante mit minimalen Kosten zu T hinzu.

Mehrere Optionen einen Schnitt auszuwählen (Prim, Kruskal).

Greedy Algorithmen

Greedy: Triff in jedem Schritt eine lokal optimale Entscheidung.

Schnitteigenschaft

- Beginne mit leerer Kantenmenge T.
- Solange T nicht zusammenhängend in G: Wähle einen Schnitt, der keine Kante aus T enthält, und füge eine Schnittkante mit minimalen Kosten zu T hinzu.

Mehrere Optionen einen Schnitt auszuwählen (Prim, Kruskal).

Kreiseigenschaft

- Beginne mit Kantenmenge $T = E$.
- Solange T nicht kreisfrei: Wähle Kreis in T und entferne kostenmaximale Kante aus dem Kreis.

Keine effiziente Implementierung bekannt.

Algorithmus von Prim

Algorithmus von Prim (1957)

Idee (Schnitteigenschaft): Betrachte den Schnitt der durch die Endknoten der bereits ausgewählten Baumkanten T induziert wird

Algorithmus von Prim (1957)

Idee (Schnitteigenschaft): Betrachte den Schnitt der durch die Endknoten der bereits ausgewählten Baumkanten T induziert wird

($=$ billigste vom bereits konstruierten Teilbaum T ausgehende Kante, die keinen Kreis mit anderen Kanten in T schließt)

Algorithmus von Prim (1957)

Idee (Schnitteigenschaft): Betrachte den Schnitt der durch die Endknoten der bereits ausgewählten Baumkanten T induziert wird

(= billigste vom bereits konstruierten Teilbaum T ausgehende Kante, die keinen Kreis mit anderen Kanten in T schließt)

Algorithmus von Prim

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

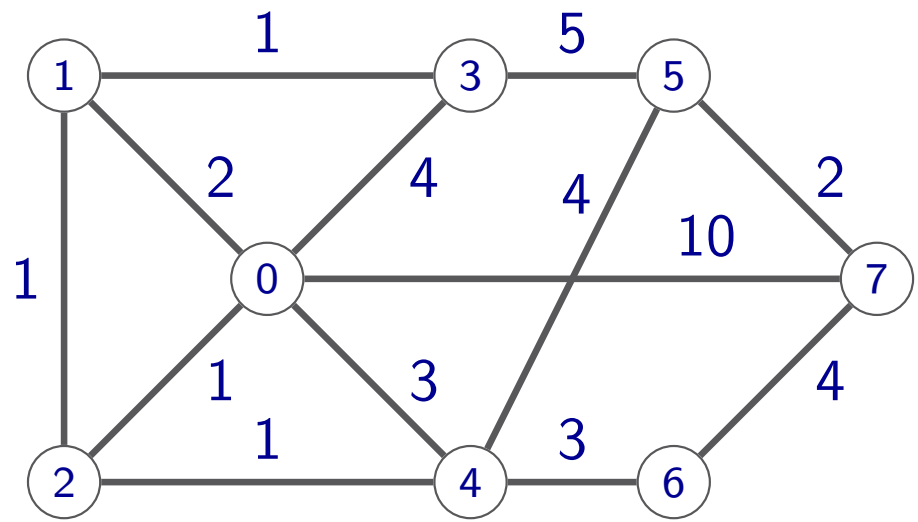
1 Setze $T = \emptyset$. (T enthält zukünftige Baumkanten)
2 Für beliebigen Startknoten s setze $S = \{s\}$.
3 **while** $|T| < |V| - 1$ **do**
4 Wähle kostenminimale Kante e aus dem Schnitt von S.
5 Füge e zu T hinzu.
6 $S \leftarrow S \cup e$
7 **return** T

Algorithmus von Prim (1957)

Algorithmus von Prim (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält zukünftige Baumkanten)
2 Für beliebigen Startknoten s setze $S = \{s\}$.
3 **while** $|T| < |V| - 1$ **do**
4 Wähle kostenminimale Kante e aus dem Schnitt von S.
5 Füge e zu T hinzu.
6 $S \leftarrow S \cup e$
7 **return** T



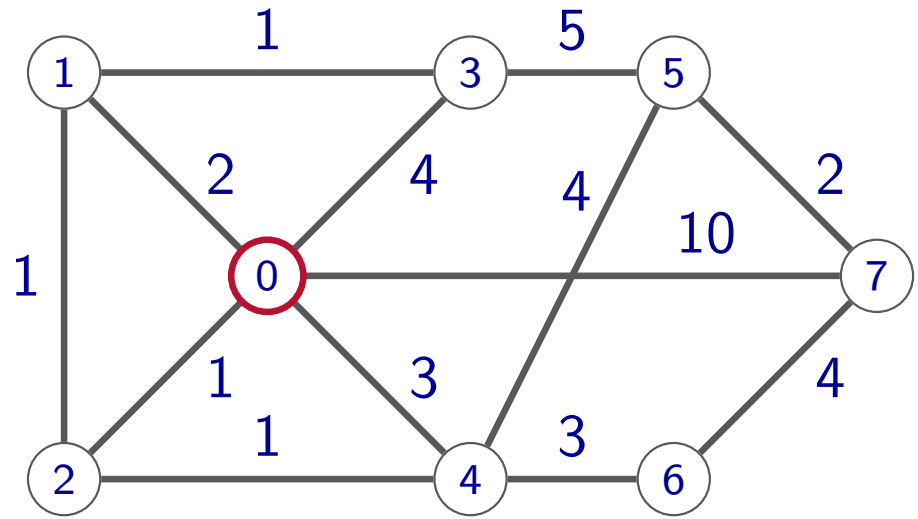
Algorithmus von Prim (1957)

Algorithmus von Prim (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält zukünftige Baumkanten)
2 Für beliebigen Startknoten s setze $S = \{s\}$.
3 **while** $|T| < |V| - 1$ **do**
4 Wähle kostenminimale Kante e aus dem Schnitt von S.
5 Füge e zu T hinzu.
6 $S \leftarrow S \cup e$
7 **return** T

$S = \{0\}$



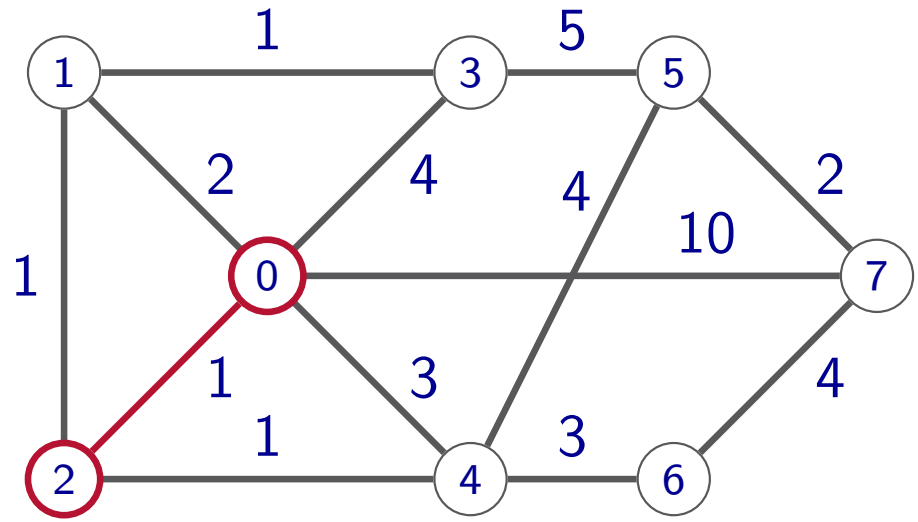
Algorithmus von Prim (1957)

Algorithmus von Prim (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält zukünftige Baumkanten)
2 Für beliebigen Startknoten s setze $S = \{s\}$.
3 **while** $|T| < |V| - 1$ **do**
4 Wähle kostenminimale Kante e aus dem Schnitt von S.
5 Füge e zu T hinzu.
6 $S \leftarrow S \cup e$
7 **return** T

$S = \{0, 2\}$



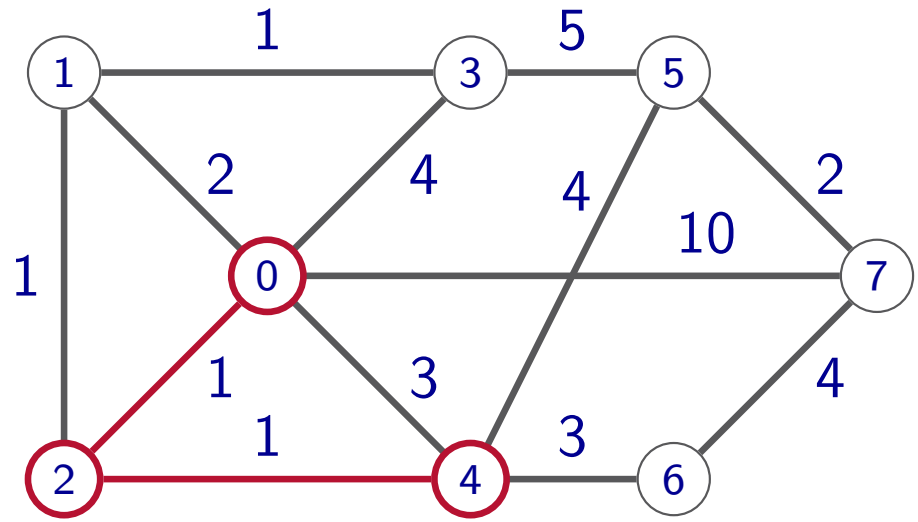
Algorithmus von Prim (1957)

Algorithmus von Prim (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält zukünftige Baumkanten)
2 Für beliebigen Startknoten s setze $S = \{s\}$.
3 **while** $|T| < |V| - 1$ **do**
4 Wähle kostenminimale Kante e aus dem Schnitt von S.
5 Füge e zu T hinzu.
6 $S \leftarrow S \cup e$
7 **return** T

$S = \{0, 2, 4\}$



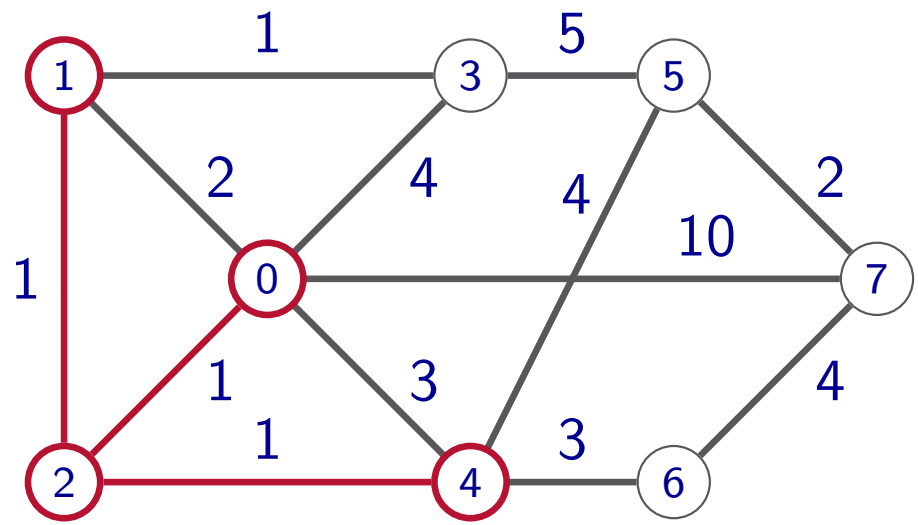
Algorithmus von Prim (1957)

Algorithmus von Prim (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält zukünftige Baumkanten)
2 Für beliebigen Startknoten s setze $S = \{s\}$.
3 **while** $|T| < |V| - 1$ **do**
4 Wähle kostenminimale Kante e aus dem Schnitt von S.
5 Füge e zu T hinzu.
6 $S \leftarrow S \cup e$
7 **return** T

$S = \{0, 2, 4, 1\}$



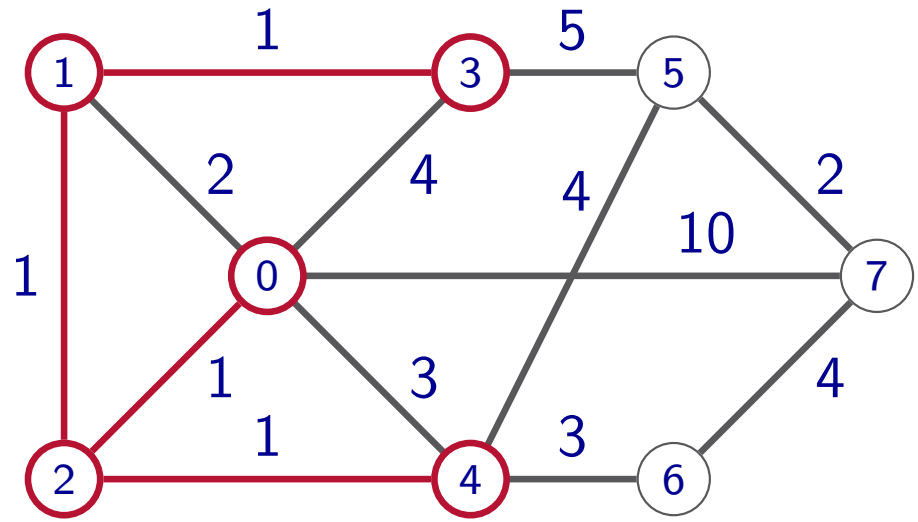
Algorithmus von Prim (1957)

Algorithmus von Prim (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält zukünftige Baumkanten)
2 Für beliebigen Startknoten s setze $S = \{s\}$.
3 **while** $|T| < |V| - 1$ **do**
4 Wähle kostenminimale Kante e aus dem Schnitt von S.
5 Füge e zu T hinzu.
6 $S \leftarrow S \cup e$
7 **return** T

$S = \{0, 2, 4, 1, 3\}$



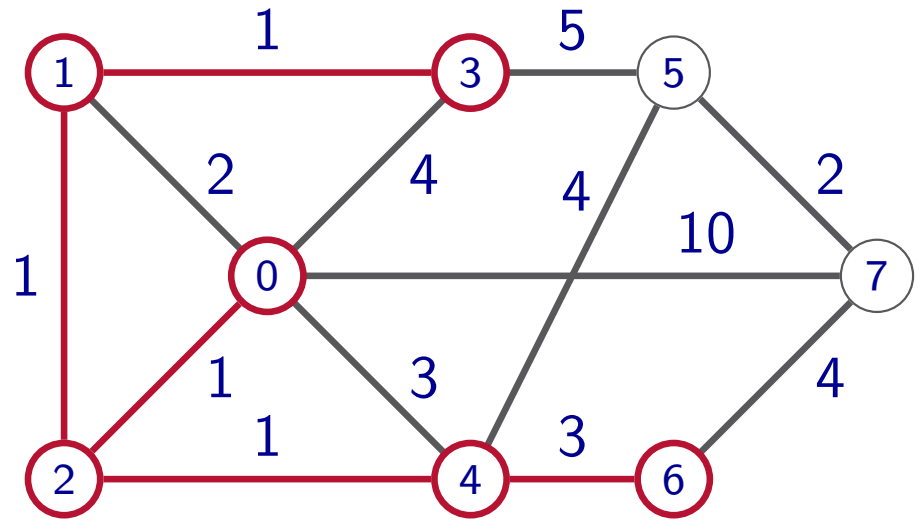
Algorithmus von Prim (1957)

Algorithmus von Prim (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält zukünftige Baumkanten)
2 Für beliebigen Startknoten s setze $S = \{s\}$.
3 **while** $|T| < |V| - 1$ **do**
4 Wähle kostenminimale Kante e aus dem Schnitt von S.
5 Füge e zu T hinzu.
6 $S \leftarrow S \cup e$
7 **return** T

$S = \{0, 2, 4, 1, 3, 6\}$



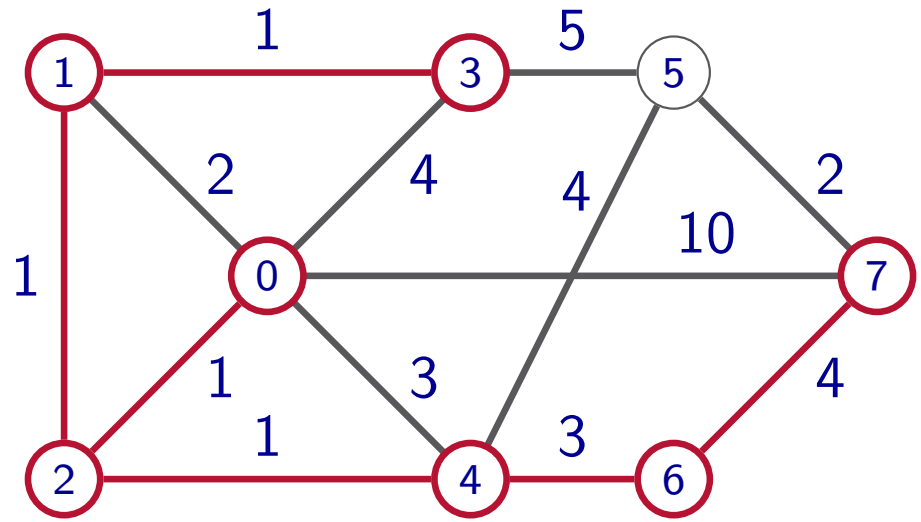
Algorithmus von Prim (1957)

Algorithmus von Prim (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält zukünftige Baumkanten)
2 Für beliebigen Startknoten s setze $S = \{s\}$.
3 **while** $|T| < |V| - 1$ **do**
4 Wähle kostenminimale Kante e aus dem Schnitt von S.
5 Füge e zu T hinzu.
6 $S \leftarrow S \cup e$
7 **return** T

$S = \{0, 2, 4, 1, 3, 6, 7\}$



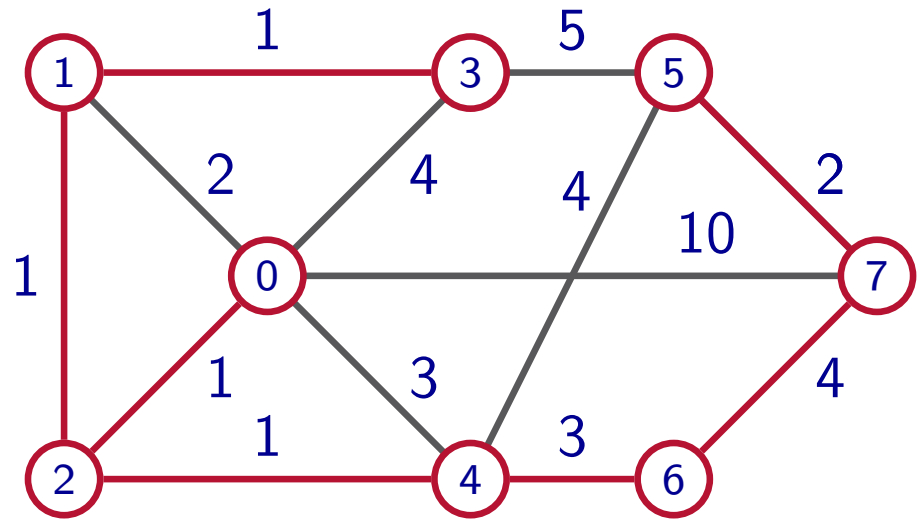
Algorithmus von Prim (1957)

Algorithmus von Prim (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält zukünftige Baumkanten)
2 Für beliebigen Startknoten s setze $S = \{s\}$.
3 **while** $|T| < |V| - 1$ **do**
4 Wähle kostenminimale Kante e aus dem Schnitt von S.
5 Füge e zu T hinzu.
6 $S \leftarrow S \cup e$
7 **return** T

$$S = \{0, 2, 4, 1, 3, 6, 7, 5\}$$



Korrektheit von Prim's Algorithmus

Satz

Sei $G = (V, E, c)$ zusammenhängend. Prim's Algorithmus liefert einen MST mit einer Laufzeit von $\mathcal{O}(n \cdot m)$.

Korrektheit von Prim's Algorithmus

Satz

Sei $G = (V, E, c)$ zusammenhängend. Prim's Algorithmus liefert einen MST mit einer Laufzeit von $\mathcal{O}(n \cdot m)$.

Beweis:

- Laufzeit: Eine kostenminimale Kante im Schnitt wird in $O(\sum_{v \in S} \deg(v)) = \mathcal{O}(m)$ gefunden und die While-Schleife wird $(n-1)$ mal durchlaufen.

Korrektheit von Prim's Algorithmus

Satz

Sei $G = (V, E, c)$ zusammenhängend. Prim's Algorithmus liefert einen MST mit einer Laufzeit von $\mathcal{O}(n \cdot m)$.

Beweis:

- Laufzeit: Eine kostenminimale Kante im Schnitt wird in $O(\sum_{v \in S} \deg(v)) = \mathcal{O}(m)$ gefunden und die While-Schleife wird $(n-1)$ mal durchlaufen.
- Korrektheit:
  - Der Algorithmus terminiert mit einem Baum (zusammenhängend, kreisfrei).
  - Jede Kante in T erfüllt die Schnitteigenschaft. □

Jarnik-Prim-Dijkstra Algorithmus

- Robert Clay Prim (geb. 1921, Amerikaner) war 1957 nicht der erste, der diesen Greedy Algorithmus für das MST-Problem beschrieben hat.

Jarnik-Prim-Dijkstra Algorithmus

- Robert Clay Prim (geb. 1921, Amerikaner) war 1957 nicht der erste, der diesen Greedy Algorithmus für das MST-Problem beschrieben hat.
- Prim's Algorithmus wurde bereits 1930 vom Tschechischen Mathematiker Vojtech Jarnik entwickelt.

Jarnik-Prim-Dijkstra Algorithmus

- Robert Clay Prim (geb. 1921, Amerikaner) war 1957 nicht der erste, der diesen Greedy Algorithmus für das MST-Problem beschrieben hat.
- Prim's Algorithmus wurde bereits 1930 vom Tschechischen Mathematiker Vojtech Jarnik entwickelt.
- Auch Dijkstra hat den Algorithmus 1959 in A Note on Two Problems in Connexion with Graphs beschrieben. (Das zweite Problem war das Kürzeste-Wege-Problem.)

Jarnik-Prim-Dijkstra Algorithmus

- Robert Clay Prim (geb. 1921, Amerikaner) war 1957 nicht der erste, der diesen Greedy Algorithmus für das MST-Problem beschrieben hat.
- Prim's Algorithmus wurde bereits 1930 vom Tschechischen Mathematiker Vojtech Jarnik entwickelt.
- Auch Dijkstra hat den Algorithmus 1959 in A Note on Two Problems in Connexion with Graphs beschrieben. (Das zweite Problem war das Kürzeste-Wege-Problem.)
- Der Algorithmus ist daher auch unter den Namen Jarnik's Algorithmus, Prim-Jarnik-, Prim-Dijkstra-Algorithmus und DJP-Algorithmus bekannt.

Detour - Binary Heaps

Kann die Laufzeit von Prim's Algorithmus verbessert werden?

Detour - Binary Heaps

Kann die Laufzeit von Prim's Algorithmus verbessert werden?

- Unser Analyse verwendet $\mathcal{O}(m)$ viele Schritte zum Finden einer Kante im Schnitt mit minimalen Kosten

Detour - Binary Heaps

Kann die Laufzeit von Prim's Algorithmus verbessert werden?

- Unser Analyse verwendet $\mathcal{O}(m)$ viele Schritte zum Finden einer Kante im Schnitt mit minimalen Kosten
- In jeder Iteration der while-Schleife starten wir eine neue Suche

Detour - Binary Heaps

Kann die Laufzeit von Prim's Algorithmus verbessert werden?

- Unser Analyse verwendet $\mathcal{O}(m)$ viele Schritte zum Finden einer Kante im Schnitt mit minimalen Kosten
- In jeder Iteration der while-Schleife starten wir eine neue Suche
- Idee: Verwalte Kanten (mit den zugehörigen Kosten) im Schnitt in geeigneter Datenstruktur

Detour - Binary Heaps

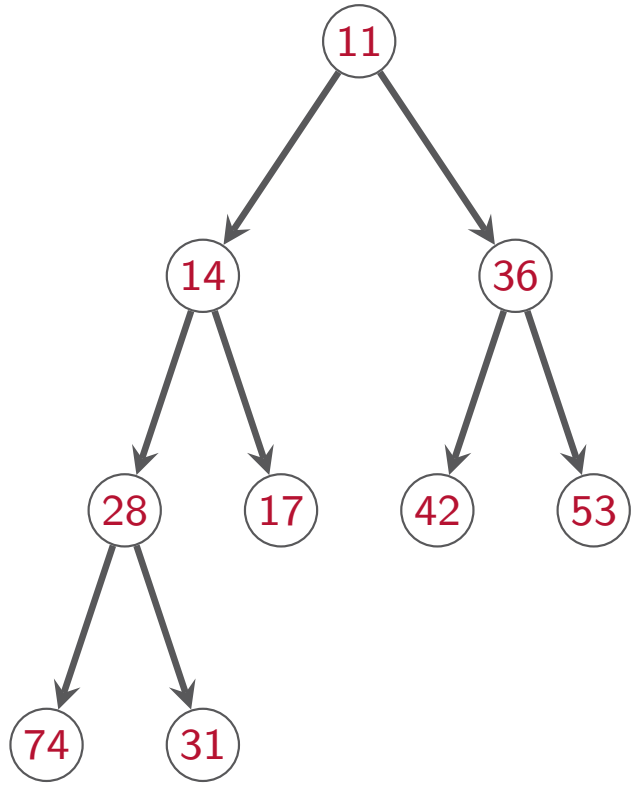
Kann die Laufzeit von Prim's Algorithmus verbessert werden?

- Unser Analyse verwendet $\mathcal{O}(m)$ viele Schritte zum Finden einer Kante im Schnitt mit minimalen Kosten
- In jeder Iteration der while-Schleife starten wir eine neue Suche
- Idee: Verwalte Kanten (mit den zugehörigen Kosten) im Schnitt in geeigneter Datenstruktur

  $\rightarrow$ Binary Heaps

Binary Heap

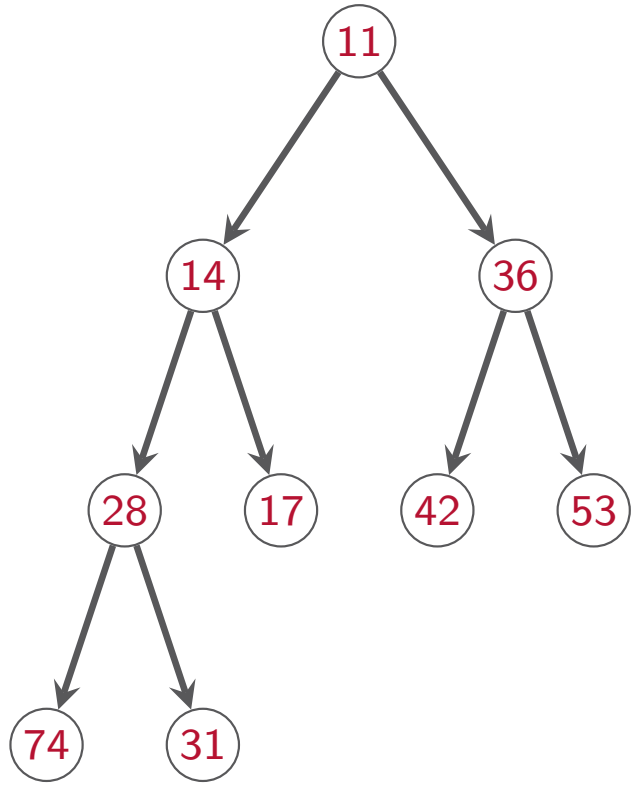
Ein Binary Heap ist eine auf Bäumen basierende Datenstruktur:



Binary Heap

Ein Binary Heap ist eine auf Bäumen basierende Datenstruktur:

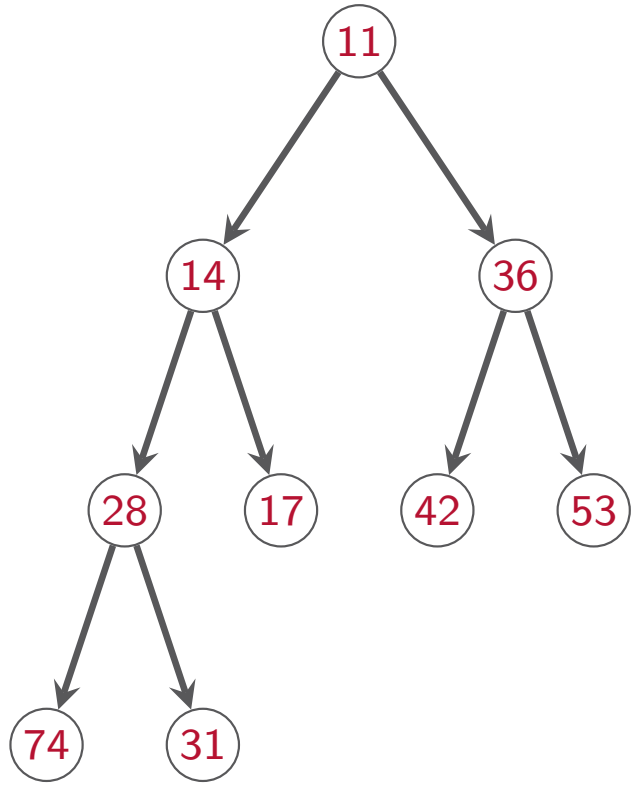
- Es gibt eine spezielle Wurzel r



Binary Heap

Ein Binary Heap ist eine auf Bäumen basierende Datenstruktur:

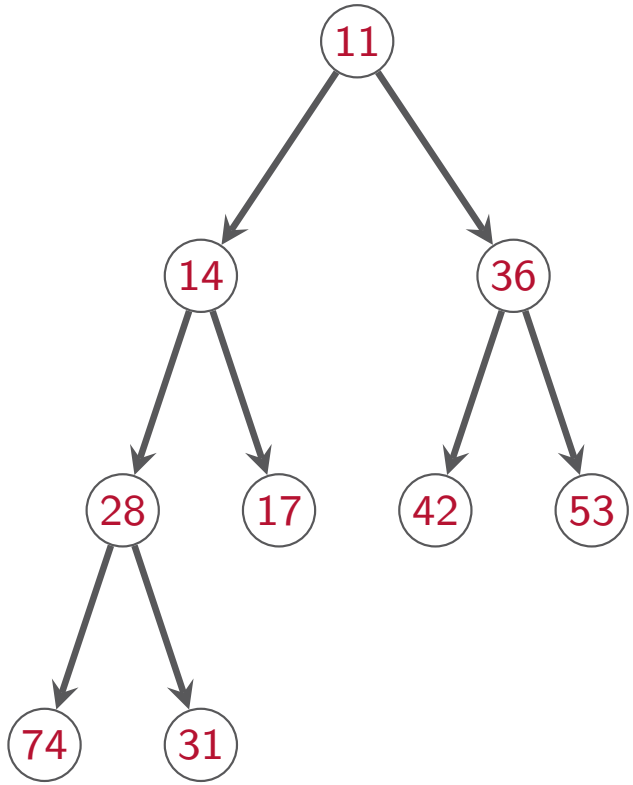
- Es gibt eine spezielle Wurzel r
- Jeder Knoten v im Baum enthält einen Wert $C[v]$



Binary Heap

Ein Binary Heap ist eine auf Bäumen basierende Datenstruktur:

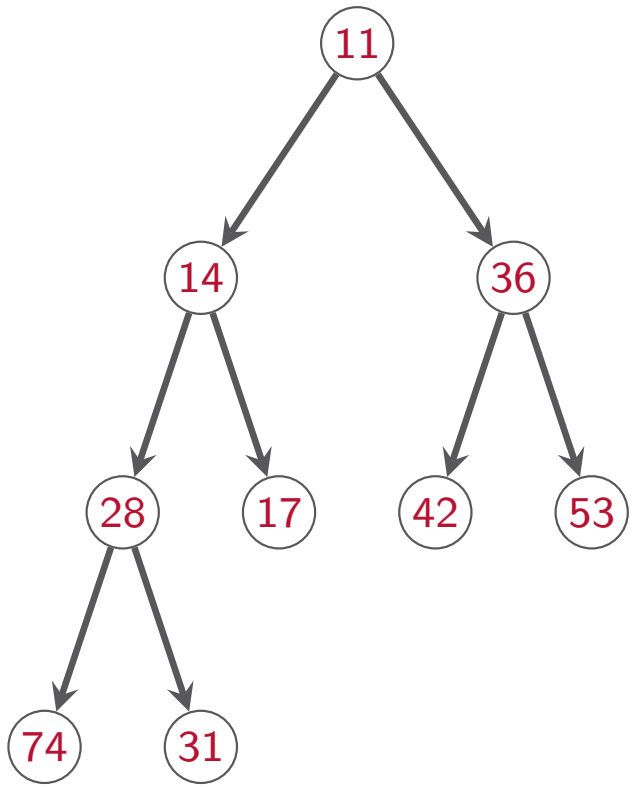
- Es gibt eine spezielle Wurzel r
- Jeder Knoten v im Baum enthält einen Wert $C[v]$
- Jeder Knoten hat genau zwei Nachfolger oder ist ein Blatt



Binary Heap

Ein Binary Heap ist eine auf Bäumen basierende Datenstruktur:

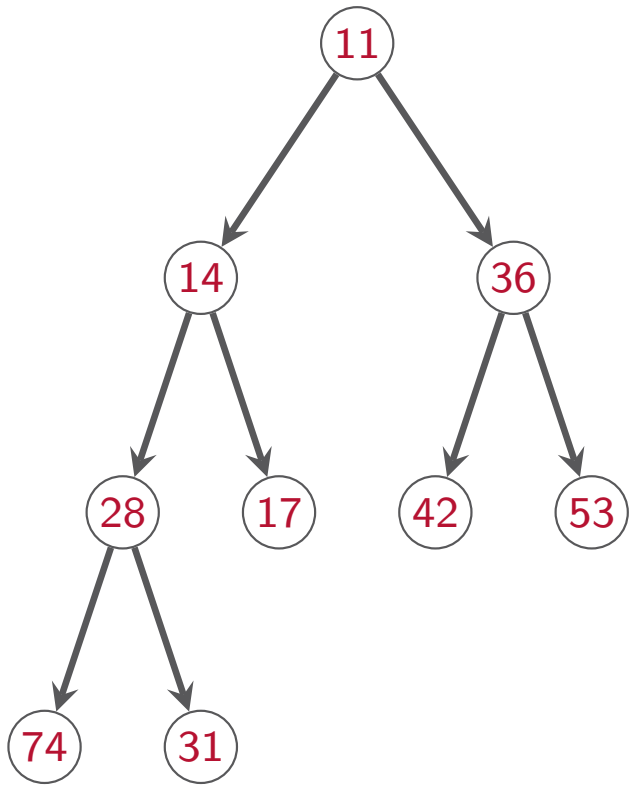
- Es gibt eine spezielle Wurzel r
- Jeder Knoten v im Baum enthält einen Wert $C[v]$
- Jeder Knoten hat genau zwei Nachfolger oder ist ein Blatt
- Nur das letzte "Level" ist nicht voll besetzt



Binary Heap

Ein Binary Heap ist eine auf Bäumen basierende Datenstruktur:

- Es gibt eine spezielle Wurzel r
- Jeder Knoten v im Baum enthält einen Wert $C[v]$
- Jeder Knoten hat genau zwei Nachfolger oder ist ein Blatt
- Nur das letzte "Level" ist nicht voll besetzt
- Das letzte "Level" ist von links besetzt

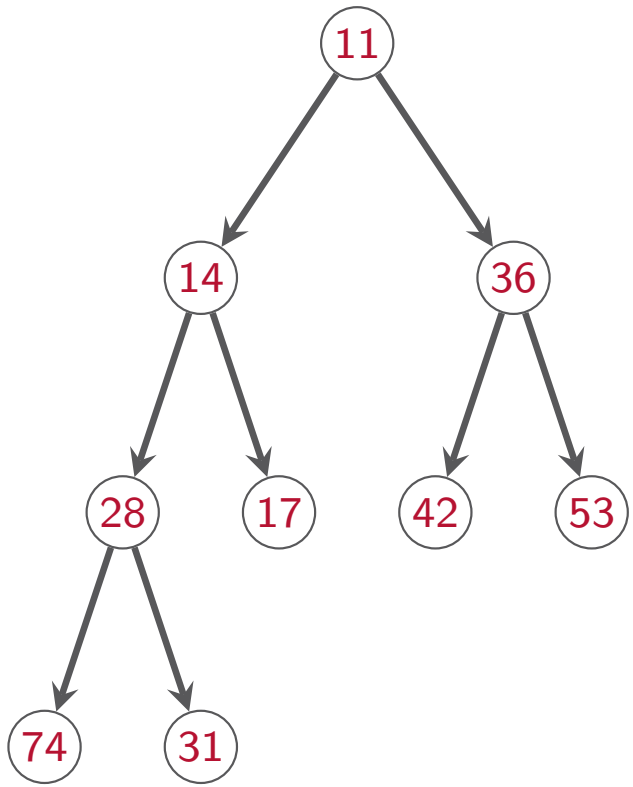


Binary Heap

Ein Binary Heap ist eine auf Bäumen basierende Datenstruktur:

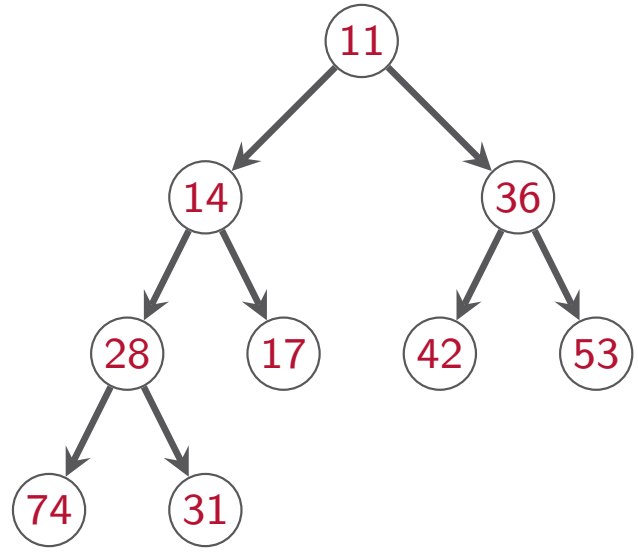
- Es gibt eine spezielle Wurzel r
- Jeder Knoten v im Baum enthält einen Wert $C[v]$
- Jeder Knoten hat genau zwei Nachfolger oder ist ein Blatt
- Nur das letzte "Level" ist nicht voll besetzt
- Das letzte "Level" ist von links besetzt
- Für jeden Knoten v mit zwei Nachfolgern w_1 und w_2 gilt

$$C[v] \leq \min(C[w_1], C[w_2])$$



Binary Heap - Repräsentation als Array

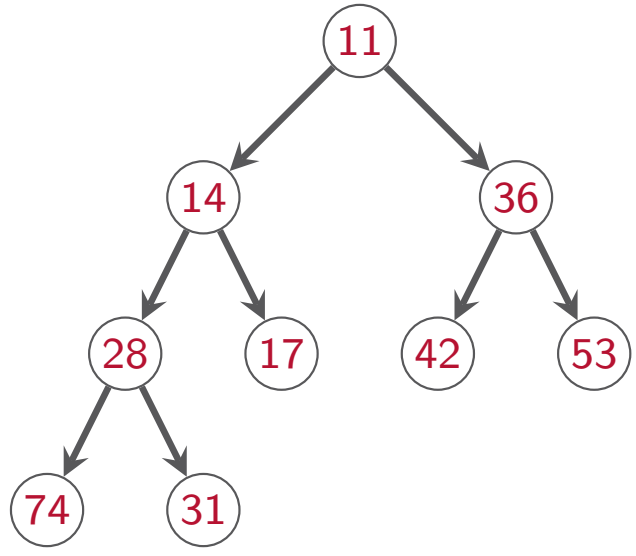
Wir können einen Binary Heap mit n Knoten als Array C der Länge n (mit Startindex 1) abspeichern:



Binary Heap - Repräsentation als Array

Wir können einen Binary Heap mit n Knoten als Array C der Länge n (mit Startindex 1) abspeichern:

- Wurzel an Position 1

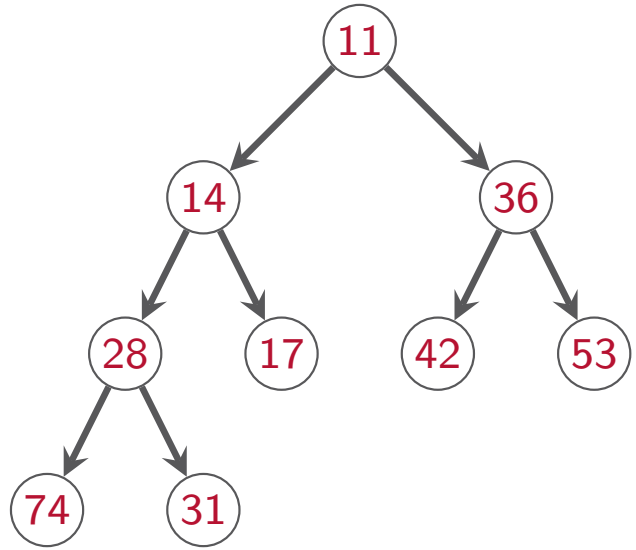


11	14	36	28	17	42	53	74	31
1	2	3	4	5	6	7	8	9

Binary Heap - Repräsentation als Array

Wir können einen Binary Heap mit n Knoten als Array C der Länge n (mit Startindex 1) abspeichern:

- Wurzel an Position 1
- Nachfolger von Knoten i sind an Positionen $2i$ und $2i + 1$



11	14	36	28	17	42	53	74	31
1	2	3	4	5	6	7	8	9

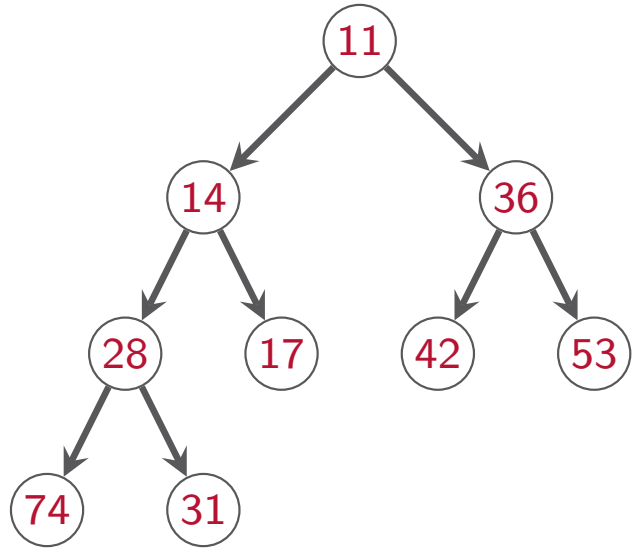
Binary Heap - Repräsentation als Array

Wir können einen Binary Heap mit n Knoten als Array C der Länge n (mit Startindex 1) abspeichern:

- Wurzel an Position 1
- Nachfolger von Knoten i sind an Positionen $2i$ und $2i+1$
- Für jedes $i \in \{1, \ldots, \lfloor n/2 \rfloor\}$ gilt

$$C[i] \leq \min(C[2i], C[2i+1])$$

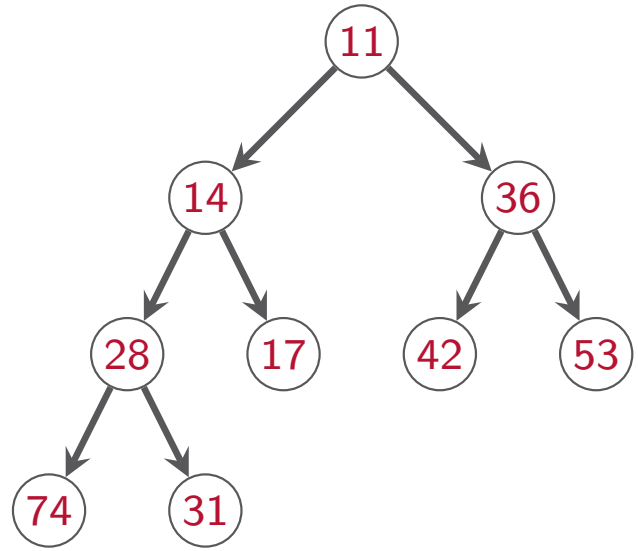
(falls $C[2i+1]$ nicht existiert, nehmen wir es als ∞ an)



11	14	36	28	17	42	53	74	31
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `FindMin`

Die Operation `FindMin` gibt das minimale Element im Binary Heap zurück.

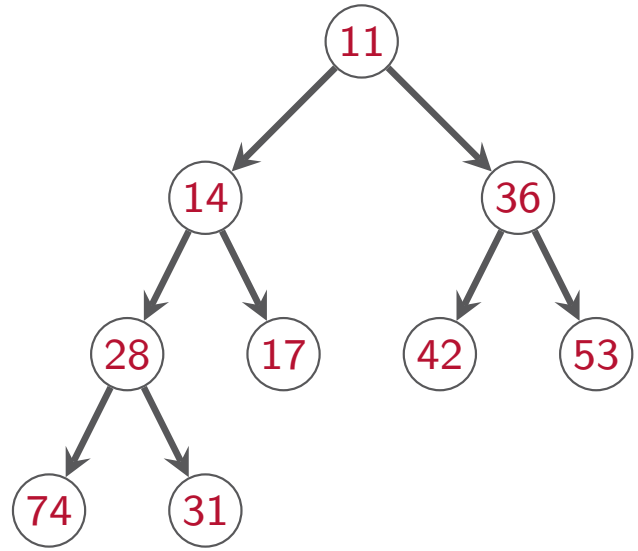


11	14	36	28	17	42	53	74	31
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `FindMin`

Die Operation `FindMin` gibt das minimale Element im Binary Heap zurück.

- Minimales Element ist immer an Position 1, wir geben also einfach $C[1]$ zurück

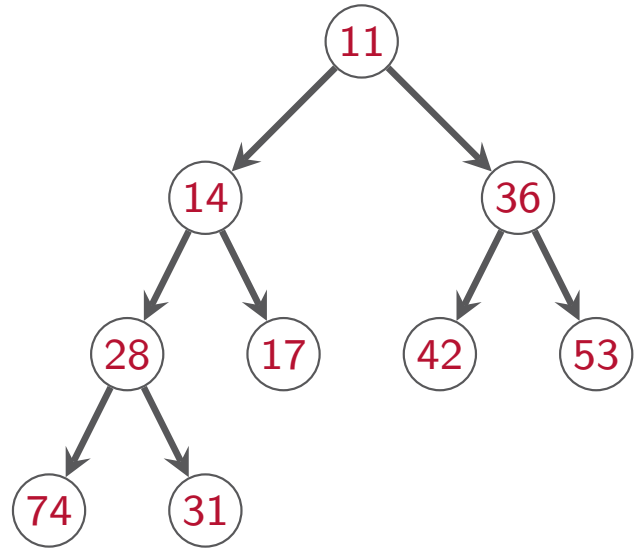


11	14	36	28	17	42	53	74	31
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `FindMin`

Die Operation `FindMin` gibt das minimale Element im Binary Heap zurück.

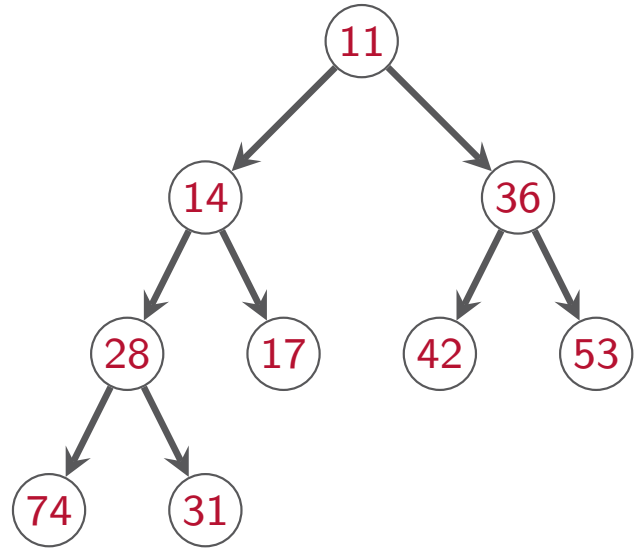
- Minimales Element ist immer an Position 1, wir geben also einfach $C[1]$ zurück
- Laufzeit: $\mathcal{O}(1)$



11	14	36	28	17	42	53	74	31
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

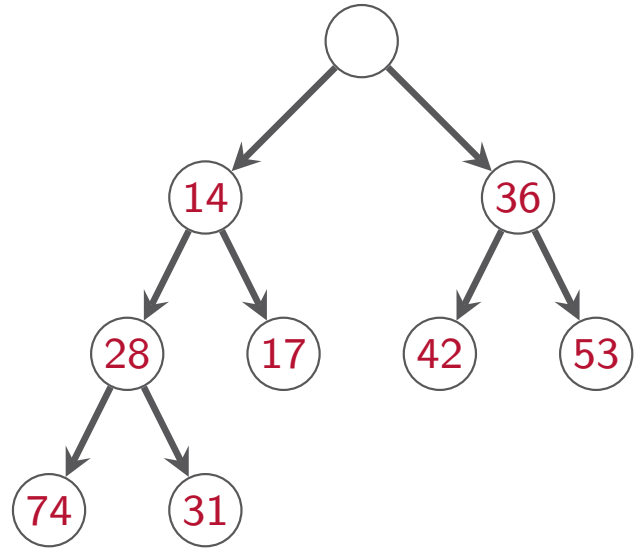
Die Operation `Delete` löscht das Element an Position (oder Knoten) i.



11	14	36	28	17	42	53	74	31
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

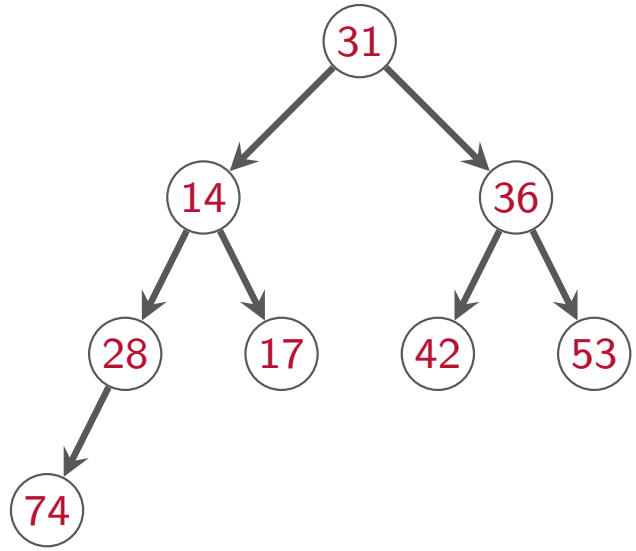


	14	36	28	17	42	53	74	31
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

- Vertausche $C[i]$ und $C[n]$ und entferne das letzte Element im Array

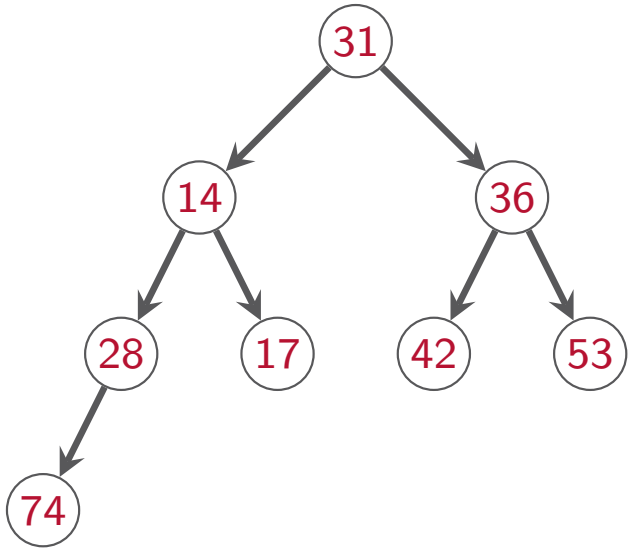


31	14	36	28	17	42	53	74	
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

- Vertausche $C[i]$ und $C[n]$ und entferne das letzte Element im Array
- Wir müssen die Heap-Eigenschaft wiederherstellen

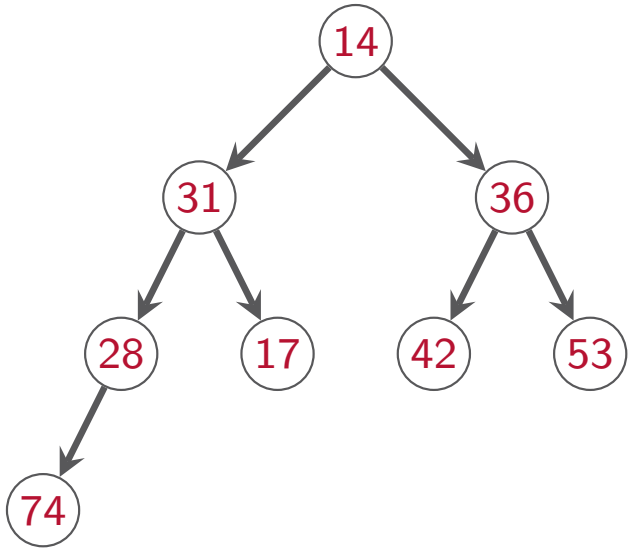


31	14	36	28	17	42	53	74	
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

- Vertausche $C[i]$ und $C[n]$ und entferne das letzte Element im Array
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Falls $C[i] > \min(C[2i], C[2i+1])$, dann Vertausche $C[i]$ mit dem kleineren der beiden Elemente

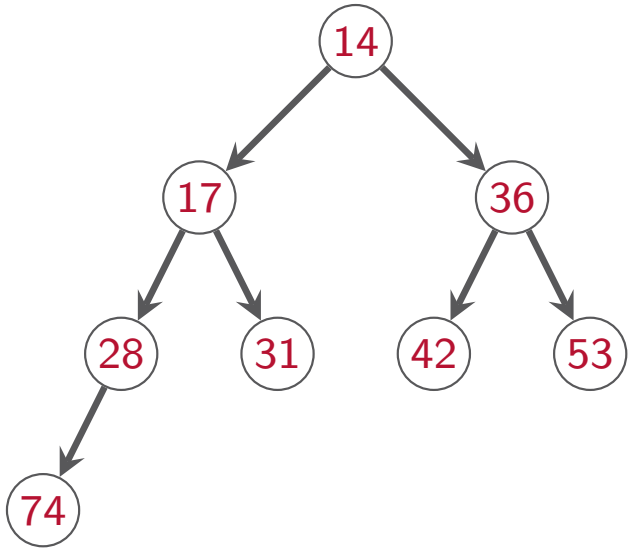


14	31	36	28	17	42	53	74	
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

- Vertausche $C[i]$ und $C[n]$ und entferne das letzte Element im Array
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Falls $C[i] > \min(C[2i], C[2i+1])$, dann Vertausche $C[i]$ mit dem kleineren der beiden Elemente
- Wiederhole rekursiv für das entsprechende Kind

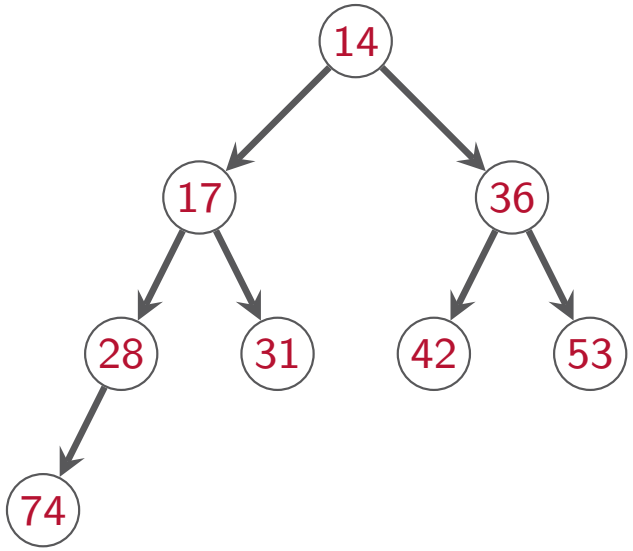


14	17	36	28	31	42	53	74	
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

- Vertausche $C[i]$ und $C[n]$ und entferne das letzte Element im Array
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Falls $C[i] > \min(C[2i], C[2i+1])$, dann Vertausche $C[i]$ mit dem kleineren der beiden Elemente
- Wiederhole rekursiv für das entsprechende Kind
- Laufzeit: $\mathcal{O}(\log n)$

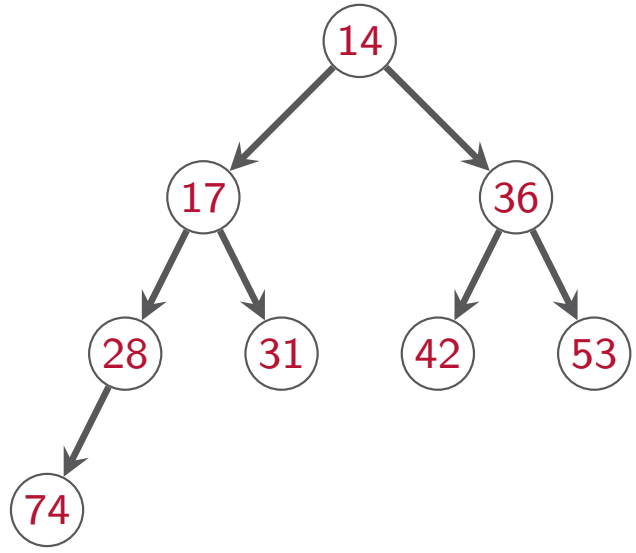


14	17	36	28	31	42	53	74	
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Insert`

Die Operation `Insert` fügt ein neues Element mit Wert c hinzu.

Annahme: Das Array hat mindestens eine leere Position am Ende.



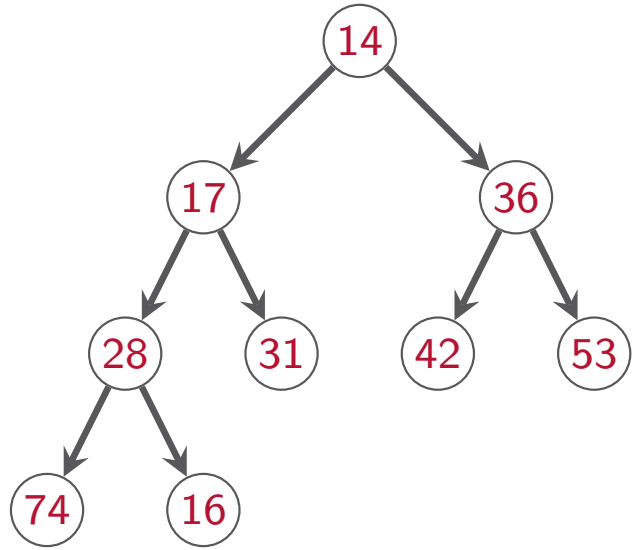
14	17	36	28	31	42	53	74	
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Insert`

Die Operation `Insert` fügt ein neues Element mit Wert c hinzu.

Annahme: Das Array hat mindestens eine leere Position am Ende.

- Setze $C[n+1] := c$



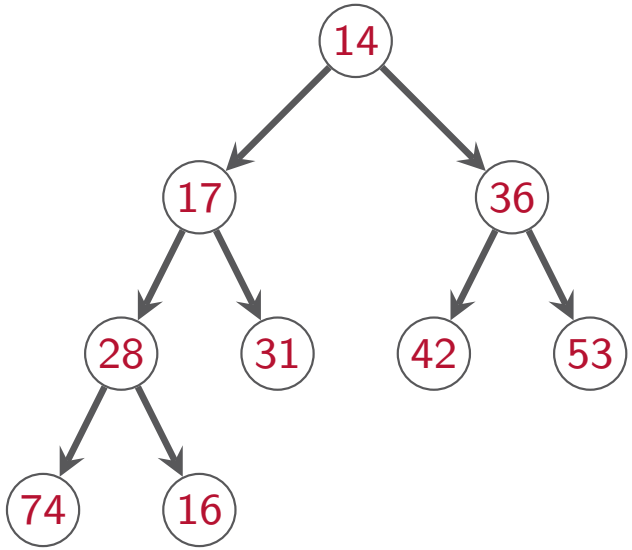
14	17	36	28	31	42	53	74	16
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Insert`

Die Operation `Insert` fügt ein neues Element mit Wert c hinzu.

Annahme: Das Array hat mindestens eine leere Position am Ende.

- Setze $C[n+1] := c$
- Wir müssen die Heap-Eigenschaft wiederherstellen



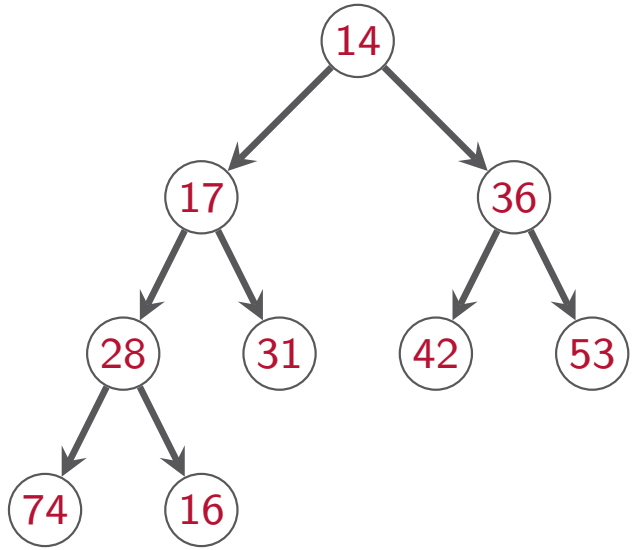
14	17	36	28	31	42	53	74	16
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Insert`

Die Operation `Insert` fügt ein neues Element mit Wert c hinzu.

Annahme: Das Array hat mindestens eine leere Position am Ende.

- Setze $C[n+1] := c$
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Sei $i := \lfloor (n+1)/2 \rfloor$ der Vorgänger



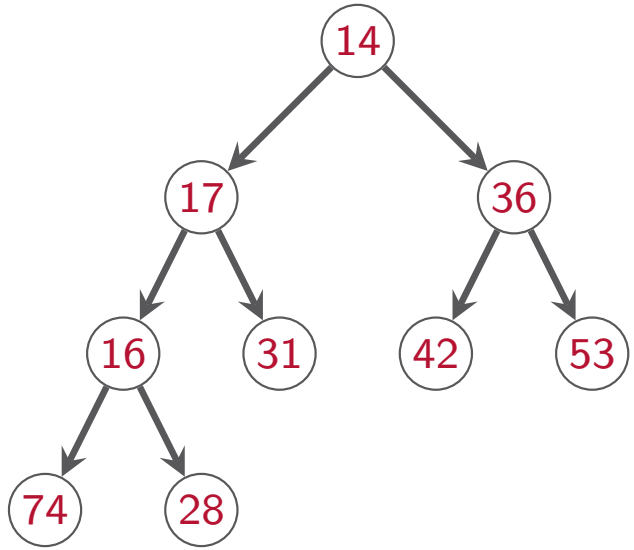
14	17	36	28	31	42	53	74	16
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Insert`

Die Operation `Insert` fügt ein neues Element mit Wert c hinzu.

Annahme: Das Array hat mindestens eine leere Position am Ende.

- Setze $C[n+1] := c$
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Sei $i := \lfloor (n+1)/2 \rfloor$ der Vorgänger
- Falls $C[n+1] < C[i]$, dann Vertausche $C[n+1]$ und $C[i]$



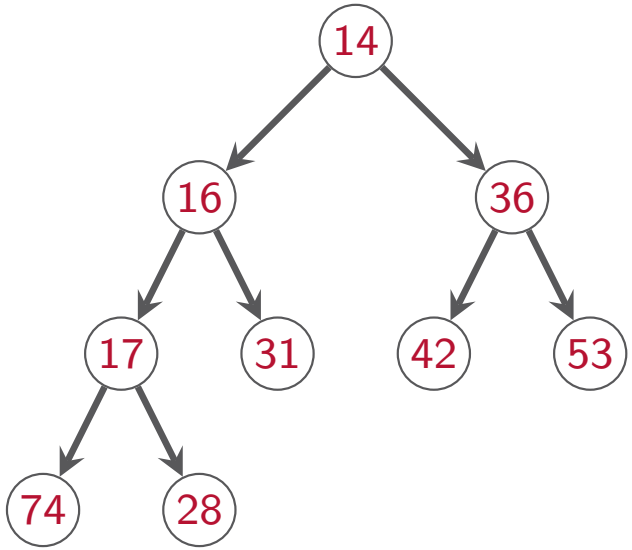
14	17	36	16	31	42	53	74	28
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Insert`

Die Operation `Insert` fügt ein neues Element mit Wert c hinzu.

Annahme: Das Array hat mindestens eine leere Position am Ende.

- Setze $C[n+1] := c$
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Sei $i := \lfloor (n+1)/2 \rfloor$ der Vorgänger
- Falls $C[n+1] < C[i]$, dann Vertausche $C[n+1]$ und $C[i]$
- Wiederhole rekursiv für Position i



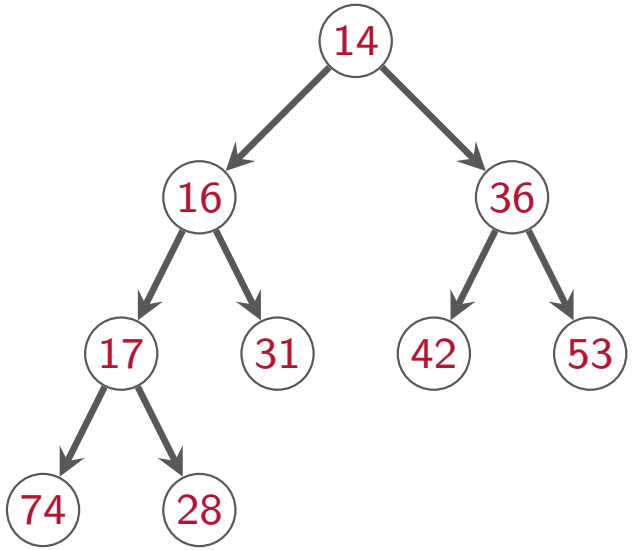
14	16	36	17	31	42	53	74	28
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Insert`

Die Operation `Insert` fügt ein neues Element mit Wert c hinzu.

Annahme: Das Array hat mindestens eine leere Position am Ende.

- Setze $C[n+1] := c$
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Sei $i := \lfloor (n+1)/2 \rfloor$ der Vorgänger
- Falls $C[n+1] < C[i]$, dann Vertausche $C[n+1]$ und $C[i]$
- Wiederhole rekursiv für Position i
- Laufzeit: $\mathcal{O}(\log n)$



14	16	36	17	31	42	53	74	28
1	2	3	4	5	6	7	8	9

Binary Heap - Zusammenfassung

Mit einem Binary Heap können wir eine Menge von Werten verwalten und die folgenden Operationen durchführen:

- `FindMin` - gebe das minimale Element zurück $\mathcal{O}(1)$
- `Delete` - entferne Element an Position i $\mathcal{O}(\log n)$
- `Insert` - füge neues Element mit Wert c hinzu $\mathcal{O}(\log n)$

Beachte: Die Werte können beliebige Objekte sein, solange wir sie miteinander vergleichen können (z.B. gewichtete Kanten in einem Graphen)

Algorithmus von Prim - Verbesserte Analyse

Algorithmus von Prim

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

```
1 Setze T = ∅.                    (T enthält zukünftige Baumkanten)
2 Für beliebigen Startknoten s setze S = {s}.
3 while |T| < |V| − 1 do
4     Wähle kostenminimale Kante e aus dem Schnitt von S.
5     Füge e zu T hinzu.
6     S ← S ∪ e
7 return T
```

Idee: Verwalte Kanten im Schnitt mit einem Binary Heap

Algorithmus von Prim - Verbesserte Analyse

Satz

Sei $G = (V, E, c)$ zusammenhängend. Prim's Algorithmus liefert einen MST mit einer Laufzeit von $\mathcal{O}((n + m) \log n)$.

Algorithmus von Prim - Verbesserte Analyse

Satz

Sei $G = (V, E, c)$ zusammenhängend. Prim's Algorithmus liefert einen MST mit einer Laufzeit von $\mathcal{O}((n + m) \log n)$.

Wir analysieren die Laufzeit erneut, wobei wir den Schnitt von S als Binary Heap verwalten:

- die While-Schleife wird $(n - 1)$ mal durchlaufen

Algorithmus von Prim - Verbesserte Analyse

Satz

Sei $G = (V, E, c)$ zusammenhängend. Prim's Algorithmus liefert einen MST mit einer Laufzeit von $\mathcal{O}((n+m)\log n)$.

Wir analysieren die Laufzeit erneut, wobei wir den Schnitt von S als Binary Heap verwalten:

- die While-Schleife wird $(n-1)$ mal durchlaufen
- wir initialisieren leeren Heap mit einem Array der Länge m in Zeit $\mathcal{O}(m)$

Algorithmus von Prim - Verbesserte Analyse

Satz

Sei $G = (V, E, c)$ zusammenhängend. Prim's Algorithmus liefert einen MST mit einer Laufzeit von $\mathcal{O}((n + m) \log n)$.

Wir analysieren die Laufzeit erneut, wobei wir den Schnitt von S als Binary Heap verwalten:

- die While-Schleife wird $(n - 1)$ mal durchlaufen
- wir initialisieren leeren Heap mit einem Array der Länge m in Zeit $\mathcal{O}(m)$
- eine kostenminimale Kante kann nun in Zeit $\mathcal{O}(1)$ gefunden werden

Algorithmus von Prim - Verbesserte Analyse

Satz

Sei $G = (V, E, c)$ zusammenhängend. Prim's Algorithmus liefert einen MST mit einer Laufzeit von $\mathcal{O}((n+m)\log n)$.

Wir analysieren die Laufzeit erneut, wobei wir den Schnitt von S als Binary Heap verwalten:

- die While-Schleife wird $(n-1)$ mal durchlaufen
- wir initialisieren leeren Heap mit einem Array der Länge m in Zeit $\mathcal{O}(m)$
- eine kostenminimale Kante kann nun in Zeit $\mathcal{O}(1)$ gefunden werden
- nach jedem Update $S \leftarrow S \cup e$ passen wir den Binary Heap an

Algorithmus von Prim - Verbesserte Analyse

Satz

Sei $G = (V, E, c)$ zusammenhängend. Prim's Algorithmus liefert einen MST mit einer Laufzeit von $\mathcal{O}((n + m) \log n)$.

Wir analysieren die Laufzeit erneut, wobei wir den Schnitt von S als Binary Heap verwalten:

- die While-Schleife wird $(n - 1)$ mal durchlaufen
- wir initialisieren leeren Heap mit einem Array der Länge m in Zeit $\mathcal{O}(m)$
- eine kostenminimale Kante kann nun in Zeit $\mathcal{O}(1)$ gefunden werden
- nach jedem Update $S \leftarrow S \cup e$ passen wir den Binary Heap an
- jede Kante wird höchstens einmal zum Binary Heap hinzugefügt und höchstens einmal aus dem Binary Heap entfernt

Algorithmus von Prim - Verbesserte Analyse

Satz

Sei $G = (V, E, c)$ zusammenhängend. Prim's Algorithmus liefert einen MST mit einer Laufzeit von $\mathcal{O}((n+m)\log n)$.

Wir analysieren die Laufzeit erneut, wobei wir den Schnitt von S als Binary Heap verwalten:

- die While-Schleife wird $(n-1)$ mal durchlaufen
- wir initialisieren leeren Heap mit einem Array der Länge m in Zeit $\mathcal{O}(m)$
- eine kostenminimale Kante kann nun in Zeit $\mathcal{O}(1)$ gefunden werden
- nach jedem Update $S \leftarrow S \cup e$ passen wir den Binary Heap an
- jede Kante wird höchstens einmal zum Binary Heap hinzugefügt und höchstens einmal aus dem Binary Heap entfernt

  $\rightarrow \mathcal{O}(n + m + m\log m) = \mathcal{O}((n+m)\log(n^2)) = \mathcal{O}((n+m)\log(n))$

Zusammenfassung

Eulertouren

- Hierholzer Algorithmus in $\mathcal{O}(n+m)$

Minimale Aufspannende Bäume (MST) Problem

- Schnitteigenschaft
- Der Algorithmen von Prim berechnet einen MST
- Binary Heap für bessere Laufzeit $\mathcal{O}((n+m)\log(n))$