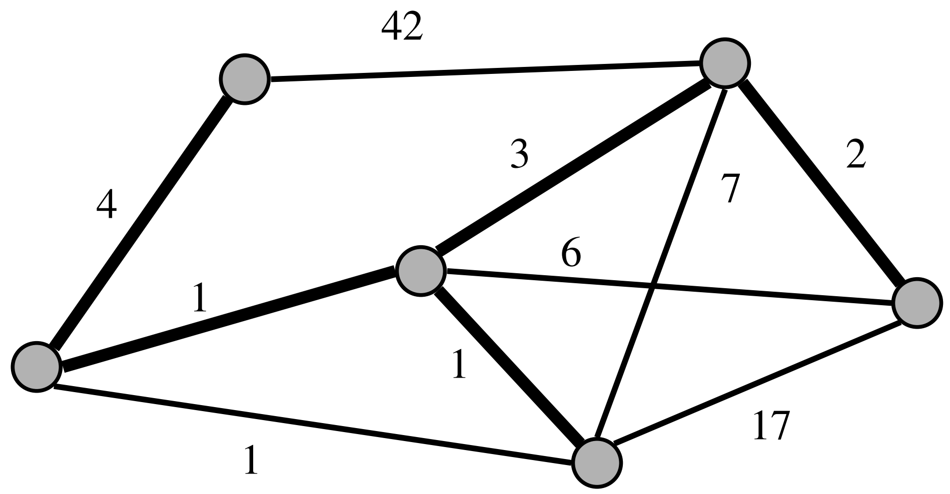
Minimale Spannbäume

Überblick:

- Was ist ein minimaler Spannbaum (MST)?
- Algorithmus von Kruskal
  - Idee, Korrektheit
  - Implementierung / Datenstrukturen
- Algorithmus von Prim
  - Idee, Korrektheit
  - Implementierung / Datenstrukturen

MST (Definition)

- Geg.:
  - zshg. Graph $G=(V,E)$; $|V|=n$, $|E|=m$
  - Kostenfunktion $c: E \rightarrow \Re^+$
- Ges.: „billigster“ Baum der alle Knoten enthält
- also $T=(V,E')$ mit $\sum_{e \in E'} c(e)$ min.



Kruskal: Idee

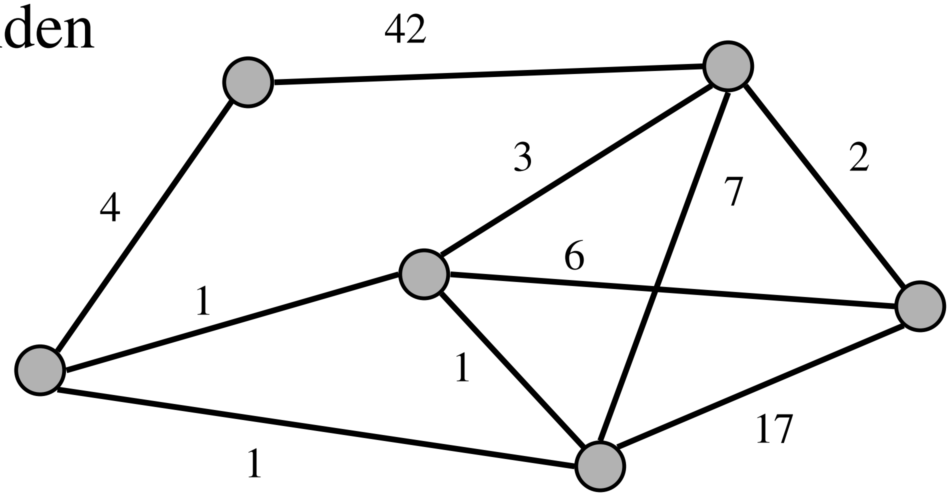
- Greedy-Vorgehen:

  immer die billigste Kante hinzufügen, falls möglich
- Initialisierung von *T*:

  Wald mit *n* Teilbäumen, sprich alle Knoten, keine Kanten
- betrachte Kanten in aufsteigender Reihenfolge

  1. Fall: Kante zw. Knoten im gleichen Teilbaum: verwerfen

  2. Fall: verschiedene Teilbäume: Bäume mit dieser Kante verbinden

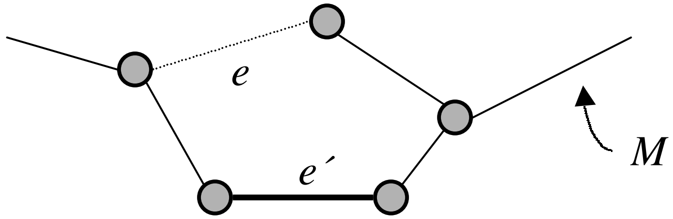


Kruskal: Korrektheit

- Zeigen: es gibt zu jedem Zeitpunkt einem MST $M=(V,E')$ der alle aktuellen Teilbäume T enthält
- Ind. über eingefügte Kanten:
  - IA: klar, bel. MST enthält alle Knoten
  - IS (i~>i+1): i Kanten wurden in T eingefügt, $T \subset M$
    e ist die nächste Kante

    1. Fall: $e \in E' \Rightarrow$ das erweiterte T ist immer noch in M

    2. Fall: $e \notin E' \Rightarrow$ in $E' \cup \{e\}$ existiert ein Kreis C



- mind. Eine Kante e' auf C ist nicht in T
- $c(e') \geq c(e)$, da sonst e' vor e gewählt würde
- $\Rightarrow M' \equiv (V, (E' \setminus \{e'\}) \cup \{e\})$ ist auch ein MST

Kruskal: Implementierg., Datenstrukt.

- Abarb. der Kanten in aufsteigender Reihenfolge
  1. Alt.: anfangs sortieren ~> $O(m \log m) = O(m \log n^2) = O(m \log n)$
  2. Alt.: Priority-Queue (Laufzeitangaben bzgl. Fib Heaps):
     - insert(e,p): e mit Priorität p einfügen ~> $O(1)$ amortisiert
     - decreasePriority(e,p): Priorität auf p verringern ~> $O(1)$ amort.
     - delMin(): kleinstes Element entnehmen ~> $O(\log n)$ amort.

     anfangs jede Kante e mit $c(e)$ als Prio. einfügen ~> $O(m)$

     Iter.: nächste Kante mit delMin() ~> $O(\log n)$ pro Kante
- 2. Alternative in der Praxis oft besser, da meist nicht alle Kanten betrachtet werden
- LEDA: `p_queue` oder `edge_pq`

Kruskal: Implementierg., Datenstrukt.

- Teilbäume partitionieren V in disj. Mengen
- Partitionen mit Union/Find Datenstrukt verwalten:
  - find(x): in welcher Partition ist x
  - union(x,y): vereinige Mengen die x bzw. y enthalten

  LZ von m union/find-Op. auf n Elementen: $O(m\alpha(m,n))$
- LEDA: `partition` oder `node_partition`
- Sortierung der Kanten dominiert Gesamt-LZ

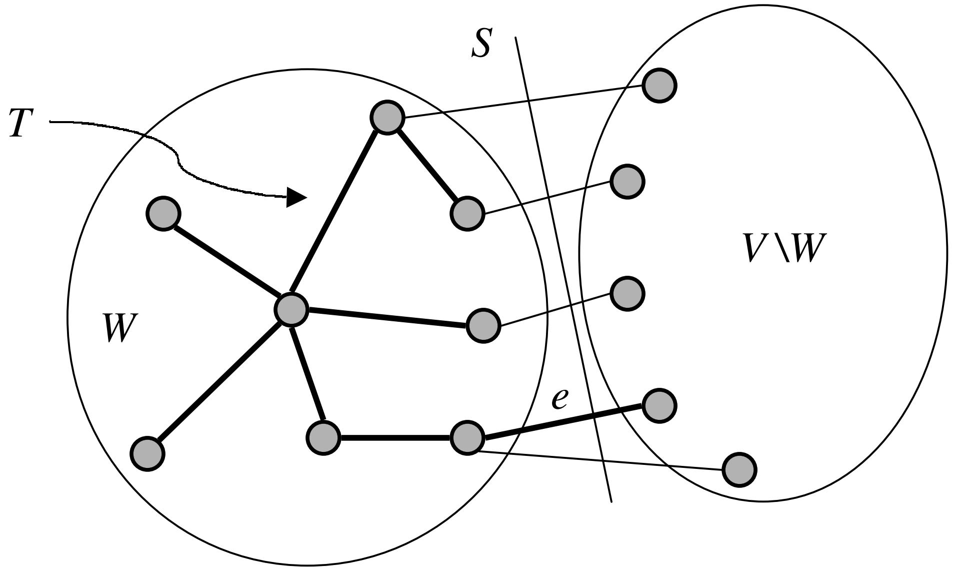
  ~> $O(m \log n)$

Prim: Idee

- Ausgehend von einem Knoten einen Teilbaum $T=(W,E')$ schrittweise zu MST erweitern
- Initialisierung: $W=v$, $E'=\{\}$
- Iteration:

  S := Kanten im Schnitt zw. W und $V\setminus W$

  übernehme Kante $e \in S$ mit min. Kosten, erweitere W



Prim: Korrektheit

Zu jedem Zeitpkt. gibt es einen MST M der T enth.

– IA: klar

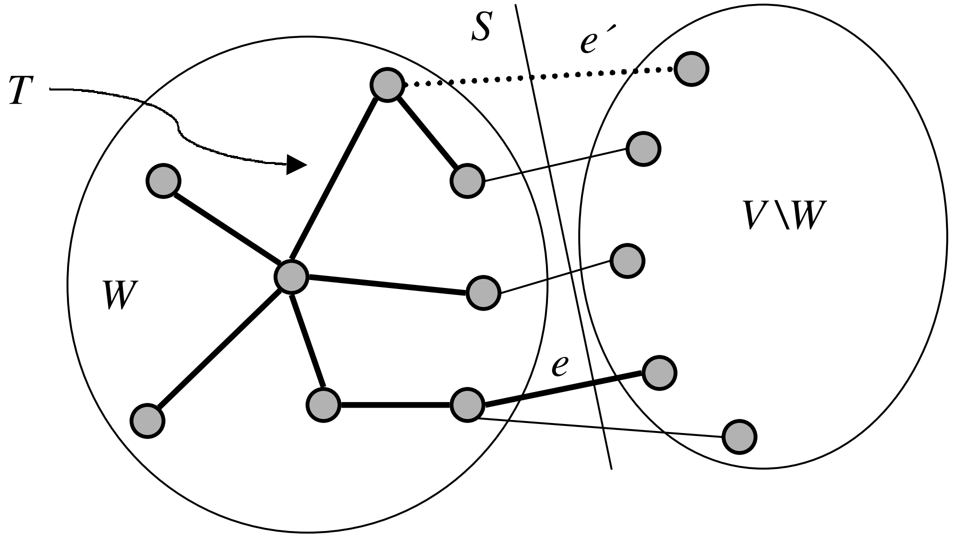
– IS: Kante e wird eingefügt

1. Fall: e in M => ok.

2. Fall: e nicht in M => $\exists e' \in S, e \neq e'$ so daß e' in M ist

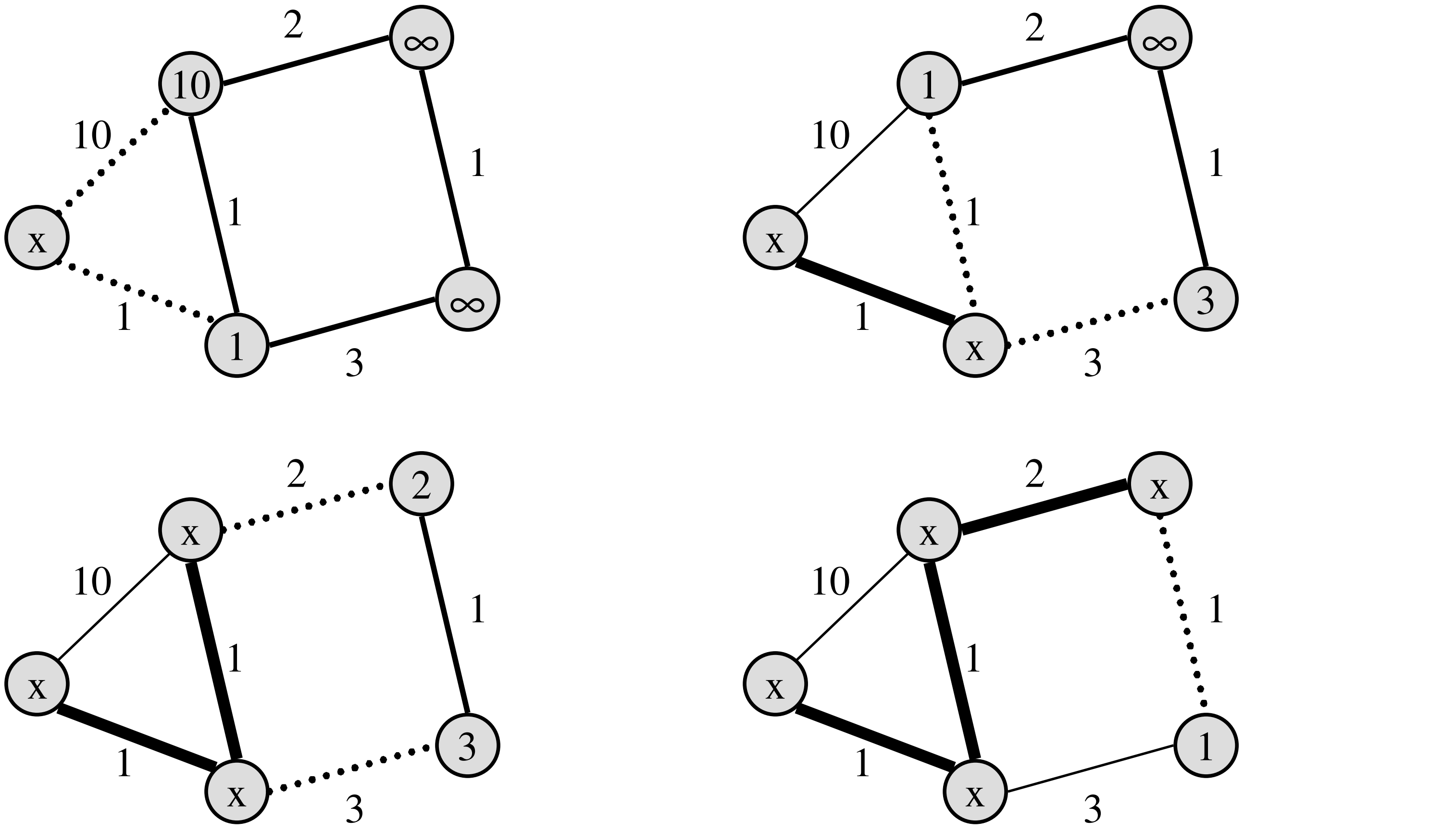
wegen Wahl von e: $c(e) \leq c(e')$

=> in M e' durch e ersetzen ~> neuer MST



Prim: Implementierg., Datenstrukt.

- Knoten in eine P-Queue
- zu jedem Knoten „bisher billigste“ Kante merken



Prim: Implementierg., Datenstrukt.

- n insert(), $O(m)$ decreasePriority(), $O(n)$ delMin()
- => LZ: $O(n + m + n \log n) = O(m + n \log n)$
- Prim besser für dichte Graphen