Graphalgorithmen

19.07.2019

Tiefen- und Breitensuche

Tiefensuche (1/3): Idee

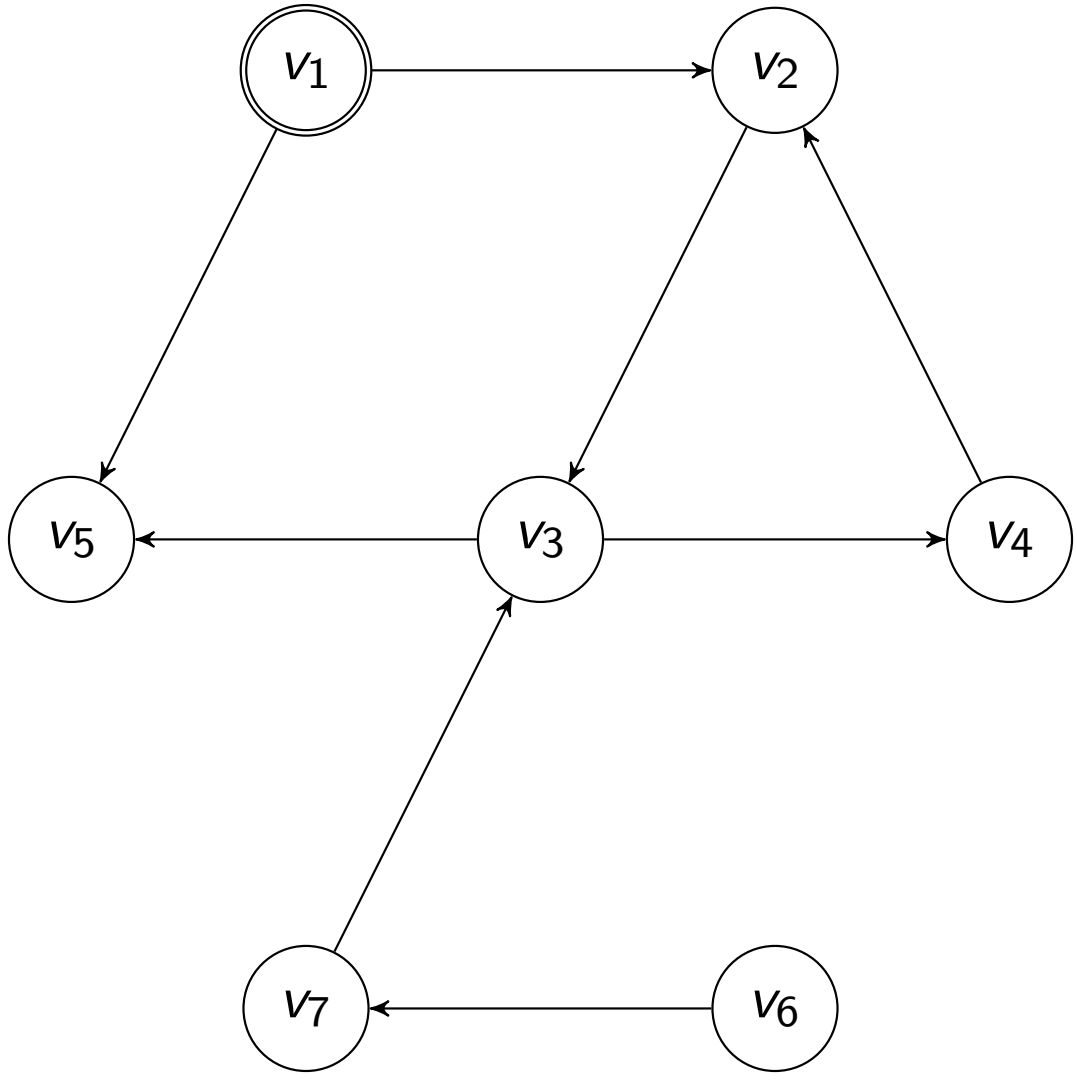
Gegeben sind ein Graph $G = (V, E)$ und ein Startknoten $v \in V$

Wir besuchen in einem Aufruf auf v alle noch nicht besuchten Nachbarn von v rekursiv:

```
tsuche(v):
  für alle Nachbarn v' von v:
    falls v' noch nicht besucht:
      markiere v' als besucht
      tsuche(v')
```

Die Funktion `tsuche()` wird für alle noch nicht besuchten Knoten aufgerufen (rekursive Aufrufe!).
Dabei kann der Wald der Tiefensuche aufgebaut werden

Tiefensuche (2/3): Beispiel



Tiefensuche (3/3): Stuff

Kantentypen: Baum-/Vorwärts/-Rückwärts und Rechts-Links-Querkanten
Anwendungen: z.B. Zusammenhang prüfen; prüfen, ob u von v aus erreichbar ist; auf Kreisfreiheit prüfen
Laufzeit: $\mathcal{O}(|V| + |E|)$ (Adjazenzliste)

Breitensuche (1/3): Idee

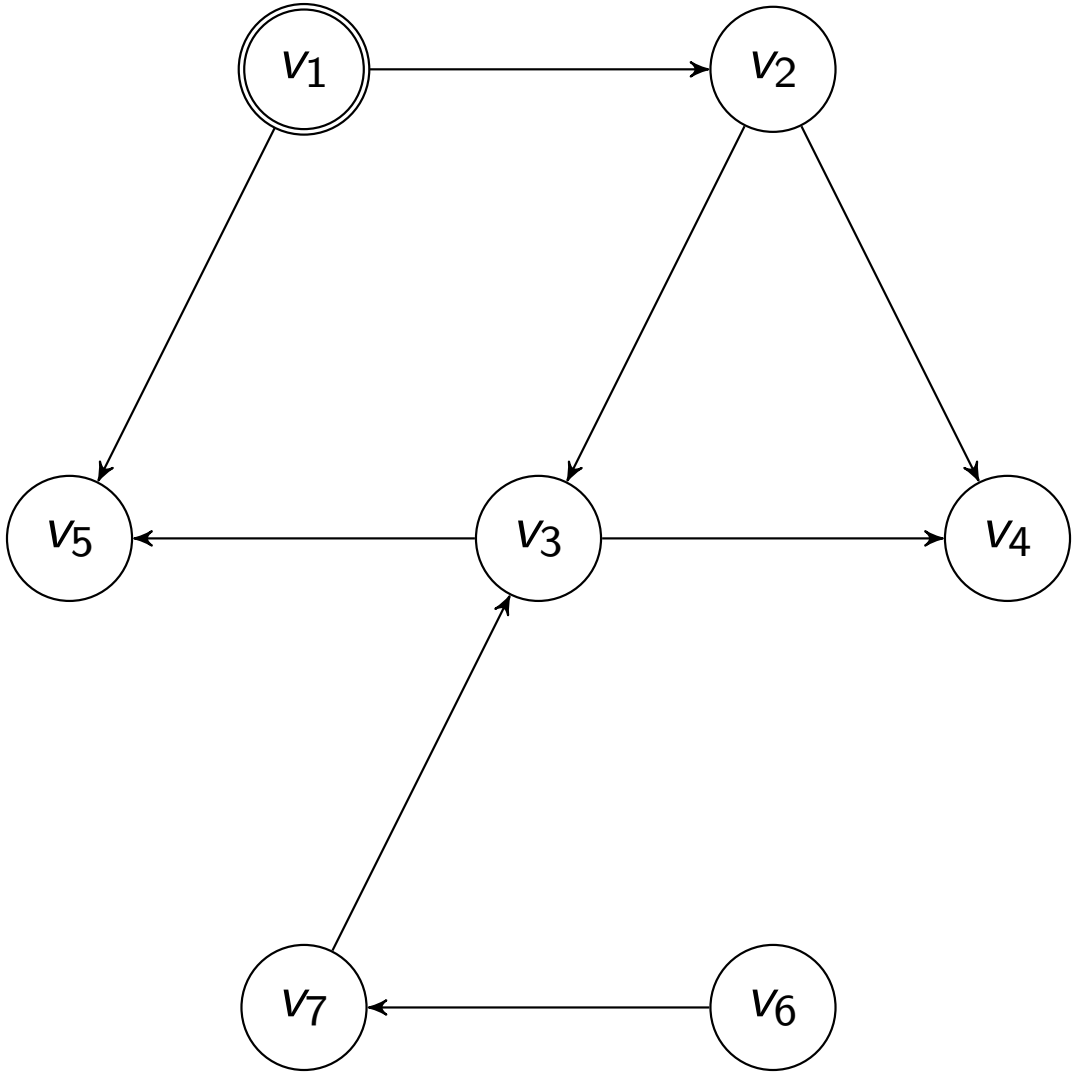
Gegeben sind ein Graph $G = (V, E)$ und ein Startknoten $v \in V$

Wir besuchen erst alle direkten Nachbarn von v, dann alle Knoten mit Abstand 2, dann alle mit Abstand 3, und so weiter.

Aufrufe können über eine Queue verwaltet werden.

Insgesamt werden nur Knoten besucht, die vom Startknoten aus erreichbar sind.

Breitensuche (2/3): Beispiel



Breitensuche (3/3): Stuff

Anwendungen: z.B. Baum der kürzesten Wege für ungewichtete Graphen
Laufzeit: $\mathcal{O}(|V| + |E|)$ (Adjazenzliste)

Dijkstra, Prim und Kruskal

Dijkstra (1/4): Idee

Eingabe: Ein gerichteter, gewichteter Graph $G = (V, E)$ mit Gewichtsfunktion länge : $E \to \mathbb{R}_{\geq 0}$ und ein Startknoten $s \in V$

Ausgabe: Baum der kürzesten Wege

Idee: verwalte die Distanzwerte aller erreichbaren Knoten und füge zu jedem Zeitpunkt einen Knoten zum Baum der kürzesten Wege hinzu, der minimale Distanz zu s hat

Dijsktra (2/4): Hilfsmittel

Wir arbeiten mit folgenden Hilfsmitteln:

- ***S*-Menge:** Verwaltung der Knoten, die sich aktuell im Baum der kürzesten Wege befinden. **Initial wird $S = \{s\}$ gesetzt, enthält also nur den Startknoten.**
- **Distanz-Array:** Verwaltung der Länge der kürzesten Wege von s zu allen erreichbaren Knoten im Graph. **Initial wird *dist* folgendermaßen gesetzt:**

$$dist[v] = \begin{cases} gewicht((s,v)), & \text{falls } (s,v) \in E \\ \infty, & \text{sonst} \end{cases}$$

(also: der Distanzwert für alle Nachbarn von s wird entsprechend dem Gewicht der verbindenden Kante gesetzt, für alle anderen Knoten wird die Distanz auf ∞ gesetzt)

Dijkstra (3/4): Pseudocode

1) Initialisierung:

$S = \{s\}$

$$\texttt{dist[v]} = \begin{cases} \texttt{länge((s,v))} & \texttt{falls } (s,v) \in E \\ \infty & \texttt{sonst} \end{cases}$$

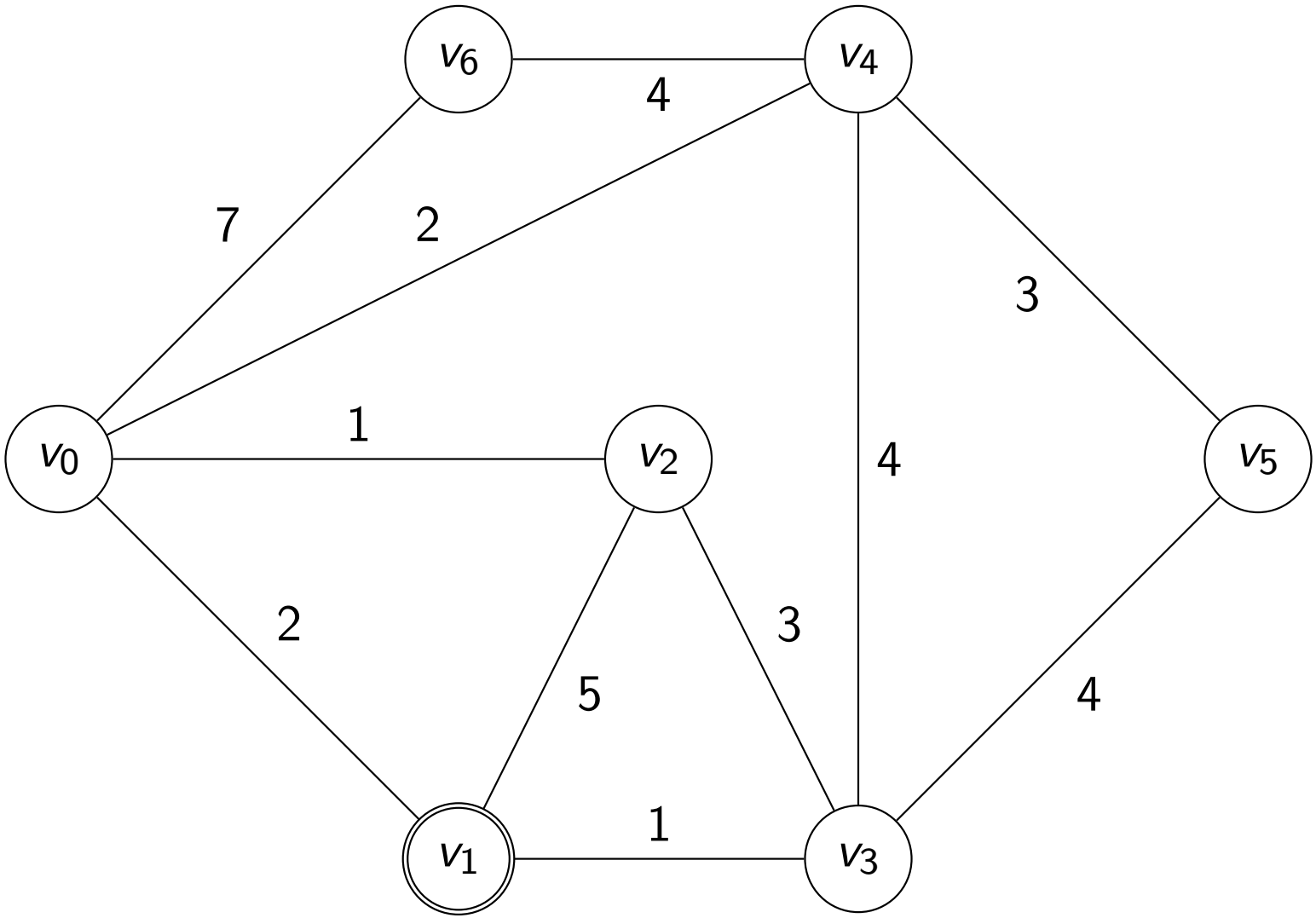
(für alle $v \in V \backslash S$)

2) Ermitteln der kürzesten Wege:

```
while (S≠V):
  finde Knoten v ∈ V\S mit minimalem Distanzwert
  füge v zu S hinzu
  für alle Nachbarn w von v:
    falls dist[v] + länge((v,w)) < dist[w]:
      setze dist[w] = dist[v]+länge((v,w))
```

Dijkstra (4/4): Beispiel

Aufgabe 5.3a)



Prim (1/4): Idee

Eingabe: Ein ungerichteter, zusammenhängender, gewichteter Graph $G = (V, E)$ mit Gewichtsfunktion länge : $E \to \mathbb{R}_{\geq 0}$ und ein Startknoten $s \in V$

Ausgabe: Minimaler Spannbaum von G

Idee: verwalte die bereits hinzugefügten Knoten in einer Menge und wähle zu jedem Zeitpunkt eine Kante minimalen Gewichts, welche diese Menge "kreuzt".

Prim (2/4): Hilfsmittel

Wir arbeiten mit folgenden Hilfsmitteln:

- ***S*-Menge:** Verwaltung der Knoten, die sich aktuell im minimalen Spannbaum befinden. **Initial wird $S = \{s\}$ gesetzt, enthält also nur den Startknoten.**
- **Heap:** Verwaltung der kürzesten S-kreuzenden Kanten. **Wird jedes Mal aktualisiert, wenn ein Knoten zu S hinzugefügt wird.**

Initial werden alle vom Startknoten s ausgehenden Kanten $\{s, t\} \in E$ in den Heap eingefügt.

Prim (3/4): Pseudocode

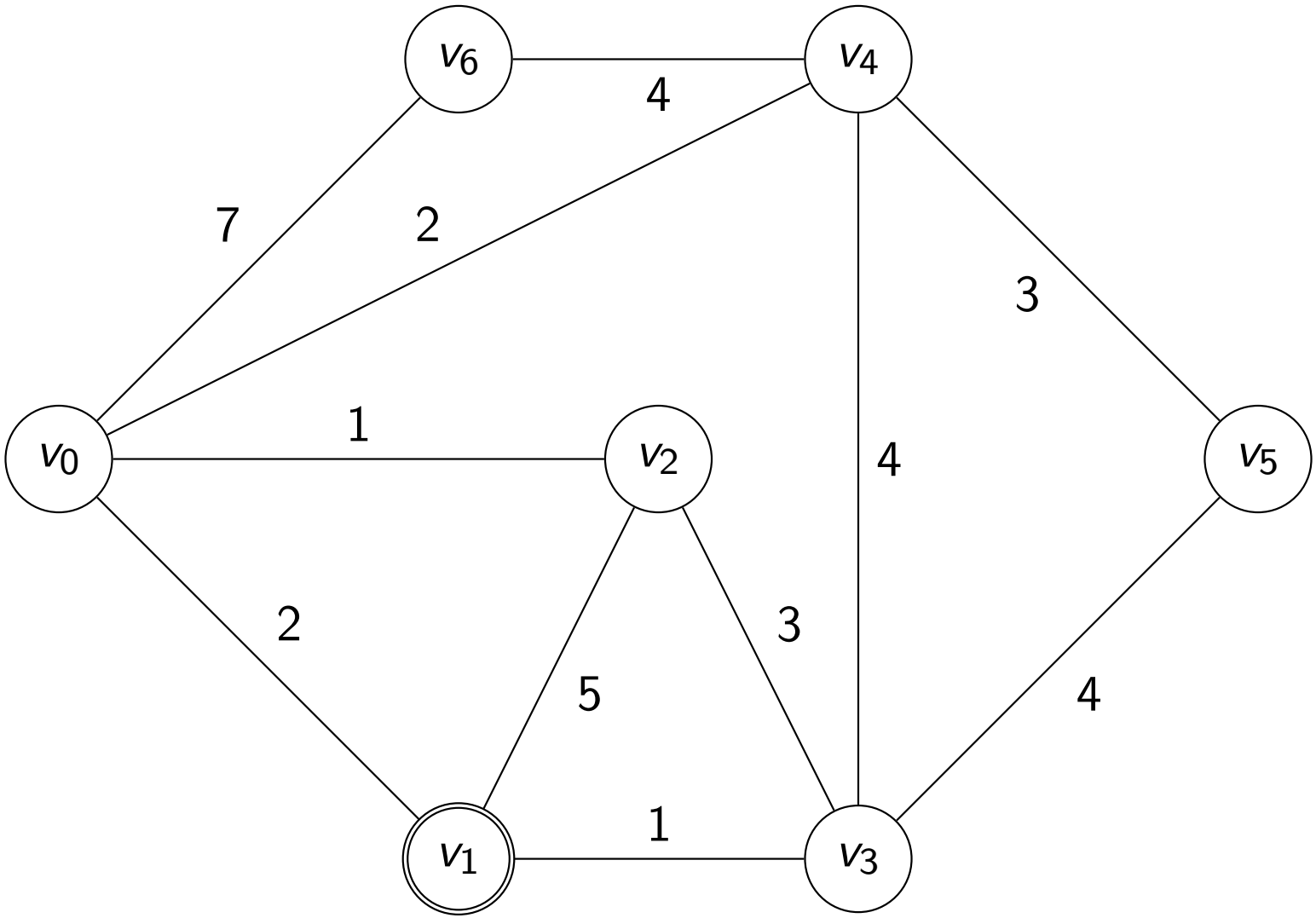
1) Initialisierung:
`S = {s}`

2) Ermitteln des minmalen Spannbaums:

```
while S ≠ V:
  finde kürzeste S-kreuzende Kante {u,v}
  if (u nicht in S):
    füge u zu S hinzu
  else if (v nicht in S):
    füge v zu S hinzu
```

Prim (4/4): Beispiel

Aufgabe 5.3b)



Kruskal (1/4): Idee

Eingabe: Ein ungerichteter, zusammenhängender, gewichteter Graph $G = (V, E)$ mit Gewichtsfunktion länge : $E \to \mathbb{R}_{\geq 0}$ und ein Startknoten $s \in V$

Ausgabe: Minimaler Spannbaum von G

Idee: sortiere die Kanten aufsteigend nach Kantengewicht und füge stets die kürzeste Kante hinzu, welche keinen Kreis schließt, bis ein minimaler Spannbaum entstanden ist.

Kruskal (2/4): Union-Find Datenstruktur (Idee)

- Zusammenhangskomponenten mit Hilfe einer Waldstruktur verwalten
- prüfen, ob Kante zwischen zwei Knoten einen Kreis schließen würde
- Achtung: Union-Find-Struktur entspricht **NICHT** dem minimalen Spannbaum

Kruskal (2/4): Union-Find Datenstruktur

Aktualisieren der Union-Find-Datenstruktur:

(u, v) ist zu prüfen und eventuell einzufügen

1. u, v im selben Baum (`wurzel(u) == wurzel(v)`)?
2. wenn ja, verwerfe die Kante
3. wenn nein:
   - (u, v) wird in den Spannbaum aufgenommen
   - die Bäume müssen in der Union-Find-Struktur vereint werden (`union(u,v)`)

Bäume vereinen: kleineren unter größeren hängen; wenn gleich groß entscheidet Index

Kruskal (3/4): Pseudocode

1) Initialisierung:

```
Sortiere alle Kanten aufsteigend nach Kantengewicht
Union-Find-Struktur:  Wald von Einzelknoten
M: Wald von Einzelknoten (wird später minimaler
Spannbaum)
```

2) Ermitteln des minimalen Spannbaums:

```
while (M kein minimaler Spannbaum):
  sei (u,v) die aktuell kürzeste Kante (Sortierung)
  if (wurzel(u) == wurzel(v)):
    verwerfe (u,v), denn sie schließt Kreis
  else:
    füge (u,v) in M ein
    union(u,v) (sind jetzt in der selben Komponente)
```

Kruskal (4/4): Beispiel

Aufgabe 5.3c)