Algorithmen und Datenstrukturen

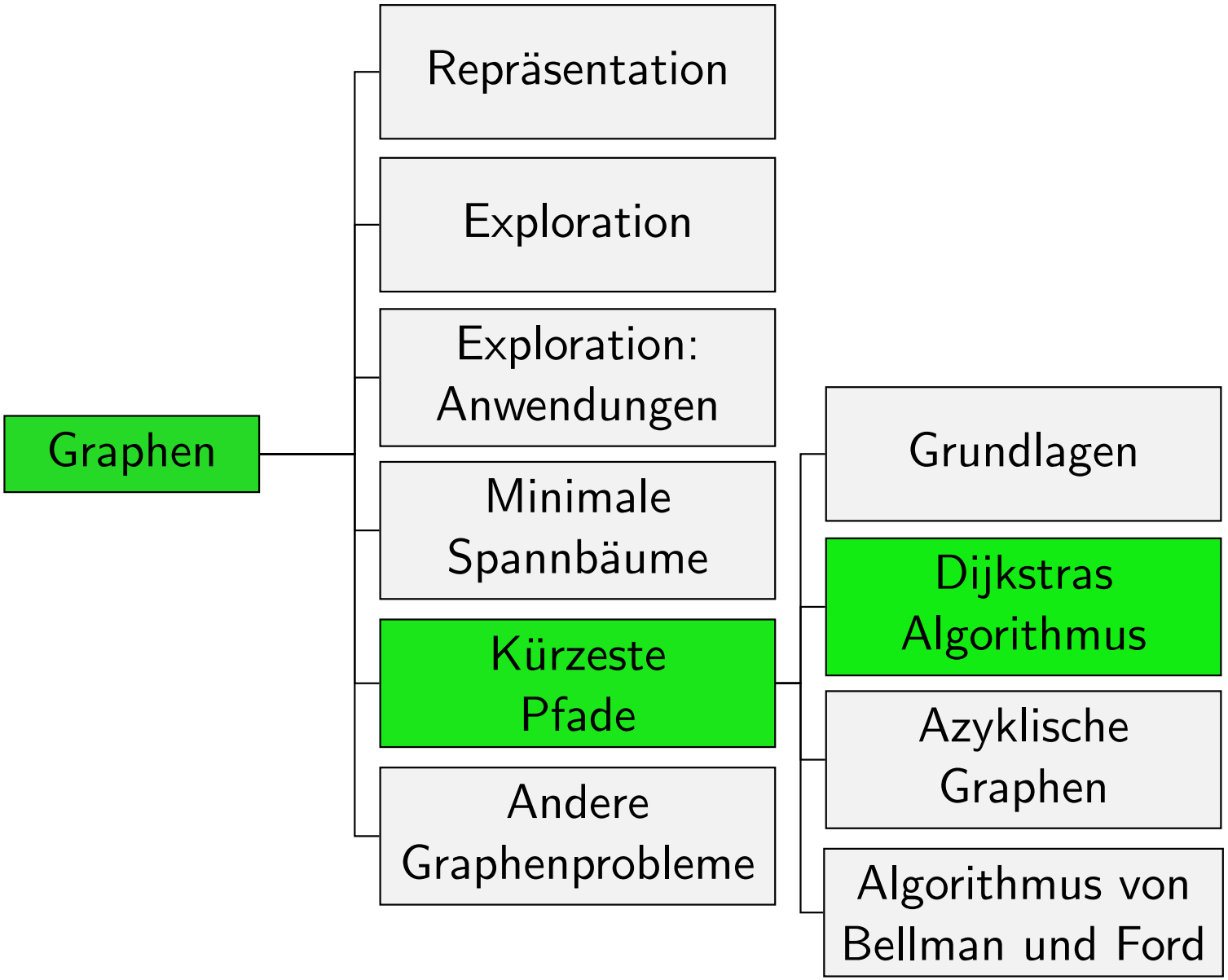
C6. Kürzeste Pfade: Algorithmen

Gabriele Röger

Universität Basel

Dijkstras Algorithmus

Graphen: Übersicht



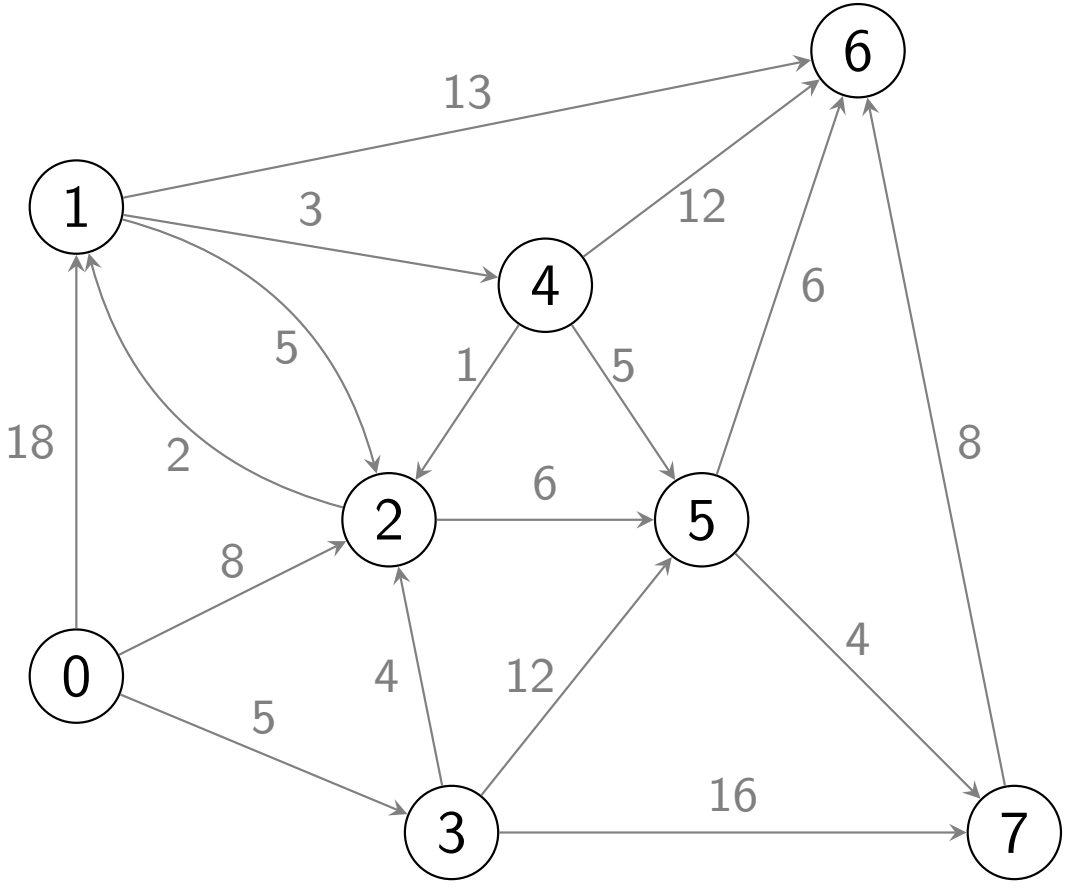
Dijkstras Algorithmus: High-Level-Perspektive

Algorithmus von Dijkstra (für nicht-negative Kantengewichte)

Baue Kürzeste-Pfade-Baum ausgehend von Startknoten s auf:

- Betrachte Knoten (die noch nicht im Baum sind) in aufsteigender Reihenfolge ihres Abstandes von s.
- Nimm Knoten in Baum auf und relaxiere ausgehende Kanten.

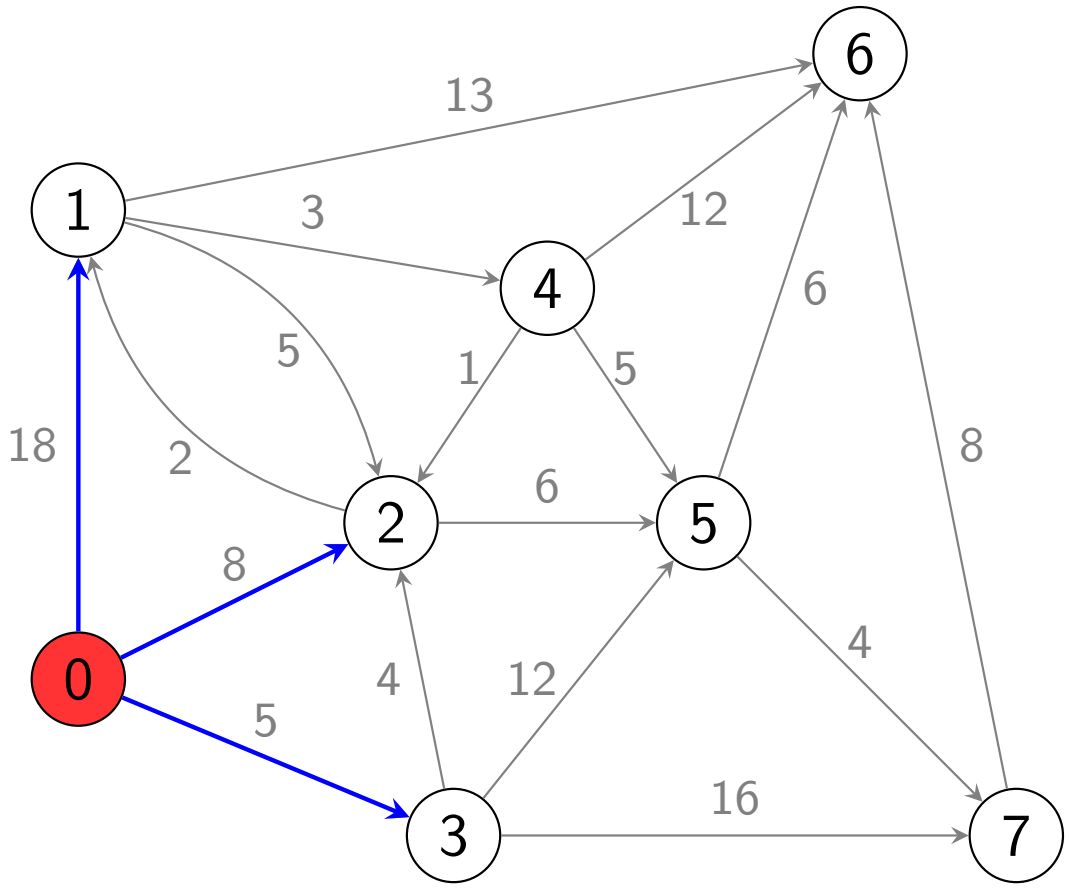
Dijkstras Algorithmus: Illustration



distance

0	0
1	∞
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞

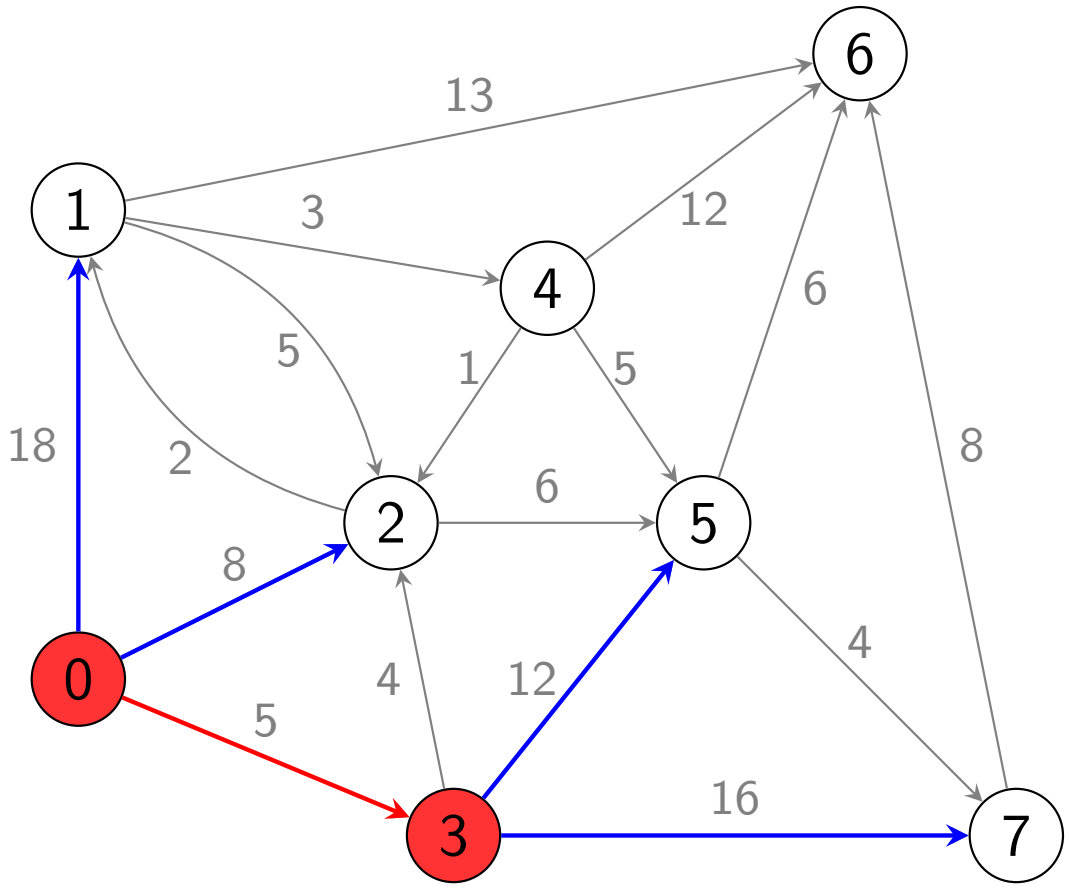
Dijkstras Algorithmus: Illustration



distance

0	0
1	18
2	8
3	5
4	∞
5	∞
6	∞
7	∞

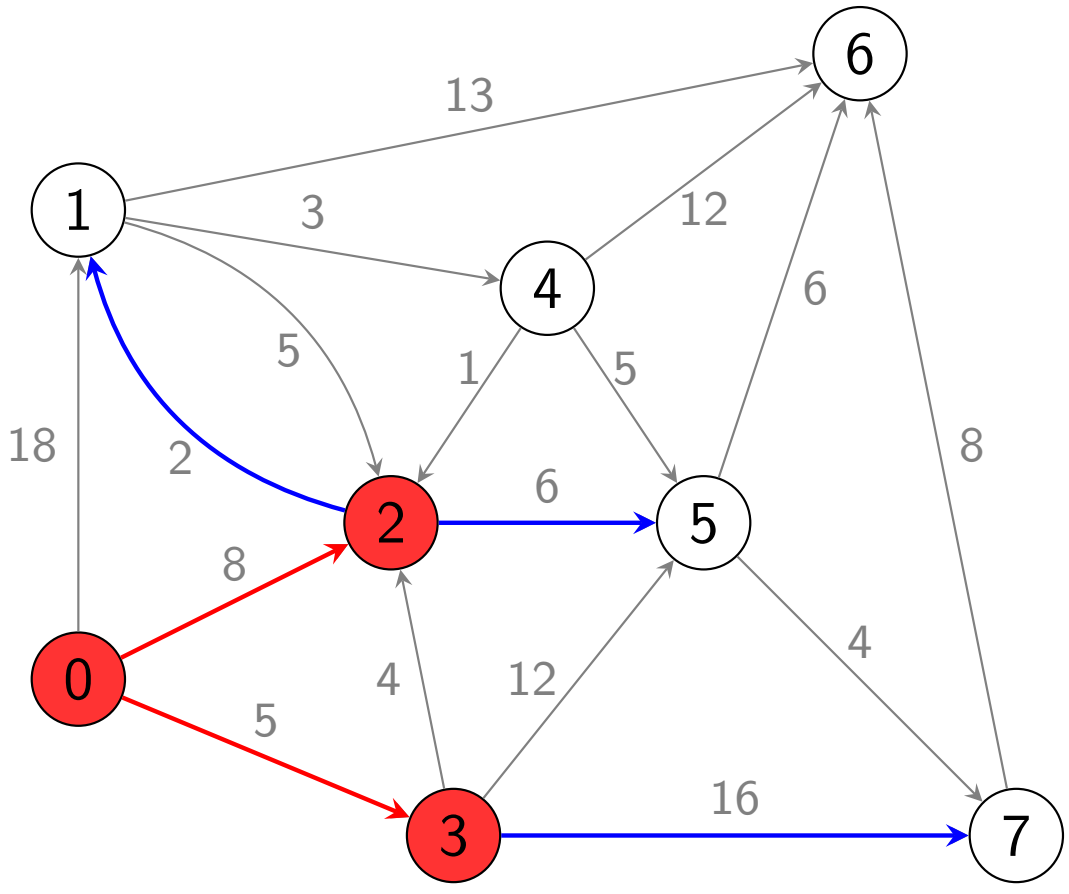
Dijkstras Algorithmus: Illustration



distance

0	0
1	18
2	8
3	5
4	∞
5	17
6	∞
7	21

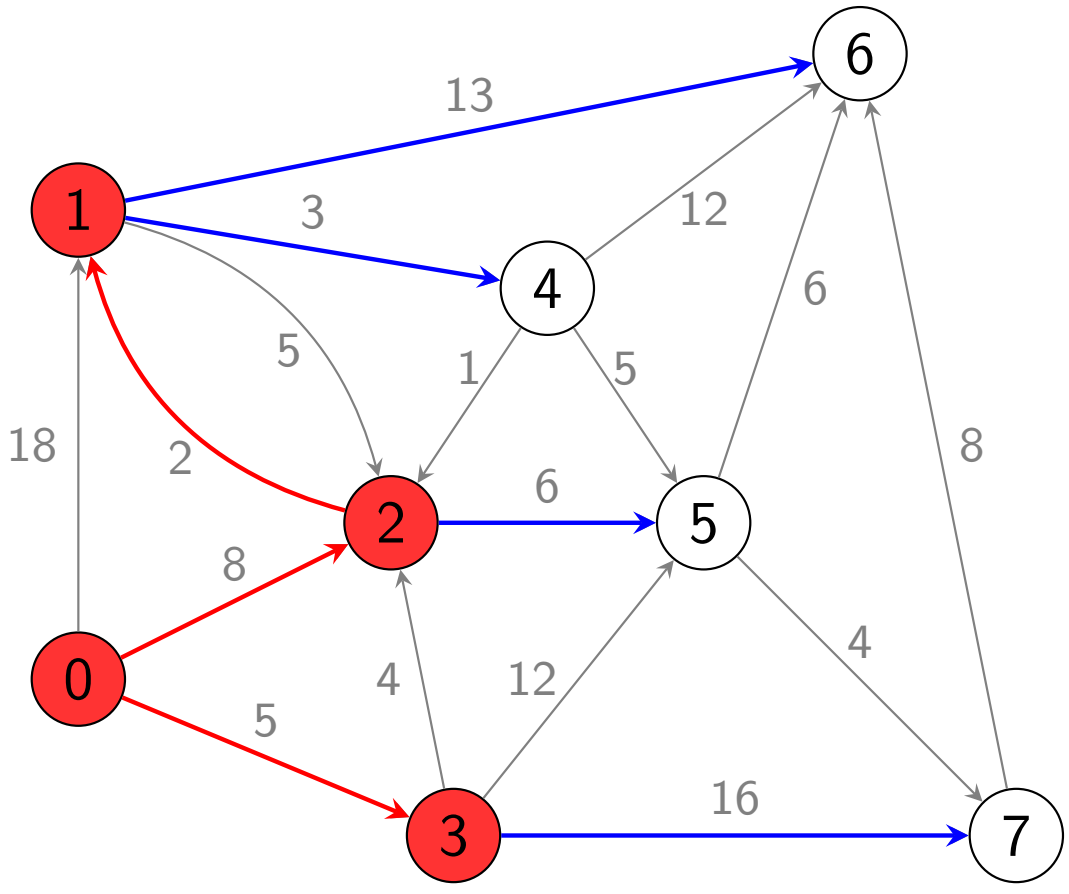
Dijkstras Algorithmus: Illustration



distance

0	0
1	10
2	8
3	5
4	∞
5	14
6	∞
7	21

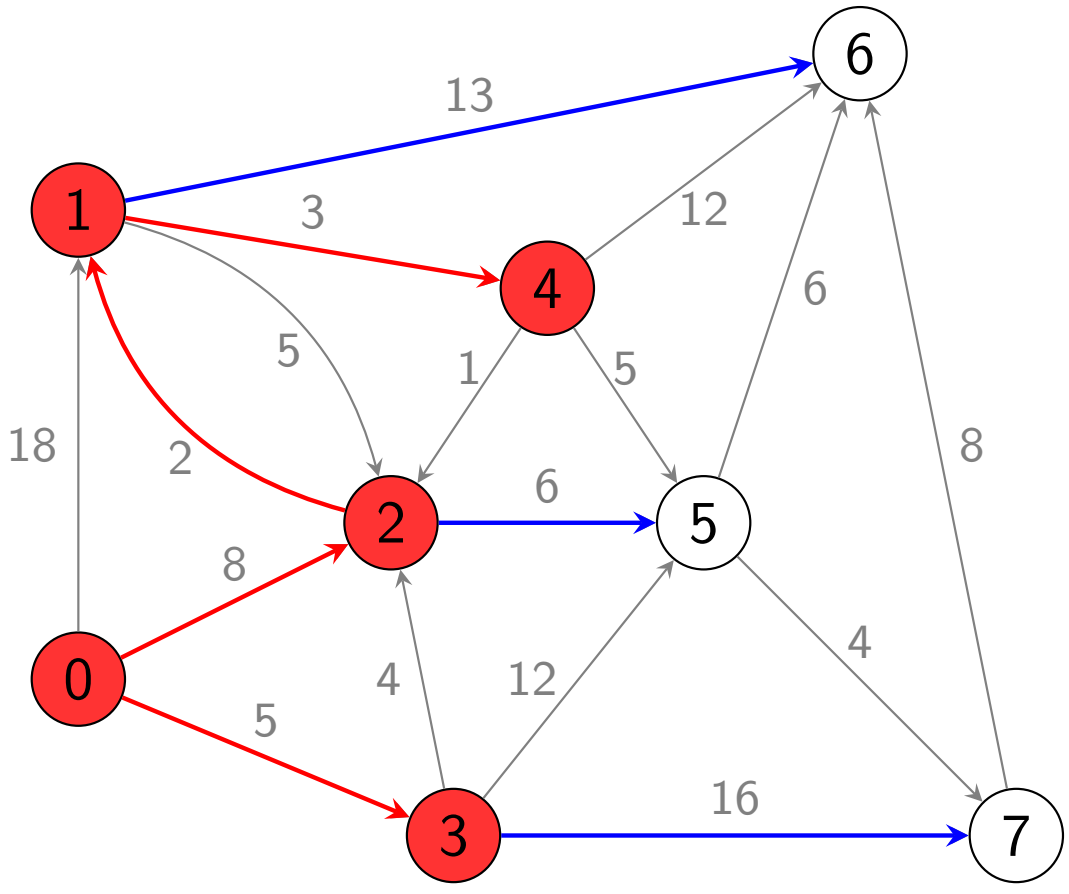
Dijkstras Algorithmus: Illustration



distance

0	0
1	10
2	8
3	5
4	13
5	14
6	23
7	21

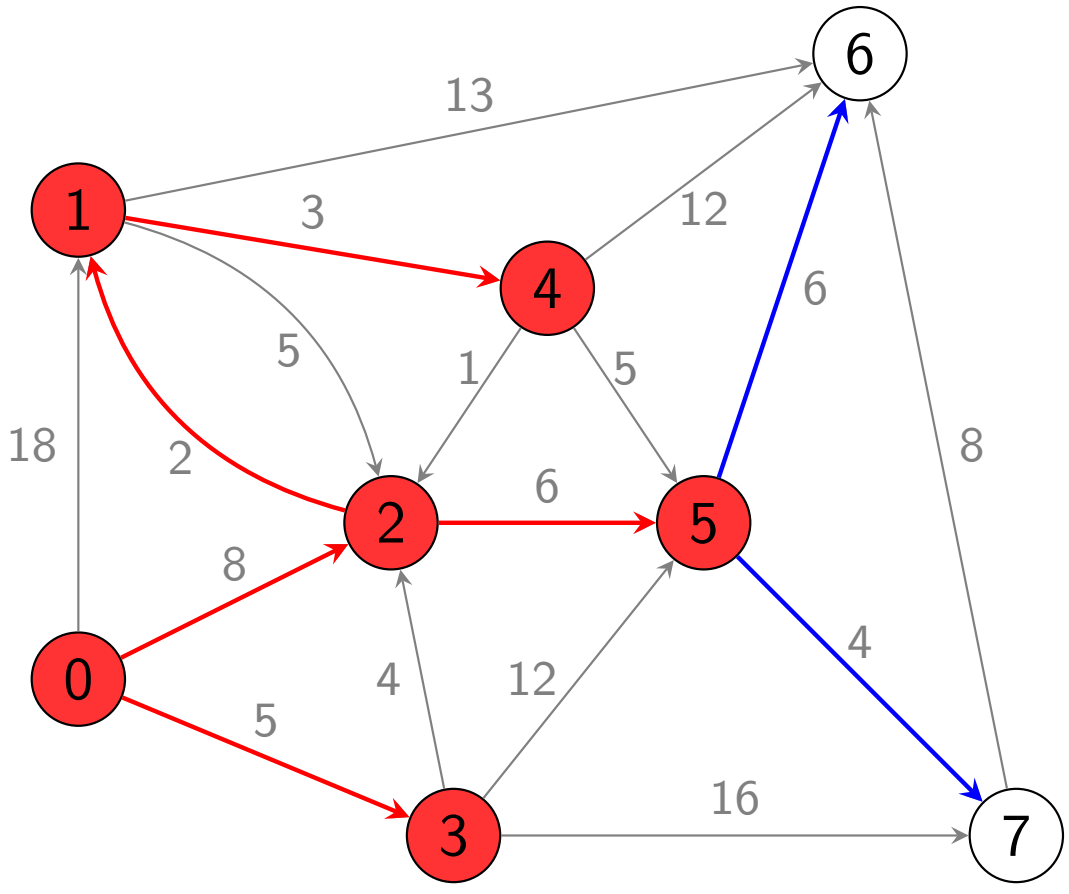
Dijkstras Algorithmus: Illustration



distance

0	0
1	10
2	8
3	5
4	13
5	14
6	23
7	21

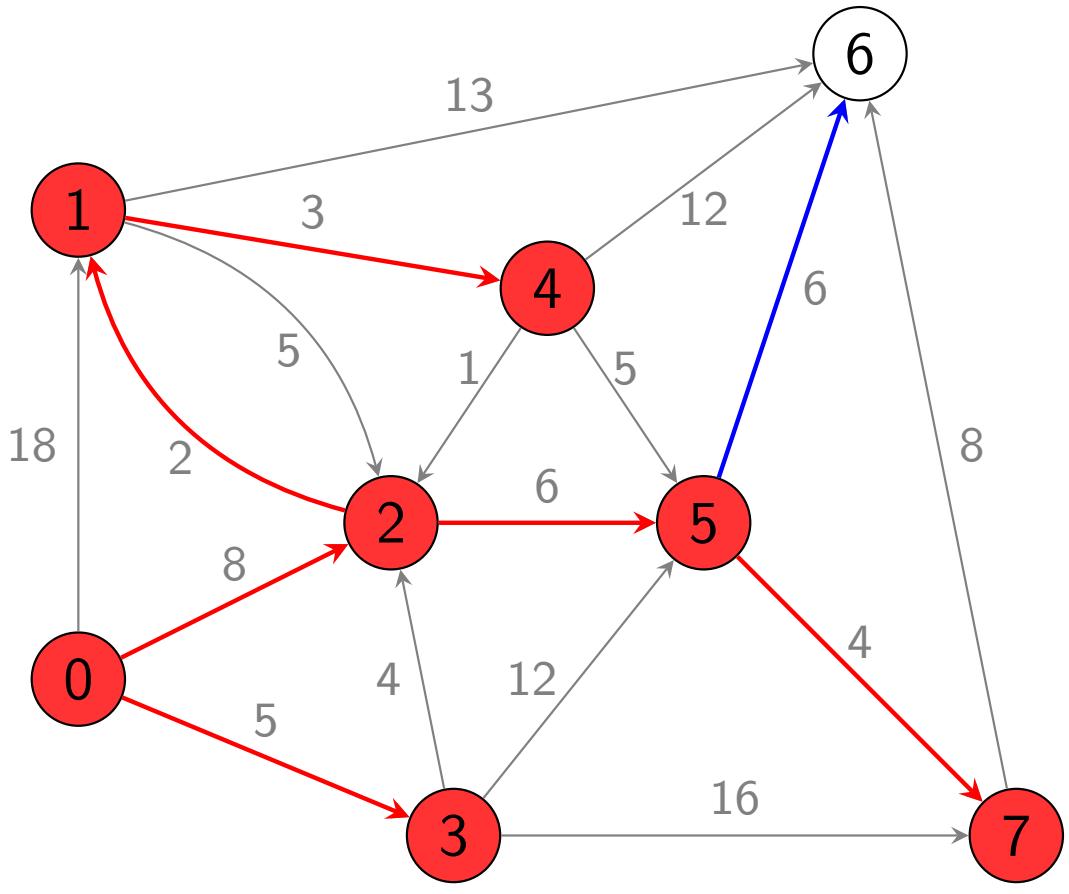
Dijkstras Algorithmus: Illustration



distance

0	0
1	10
2	8
3	5
4	13
5	14
6	20
7	18

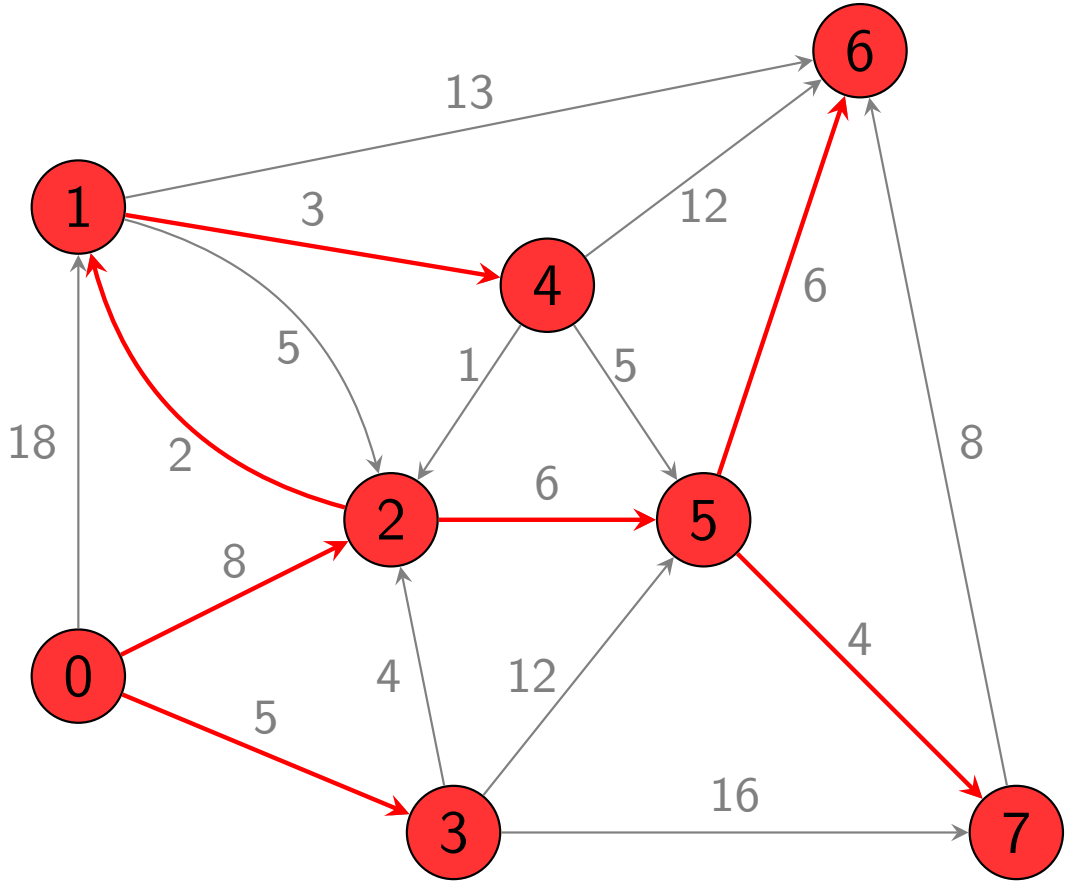
Dijkstras Algorithmus: Illustration



distance

0	0
1	10
2	8
3	5
4	13
5	14
6	20
7	18

Dijkstras Algorithmus: Illustration



distance

0	0
1	10
2	8
3	5
4	13
5	14
6	20
7	18

Datenstrukturen

- `edge_to`: knotenindiziertes Array, das an Stelle v die letzte Kante des kürzesten bekannten Pfades enthält.

Datenstrukturen

- `edge_to`: knotenindiziertes Array, das an Stelle v die letzte Kante des kürzesten bekannten Pfades enthält.
- `distance`: knotenindiziertes Array, das an Stelle v die Kosten des kürzesten bekannten Pfades vom Startknoten zu v enthält.

Datenstrukturen

- `edge_to`: knotenindiziertes Array, das an Stelle v die letzte Kante des kürzesten bekannten Pfades enthält.
- `distance`: knotenindiziertes Array, das an Stelle v die Kosten des kürzesten bekannten Pfades vom Startknoten zu v enthält.
- `pq`: indizierte Priority-Queue von Knoten
  - Knoten noch nicht im Baum
  - Bereits ein Pfad zu dem Knoten bekannt
  - Sortiert nach Kosten des kürzesten bekannten Pfades zu dem Knoten.

Dijkstras Algorithmus

```
1  class DijkstraSSSP:
2      def __init__(self, graph, start_node):
3          self.edge_to = [None] * graph.no_nodes()
4          self.distance = [float('inf')] * graph.no_nodes()
5          pq = IndexMinPQ()
6          self.distance[start_node] = 0
7          pq.insert(start_node, 0)
8          while not pq.empty():
9              self.relax(graph, pq.del_min(), pq)
10
11     def relax(self, graph, v, pq):
12         for edge in graph.adjacent_edges(v):
13             w = edge.to_node()
14             if self.distance[v] + edge.weight() < self.distance[w]:
15                 self.edge_to[w] = edge
16                 self.distance[w] = self.distance[v] + edge.weight()
17                 if pq.contains(w):
18                     pq.change(w, self.distance[w])
19                 else:
20                     pq.insert(w, self.distance[w])
```

Korrektheit

Theorem

Dijkstras Algorithmus löst das Single-Source-Shortest-Paths-Problem in Digraphen mit nicht-negativen Gewichten.

Beweis.

- Ist v von Startknoten erreichbar, wird jede ausgehende Kante $e = (v, w)$ genau einmal relaxiert (wenn v relaxiert wird).
- Dann gilt $\mathit{distance}[w] \leq \mathit{distance}[v] + \mathit{weight}(e)$.
- Ungleichung bleibt erfüllt:
  - $\mathit{distance}[v]$ wird nicht mehr verändert, da Wert minimal war und es keine negativen Kantengewichte gibt.
  - $\mathit{distance}[w]$ wird höchstens kleiner.
- Sind alle erreichbaren Knoten relaxiert, ist Optimalitätsbedingung erfüllt.

□

Vergleich zu Eager Prim-Algorithmus

Dijkstras Algorithmus sehr ähnlich zu Eager Prim-Algorithmus für minimale Spannbäume

- Beide bauen sukzessive einen Baum auf
- nächster Knoten Prim: minimale Distanz zu bisherigem Baum.
- nächster Knoten Dijkstra: minimale Distanz vom Startknoten.
- `included_nodes` von Prim bei Dijkstra nicht notwendig, da bei bereits erledigten Knoten die `if`-Bedingung in Zeile 14 immer falsch ist.

Vergleich zu Eager Prim-Algorithmus

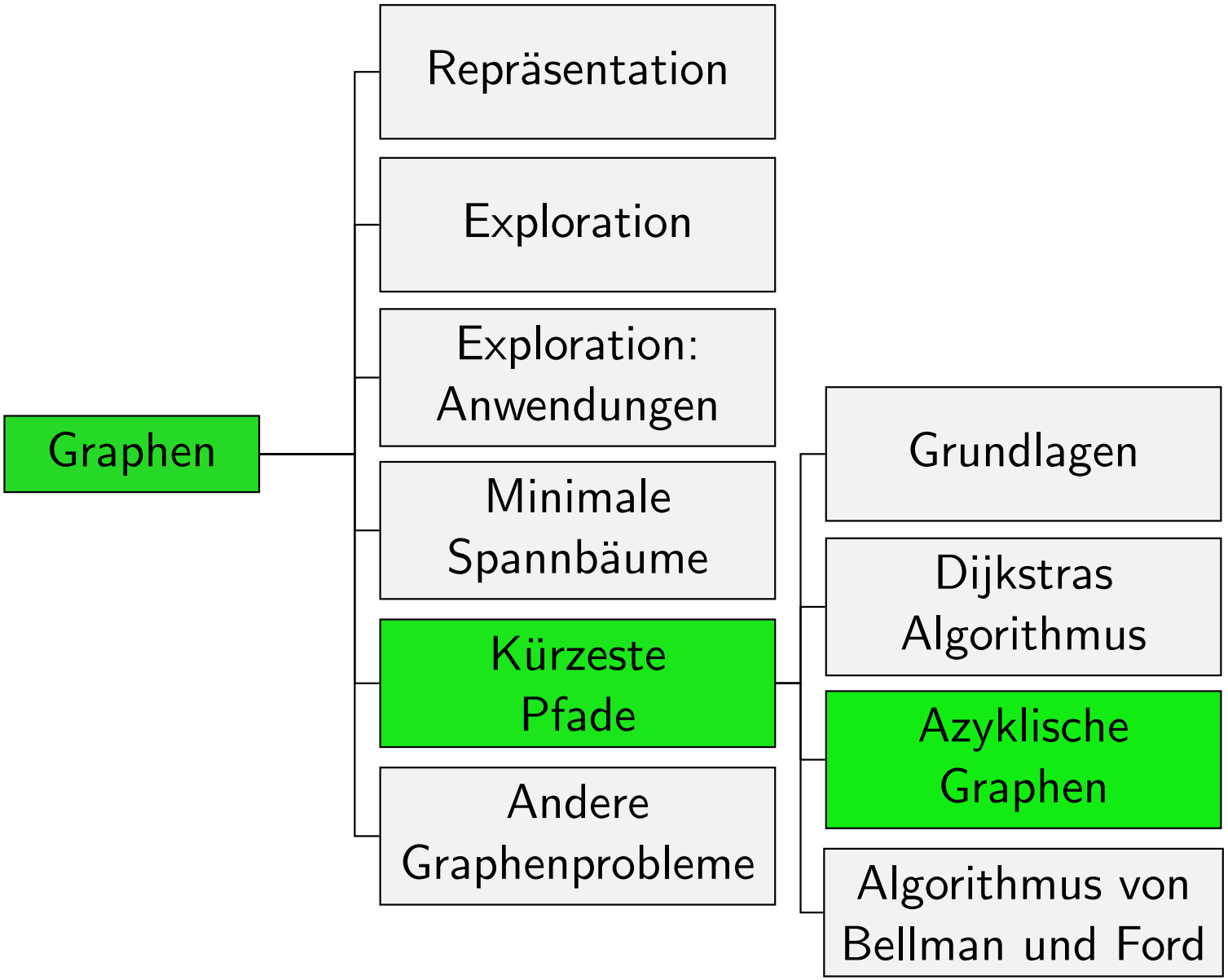
Dijkstras Algorithmus sehr ähnlich zu Eager Prim-Algorithmus für minimale Spannbäume

- Beide bauen sukzessive einen Baum auf
- nächster Knoten Prim: minimale Distanz zu bisherigem Baum.
- nächster Knoten Dijkstra: minimale Distanz vom Startknoten.
- `included_nodes` von Prim bei Dijkstra nicht notwendig, da bei bereits erledigten Knoten die `if`-Bedingung in Zeile 14 immer falsch ist.

Laufzeit $O(|E| \log |V|)$ und Platzbedarf $O(|V|)$ direkt übertragbar.

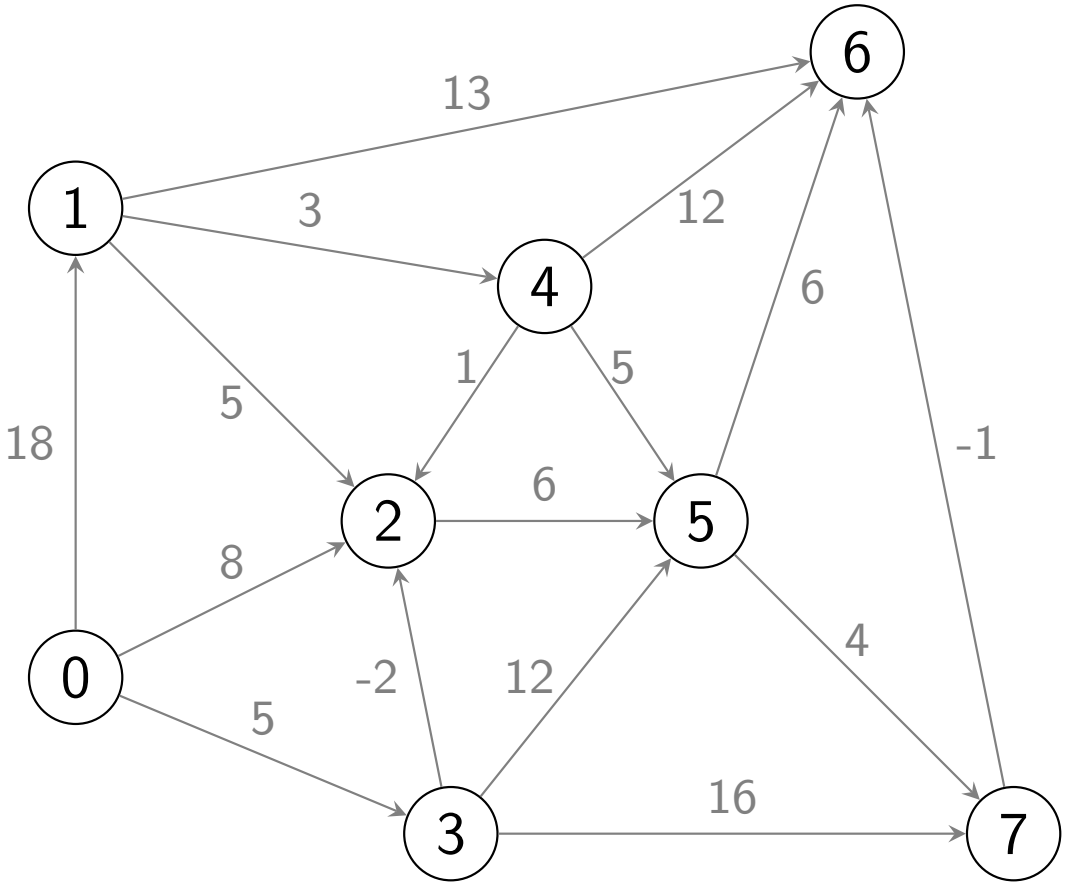
Azyklische Graphen

Graphen: Übersicht



Zykelfreiheit ausnutzen

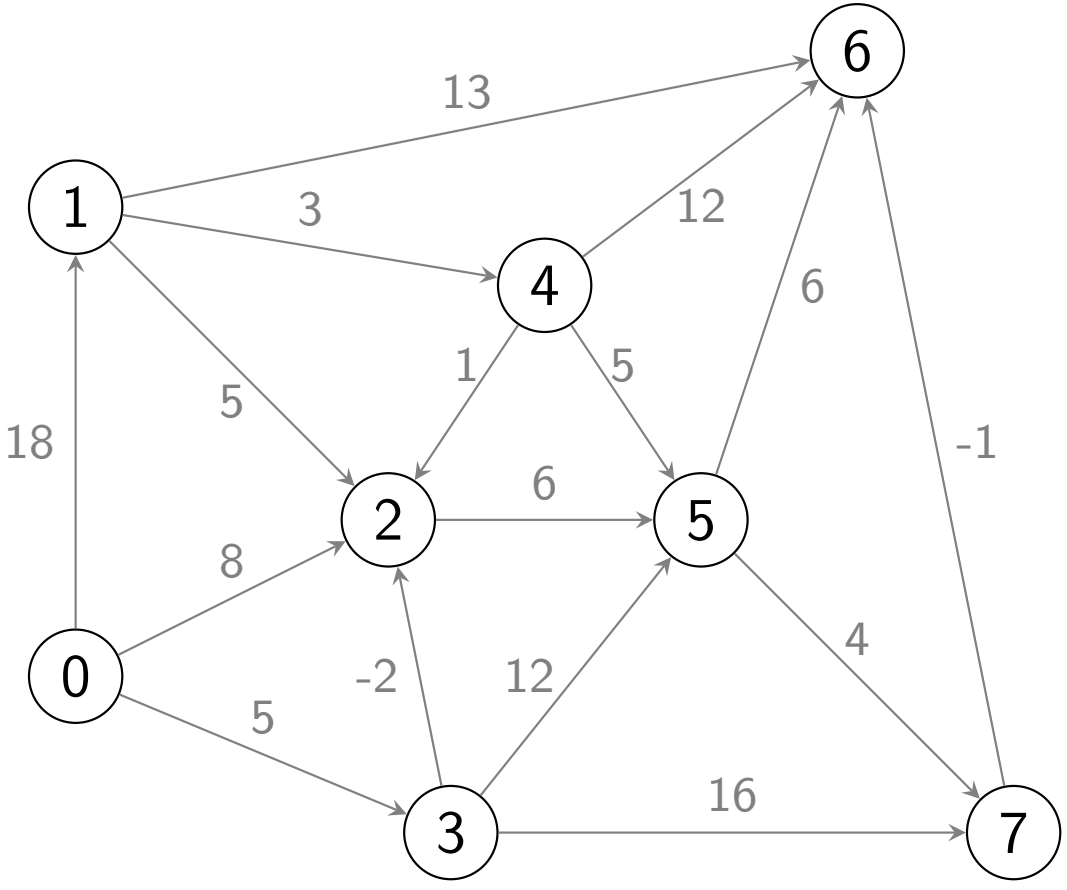
Gegeben: Azyklischer, gewichteter Digraph



Können wir die Zykelfreiheit beim Finden kürzester Pfade nutzen?

Beispiel

Idee: Relaxiere Knoten in topologischer Reihenfolge
z.B. 0, 1, 3, 4, 2, 5, 7, 6

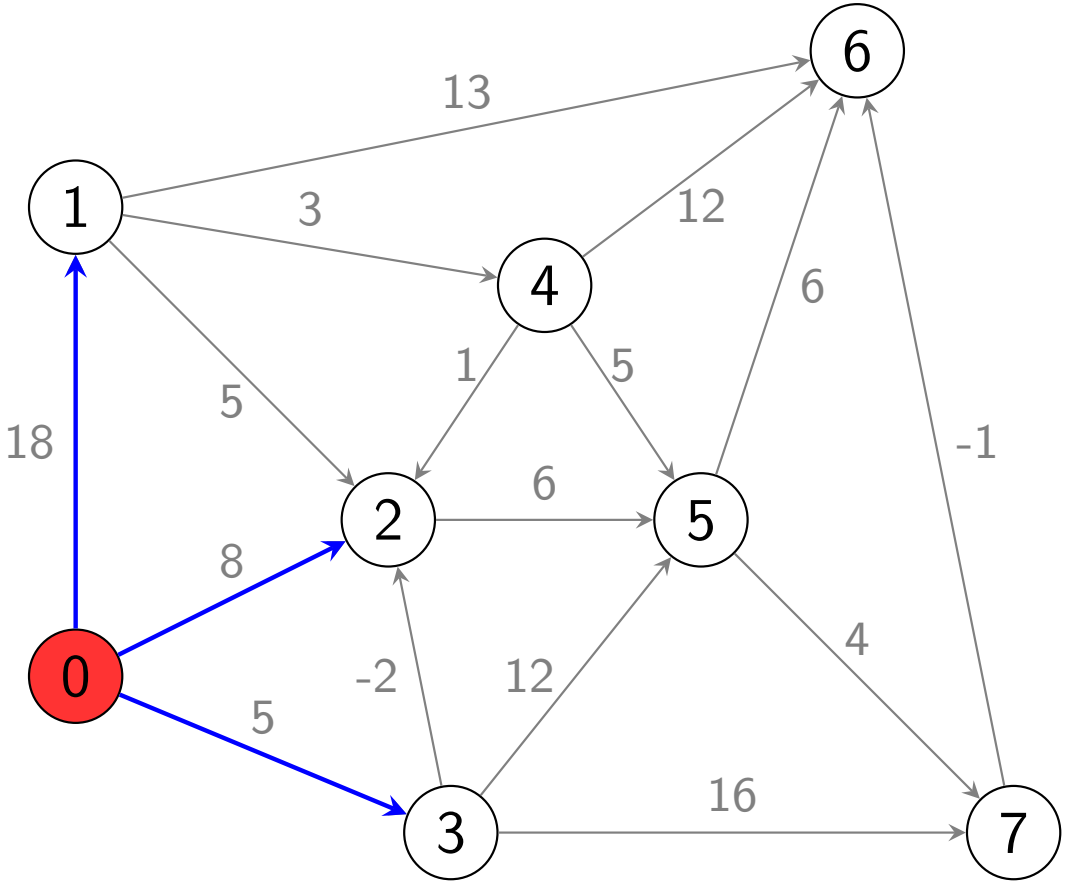


distance

0	0
1	∞
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞

Beispiel

Idee: Relaxiere Knoten in topologischer Reihenfolge
z.B. 0, 1, 3, 4, 2, 5, 7, 6

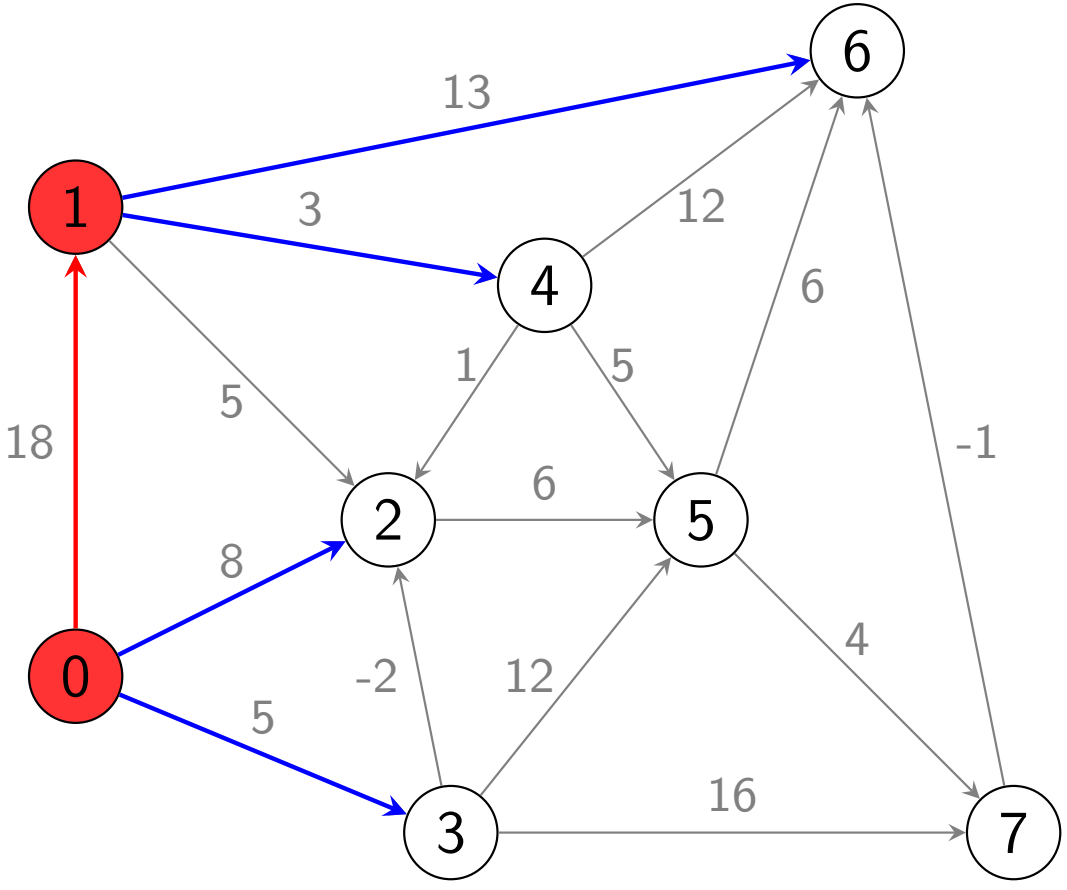


distance

0	0
1	18
2	8
3	5
4	∞
5	∞
6	∞
7	∞

Beispiel

Idee: Relaxiere Knoten in topologischer Reihenfolge
z.B. 0, 1, 3, 4, 2, 5, 7, 6

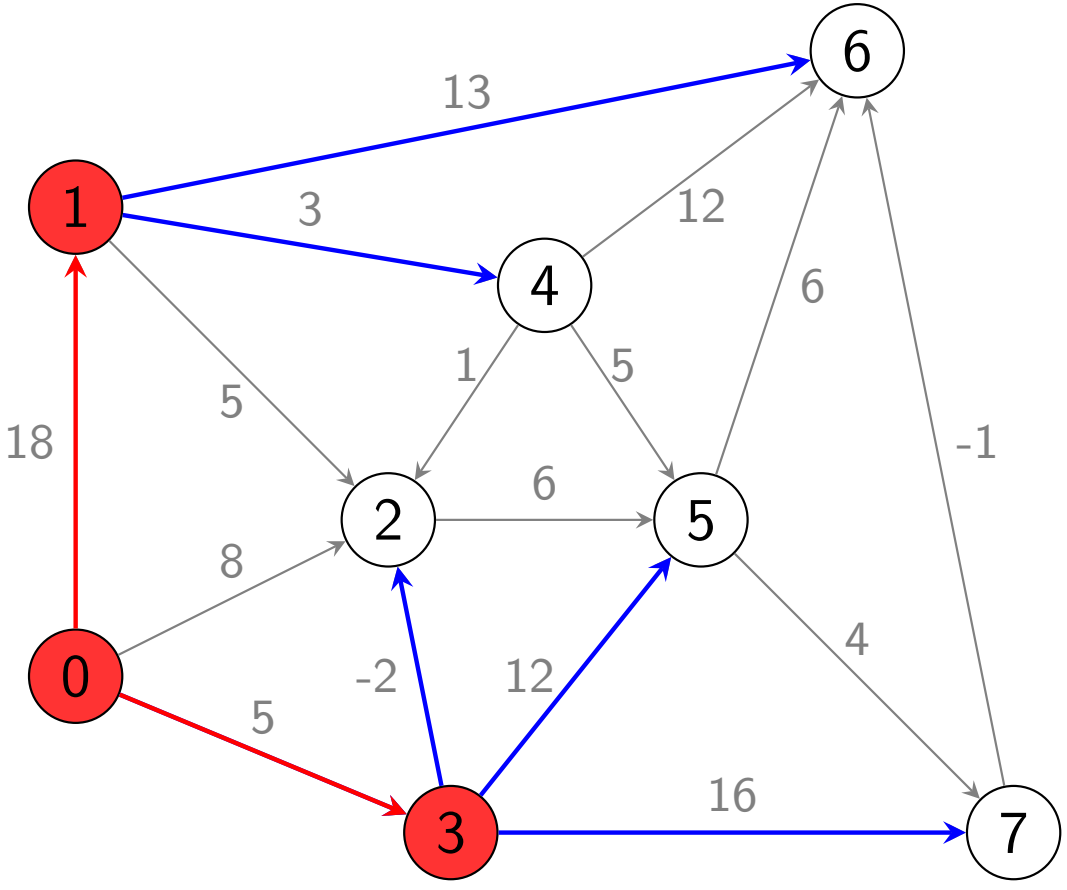


distance

0	0
1	18
2	8
3	5
4	21
5	∞
6	31
7	∞

Beispiel

Idee: Relaxiere Knoten in topologischer Reihenfolge
z.B. 0, 1, 3, 4, 2, 5, 7, 6

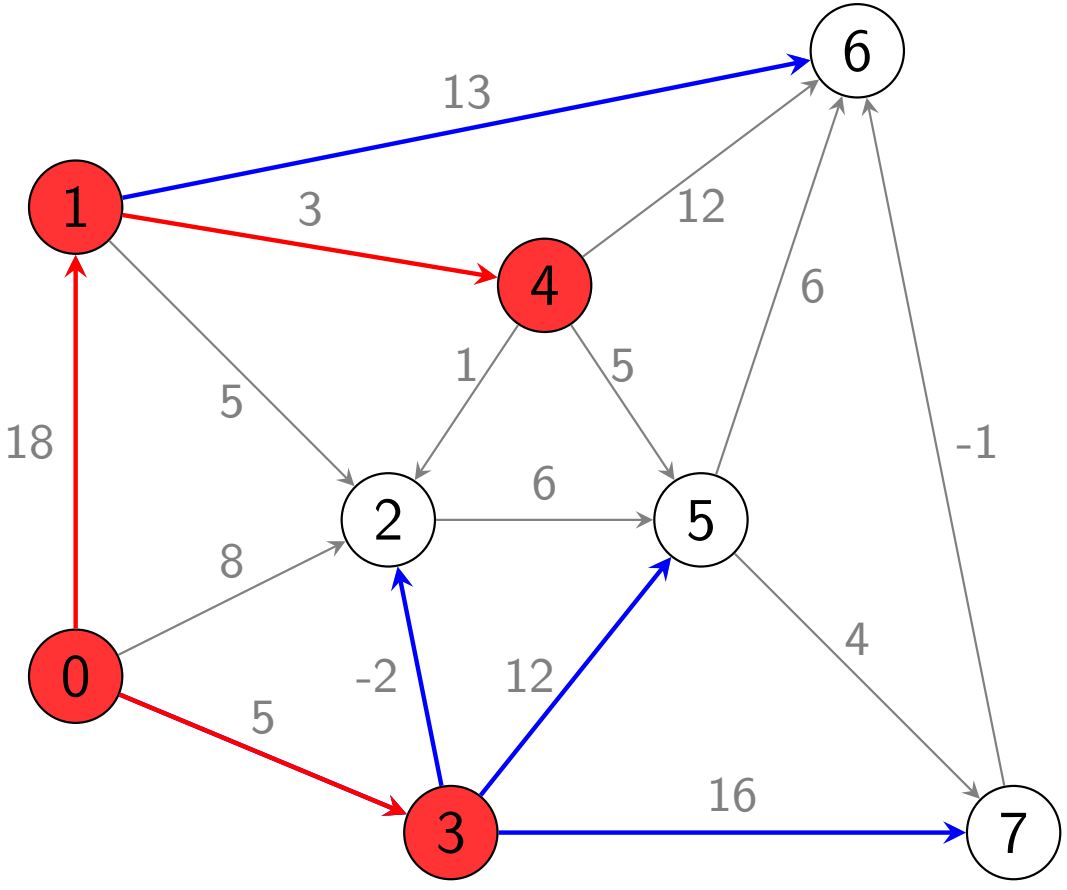


distance

0	0
1	18
2	3
3	5
4	21
5	17
6	31
7	21

Beispiel

Idee: Relaxiere Knoten in topologischer Reihenfolge
z.B. 0, 1, 3, 4, 2, 5, 7, 6

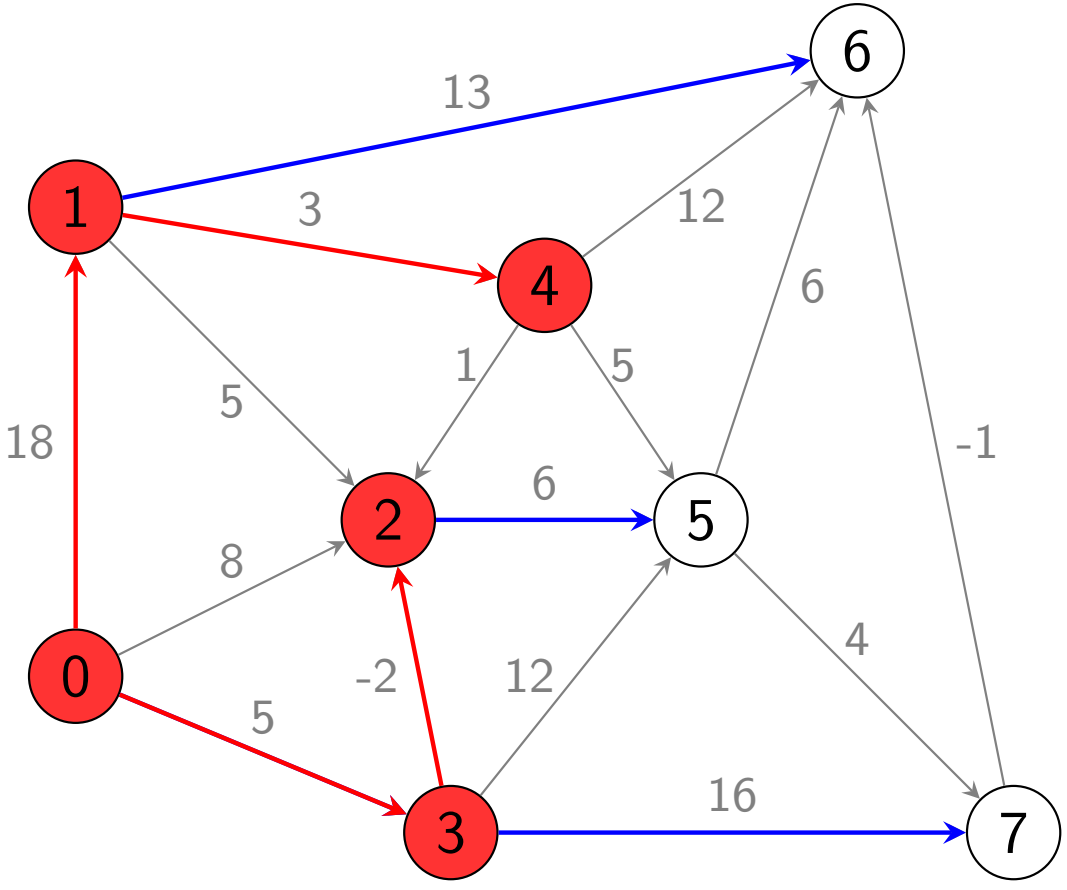


distance

0	0
1	18
2	3
3	5
4	21
5	17
6	31
7	21

Beispiel

Idee: Relaxiere Knoten in topologischer Reihenfolge
z.B. 0, 1, 3, 4, 2, 5, 7, 6

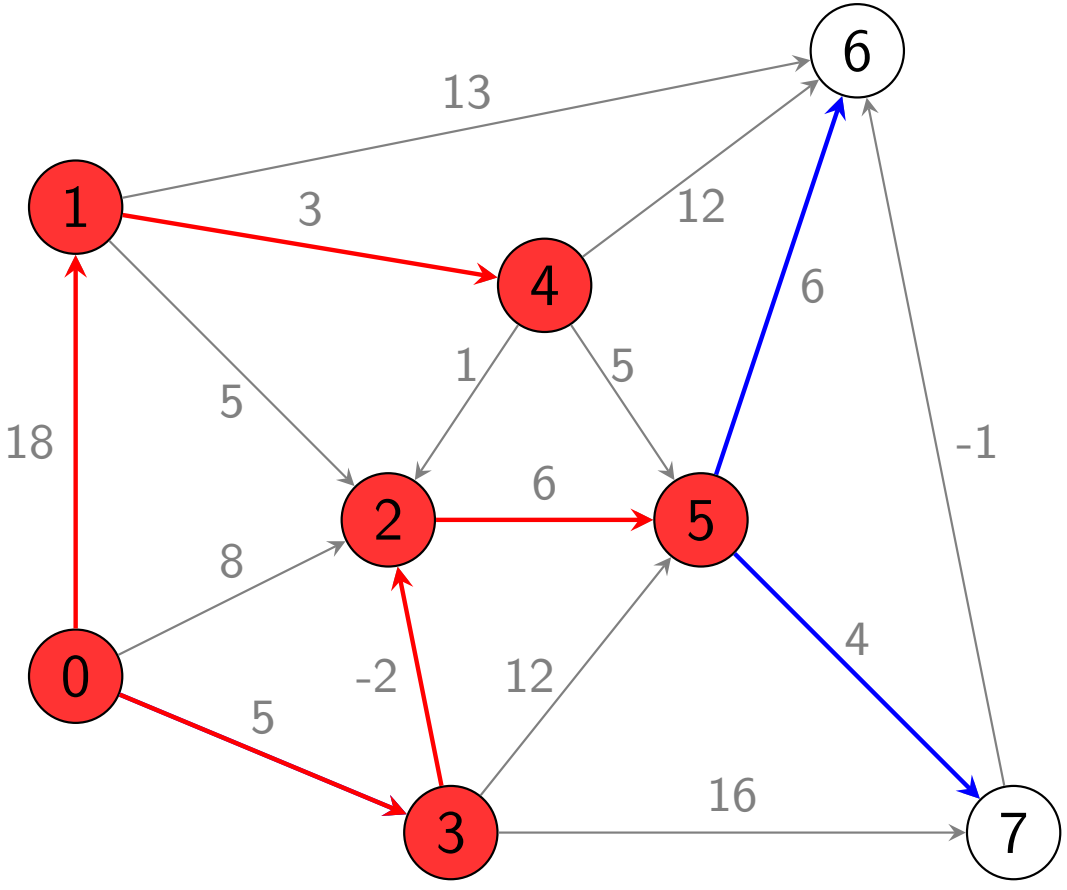


distance

0	0
1	18
2	3
3	5
4	21
5	9
6	31
7	21

Beispiel

Idee: Relaxiere Knoten in topologischer Reihenfolge
z.B. 0, 1, 3, 4, 2, 5, 7, 6

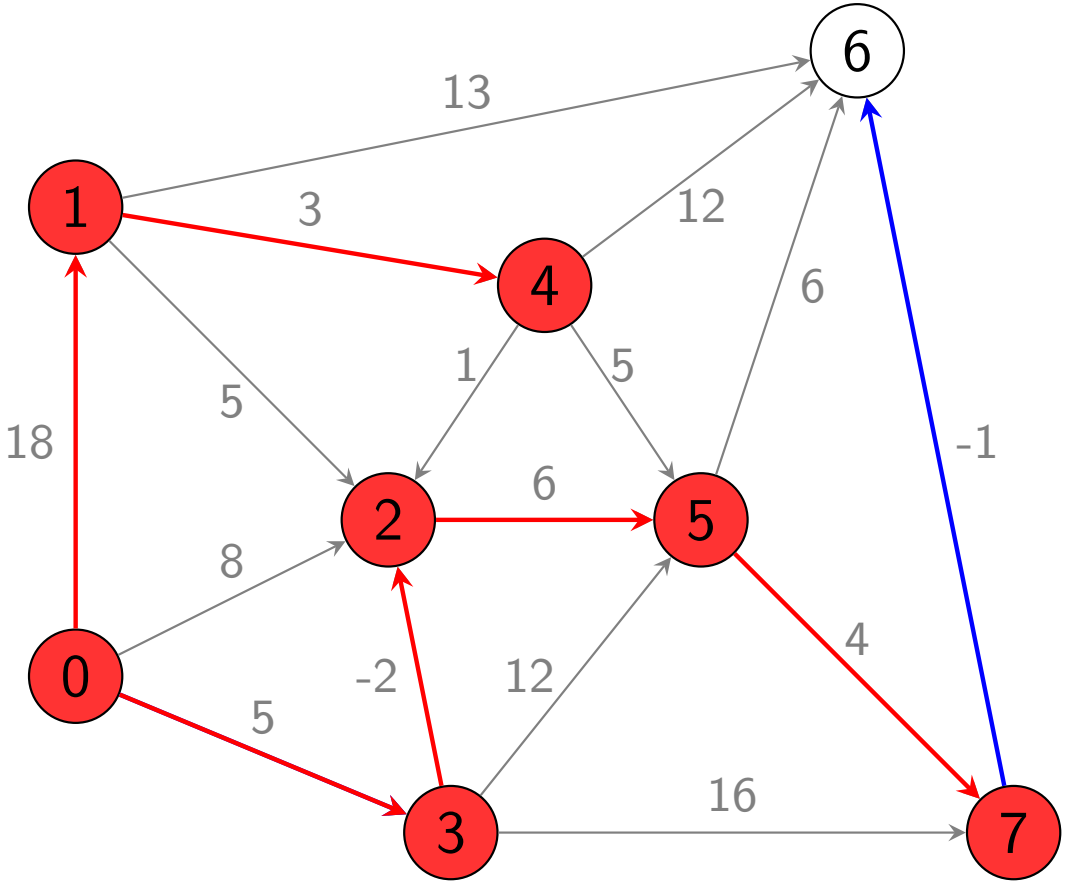


distance

0	0
1	18
2	3
3	5
4	21
5	9
6	15
7	13

Beispiel

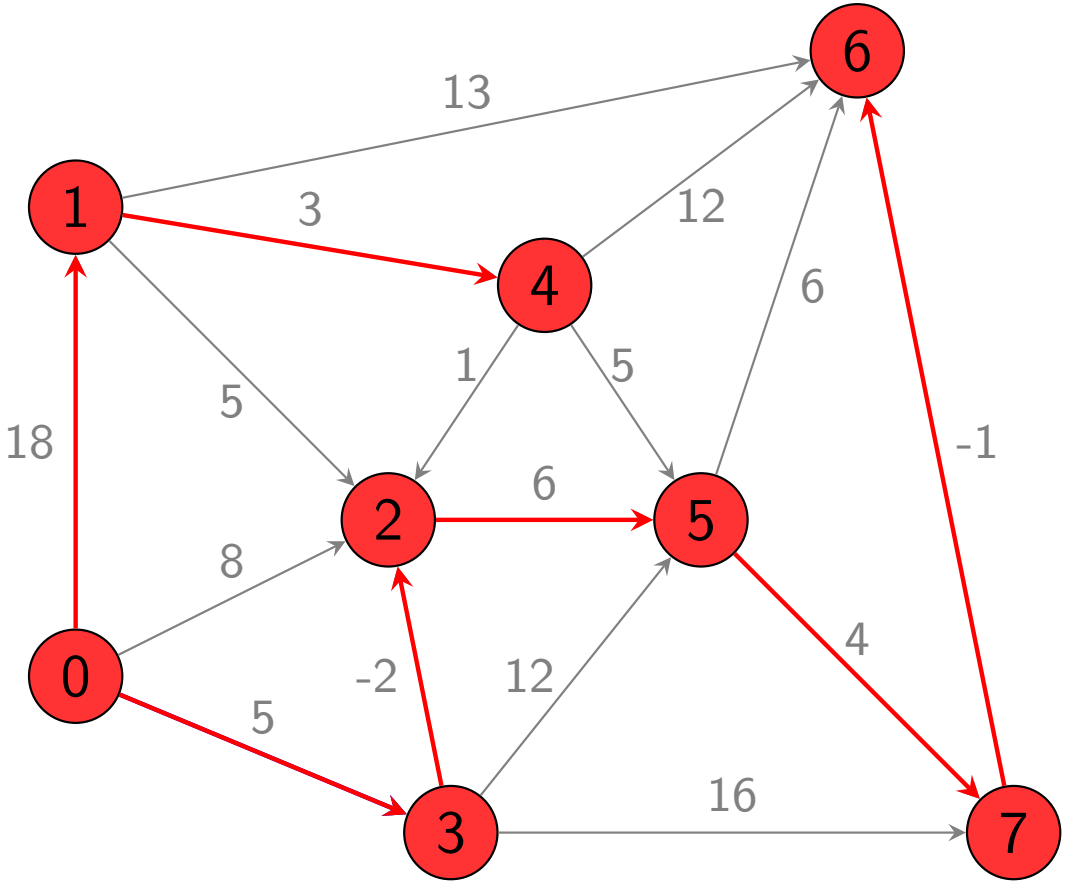
Idee: Relaxiere Knoten in topologischer Reihenfolge
z.B. 0, 1, 3, 4, 2, 5, 7, 6



	distance
0	0
1	18
2	3
3	5
4	21
5	9
6	12
7	13

Beispiel

Idee: Relaxiere Knoten in topologischer Reihenfolge
z.B. 0, 1, 3, 4, 2, 5, 7, 6



distance

0	0
1	18
2	3
3	5
4	21
5	9
6	12
7	13

Theorem

Theorem

Durch Relaxieren der Knoten in topologischer Reihenfolge wird das Single-Source-Shortest-Paths-Problem für kantengewichtete, azyklische Digraphen in Zeit $O(|E| + |V|)$ gelöst.

Theorem

Theorem

Durch Relaxieren der Knoten in topologischer Reihenfolge wird das Single-Source-Shortest-Paths-Problem für kantengewichtete, azyklische Digraphen in Zeit $O(|E| + |V|)$ gelöst.

Beweis.

- Jede Kante $e = (v, w)$ wird genau einmal relaxiert. Direkt danach gilt $\texttt{distance}[w] \leq \texttt{distance}[v] + \mathit{weight}(e)$.
- Ungleichung gilt bis zur Terminierung
  - $\texttt{distance}[w]$ wird nie grösser.
  - $\texttt{distance}[v]$ wird nicht mehr verändert, da alle eingehenden Kanten aufgrund der topologischen Sortierung bereits relaxiert wurden.

$\rightarrow$ Optimalitätskriterium ist bei Terminierung erfüllt. □

Verwandte Probleme: Längste Pfade

Definition (Längste Pfade in azylischen Graphen)

Gegeben: Kantengewichteter, azyklischer Digraph, Startknoten s
Gefragt: Gibt es einen Pfad von s zu Knoten v?
Falls ja, finde den Pfad mit maximalem Gewicht.

Verwandte Probleme: Längste Pfade

Definition (Längste Pfade in azylischen Graphen)

Gegeben: Kantengewichteter, azyklischer Digraph, Startknoten s
Gefragt: Gibt es einen Pfad von s zu Knoten v?
Falls ja, finde den Pfad mit maximalem Gewicht.

Multipliziere alle Kantengewichte mit -1 und verwende Kürzeste-Pfade-Algorithmus.

Verwandte Probleme: Kritischer Pfad

Gegeben:

- Menge von Aufgaben a, jede benötigt gegebene Zeit t_a
- Bedingungen $a \to a'$, dass a fertiggestellt sein muss, bevor a' begonnen werden kann (in lösbaren Problemen zykelfrei).

Verwandte Probleme: Kritischer Pfad

Gegeben:

- Menge von Aufgaben a, jede benötigt gegebene Zeit t_a
- Bedingungen $a \to a'$, dass a fertiggestellt sein muss, bevor a' begonnen werden kann (in lösbaren Problemen zykelfrei).

Frage:

- Annahme: Beliebig viele Aufgaben parallel ausführbar
- Wie lange benötigen Sie für die Erledigung aller Aufgaben?

Verwandte Probleme: Kritischer Pfad

Erstelle kantengewichteten Digraphen

- Knoten s, e + für jede Aufgabe a zwei Knoten a_s und a_e
- für alle a:
  - Kante (s, a_s) mit Gewicht 0
  - Kante (a_e, e) mit Gewicht 0
  - Kante $(a_\mathrm{s}, a_\mathrm{e})$ mit Gewicht t_a
- für jede Bedingung $a \to a'$ Kante $(a_\mathrm{e}, a'_\mathrm{s})$ mit Gewicht 0

Verwandte Probleme: Kritischer Pfad

Erstelle kantengewichteten Digraphen

- Knoten s, e + für jede Aufgabe a zwei Knoten a_s und a_e
- für alle a:
  - Kante (s, a_s) mit Gewicht 0
  - Kante (a_e, e) mit Gewicht 0
  - Kante (a_s, a_e) mit Gewicht t_a
- für jede Bedingung $a \to a'$ Kante (a_e, a'_s) mit Gewicht 0

Kritischer Pfad für Aufgabe a ist längster Pfad von s zu a_s.
Wähle Startzeit für a als Gewicht eines kritischen Pfades.

Verwandte Probleme: Kritischer Pfad

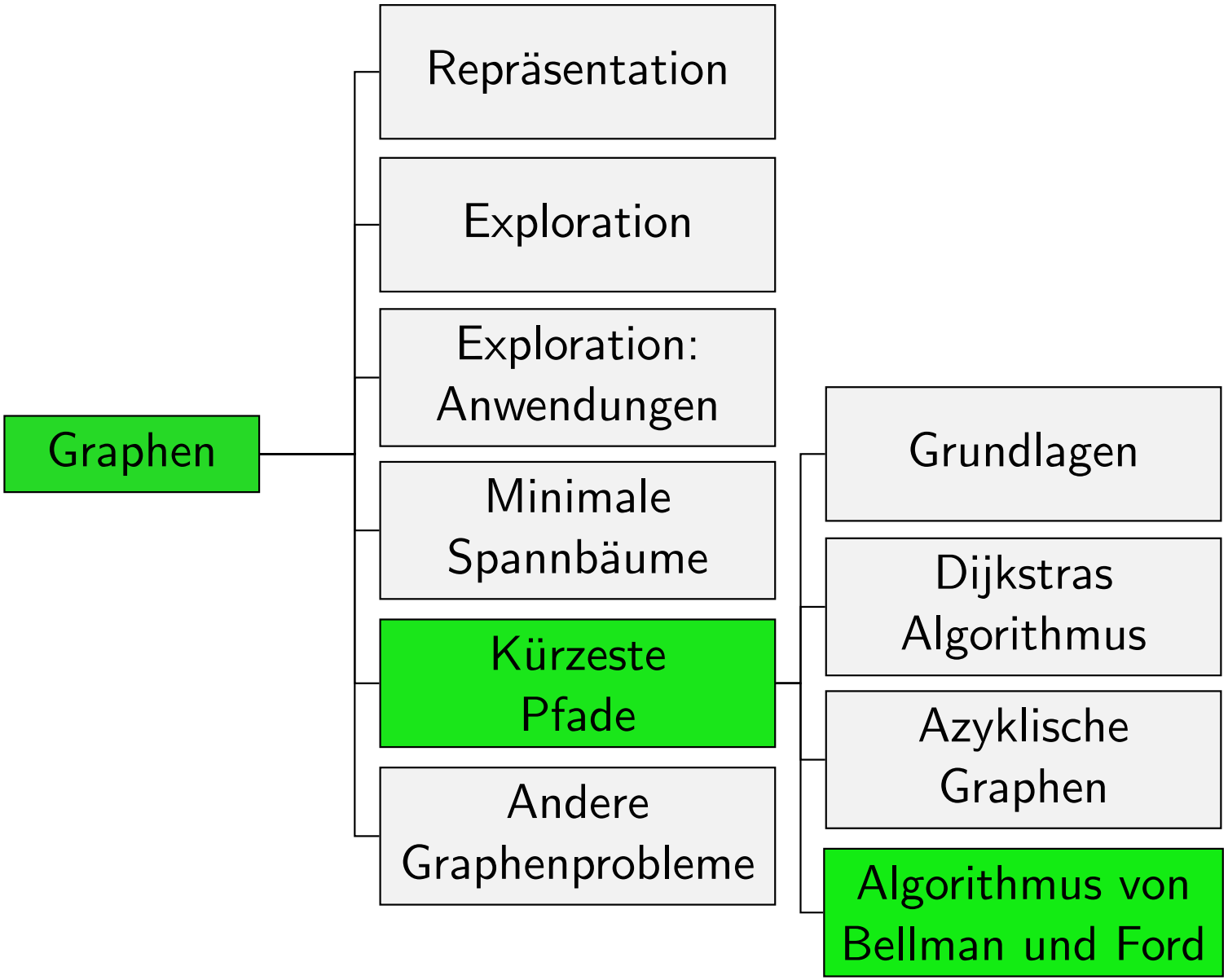
Erstelle kantengewichteten Digraphen

- Knoten s, e + für jede Aufgabe a zwei Knoten a_s und a_e
- für alle a:
  - Kante (s, a_s) mit Gewicht 0
  - Kante (a_e, e) mit Gewicht 0
  - Kante (a_s, a_e) mit Gewicht t_a
- für jede Bedingung $a \to a'$ Kante (a_e, a'_s) mit Gewicht 0

Kritischer Pfad für Aufgabe a ist längster Pfad von s zu a_s.
Wähle Startzeit für a als Gewicht eines kritischen Pfades.
$\to$ Ergibt optimale Gesamtausführungszeit
(= Gewicht von längstem Pfad von s zu e)

Bellman-Ford-Algorithmus

Graphen: Übersicht



Problem

- Bei negativen Kantengewichten kann es negative Zyklen geben, d.h. Zyklen, bei denen die Summe der Kantengewichte negativ ist.
- Liegt ein Knoten eines solchen Zyklus auf einem Pfad von s nach v, können wir Pfade finden, deren Gewicht niedriger als jeder gegebene Wert ist.
  $\rightarrow$ kein korrekt gestelltes Problem
- Alternative Fragestellung: Finde kürzesten einfachen Pfad?
  $\rightarrow$ NP-schweres (= sehr schwieriges) Problem

Fragestellung

In vielen praktischen Anwendungen sind negative Zyklen ein Hinweis auf einen Modellierungsfehler.

Neue Fragestellung

Gegeben: Gewichteter Digraph, Startknoten s

Gefragt: Ist von s aus ein negativer Zyklus erreichbar?
Falls nein, berechne den Kürzeste-Pfade-Baum zu allen erreichbaren Knoten.

Bellman-Ford-Algorithmus: High-Level-Perspektive

In Graphen ohne negative Zyklen (aber mit negativen Gewichten):

Bellman-Ford-Algorithmus

- Initialisiere $\mathit{distance}[s] = 0$ für Startknoten s, $\mathit{distance}[n] = \infty$ für alle anderen Knoten.
- Dann $|V|$ Durchläufe, in denen jeweils alle Kanten relaxiert werden.

Bellman-Ford-Algorithmus: High-Level-Perspektive

In Graphen ohne negative Zyklen (aber mit negativen Gewichten):

Bellman-Ford-Algorithmus

- Initialisiere $distance[s] = 0$ für Startknoten s, $distance[n] = \infty$ für alle anderen Knoten.
- Dann $|V|$ Durchläufe, in denen jeweils alle Kanten relaxiert werden.

Proposition

Das Verfahren löst das Single-Source-Shortest-Paths-Problem für Graphen ohne negative Zyklen in Zeit $O(|E||V|)$ und mit zusätzlichem Speicher $O(|V|)$.

Beweisidee: Nach i Durchgängen ist jeder Pfad zu v mindestens so kurz wie jeder Pfad zu v mit höchstens i Kanten.

Effizientere Variante

- Ändert sich *distance*[v] in Durchgang i nicht, ändert auch keine Relaxierung einer von v ausgehenden Kante in Durchgang $i + 1$ etwas.
- Idee: Merke dir Knoten mit veränderter *distance* in Queue.
- In der Praxis deutlich schneller, auch wenn sich das Worst-Case-Verhalten nicht verbessert.

Was ist mit negativen Zyklen?

- Ist von s aus kein negativer Zyklus erreichbar, wird im $|V|$-ten Durchgang keine Knotendistanz mehr geupdated.
- Gibt es einen negativen Zyklus, führt dies zu einem Zyklus mit den in `edge_to` gespeicherten Kanten.
- In der Praxis testen wir das nach jedem Durchlauf.

Bellman-Ford-Algorithmus

```
1  class BellmanFordSSSP:
2      def __init__(self, graph, start_node):
3          self.edge_to = [None] * graph.no_nodes()
4          self.distance = [float('inf')] * graph.no_nodes()
5          self.in_queue = [False] * graph.no_nodes()
6          self.queue = deque()
7          self.calls_to_relax = 0
8          self.cycle = None
9
10         self.distance[start_node] = 0
11         self.queue.append(start_node)
12         self.in_queue[start_node] = True
13         while (not self.has_negative_cycle() and
14                 self.queue):  # queue not empty
15             node = self.queue.popleft()
16             self.in_queue[node] = False
17             self.relax(graph, node)
18
```

Bellman-Ford-Algorithmus (Fortsetzung)

```
19      def relax(self, graph, v):
20          for edge in graph.adjacent_edges(v):
21              w = edge.to_node()
22              if self.distance[v] + edge.weight() < self.distance[w]:
23                  self.edge_to[w] = edge
24                  self.distance[w] = self.distance[v] + edge.weight()
25                  if not self.in_queue[w]:
26                      self.queue.append(w)
27                      self.in_queue[w] = True
28          self.calls_to_relax += 1
29          if self.calls_to_relax % graph.no_nodes() == 0:
30              self.find_negative_cycle()
31
```

Bellman-Ford-Algorithmus (Fortsetzung)

```
32      def has_negative_cycle(self):
33          return self.cycle is not None
34
35      def find_negative_cycle(self):
36          no_nodes = len(self.distance)
37          graph = EdgeWeightedDigraph(no_nodes)
38          for edge in self.edge_to:
39              if edge is not None:
40                  graph.add_edge(edge)
41
42          cycle_finder = WeightedDirectedCycle(graph)
43          self.cycle = cycle_finder.get_cycle()
```

`WeightedDirectedCycle` detektiert gerichtete Zykel in gewichteten Graphen.

→ Folge von Tiefensuchen wie in `DirectedCycle` (C2)

Zusammenfassung

Zusammenfassung

- Nicht-negative Gewichte
  - Sehr häufiges Problem
  - Dijkstras Algorithmus mit Laufzeit $O(|E| \log |V|)$
- Azyklische Graphen
  - Kommt in manchen Anwendungen vor und sollte ausgenutzt werden.
  - Mit topologischer Sortierung in linearer Zeit $O(|E| + |V|)$
- Negative Gewichte oder negative Zykel
  - Gibt es keinen negativen Zyklus findet der Bellman-Ford-Algorithmus kürzeste Pfade.
  - Sonst findet er einen negativen Zyklus.