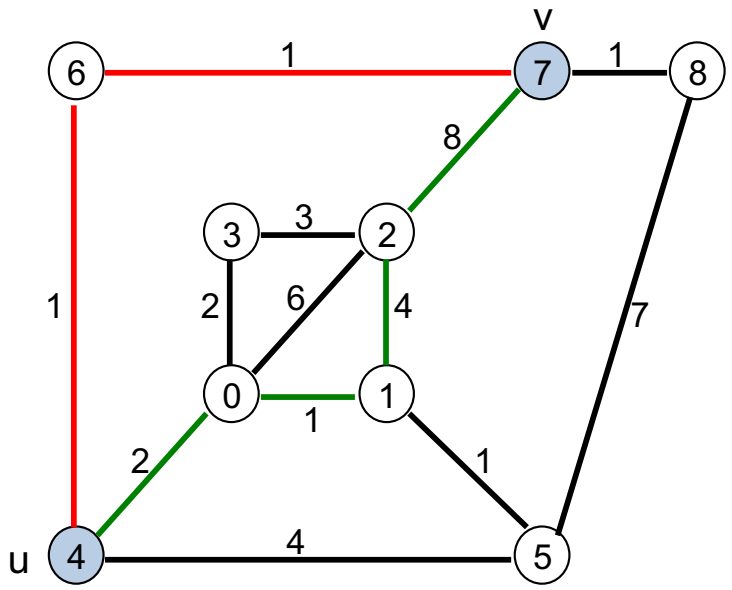
9. Kürzeste Wege in Graphen

- Problemstellung
- Erweiterte Breitensuche
- Dijkstra- und A*-Algorithmus
- Moore-Ford-Algorithmus
- Netzpläne und erweiterte topologische Sortierung
- Alle kürzesten Wege mit Floyd-Algorithmus

Problemstellung kürzeste Wege (1)

- Im allgemeinen gibt es zwischen zwei Knoten u und v mehrere Wege.
- Ein Weg von u nach v heißt kürzester Weg, falls der Weg minimale Länge hat.
- Weglänge bei gewichtetem Graph ist Summe der Kantengewichte.
- Weglänge bei ungewichtetem Graph ist Anzahl der Kanten.
- Die Distanz dist(u,v) zwischen u und v ist die Länge eines kürzesten Weges. dist(u,v) = ∞, falls es keinen Weg von u nach v gibt.



Weg von 4 nach 7 mit Länge 15.

Kürzester Weg von 4 nach 7 mit Länge 2.

Damit ist Distanz dist(4,7) = 2.

Problemstellung kürzeste Wege (2)

- Länge eines Weges muss nicht die wörtliche Bedeutung haben.
- Die Gewichte (Kosten) der Kanten und damit die Weglängen können in Abhängigkeit von der Anwendung ganz unterschiedliche Bedeutungen haben.

  Beispiele:

  – Streckenlänge
  – Zeitspannen
  – Kosten
  – Profit: Gewinn/Verlust (Gewichte können positiv und negativ sein)
  – Wahrscheinlichkeiten

Verschiedene Problemvarianten

(1) Single pair shortest path:
Kürzeste Wege zwischen zwei Knoten:

(2) Single source shortest path:
Kürzeste Wege zwischen einem Startknoten s und allen anderen Knoten

(3) All pairs shortest path:
Kürzeste Wege zwischen allen Knotenpaaren.

- Für Problem (1) kennt man keine bessere Lösung, als einen Algorithmus für Lösung (2) zu nehmen, der abgebrochen wird, sobald der Zielknoten erreicht wird.
- Problem (3) kann auf Problem (2) zurückgeführt werden.
  Bei dicht besetzten Graphen gibt es jedoch eine effizientere Lösung, die zudem auch negative Kantengewichte zulässt.

Verschiedene Algorithmen

Algorithmus	Problem-variante	Ungerichtet/ gerichtet	Kanten-Gewichte	Zyklenfreiheit
Breitensuche	Single Source Shortest Path	beides	ungewichtet	egal
Algorithmus von Dijkstra; A*-Verfahren	Single Source Shortest Path	beides	pos.	egal
Algorithmus von Moore und Ford	Single Source Shortest Path	gerichtet	pos./neg.	keine Zyklen neg. Länge
Erweiterte topologische Sortierung	Single Source Shortest Path	gerichtet	pos./neg.	zyklenfrei
Algorithmus von Floyd	All Pairs Shortest Path	ungerichtet	pos.	egal
		gerichtet	pos./neg.	keine Zyklen neg. Länge

9. Kürzeste Wege in Graphen

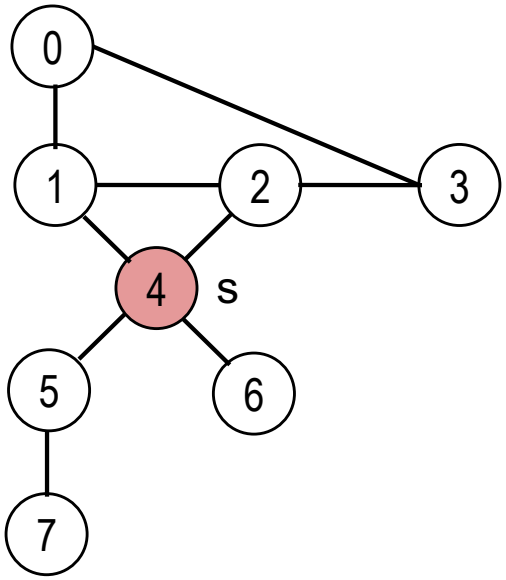
- Problemstellung
- Erweiterte Breitensuche
- Dijkstra- und A*-Algorithmus
- Moore-Ford-Algorithmus
- Netzpläne und erweiterte topologische Sortierung
- Alle kürzesten Wege mit Floyd-Algorithmus

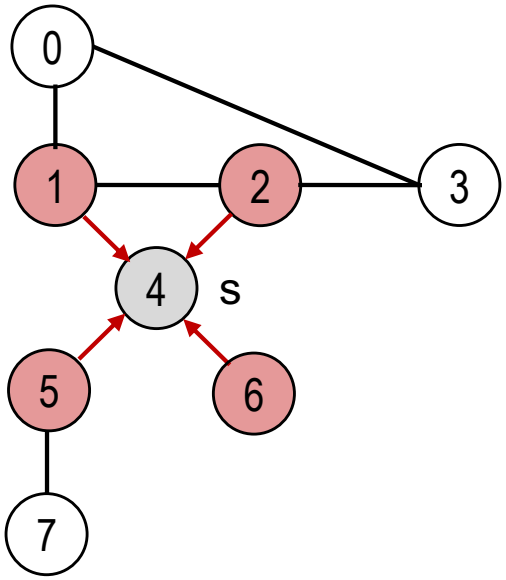
Verschiedene Algorithmen

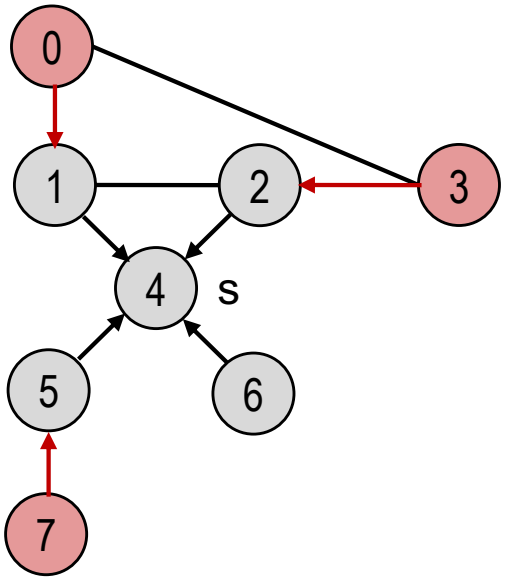
Algorithmus	Problem-variante	Ungerichtet/ gerichtet	Kanten-Gewichte	Zyklenfreiheit
Breitensuche	Single Source Shortest Path	beides	ungewichtet	egal
Algorithmus von Dijkstra; A*-Verfahren	Single Source Shortest Path	beides	pos.	egal
Algorithmus von Moore und Ford	Single Source Shortest Path	gerichtet	pos./neg.	keine Zyklen neg. Länge
Erweiterte topologische Sortierung	Single Source Shortest Path	gerichtet	pos./neg.	zyklenfrei
Algorithmus von Floyd	All Pairs Shortest Path	ungerichtet	pos.	egal
		gerichtet	pos./neg.	keine Zyklen neg. Länge

Erweiterte Breitensuche

- Beginne bei Startknoten s und durchlaufe Graph mit Breitensuche.
- Speichere in einer Queue die Knoten, die als nächstes besucht werden können (Kandidaten).
- Distanzfeld d: Für jeden besuchten Knoten oder Kandidat v ist d[v] = dist(s,v). Für die restlichen Knoten v ist d[v] = ∞.
- Kürzeste Wege werden im Vorgängerfeld p[v] gespeichert (p = predecessor).







v	0	1	2	3	4	5	6	7
d[v]	∞	∞	∞	∞	0	∞	∞	∞
p[v]	-	-	-	-	-	-	-	-

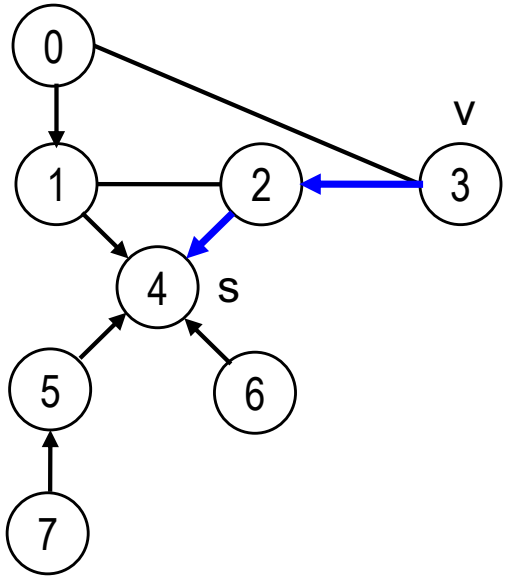
v	0	1	2	3	4	5	6	7
d[v]	∞	1	1	∞	0	1	1	∞
p[v]	-	4	4	-	-	4	4	-

v	0	1	2	3	4	5	6	7
d[v]	2	1	1	2	0	1	1	2
p[v]	1	4	4	2	-	4	4	5

Vorgängerfeld p

- p[v] = Vorgängerknoten im kürzesten Weg von s nach v.
- p enthält alle kürzeste Wege mit Startknoten s.
- p stellt ein (Wurzel-)Baum mit Wurzel s dar: p[v] ist der Elternknoten von v.
- Der kürzeste Weg von s nach v ergibt sich in umgekehrter Reihenfolge:

  v, p[v], p[p[v]], …, p[…p[p[v]]…] = s



v	0	1	2	3	4	5	6	7
d[v]	2	1	1	2	0	1	1	2
p[v]	1	4	4	2	-	4	4	5

kürzester Weg von s = 4 nach v = 3 in umgekehrter Reihenfolge:

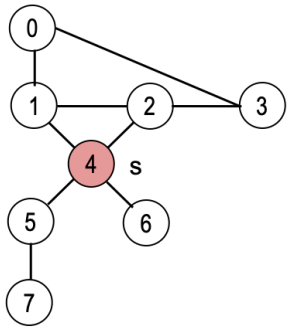
3, 2, 4

Algorithmus für erweiterte Breitensuche

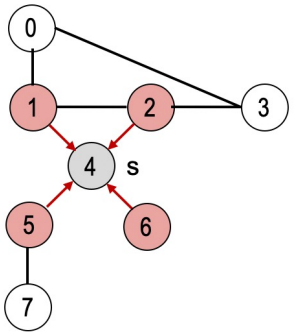
Eingabe: Startknoten s und ungewichteter Graph G

Ausgabe: Distanzfeld d und Vorgängerfeld p

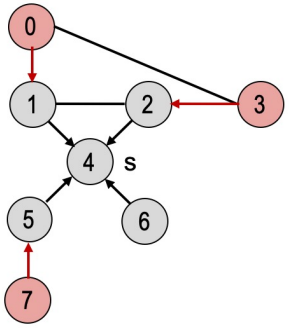
```
void shortestPath (Vertex s, Graph G, int[ ] d, Vertex[ ] p) {
        for ( jeden Knoten v ) {
                d[v] = ∞;
                p[v] = undef;
        }
        d[s] = 0; // Distanz für Startknoten

        Queue<Vertex> q;
        q.add(s);

        while (! q.empty() )  {
                v = q.remove();
                for ( jeden adjazenten Knoten w von v  ) {
                        if (d[w] == ∞) { // w noch nicht registriert
                                d[w] = d[v] + 1;
                                p[w] = v;
                                q.add(w);
                        }
                }
        }
}
```



v	0	1	2	3	4	5	6	7
d[v]	∞	∞	∞	∞	0	∞	∞	∞
p[v]	-	-	-	-	-	-	-	-



v	0	1	2	3	4	5	6	7
d[v]	∞	1	1	∞	0	1	1	∞
p[v]	-	4	4	-	-	4	4	-



v	0	1	2	3	4	5	6	7
d[v]	2	1	1	2	0	1	1	2
p[v]	1	4	4	2	-	4	4	5

Analyse der Breitensuche

- Für jeden Knoten w kann d[w] höchstens einmal von ∞ auf einen endlichen Wert gesetzt werden. Damit kommen in die Queue q höchstens |V| Knoten. Die while-Schleife läuft damit höchstens |V|-mal.

- Jede Kantenrichtung wird höchstens einmal in einer der for-Schleifen betrachtet.
  Werden Adjazenzlisten verwendet, ist der Aufwand für den Durchlauf aller for-Schleifen gleich O(|E|).

- Damit ergibt sich mit Adjazenzlisten insgesamt:

  $T = O(|V| + |E|)$

9. Kürzeste Wege in Graphen

- Problemstellung
- Erweiterte Breitensuche
- Dijkstra- und A*-Algorithmus
- Moore-Ford-Algorithmus
- Netzpläne und erweiterte topologische Sortierung
- Alle kürzesten Wege mit Floyd-Algorithmus

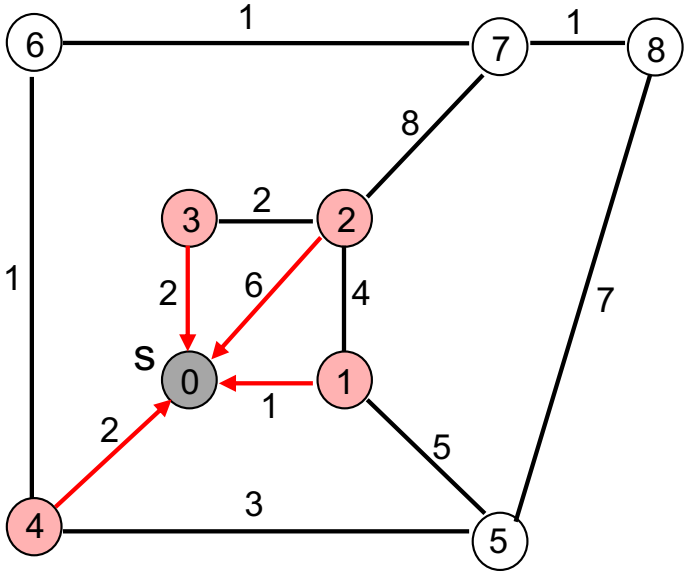
Verschiedene Algorithmen

Algorithmus	Problem-variante	Ungerichtet/ gerichtet	Kanten-Gewichte	Zyklenfreiheit
Breitensuche	Single Source Shortest Path	beides	ungewichtet	egal
Algorithmus von Dijkstra; A*-Verfahren	Single Source Shortest Path	beides	pos.	egal
Algorithmus von Moore und Ford	Single Source Shortest Path	gerichtet	pos./neg.	keine Zyklen neg. Länge
Erweiterte topologische Sortierung	Single Source Shortest Path	gerichtet	pos./neg.	zyklenfrei
Algorithmus von Floyd	All Pairs Shortest Path	ungerichtet	pos.	egal
		gerichtet	pos./neg.	keine Zyklen neg. Länge

Graphen mit positiven Gewichten werden auch Distanzgraphen genannt.

Algorithmus von Dijkstra – Idee (1)

- Drei Arten von Knoten:
  - Bereits besuchte Knoten (in grau).
  - Kandidaten, die als nächstes besucht werden können (in rot)
  - Restliche Knoten (in weiss).
- Distanzfeld d:
  - d[v] = dist(s,v) für bereits besuchte Knoten
  - d[v] ≥ dist(s,v) und d[v] < ∞ für Kandidaten. d-Wert kann sich noch verbessern.
  - d[v] = ∞ für die restlichen Knoten
- Kürzeste Wege werden im Vorgängerfeld p gespeichert (wie bei erweiterter Breitensuche).



v	0	1	2	3	4	5	6	7	8
d[v]	0	1	6	2	2	∞	∞	∞	∞
p[v]	-	0	0	0	0	-	-	-	-

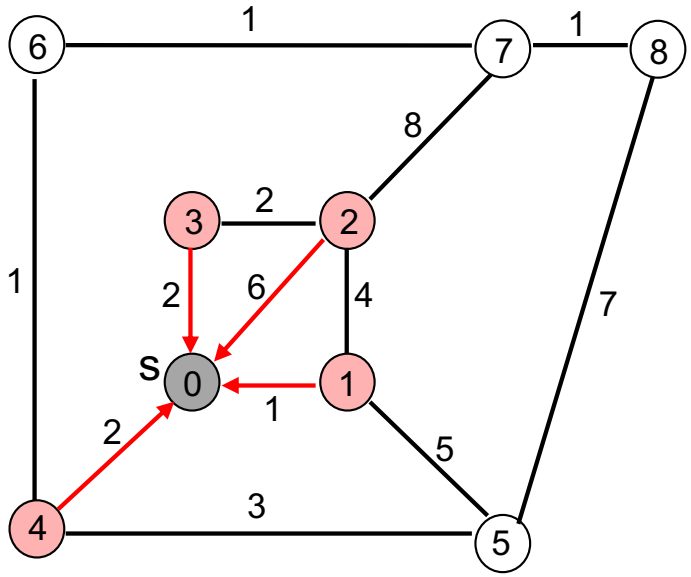
Kandidaten

Bereits besucht

Startknoten s = 0.

Algorithmus von Dijkstra – Idee (2)

- Aus der Kandidatenliste wird immer der Knoten v mit dem kleinsten d-Wert als nächstes besucht.
- Genau dann ist ein kürzester Weg von s nach v bekannt.
  Also d[v] = dist(s,v).
- Der kürzeste Weg nach von s nach v kann dann aus dem p-Feld abgelesen werden.



Als nächstes wird Knoten 1 mit Distanzwert 1 besucht.

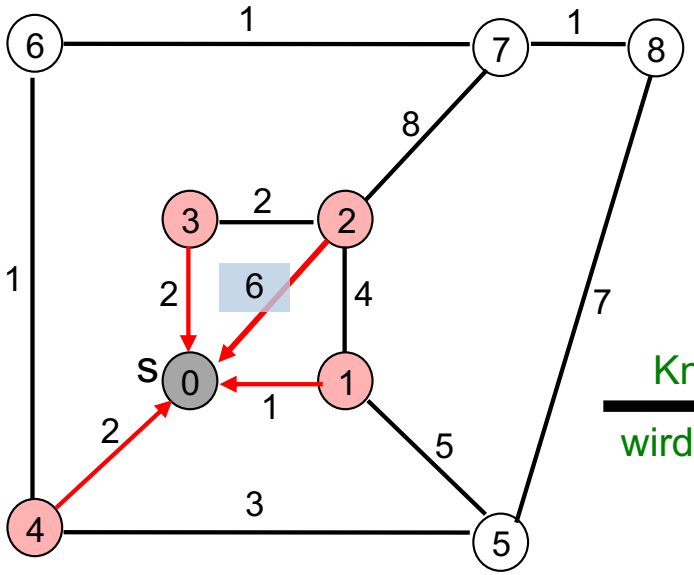
v	0	1	2	3	4	5	6	7	8
d[v]	0	1	6	2	2	∞	∞	∞	∞
p[v]	-	0	0	0	0	-	-	-	-

Kandidaten

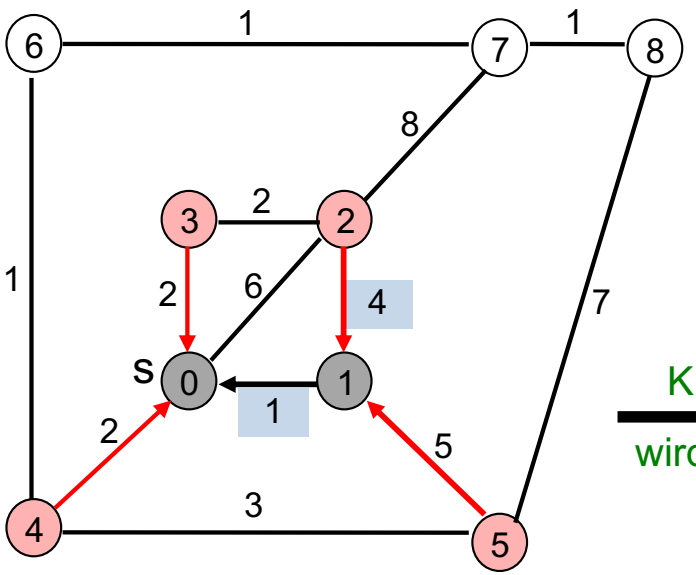
Bereits besucht

Algorithmus von Dijkstra – Idee (3)

- Sobald ein Knoten v besucht wird, werden alle Nachbarknoten w inspiziert:
  - falls w noch nie gesehen wurde (restlicher Knoten), dann wird w zur Kandidatenliste hinzugefügt
  - falls w schon in Kandidatenliste enthalten ist, dann wird geprüft, ob ein Weg über v einen kürzeren Weg ergibt.



Knoten 1 wird besucht



Knoten 3 wird besucht …

v	0	1	2	3	4	5	6	7	8
d[v]	0	1	6	2	2	∞	∞	∞	∞
p[v]	-	0	0	0	0	-	-	-	-

v	0	1	2	3	4	5	6	7	8
d[v]	0	1	5	2	2	6	∞	∞	∞
p[v]	-	0	1	0	0	1	-	-	-

Knoten 5 ist zur Kandidatenliste neu hinzugefügt worden.

Der Distanzwert von Knoten 2 hat sich von 6 auf 5 verbessert, da der Weg über Knoten 1 kürzer ist als der bisherige.

Algorithmus von Dijkstra

```
void shortestPath (Vertex s, DistanzGraph G,  double[ ] d, Vertex[ ] p ) {

        kl = ∅;    //  Kandidatenliste
        for (jeden Knoten v) {
              d[v] = ∞;
              p[v] = undef;
        }
        d[s] = 0;   // Startknoten
(1)     kl.insert(s, 0);

        while (! kl.empty() )  {
(2)           v = kl.delMin();   // lösche Knoten v mit kleinstem d-Wert
              for ( jeden adjazenten Knoten w von v )
                    if (d[w] == ∞)  { // w noch nicht besucht und nicht in Kandidatenliste
                          p[w] = v;
                          d[w] = d[v] + c(v,w);
(3)                       kl.insert(w, d[w]);
                    } else if (d[v] + c(v,w) < d[w]) {
                          p[w] = v;
                          d[w] = d[v] + c(v,w);
(4)                       kl.change(w, d[w]);
                    }
        }
}
```

Eingabe: Startknoten s und Distanzgraph G
Ausgabe: Distanzfeld d und Vorgängerfeld p

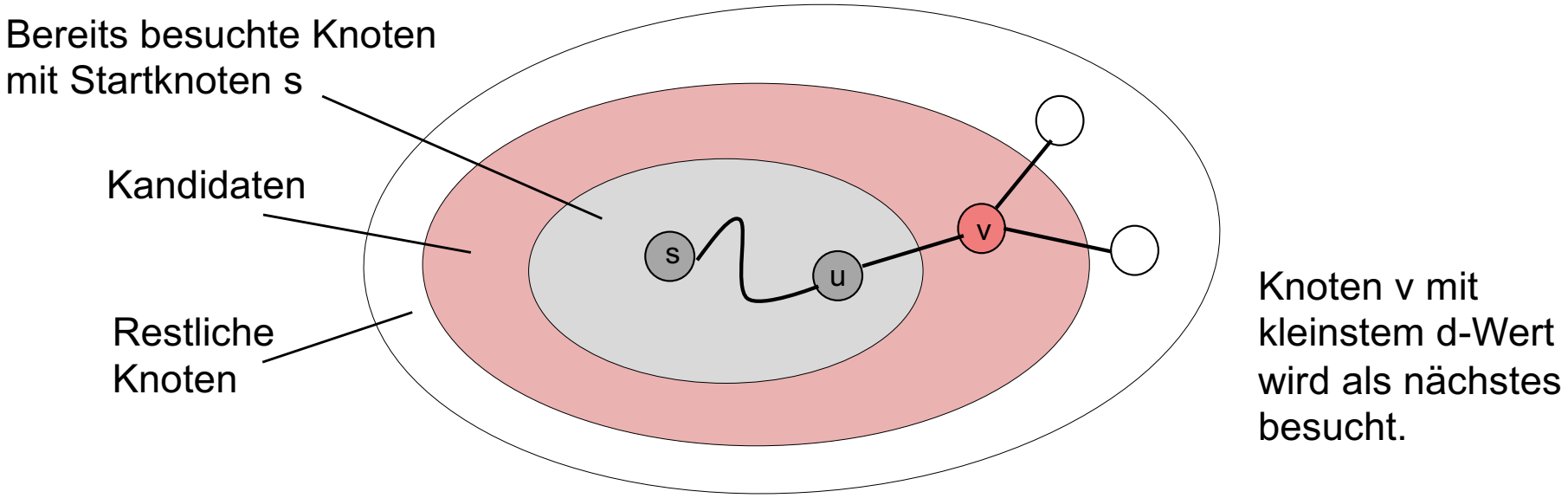
- Kandidatenliste kl (Knoten die als nächstes besucht werden können)
- Implementierungen:
  - Index-Heap
  - Unsortierte Liste

Knoten v wird besucht!
d[v] und p[v] sind nun endgültig.

Knoten w kommt zu den Kandidaten dazu.
Kürzester Weg nach w geht über v.

Distanzwert für w verbessert sich.
Kürzester Weg nach w geht nun über v.

Dijkstra berechnet kürzeste Wege - Beweis (1)



Behauptung:

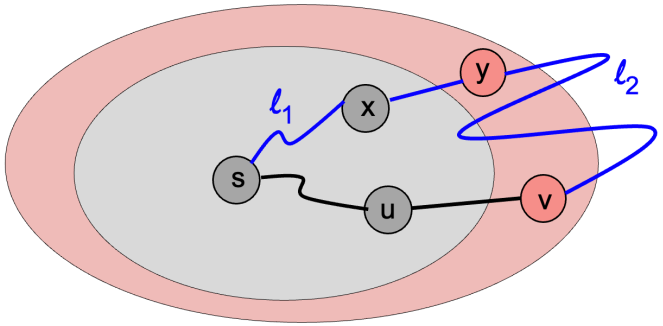
Falls Knoten v als nächstes besucht wird, dann ist d[v] = dist(s,v).
D.h. d[v] ist dann die Länge eines kürzesten Wegs von s nach v.

- Beweis induktiv über die Anzahl der besuchten Knoten.
- Induktionsanfang: v = s. Es gilt d[s] = dist(s,s) = 0.
- Induktionsschritt: Für bereits besuchte Knoten gelte die Behauptung (IndVor).
  Die Behauptung ist dann für v zu beweisen. → nächste Seite

Dijkstra berechnet kürzeste Wege - Beweis (2)

- Beweise Induktionsschritt indirekt.
- Annahme: es gibt einen kürzeren Weg p von s nach v mit Länge $\ell < d[v]$.
- Weg p muss über einen Kandidaten gehen.
  y sei der erste Kandidaten-Knoten im Weg p.
  (y kann auch v sein.)
  Vorgänger von y in diesem Weg sei x.
  x muss bereits besucht sein.

Kürzerer Weg p von s nach v mit Länge $\ell = \ell_1 + c(x,y) + \ell_2 < d[v]$.



- Für Länge ℓ des Wegs p gilt:

$\ell = \ell_1 + c(x,y) + \ell_2$

$\geq dist(s,x) + c(x,y)$	$\ell_1 \geq dist(s,x)$ und $\ell_2 \geq 0$
$= d[x] + c(x,y)$	(IndVor) für besuchten Knoten x
$\geq d[y]$	d[y] wurde über alle besuchten Nachbarn minimiert
$\geq d[v]$	v ist Kandidat mit kleinstem d-Wert

- Also ist $\ell \geq d[v]$. Damit Widerspruch zur Annahme.

Wichtige Folgerung: für bereits besuchte Knoten kann sich d-Wert nicht mehr verbessern

```
void shortestPath (Vertex s, DistanzGraph G,  double[ ] d, Vertex[ ] p ) {

        kl = ∅;    //  Kandidatenliste
        for (jeden Knoten v) {
              d[v] = ∞;
              p[v] = undef;
        }
        d[s] = 0;   // Startknoten
(1)     kl.insert(s, 0);

        while (! kl.empty() )  {
(2)           v = kl.delMin();   // lösche Knoten v mit kleinstem d-Wert
              for ( jeden adjazenten Knoten w von v )
                    if (d[w] == ∞)  { // w noch nicht besucht und nicht in Kandidatenliste
                          p[w] = v;
                          d[w] = d[v] + c(v,w);
(3)                       kl.insert(w, d[w]);
                    } else if (d[v] + c(v,w) < d[w]) {
                          p[w] = v;
                          d[w] = d[v] + c(v,w);
(4)                       kl.change(w, d[w]);
                    }
        }
}
```

- Distanzwert für w verbessert sich.
- w kann kein bereits besuchter Knoten sein, denn die besuchten Knoten haben schon kleinstmögliche d-Werte.
- w muss also ein Kandidat sein, d.h. w muss sich in in der Prioritätsliste kl befinden!

Laufzeit-Analyse - Kandidatenliste als Index-Heap

a) Ein Index-Heap (spezielle Implementierung einer Prioritätsliste, Kap. 6) bietet eine effiziente Realisierung folgender Operationen an:

- insert(v, d) in O(log|V|):
  fügt Element v mit Priorität d ein. Aufruf in (1) und (3).
- v = delMin() in O(log|V|):
  löscht Element mit kleinster Priorität. Aufruf in (2).
- change(v, dNeu) in O(log|V|):
  ändert den Vorrangwert des Elements v in dNeu. Aufruf in (4).

b) Die while-Schleife wird maximal |V|-mal durchlaufen.
Damit ist Aufwand für (2) insgesamt O(|V| log|V|).

c) Jede Kantenrichtung wird genau einmal in einer der for-Schleifen betrachtet.
Da (3) und (4) in O(log|V|) geht, ist für alle for-Schleifen insgesamt O(|E| log|V|) notwendig.

d) Größenordnungen aus b) und c) summiert: T = O(|E| log|V|)

Laufzeit-Analyse - Kandidatenliste als unsortiertes Feld

a) Kandidatenliste als unsortiertes Feld:
   - Zeile (2) (Minimum bestimmen) benötigt O(|V|)
   - Operationen in (1), (3) und (4) benötigen O(1).

b) Die while-Schleife wird höchstens |V|-mal durchgeführt.
   Für (2) ist O(|V|) notwendig. Daher Aufwand für (2) insgesamt $O(|V|^2)$.

c) Jede Kantenrichtung wird genau einmal in einer der for-Schleifen betrachtet. Da (3) und (4) in O(1) geht, ist für alle for-Schleifen insgesamt O(|E|) notwendig.

d) Größenordnungen aus b) und c) summiert: $T = O(|V|^2)$

Analyse - Fazit

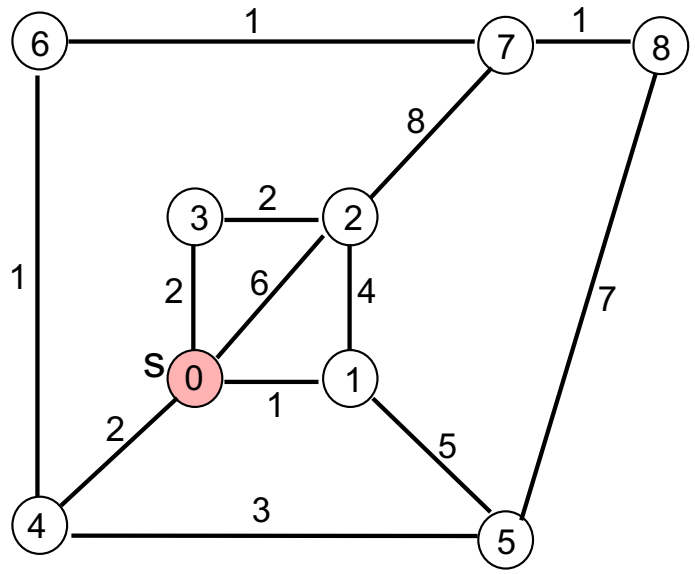
- Ist der Graph dicht besetzt, d.h. $|E| = O(|V|^2)$,
  dann ist Variante mit Kandidatenliste als unsortiertes Feld besser.

  Damit: $T = O(|V|^2)$

- Ist der Graph dünn besetzt, d.h. $|E| = O(|V|)$,
  dann ist die Variante mit Kandidatenliste als Prioritätsliste besser.

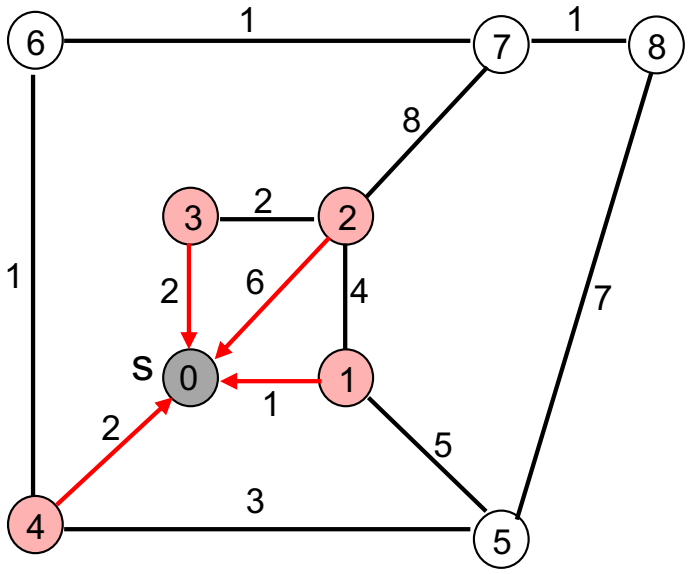
  Damit: $T = O(|V| \log|V|)$

Beispiel zu Dijkstra-Algorithmus (1)



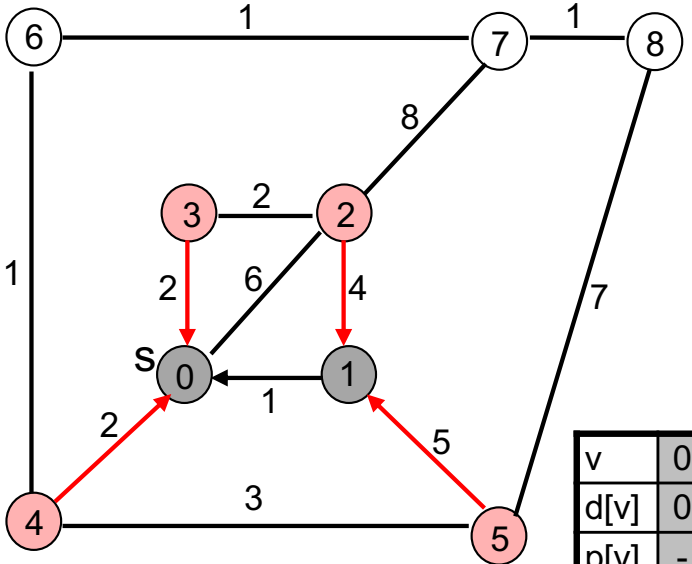
v	0	1	2	3	4	5	6	7	8
d[v]	0	∞	∞	∞	∞	∞	∞	∞	∞
p[v]	-	-	-	-	-	-	-	-	-

Knoten 0 wird besucht



v	0	1	2	3	4	5	6	7	8
d[v]	0	1	6	2	2	∞	∞	∞	∞
p[v]	-	0	0	0	0	-	-	-	-

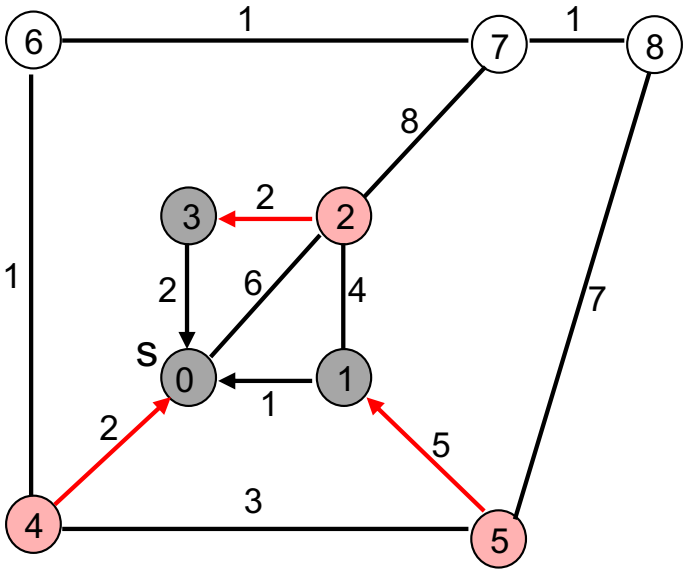
Knoten 1 wird besucht



v	0	1	2	3	4	5	6	7	8
d[v]	0	1	5	2	2	6	∞	∞	∞
p[v]	-	0	1	0	0	1	-	-	-

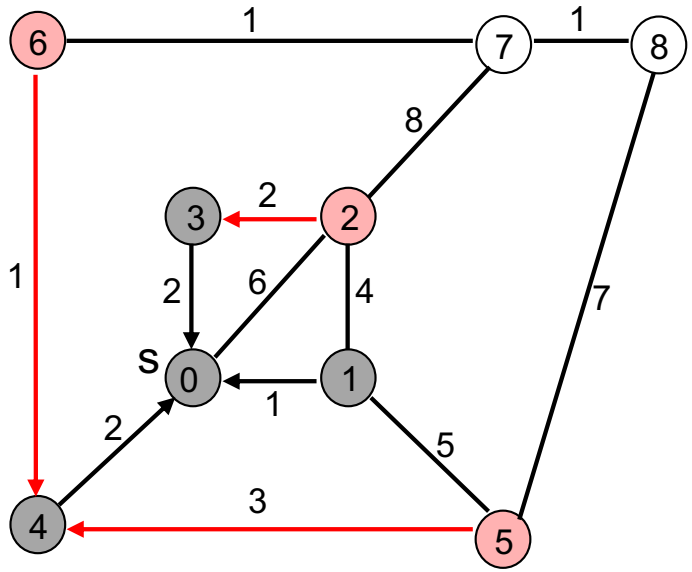
Knoten 3 wird besucht

Beispiel zu Dijkstra-Algorithmus (2)



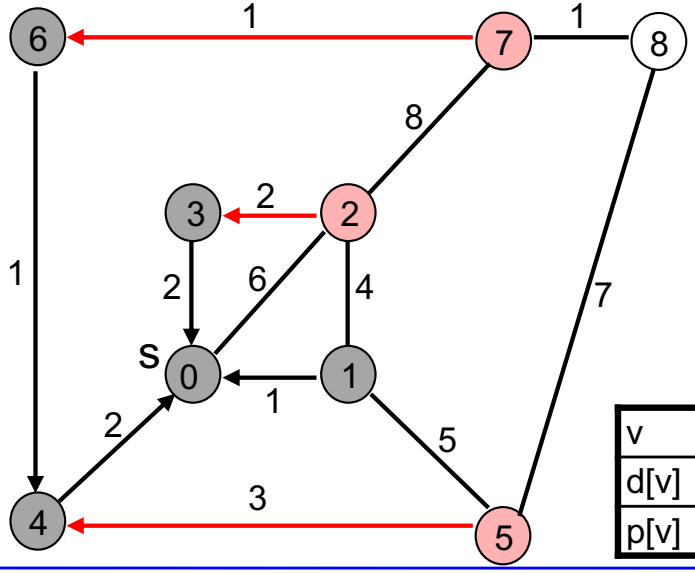
v	0	1	2	3	4	5	6	7	8
d[v]	0	1	4	2	2	6	∞	∞	∞
p[v]	-	0	3	0	0	1	-	-	-

Knoten 4
wird besucht



v	0	1	2	3	4	5	6	7	8
d[v]	0	1	4	2	2	5	3	∞	∞
p[v]	-	0	3	0	0	4	4	-	-

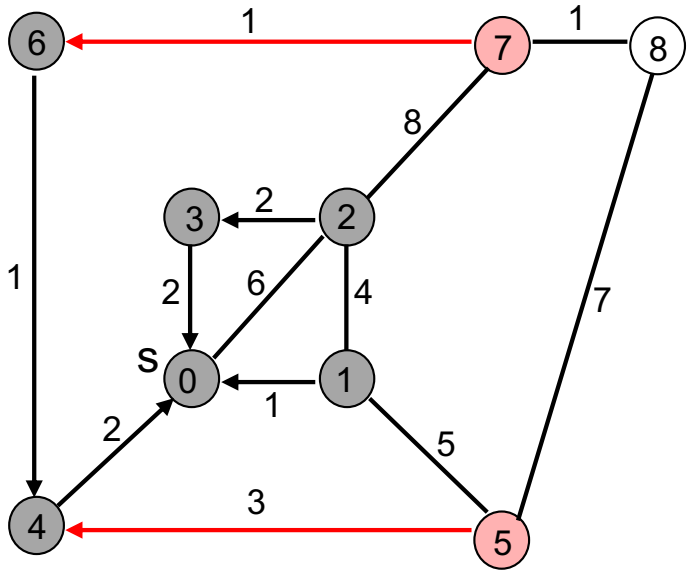
Knoten 6
wird besucht



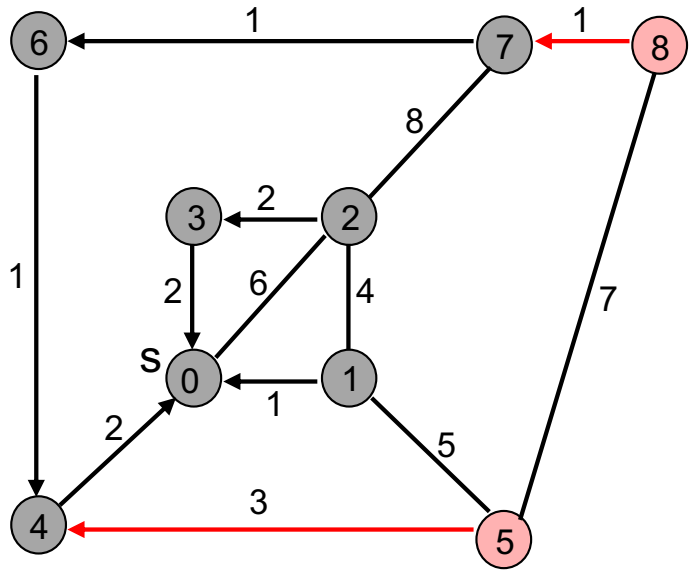
Knoten 2
wird besucht

v	0	1	2	3	4	5	6	7	8
d[v]	0	1	4	2	2	5	3	4	∞
p[v]	-	0	3	0	0	4	4	6	-

Beispiel zu Dijkstra-Algorithmus (3)



Knoten 7
wird besucht



v	0	1	2	3	4	5	6	7	8
d[v]	0	1	4	2	2	5	3	4	∞
p[v]	-	0	3	0	0	4	4	6	-

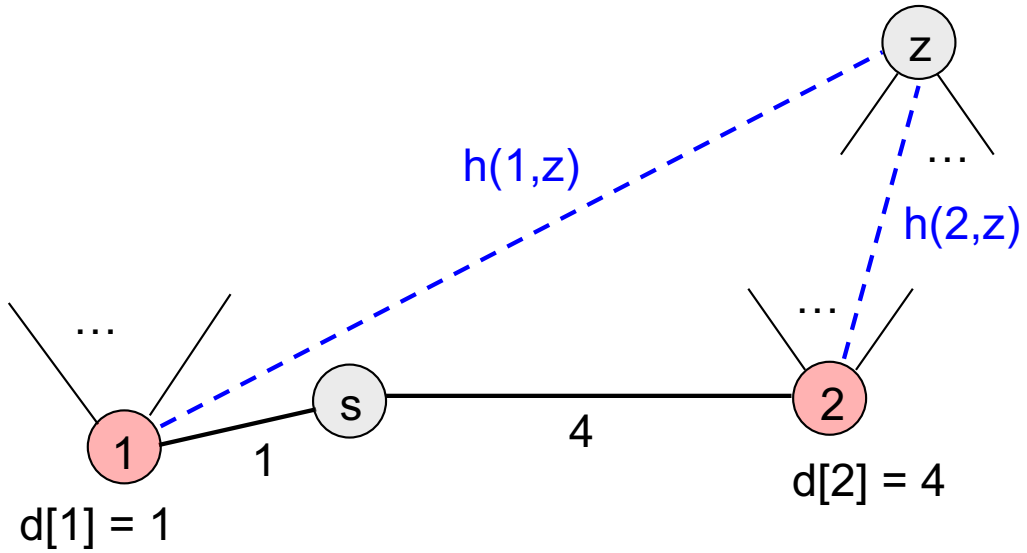
v	0	1	2	3	4	5	6	7	8
d[v]	0	1	4	2	2	5	3	4	5
p[v]	-	0	3	0	0	4	4	6	7

Kürzeste Wege für Startknoten s = 0 sind nun bekannt für die Zielknoten: 0, 1, 2, 3, 4, 6, 7.

Kürzester Weg von 0 nach 7 ist 0, 4, 6, 7 mit Distanz d = 4.

A*-Algorithmus

- Es sei der kürzeste Weg von s zu genau einem Zielknoten z gesucht.
- Annahme: es gibt eine Schätzfunktion $h(v,z) \leq dist(v,z)$,
  mit der sich die Distanz dist(v,z) nach unten abschätzen lässt.
- Typisches Beispiel: h(v,z) = Euklidische Abstand zwischen v und z.
- Besuche als nächstes Knoten v mit d[v] + h(v,z) minimal.
- Bei h als Euklidischen Abstand: bevorzuge Knoten, die näher zum Ziel liegen.



Von den Kandidaten 1 und 2 wird als nächstes Knoten 2 besucht, weil 2 näher zum Ziel z liegt.

Beachte, dass der Dijkstra-Algorithmus als nächstes Knoten 1 besuchen würde!

A*- Algorithmus

```
boolean shortestPath (Vertex s, Vertex z, DistanzGraph G,  double[ ] d, Vertex[ ] p ) {
        kl = ∅;      // leere Kandidatenliste
        for (jeden Knoten v) {
                d[v] = ∞;
                p[v] = undef;
        }
        d[s] = 0;    // Startknoten
        kl.insert(s, 0 + h(s,z));
        while (! kl.empty() ) {
                v = kl.delMin();   // lösche Knoten v mit d[v] + h(v,z) minimal;
                if (v == z) return true;          // Zielknoten z erreicht
                for ( jeden adjazenten Knoten w von v )
                        if (d[w] == ∞)  { // w noch nicht besucht und nicht in Kandidatenliste
                                p[w] = v;
                                d[w] = d[v] + c(v,w);
                                kl.insert(w, d[w] + h(w,z));
                        } else if (d[v] + c(v,w) < d[w]) {
                                p[w] = v;
                                d[w] = d[v] + c(v,w);
                                kl.change(w, d[w] + h(w,z));
                        }
        }
        return false;
}
```

Eingabe: Startknoten s, Zielknoten z und Distanzgraph G

Ausgabe: Distanzfeld d und Vorgängerfeld p

Kandidatenliste enthält alle als nächstes besuchbaren Knoten v mit d[v] + h(v,z) als Prioritätswert.

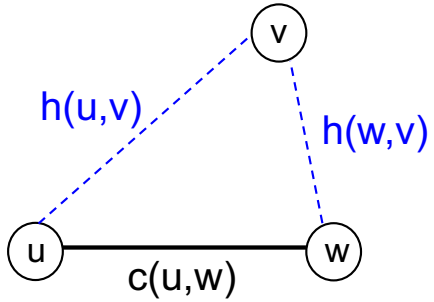
Änderungen gegenüber dem Diskstra-Algorithmus sind blau gekennzeichnet.

Korrektheit des A*- Algorithmus

- Das A*-Verfahren liefert immer einen kürzesten Weg, falls der Zielknoten z erreichbar ist und die Schätzfunktion h folgende Eigenschaften erfüllt:

  (1) h ist zulässig:
  $0 \leq h(u,v) \leq dist(u,v)$ für alle Knoten u, v
  (h ist positiv und ist eine untere Schranke für die Distanz dist)

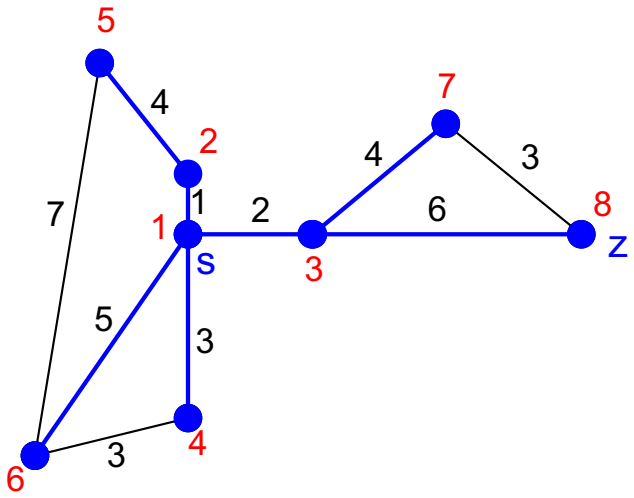
  (2) h ist monoton:
  $h(u,v) \leq c(u,w) + h(w,v)$ für alle Knoten u, v, w



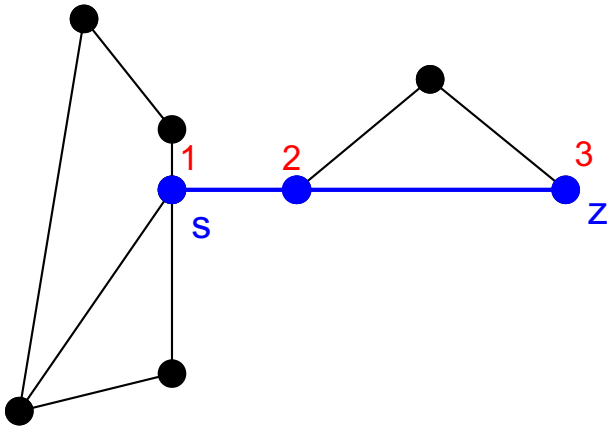
- Die Euklidische Abstandsfunktion erfüllt die beiden oberen Bedingungen.
- Die Schätzfunktion h(u,v) = 0 erfüllt trivialerweise beide Bedingungen. Man erhält dann genau den Algorithmus von Dijkstra.
- In der Literatur wird die Schätzfunktion auch Heuristik genannt
  (Heuristik = Strategie zur Lösungssuche)

Vergleich A*- und Dijkstra-Algorithmus

Dijkstra-Algorithmus



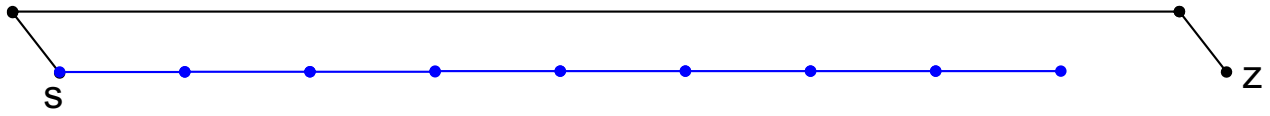
A*-Algorithmus



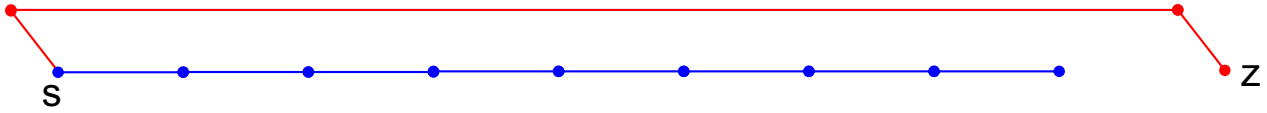
- Gesucht: kürzester Weg von Startknoten s nach Zielknoten z
- Reihenfolge der besuchten Knoten in rot.
- Schätzfunktion h und Kantengewicht c sind Euklidische Abstände
- Berechnete kürzeste Wege sind blau markiert.

Schlechtes Verhalten von A* bei Sackgassen

- A* geht direkt auf das Ziel z zu, läuft aber in eine Sackgasse.



- A* besucht nun einen Knoten, der sich anfangs vom Ziel entfernt und daher nicht früher betrachtet wurde.



9. Kürzeste Wege in Graphen

- Problemstellung
- Erweiterte Breitensuche
- Dijkstra- und A*-Algorithmus
- Moore-Ford-Algorithmus
- Netzpläne und erweiterte topologische Sortierung
- Alle kürzesten Wege mit Floyd-Algorithmus

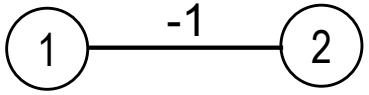
Verschiedene Algorithmen

Algorithmus	Problem-variante	Ungerichtet/ gerichtet	Kanten-Gewichte	Zyklenfreiheit
Breitensuche	Single Source Shortest Path	beides	ungewichtet	egal
Algorithmus von Dijkstra; A*-Verfahren	Single Source Shortest Path	beides	pos.	egal
Algorithmus von Moore und Ford	Single Source Shortest Path	gerichtet	pos./neg.	keine Zyklen neg. Länge
Erweiterte topologische Sortierung	Single Source Shortest Path	gerichtet	pos./neg.	zyklenfrei
Algorithmus von Floyd	All Pairs Shortest Path	ungerichtet	pos.	egal
		gerichtet	pos./neg.	keine Zyklen neg. Länge

Graphen mit positiven Gewichten werden auch Distanzgraphen genannt.

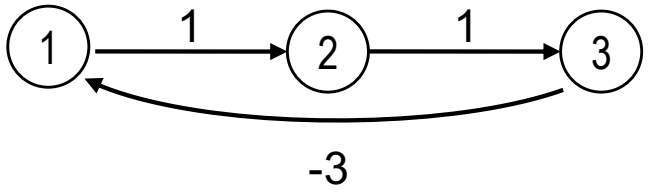
Graphen mit negativen Gewichten

- Problem 1: bei einem ungerichteten Gaphen mit negativen Gewichten ergeben sich beliebig kleine Wege:



Weg 1, 2, 1 mit Länge -2.
Weg 1, 2, 1, 2, 1 mit Länge -4.
...

- Problem 2: Zyklen mit negativer Länge in einem gerichteten Graphen führen zum gleichen Problem:

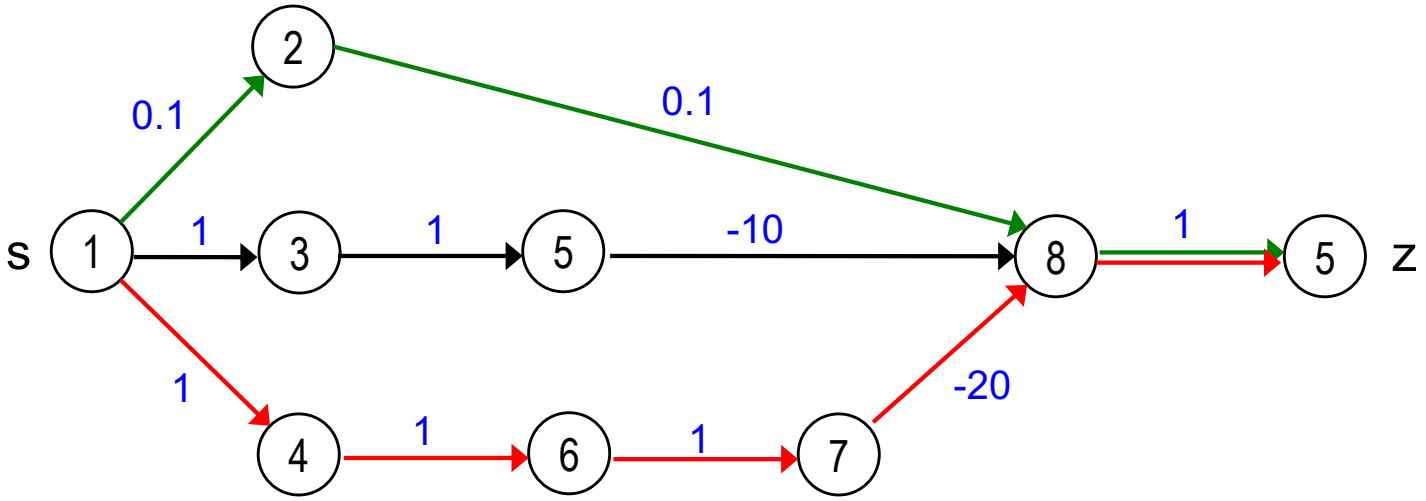


Zyklus 1, 2, 3, 1 mit Länge -1.

- Die Eindeutigkeit der Länge kürzester Wege ist daher nicht garantiert.
- Daher: Einschränkung auf gerichtete Graphen mit beliebigen Gewichten ohne Zyklen negativer Länge.

Dijkstra-Algorithmus bei negativen Gewichten nicht geeignet

Mit Algorithmus von Dijkstra
berechneter Weg mit Länge = 1.2.



Kürzester Weg von s nach z
mit Länge = -16.

Annahme: Dijkstra-Algorithmus stoppt, sobald Zielknoten z erreicht wird.

Was würde passieren, wenn Algorithmus weiterliefe?

Algorithmus von Moore und Ford - Idee

- Ausgangspunkt ist der Algorithmus von Dijkstra.
- Ein Knoten v muss jedoch mehrere Male besucht werden können. Und zwar immer dann, wenn sich für einen Vorgänger von v einen noch kürzeren Weg ergeben hat.
- Die Reihenfolge, in der die Knoten besucht werden, spielt keine Rolle mehr.
- Daher werden Kandidaten in einer einfachen Schlange gespeichert.

Algorithmus von Moore und Ford

```
void shortestPath (Vertex s, WeightedDiGraph G,  double[ ] d, Vertex[ ] p) {

        Queue<Vertex> kl = ∅;

        for (jeden Knoten v) {
              d[v] = ∞;
              p[v] = undef;
        }
        d[s] = 0;  // Startknoten
        kl.add(s);

        while (! kl.empty())  {
              v = kl.remove();
              for ( jeden adjazenten Knoten w von v ) {
(1)                 if (d[v] + c(v,w) < d[w]) {
(2)                       p[w] = v;
(3)                       d[w] = d[v] + c(v,w);
(4)                       if (! kl.contains(w) )
(5)                             kl.add(w);
                    }
              }
        }
}
```

Ausgabe: Distanzfeld d und Vorgängerfeld p

Eingabe: Startknoten s und gewichteter und gerichteter Graph g

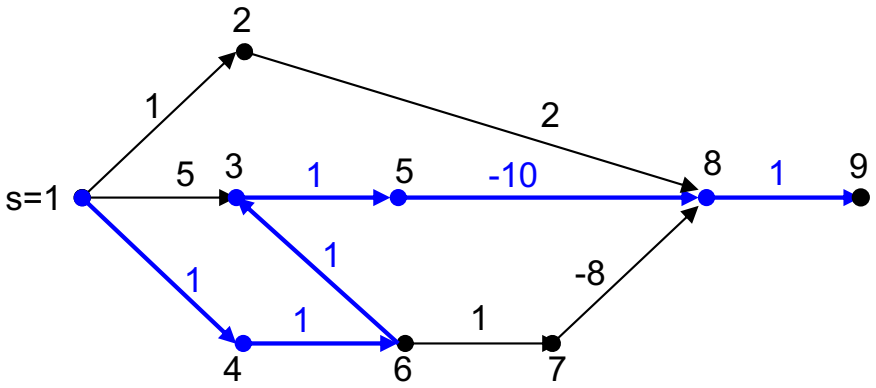
In der Kandidatenliste sind alle Knoten abgespeichert, die als nächstes besucht werden können.

Die Zugriffsoperationen auf die Kandidaten-liste sind rot gekennzeichnet.

Distanzwert für w verbessert sich. Kürzester Weg nach w geht nun über v.

w muss nochmals besucht werden. Prüfung auf negativen Zyklus muss noch ergänzt werden!

Beispiel zu Moore-Ford-Algorithmus



Nächster besuchter Knoten ist rot gekennzeichnet.

v	1	2	3	4	5	6	7	8	9
d[v]	0	∞	∞	∞	∞	∞	∞	∞	∞
		1	5	1					
								3	
					6				
						2			
									4
								-4	
			3				3		
									-3
					4				
								-5	
								-6	
									-5

v	1	2	3	4	5	6	7	8	9
p[v]	-	-	-	-	-	-	-	-	-
		1	1	1					
								2	
					3				
						4			
									8
								5	
			6				6		
									8
					3				
								7	
								5	
									8

Kandiatenliste kl
1
2, 3, 4
3, 4, 8
4, 8, 5
8, 5, 6
5, 6, 9
6, 9, 8
9, 8, 3, 7
8, 3, 7
3, 7, 9
7, 9, 5
9, 5, 8
5, 8
8
9

Analyse des Moore-Ford-Algorithmus

- Enthält der Graph keine Zyklen mit negativer Länge, dann wird jeder Knoten maximal (|V|-1)-mal in die Schlange kl eingefügt (Zeile (5) im Algorithmus). (Begründung siehe [Turau]).
- Wird ein Knoten |V|-mal eingefügt, dann muss ein Zyklus mit negativer Länge existieren und es kann im Algorithmus eine entsprechende Ausgabe erfolgen.
- Würde jeder Knoten genau einmal in die Schlange kl eingefügt werden, wäre der Aufwand für alle for-Schleifen gerade O(|E|). Denn jede Kante im Graph wird genau einmal in einer der for-Schleifen betrachtetet.
  (Zeile (5) kl.contains(w) kann mit Hilfe eines Booleschen Felds in O(1) realisiert weden).
- Da nun aber jeder Knoten bis zu (|V|-1)-mal in die Schlange kl eingefügt werden kann, erhält man insgesamt:

  $T = O(|E|*|V|)$.

9. Kürzeste Wege in Graphen

- Problemstellung
- Erweiterte Breitensuche
- Dijkstra- und A*-Algorithmus
- Moore-Ford-Algorithmus
- Netzpläne und erweiterte topologische Sortierung
- Alle kürzesten Wege mit Floyd-Algorithmus

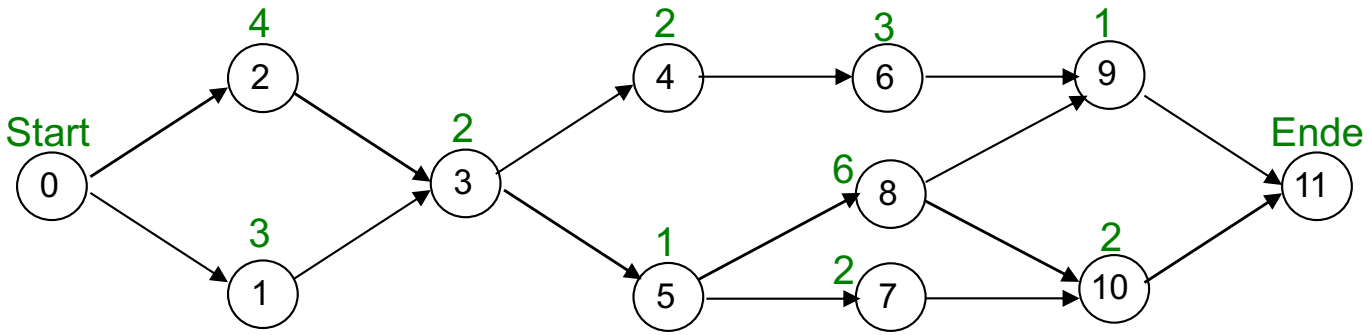
Verschiedene Algorithmen

Algorithmus	Problem-variante	Ungerichtet/ gerichtet	Kanten-Gewichte	Zyklenfreiheit
Breitensuche	Single Source Shortest Path	beides	ungewichtet	egal
Algorithmus von Dijkstra; A*-Verfahren	Single Source Shortest Path	beides	pos.	egal
Algorithmus von Moore und Ford	Single Source Shortest Path	gerichtet	pos./neg.	keine Zyklen neg. Länge
Erweiterte topologische Sortierung	Single Source Shortest Path	gerichtet	pos./neg.	zyklenfrei
Algorithmus von Floyd	All Pairs Shortest Path	ungerichtet	pos.	egal
		gerichtet	pos./neg.	keine Zyklen neg. Länge

Graphen sind jetzt sogenannte Netzpläne.

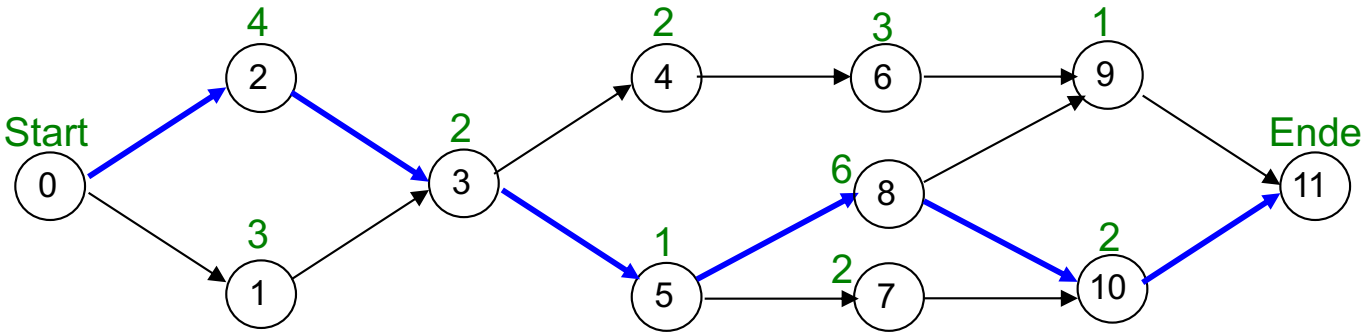
Netzpläne

- Netzpläne sind azyklische gerichtete Graphen mit genau einem Start und einem Endknoten.
- Die Knoten (außer Start- und Endknoten) stellen oft gewisse Aktivitäten dar und sind mit einem Gewicht (Aufwand wie z.B. Zeitangaben) versehen.
- Die gerichteten Kanten legen Reihenfolgenbeziehungen fest.



Kritischer Pfad

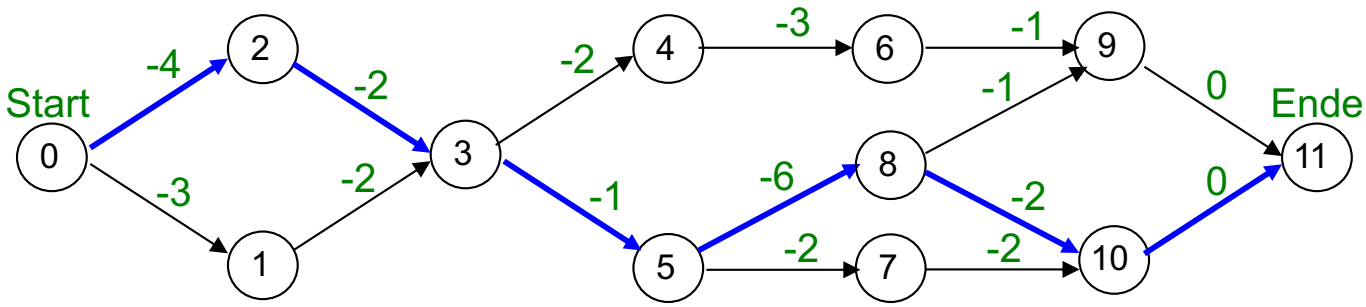
- Kritischer Pfad ist ein Weg vom Start- zum Ziel-Knoten mit maximalen Aufwand (längster Weg!).
- In einem Projektplan gibt die Länge eines kritischen Pfades die frühestmögliche Projektfertigstellung an.



Kritischer Pfad mit Zeitaufwand 15

Bestimmung eines kritischen Pfades

- Werden die an den Knoten notierten Gewichte an die einmündenden Kanten mit umgekehrtem Vorzeichen „verschoben“, erhält man einen gewichteten azyklischen gerichteten Graphen.
- Die Bestimmung des kritischen Pfades lässt sich dann zurückführen auf die Berechnung eines kürzesten Weges vom Start- zum Zielknoten.



Kürzester Weg mit Länge -15.

- Prinzipiell ließe sich der kritische Pfad mit dem Algorithmus von Moore und Ford bestimmen.
- Für den Spezialfall der Netzpläne ist jedoch ein auf toplogischer Sortierung basierender Algorithmus wesentlich effizienter.

Idee der erweiterten topologischen Sortierung

- Besuche die Knoten wie bei der topologischen Sortierung.
- Dazu werden in der Schlange alle noch nicht besuchten Knoten gehalten, deren (auch die unmittelbaren) Vorgänger bereits besucht worden sind.
- Zusätzlich wird ein Distanzfeld d und ein Vorgängerfeld p berechnet.
- Bei jedem Knotenbesuch werden dann wie beim Algorithmus von Moore und Ford für alle Nachfolgerknoten die Distanzen und der gefundene kürzeste Weg gegebenenfalls aktualisiert.
- Analyse (wie bei der topologischen Sortierung):

  $T = O(|E|+|V|)$.

Erweiterte topologische Sortierung

```
void allShortestPath (WeightedDiGraph g, double[ ] d, Vertex[ ] p)
{

    int[ ] inDegree;   // Anz. noch nicht besuchten Vorgänger
    Queue<Vertex> q;

    for (jeden Knoten v) {
        d[v] = ∞;
        p[v] = undef;
        inDegree[v] = Anzahl der Vorgänger;
        if (inDegree[v]  == 0) {
            q.add(v);
            d[v] = 0;
        }
    }

    if (q.size() > 1) {
        println("Fehler; mehr als ein Startknoten");
        return;
    }
```

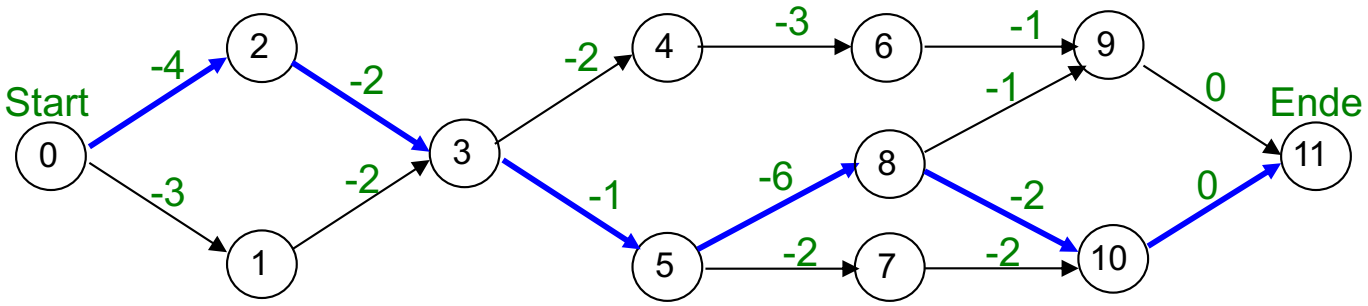
Ausgabe:
Distanzfeld d und Vorgängerfeld p

Eingabe:
Gewichteter Digraph G mit genau einem Startknoten.

```
    int k = 0;

    while (! q.empty() ) {
        v = q.remove(); k++;
        for ( jeden Nachfolger w von v ) {
            if (d[v] + c(v,w) < d[w]) {
                p[w] = v;
                d[w] = d[v] + c(v,w);
            }
            if (--inDegree[w] == 0)
                q.add(w);
        }
    }

    if (k != n)
        println( "Fehler; Graph ist zyklisch");
}
```

Programmteile, um die die topologische Sortierung erweitert wurde, sind blau gekennzeichnet.

Beispiel



Kürzester Weg
(Kritischer Pfad)

Kand.Liste q
0
0
1, 2
2
3
4, 5
5, 6
6, 7, 8
7, 8
8
9, 10
10
11

v	0	1	2	3	4	5	6	7	8	9	10	11
d[v]	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
		-3	-4									
				-5								
				-6								
					-8	-7						
							-11					
								-9	-13			
										-12		
											-11	
										-14	-15	
												-14
												-15

v	0	1	2	3	4	5	6	7	8	9	10	11
p[v]	-	-	-	-	-	-	-	-	-	-	-	-
		0	0									
				1								
				2								
					3	3						
							4					
								5	5			
										6		
											7	
										8	8	
												9
												10

Nächster besuchter Knoten ist rot gekennzeichnet.

9. Kürzeste Wege in Graphen

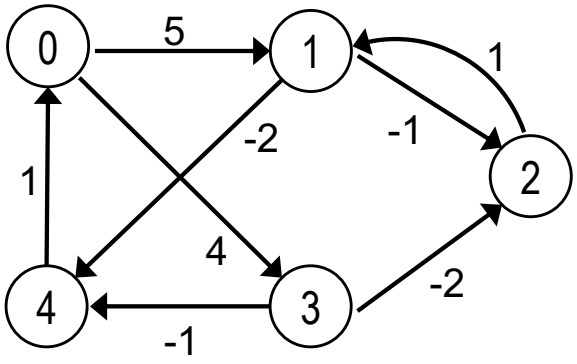
- Problemstellung
- Erweiterte Breitensuche
- Dijkstra- und A*-Algorithmus
- Moore-Ford-Algorithmus
- Netzpläne und erweiterte topologische Sortierung
- Alle kürzesten Wege mit Floyd-Algorithmus

Verschiedene Algorithmen

Algorithmus	Problem-variante	Ungerichtet/ gerichtet	Kanten-Gewichte	Zyklenfreiheit
Breitensuche	Single Source Shortest Path	beides	ungewichtet	egal
Algorithmus von Dijkstra; A*-Verfahren	Single Source Shortest Path	beides	pos.	egal
Algorithmus von Moore und Ford	Single Source Shortest Path	gerichtet	pos./neg.	keine Zyklen neg. Länge
Erweiterte topologische Sortierung	Single Source Shortest Path	gerichtet	pos./neg.	zyklenfrei
Algorithmus von Floyd	All Pairs Shortest Path	ungerichtet	pos.	egal
		gerichtet	pos./neg.	keine Zyklen neg. Länge

All Pairs Shortest Path

- Gesucht: Matrix die für alle möglichen Start-Ziel-Paare (s,z) die Distanz dist(s,z) enthält.
- Gerichteter Graph darf zwar negative Gewichte haben jedoch keine negativen Zyklen!
- Ungerichteter Graph geht auch. Jedoch dann keine negativen Gewichte.



s\z	0	1	2	3	4
0	0	3	2	4	1
1	-1	0	-1	3	-2
2	0	1	0	4	-1
3	-2	-1	-2	0	-3
4	1	4	3	5	0

dist(4,1) = 4

dist(2,4) = -1

Idee des Floyd-Algorithmus

- Optimalitätsprinzip von Bellman (gilt für Graphen ohne negative Zyklen)

> Für jeden Knoten w auf einem kürzesten Weg von u nach v gilt:
>
> dist(u,v) = dist(u,w) + dist(w,v).
>
> D.h. Teilwege von kürzesten Wegen müssen ebenfalls kürzeste Wege sein.

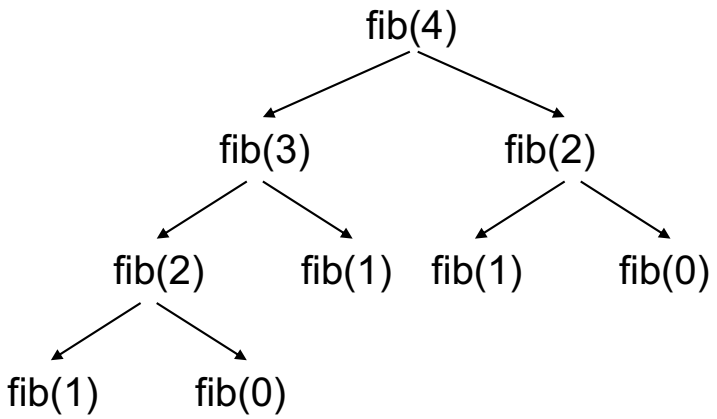
- Das Bellman-Prinzip führt auf eine rekursiv formulierte Problem-Stellung.
- Der Floyd-Algorithmus löst das rekursiv formulierte Problem nach dem Prinzip des dynamischen Programmierens.

Einschub - dynamisches Programmieren (1)

- Problem mit rekursiven Funktionen:
  Bei rekursiven Funktionen kann es zu mehrfacher Berechnung des gleichen Funktionswerts kommen.
- Beispiel: Fibonacci-Funktion

```
int fib(int n) {
        if (n == 0)
                return 0;
        else if (n == 1)
                return 1;
        else
                return fib(n-1) + fib(n-2);
}
```

Aufrufstruktur für fib(4):



- Bei Aufruf von fib(4) wird 2-mal fib(2), 3-mal fib(1) und 2-mal fib(0) berechnet.
- Bei Aufruf von fib(24) wird 2-mal fib(22), 3-mal fib(21), 5-mal fib(20), 9-mal fib(19), 13-mal fib(18) etc. berechnet.
- Allgemein ist bei fib(n) die Anzahl der fib-Aufrufe:

  $\approx (2/\sqrt{5})*\phi^{n+1} - 1 \approx 0.89*1.62^{n+1}$ (ϕ wird auch „Goldener Schnitt“ genannt.)

Einschub - dynamisches Programmieren (2)

Dynamisches Programmieren:

- Ausgangspunkt: Rekursiv definierte Funktion.
- Berechne rekursive Funktion mit Iteration und Tabelle statt Rekursion:
- Beginne mit den Basisfällen und wende dann Rekursionsschritte „aufwärts“ an, wobei Zwischenergebnisse in einer Tabelle zur späteren Verwendung abgespeichert werden.
- Dynamisches Programmieren geht auf ein Verfahren zur Optimierung technischer Prozesse von Bellman 1957 zurück.

Beispiel: Fibonacci-Funktion

```
int[ ] fib = new int[N+1];

fib[0] = 0;
fib[1] = 1;

for (int n = 2; n <= N; n++)
      fib[n] = fib[n-1] + fib[n-2];
```

Tabelle

Basisfälle

Rekursionsschritt aufwärts angewendet

Analyse der Fibonacci-Funktion:

- dynamisch programmiertes fib: O(n)
- rekursives fib: $O(1.62^n)$

Einschub - dynamisches Programmieren (3)

Weiteres Beispiel aus der Kombinatorik:

- Eine Auswahl (Teilmenge) von k verschiedenen Elementen aus einer Menge von n Elementen heißt Kombination.
- Die Anzahl der Kombinationen schreibt man in der Mathematik auch als:

$$\binom{n}{k}$$

Dieser Ausdruck ist nur für n ≥ k ≥ 0 definiert.

- Beispiel: die Anzahl der möglichen Lottokombinationen ist gleich $\binom{49}{6}$

Rekursive Formulierung

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Rekursionsfall:

Um eine k-elementige Teilmenge aus einer Menge mit n Elementen auszuwählen, kann entweder das n-te Element weggelassen oder ausgewählt werden.

$$\binom{n}{k} = 1 \text{ falls } k = 0$$

$$\binom{n}{k} = 1, \text{falls } k = n$$

Basisfälle:

Bei k = 0 gibt es nur eine Teilmenge als Auswahl, nämlich die leere Teilmenge. Bei k = n ist die Menge selbst die einzige Teilmenge.

Einschub - dynamisches Programmieren (4)

Iterative Berechnung mit Tabelle

```
int[ ][ ]  komb = new int[N+1][N+1];

for (int n = 0; n <= N; n++) {
      komb[n][0] = 1;       // Basisfall (1)
      komb[n][n] = 1;       // Basisfall (2)
}

// Rekursionsschritt:
for (int n = 2; n <= N; n++)
      for (int k = 1; k < n; k++)
            komb[n][k] = komb[n-1][k] + komb[n-1][k-1];
```

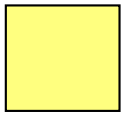
Tabelle zum Speichern der Zwischenergebnisse:

komb[n][k] = $\binom{n}{k}$

Einschub - dynamisches Programmieren (5)

- Beispiel: für N = 4 ergibt sich die Tabelle komb wie folgt:

n \ k	0	1	2	3	4
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1



komb[n][k] = komb[n-1][k] + komb[n-1][k-1];

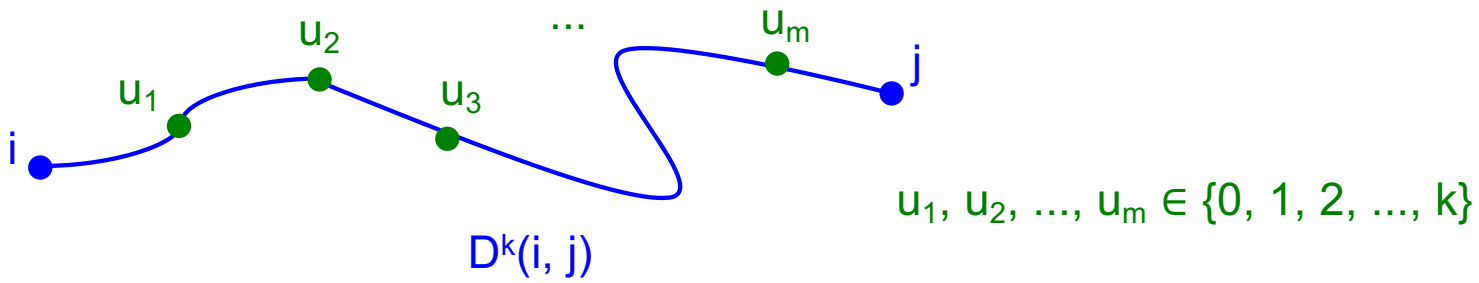
Pascalsches Dreieck!

- Bemerkung: Für die Berechnung von komb[n][k] würde es genügen nur die jeweils letzte Zeile der Matrix zu speichern. Daher eindimensionales Feld statt Matrix.

Problemstellung für alle kürzeste Wege

- Nochmals die Problemstellung
  - Berechne kürzeste Wege zwischen allen Knotenpaaren in einem gewichteten Graphen. In einem gerichteten Graphen dürfen Gewichte negativ sein, aber keine Zyklen negativer Länge.
  - Knoten sind durchnummeriert: 0, 1, 2, ..., n-1.

- Formulierung des Problems als Funktion
  - $D^k(i, j)$ = Distanz (Länge eines kürzesten Weges) von Knoten i nach j, wobei nur Wege betrachtet werden, die über Knoten aus {0, 1, …, k} gehen.
  - $D^{n-1}(i, j)$ löst unser Problem!



Rekursive Formulierung von $D^k(i,j)$

- Basisfall $k = -1$ (nur die direkte Verbindung von i nach j ist erlaubt):

  $D^{-1}(i, j) = c(i, j)$ (Gewicht der Kante (i, j))

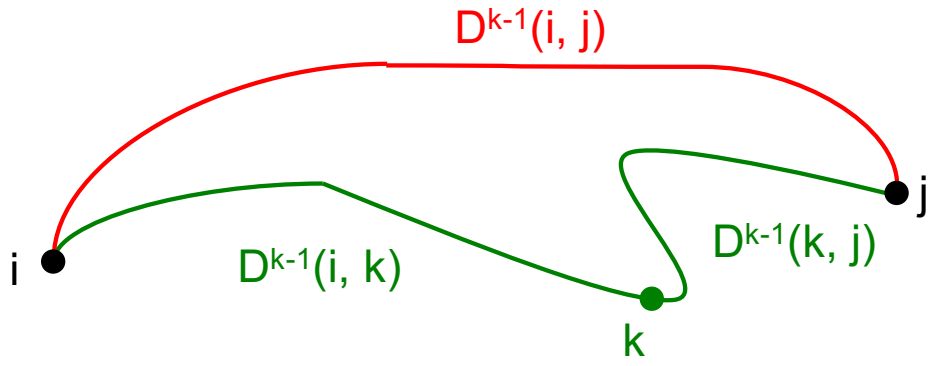
- Rekursionfall ($k \geq 0$):

  $D^k(i, j) = \min \{ D^{k-1}(i, j) , D^{k-1}(i, k) + D^{k-1}(k, j) \}$

Länge eines kürzesten Wegs von i nach j ohne über k zu gehen.

Länge eines kürzesten Wegs von i nach j über k setzt sich zusammen aus kürzestem Weg von i nach k und kürzestem Weg von k nach j.

(Optimalitätsprinzip von Bellman)



Berechnung von D(i,j) mit dynamischer Programmierung

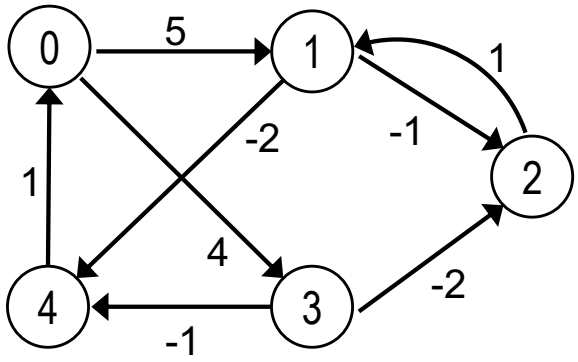
- $D^k(i, j)$ lässt sich für jedes k durch eine 2-dimensionale Matrix D[i][j] darstellen.
  D^{-1} wird gemäß Basisfall initialisiert;
  Nach dem Rekursionsfall ergibt sich D^0 aus D^{-1}, D^1 aus D^0, usw.
  D^{n-1} ist die Lösung unseres Problems.
- Es genügt, D^k nur für den jeweils aktuellen Wert von k abzuspeichern.
  Also nur eine Matrix D!

$$D^k(i, j) = \min \{ D^{k-1}(i, j) , \; D^{k-1}(i, k) + D^{k-1}(k, j) \}$$
$$= \min \{ D^{k-1}(i, j) , \; D^k(i, k) + D^k(k, j) \} \quad (\text{denn } D^k(i, k) = D^{k-1}(i, k) \text{ und } D^k(k, j) = D^{k-1}(k, j))$$

```
// Tabelle
int[ ][ ] D = new int[n][n];

// Starte mit D^-1:
for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
                if (i == j) D[i][j] = 0; else D[i][j] = c(v_i, v_j);
}
for (int k = 0; k < n; k++)
        // Berechne D^k:
        for (int i = 0; i < n; i++)
                for (int j = 0; j < n; j++)
                        if (D[i][j] > D[i][k] + D[k][j])
                                D[i][j] = D[i][k] + D[k][j];
```

D[i][j] = min {D[i][j] , D[i][k] + D[k][j] };

Beispiel



i\j	0	1	2	3	4
0	0	5	∞	4	∞
1	∞	0	-1	∞	-2
2	∞	1	0	∞	∞
3	∞	∞	-2	0	-1
4	1	∞	∞	∞	0

D^{-1}

i\j	0	1	2	3	4
0	0	5	∞	4	∞
1	∞	0	-1	∞	-2
2	∞	1	0	∞	∞
3	∞	∞	-2	0	-1
4	1	6	∞	5	0

D^{0}

i\j	0	1	2	3	4
0	0	5	4	4	3
1	∞	0	-1	∞	-2
2	∞	1	0	∞	-1
3	∞	∞	-2	0	-1
4	1	6	5	5	0

D^{1}

i\j	0	1	2	3	4
0	0	5	4	4	3
1	∞	0	-1	∞	-2
2	∞	1	0	∞	-1
3	∞	-1	-2	0	-3
4	1	6	5	5	0

D^{2}

i\j	0	1	2	3	4
0	0	3	2	4	1
1	∞	0	-1	∞	-2
2	∞	1	0	∞	-1
3	∞	-1	-2	0	-3
4	1	4	3	5	0

D^{3}

i\j	0	1	2	3	4
0	0	3	2	4	1
1	-1	0	-1	3	-2
2	0	1	0	4	-1
3	-2	-1	-2	0	-3
4	1	4	3	5	0

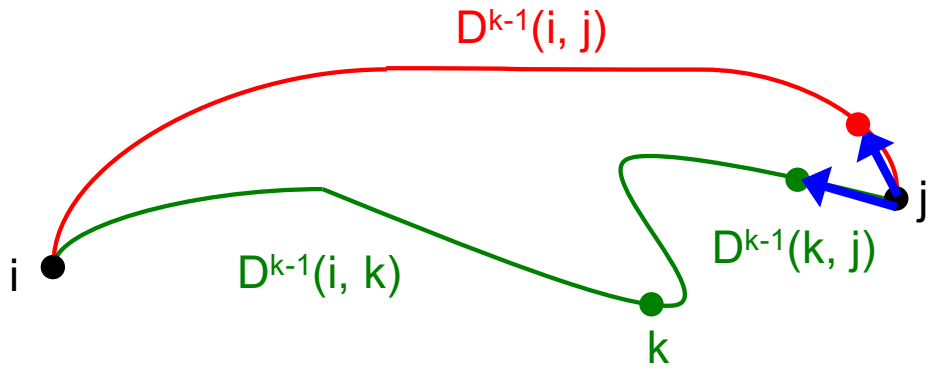
D^{4}

```
if ( D[i][j]  > D[i][k] + D[k][j] )
      D[i][j]  = D[i][k] + D[k][j];
```

Speicherung der kürzesten Wege

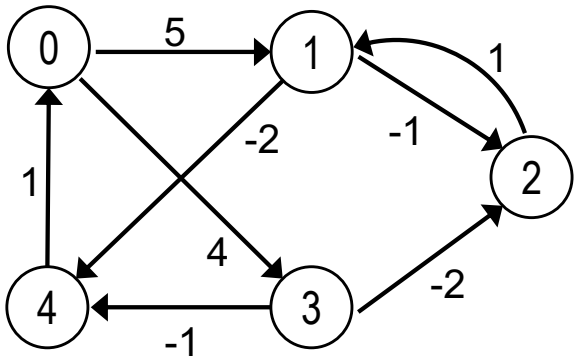
- Zur Speicherung der gefundenen Wege wird eine Vorgängermatrix P eingesetzt:

  P[i][j] = Vorgängerknoten von j im kürzesten Weg von i nach j.




```
if ( D[i][j]  > D[i][k] + D[k][j] ) {
        D[i][j]  = D[i][k] + D[k][j];
        P[i][j]  = P[k][j];
} else {
        // D[i][j] und P[i][j] ändern sich nicht
}
```

Beispiel mit Vorgängermatrix P




```
if ( D[i][j]  > D[i][k] + D[k][j] ) {
      D[i][j]  = D[i][k] + D[k][j];
      P[i][j]  = P[k][j];
}
```

i\j	0	1	2	3	4
0	0	5	∞	4	∞
1	∞	0	-1	∞	-2
2	∞	1	0	∞	∞
3	∞	∞	-2	0	-1
4	1	∞	∞	∞	0

D^{-1}

i\j	0	1	2	3	4
0	0	5	∞	4	∞
1	∞	0	-1	∞	-2
2	∞	1	0	∞	∞
3	∞	∞	-2	0	-1
4	1	6	∞	5	0

D^{0}

i\j	0	1	2	3	4
0	0	5	4	4	3
1	∞	0	-1	∞	-2
2	∞	1	0	∞	-1
3	∞	∞	-2	0	-1
4	1	6	5	5	0

D^{1}

i\j	0	1	2	3	4
0	0	5	4	4	3
1	∞	0	-1	∞	-2
2	∞	1	0	∞	-1
3	∞	-1	-2	0	-3
4	1	6	5	5	0

D^{2}

i\j	0	1	2	3	4
0	0	3	2	4	1
1	∞	0	-1	∞	-2
2	∞	1	0	∞	-1
3	∞	-1	-2	0	-3
4	1	4	3	5	0

D^{3}

i\j	0	1	2	3	4
0	0	3	2	4	1
1	-1	0	-1	3	-2
2	0	1	0	4	-1
3	-2	-1	-2	0	-3
4	1	4	3	5	0

D^{4}

i\j	0	1	2	3	4
0	-	0	-	0	-
1	-	-	1	-	1
2	-	2	-	-	-
3	-	-	3	-	3
4	4	-	-	-	-

P^{-1}

i\j	0	1	2	3	4
0	-	0	-	0	-
1	-	-	1	-	1
2	-	2	-	-	-
3	-	-	3	-	3
4	4	0	-	0	-

P^{0}

i\j	0	1	2	3	4
0	-	0	1	0	1
1	-	-	1	-	1
2	-	2	-	-	1
3	-	-	3	-	3
4	4	0	1	0	-

P^{1}

i\j	0	1	2	3	4
0	-	0	1	0	1
1	-	-	1	-	1
2	-	2	-	-	1
3	-	2	3	-	1
4	4	0	1	0	-

P^{2}

i\j	0	1	2	3	4
0	-	2	3	0	1
1	-	-	1	-	1
2	-	2	-	-	1
3	-	2	3	-	1
4	4	2	3	0	-

P^{3}

i\j	0	1	2	3	4
0	-	2	3	0	1
1	4	-	1	0	1
2	4	2	-	0	1
3	4	2	3	-	1
4	4	2	3	0	-

P^{4}

Algorithmus von Floyd

```
void computeAllShortestPath (WightedDiGraph G,  double[ ][ ] D, Vertex[ ][ ] P) {
    // Initialisiere D und P:
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            P[i][j] = undef;
            if (i == j)
                D[i][j] = 0
            else {
                D[i][j] = c(v_i, v_j); // ∞, falls keine Kante (v_i,v_j) existiert
                if (D[i][j] != ∞)
                    P[i][j] = i;
            }
        }
    }

    for (int k = 0; k < n; k++)
        // Berechne D^k:
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                if (D[i][j] > D[i][k] + D[k][j]) {
                    D[i][j] = D[i][k] + D[k][j];
                    P[i][j] = P[k][j];
                }
}
```

Ausgabe:
Distanzmatrix D und Vorgängermatrix P

Eingabe:
Gewichteter Digraph G mit n Knoten.

Analyse des Floyd-Algorithmus

- Enthält der Graph Zyklen mit negativer Länge,
  dann wird ein Diagonalelement von D negativ.
  Der Algorithmus kann leicht um eine passende Prüfung
  mit entsprechender Ausgabe erweitert werden.
- Analyse: die dreifach geschachtelten for-Schleifen ergeben:

  $T = O(|V|^3)$.
- Vergleich zu Dijkstra-Algorithmus:
  - Ist der Graph dünn-besetzt (d.h. $|E| = O(|V|)$ und alle Gewichte positiv,
    dann kann der Dijkstra-Algorithmus mit einem Index-Heap auch
    für jeden Knoten aus V als Startknoten durchgeführt werden.
    Damit: $T = O(|V| \, |E| \log(|V|) = O(|V|^2 \log(|V|)$.
    Also: Dijkstra-Algorithmus besser als Floyd-Algorithmus.
  - Ist der Graph dicht besetzt und alle Gewichte positiv,
    dann würde das |V|-malige Durchführen des Dijkstra-Algorithmus
    (mit unsortierter Prioritätsliste) ebenfalls zu $T = O(|V|^3)$ führen.
    Dann ist aber aufgrund der einfacheren Datenstruktur
    der Floyd-Algorithmus besser als der Dijkstra-Algorithmus.