Teil 2: Graphenalgorithmen

- Anwendungen
- Definitionen
- Datenstrukturen für Graphen
- Elementare Algorithmen
- Topologisches Sortieren
- Kürzeste Wege
- Minimal aufspannende Bäume
- Flüsse in Netzwerken
- Zusammenhangskomponenten

Topologische Sortierung (1)

Definition:

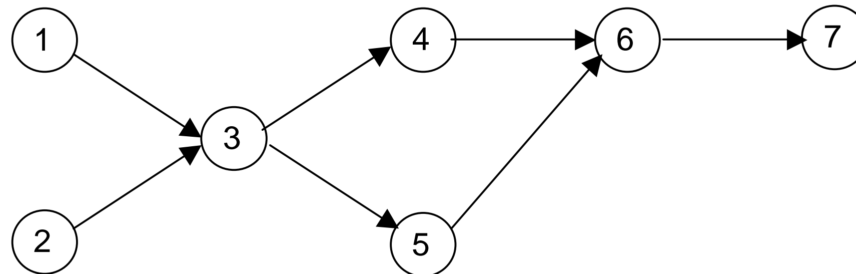
Eine Folge aller Knoten eines gerichteten Graphen G

$v_0, v_1, v_2, \ldots, v_{n-1}$

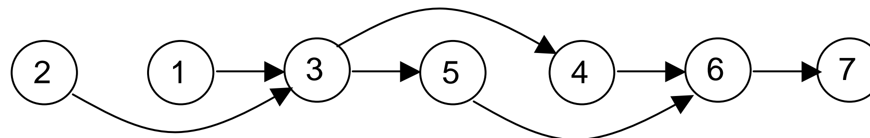
heißt topologische Sortierung, falls für alle Knoten u, v folgende Bedingung gilt:

falls es einen Weg in G von u nach v gibt, dann steht in der Folge u links von v.

Beispiel:



Topologische Sortierung



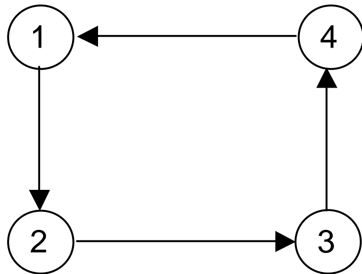
Anschaulich:

Unter Einhaltung der Nachbarschaftsbeziehung lässt sich der Graph in eine lineare Kette so „verbiegen“, dass die Pfeile nur nach rechts gehen.

Topologische Sortierung (2)

Eigenschaften:

- Falls der Graph einen Zyklus enthält, dann existiert keine topologische Sortierung.
  Beispiel:



- Falls ein Graph topologisch sortiert werden kann, dann ist im allgemeinen die topologische Sortierung nicht eindeutig.

Beispiele für Anwendungen:

- Stelle fest, ob es eine Durchführungsreihenfolge für die Aktivitäten oder auch Prozesse in einem Vorranggraphen gibt.
- Prüfe, ob Vererbungsgraph oder include-Graph zyklenfrei ist.

Topologische Sortierung (3)

Idee:

- Speichere für jeden Knoten v:

  inDegree[v] = Anzahl der noch nicht besuchten Vorgänger.

- Speichere alle Knoten v, für die InDegree[v] == 0 gilt, in einer Schlange q. q enthält also alle Knoten, die als nächstes besucht werden dürfen.
- Besuche als nächstes den vorderstes Knoten in der Schlange q und entferne ihn aus q.

Liefert zu einem gerichteten Graphen G eine topologische Sortierung ts.

Algorithmus:

```
void topSort(DiGraph G, Vertex ts[ ])
{
    int inDegree[n];   // n ist Anz. der Knoten in G
    Queue<Vertex> q;

    for (jeden Knoten v) {
        inDegree[v] = Anzahl der Vorgänger;
        if (inDegree[v]  == 0)
            q.push(v);
    }

    int k = 0;
    while (! q.empty() ) {
        v = q.front(); q.pop();
        ts[k++] = v;

        for (jeden Nachfolger w von v )
            if(--inDegree[w] == 0)
                q.push(w);
    }
    if (k != n)
        cout << “Graph ist zyklisch; ”
             << “keine top. Sortierung
                moeglich”;
}
```

Topologische Sortierung (4)

Bemerkungen:

- Der Algorithmus zur topologischen Sortierung ist eine Modifikation der Breitensuche:

Breiten-suche:

```
for (jeden Nachfolger w von v )
        if ((val[w] == “nichtBesucht”)
                q.push(w);
```

Füge alle noch nicht besuchten Nachbarn w in die Schlange q ein.

Topolog. Sortieren:

```
for (jeden Nachfolger w von v )
        if(--inDegree[w] == 0)
                q.push(w);
```

Füge alle Nachbarn w, für die --inDegree[w] == 0 gilt, in die Schlange q ein.
Das bedeutet, dass alle Nachbarn w in die Schlange eingefügt werden, für die gilt:

- w noch nicht besucht und
- alle Vorgänger von w wurden bereits besucht

- Lässt man das Abspeichern der besuchten Knoten in das Feld ts weg, dann erhält man einen einfachen Algorithmus zur Prüfung von Zyklenfreiheit.

- Es gibt auch eine modifizierte Tiefensuche zur Prüfung von Zyklenfreiheit. Dieser Algorithmus arbeitet den Graph von „rechts nach links“ ab, während der hier vorgestellte Algorithmus den Graphen von „links nach rechts“ abarbeitet. Der Tiefensuch-Algorithmus benutzt Rekursion oder einen Stack und ist weniger intuitiv (siehe [Turau]; Seite 95).

Topologische Sortierung (5)

Analyse

- Die while-Schleife wird für jeden Knoten genau einmal durchlaufen; also |V|-mal.
- Die for-Schleife wird für jede von v ausgehende Kante genau einmal durchlaufen.
  Verwendet man die Adjazenzlistendarstellung, ist der Aufwand für den Durchlauf <u>aller</u> for-Schleifen gleich O(|E|).

Damit ergibt sich mit <u>Adjazenzlistendarstellung</u> insgesamt:

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

```
void topSort(DiGraph G, Vertex ts[ ])
{
	int inDegree[n];   // n ist Anz. der Knoten in G
	Queue<Vertex> q;

	for (jeden Knoten v) {
		inDegree[v] = Anzahl der Vorgänger;
		if (inDegree[v]  == 0)
			q.push(v);
	}

	int k = 0;
	while (! q.empty() ) {
		v = q.front(); q.pop();
		ts[k++] = v;

		for (jeden Nachfolger w von v )
			if(--inDegree[w] == 0)
				q.push(w);
	}
	if (k != n)
		cout << “Graph ist zyklisch; ”
		      << “keine top. Sortierung
moeglich”;
}
```

Teil 2: Graphenalgorithmen

- Anwendungen
- Definitionen
- Datenstrukturen für Graphen
- Elementare Algorithmen
- Topologisches Sortieren
- Kürzeste Wege
  - Problemstellung
  - Ungewichtete Graphen
  - Distanzgraphen
  - Gewichtete Digraphen
  - Netzpläne
  - Alle kürzeste Wege in gewichteten Digraphen
- Flüsse in Netzwerken
- Minimal aufspannende Bäume

Problemstellung (1)

Länge eines Weges

- Gewichteter Graph: Summe der Kantengewichte (Kantenkosten)
- Ungewichteter Graph: Anzahl der Kanten.
  (Ein ungewichteter Graph entspricht damit einem gewichteten Graphen, bei dem jede Kante die Kosten 1 hat)

Kürzester Weg und Distanz

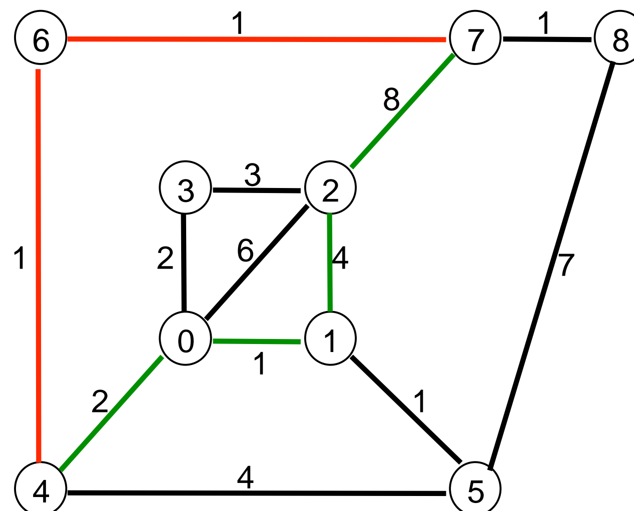
Im allgemeinen gibt es zwischen zwei Knoten u und v mehrere Wege.

Ein Weg von u nach v heißt kürzester Weg, falls der Weg minimale Länge hat.

Die Distanz d(u,v) zwischen u und v wird als die Länge eines kürzesten Weges definiert.

Falls es keinen Weg von u nach v gibt, ist $d(u,v) = \infty$.

Beispiel



Weg von 4 nach 7 mit Länge 15.

Kürzester Weg von 4 nach 7 mit Länge 2.

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

Problemstellung (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

Problemstellung (3)

Verschiedene Problemvarianten:

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

(2) Kürzeste Wege zwischen einem Knoten und allen anderen Knoten (Single-source shortest-path problem)

(3) Kürzeste Wege zwischen allen Knotenpaaren (all pairs shortest path)

- 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 (d.h. viele Kanten) gibt es jedoch eine effizientere Lösung, die zudem auch negative Kantengewichte zulässt.

Problemstellung (4)

Übersicht über die hier präsentierten Algorithmen:

	Problemtyp	**Algorithmus**
A	Problem (2) für ungewichtete Graphen	Erweiterte Breitensuche
B	Problem (2) für Distanz-Graphen (Graphen mit nur positiven Gewichten)	Algorithmus von Dijkstra; A*-Verfahren
C	Problem (2) für gewichtete Digraphen (gerichtete Graphen mit beliebigen d.h. auch negativen Gewichten)	Algorithmus von Moore und Ford
D	Problem (2) für gewichtete, azyklische Digraphen mit einem Start- und einem Endknoten (sogenannte Netzpläne)	Erweiterte topologische Sortierung
E	Problem (3) für gewichtete Digraphen (auch negative Gewichte zulässig)	Algorithmus von Floyd

Problem (2): Kürzeste Wege zwischen einem Knoten und allen anderen Knoten

Problem (3): Kürzeste Wege zwischen allen Knotenpaaren

<u>Beachte:</u>

Problemtyp D ist ein Spezialfall von C. Der auf Problemtyp D zugeschnittene Algorithmus ist wesentlich effizienter als der Algorithmus für Problemtyp C.

Teil 2: Graphenalgorithmen

- Anwendungen
- Definitionen
- Datenstrukturen für Graphen
- Elementare Algorithmen
- Topologisches Sortieren
- Kürzeste Wege
  - Problemstellung
  - Ungewichtete Graphen
  - Distanzgraphen
  - Gewichtete Digraphen
  - Netzpläne
  - Alle kürzeste Wege in gewichteten Digraphen
- Minimal aufspannende Bäume
- Flüsse in Netzwerken
- Zusammenhangskomponenten

Erweiterte Breitensuche (1)

Problem:

Eingabe: Ungewichteter Graph mit Startknoten s.
Ausgabe: Kürzeste Wege zwischen s und allen anderen Knoten

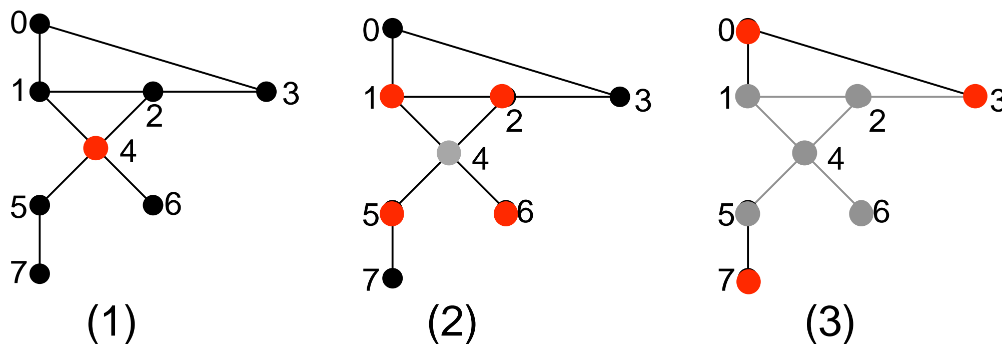
Idee: Breitensuche mit Abspeichern der gefundenen Wege und Distanzen

- Beginne bei s und durchlaufe Graph mit Breitensuche:
  zuerst werden alle Knoten mit Distanz 1 besucht, dann alle Knoten mit Distanz 2, usw.
- Distanzfeld d: Speichere für jeden besuchten Knoten v die Distanz d[v] zu Startknoten s.
- Vorgängerfeld p: p[v] = Vorgängerknoten im kürzesten Weg nach v.

  Damit ergibt sich für v folgender kürzester Weg (in umgekehrter Reihenfolge):

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

Beispiel: Startknoten s = 4



(1)

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

(2)

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

(3)

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

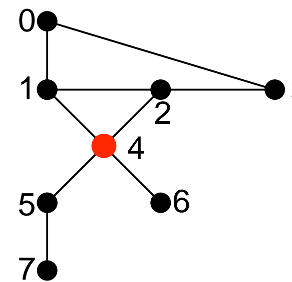
Erweiterte Breitensuche (2)

Algorithmus

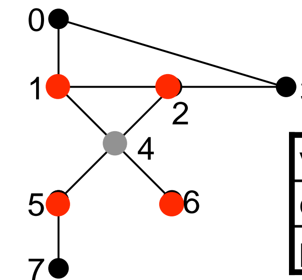
Eingabe: Startknoten s und ungewichteter Graph G

Ausgabe: Distanzfeld d und Vorgängerfeld p

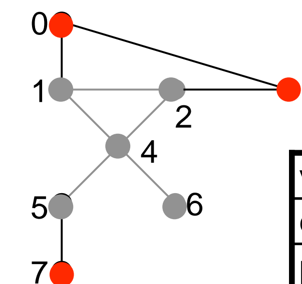
```
void shortestPath  (Vertex s, Graph G,
                    int d[ ], Vertex p[ ])
{
      for (jeden Knoten v) {
            d[v] = ∞;
            p[v] = undef;
      }
      d[s] = 0; // Distanz für Startknoten

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

      while (! q.empty() )
      {
            v = q.front(); q.pop();
            for (jeden Nachbarn w von v )
                  if (d[w] == ∞) { // w noch nicht besucht
                        d[w] = d[v] + 1;
                        p[w] = v;
                        q.push(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

Erweiterte Breitensuche (3)

Analyse:

Mit Adjazenlistendarstellung ergibt sich eine Laufzeit von:

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

(Begründung wie bei der topologischen Sortierung.)

Teil 2: Graphenalgorithmen

- Anwendungen
- Definitionen
- Datenstrukturen für Graphen
- Elementare Algorithmen
- Topologisches Sortieren
- Kürzeste Wege
  - Problemstellung
  - Ungewichtete Graphen
  - Distanzgraphen
  - Gewichtete Digraphen
  - Netzpläne
  - Alle kürzeste Wege in gewichteten Digraphen
- Minimal aufspannende Bäume
- Zusammenhangskomponenten
- Flüsse in Netzwerken
- Schwierige Probleme

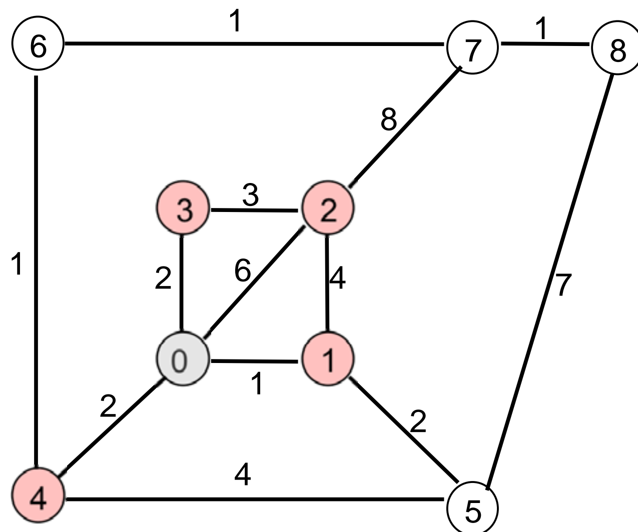
Algorithmus von Dijkstra (1)

Problem:

Eingabe: Distanz-Graph (Gewichte sind positiv) mit Startknoten s.
Ausgabe: Kürzeste Wege zwischen s und allen anderen Knoten

Idee

- Ähnlich wie bei der Breitensuche werden alle Knoten, die als nächstes besucht werden müssen, in einer Liste K (Kandidatenliste) gehalten.
- Aus der Kandidatenliste wird immer der Knoten mit der kleinsten Distanz als nächster besucht.



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

Als nächstes wird Knoten 1 mit Distanzwert 1 besucht.

Kandidatenliste

Bereits besucht

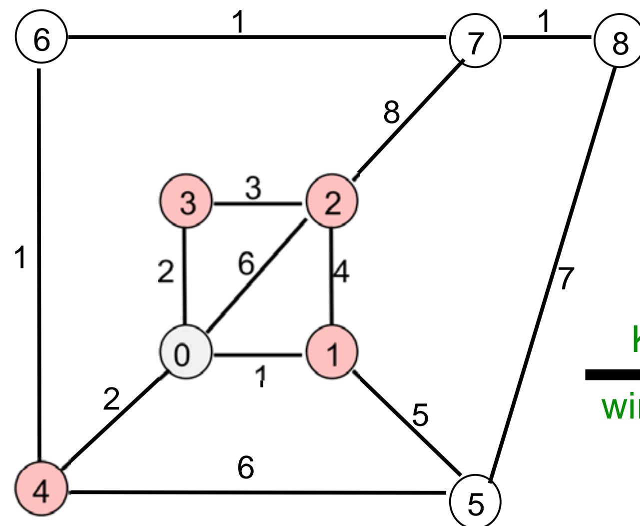
d[v] Distanz zwischen v und Startknoten s = 0

p[v] Vorgängerknoten im kürzesten Weg nach v

Algorithmus von Dijkstra (2)

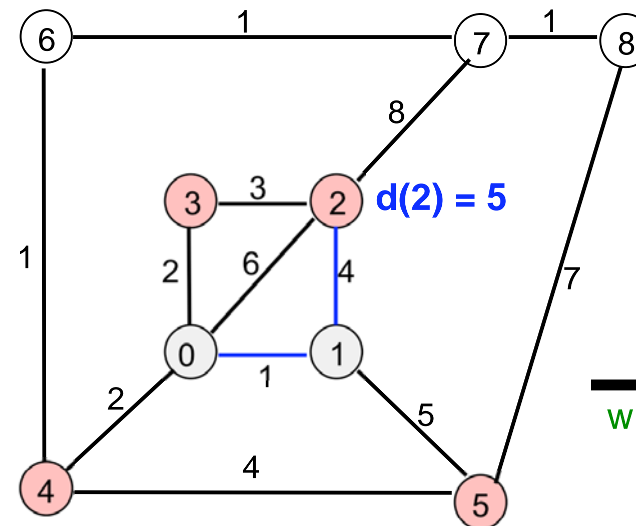
Idee (Fortsetzung)

- Wird ein neuer Knoten v besucht, dann kommen eventuell neue Nachbarknoten zur Kandidatenliste.
- Bei allen Nachbarknoten 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 dazu gekommen.

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 (3)

```
void shortestPath  (Vertex s, DistanzGraph G,  int d[ ], Vertex p[ ] )
{
        Set kl;   // Kandidatenliste

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

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

Eingabe: Startknoten s und Distanzgraph G

Ausgabe: Distanzfeld d und Vorgängerfeld p

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

Die Zugriffsoperationen auf die Kandidatenliste sind rot gekennzeichnet.

Mögliche Implementierungen werden auf der nächsten Folie diskutiert.

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

Man beachte, dass sich der Distanzwert nur für Knoten w aus kl verbessern kann.

Algorithmus von Dijkstra (4)

(a) Implementierung der Kandidatenliste als Vorrangswarteschlange

Eine Vorrangswarteschlange (Priority Queue) bietet eine effiziente Realisierung folgender Operationen an:

- insert(v, d); — fügt Element mit Vorrangswert (hier Distanzwert) d ein. Aufruf in (1) und (3).
- v = delMin(); — löscht Element mit kleinstem Vorrangswert. Aufruf in (2).
- change(v, dNeu); — ändert den Vorrangwert des Elements v auf dNeu. Aufruf in (4).

Eine Vorrangswarteschlange kann mit einer Heap-Struktur (siehe HeapSort aus Programmiertechnik 2) implementiert werden. Damit können insert, delMin und change in O(log n) durchgeführt werden. Für change wird noch zusätzlich eine Suchstruktur benötigt, um den Knoten v finden zu können.

(b) Implementierung der Kandidatenliste als einfaches (unsortiertes) Feld

- insert-Operation in O(1). Siehe (1) und (3).
- Löschen des kleinsten Element in O(n). Siehe (2).
- In Schritt (4) ist keine Umorganisation des Feldes notwendig: daher O(1) für Schritt (4).

Algorithmus von Dijkstra (5)

Analyse:

(a) Kandidatenliste als einfaches (unsortiertes) Feld

Die while-Schleife wird |V|-mal durchgeführt.

Für (2) ist O(|V|) notwendig. Die for-Schleife braucht ebenfalls O(|V|) . Insgesamt:

$\mathbf{T = O(|V|^2)}$

(b) Kandidatenliste als Vorrangswarteschlange

- (3) und (4) (change-Operation) benötigen O(log|V|).

  Jede Kante im Graph wird genau einmal in einer der for-Schleifen betrachtetet.

  Damit ist für alle for-Schleifen insgesamt O(E log|V|) notwendig.
- Hinzu kommt |V|-mal die Ausführung von (2), die in O(log|V|) realisiert werden kann.
- Insgesamt:

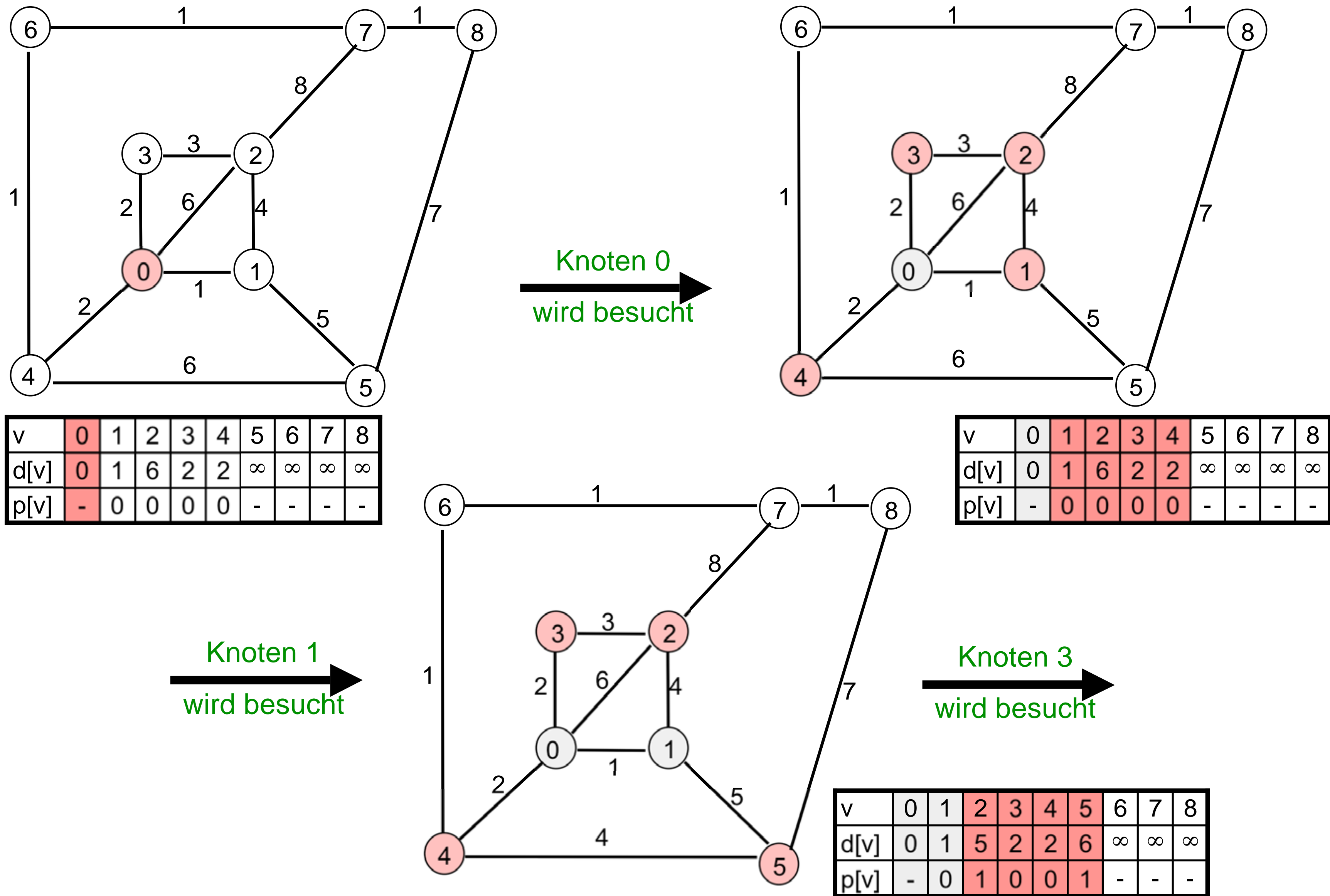
$\mathbf{T = O(E \log|V|)}$

Fazit:

Ist der Graph dicht besetzt, d.h. $|E| = O(|V|^2)$, dann ist die Variante (a) besser.

Ist der Graph dünn besetzt, d.h. |E| = O(|V|), dann ist die Variante (b) besser.

Beispiel zu Algorithmus von Dijkstra (1)

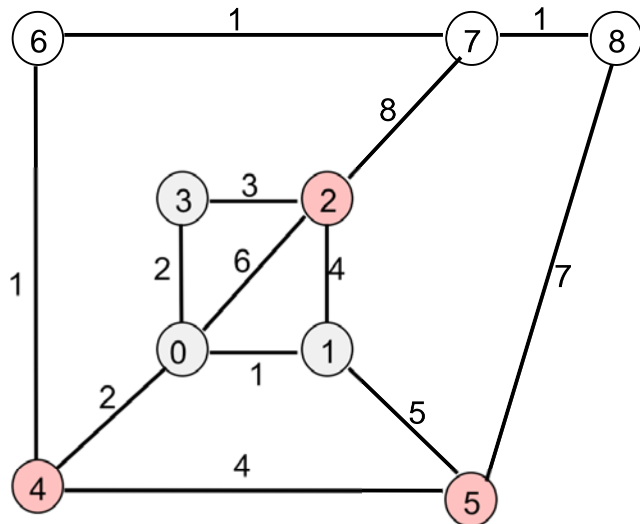


v	0	1	2	3	4	5	6	7	8
d[v]	0	1	6	2	2	∞	∞	∞	∞
p[v]	-	0	0	0	0	-	-	-	-

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

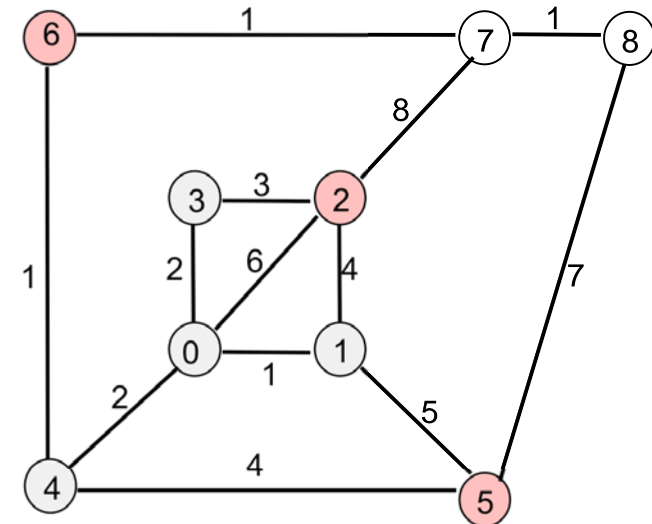
v	0	1	2	3	4	5	6	7	8
d[v]	0	1	5	2	2	6	∞	∞	∞
p[v]	-	0	1	0	0	1	-	-	-

Beispiel zu Algorithmus von Dijkstra (2)



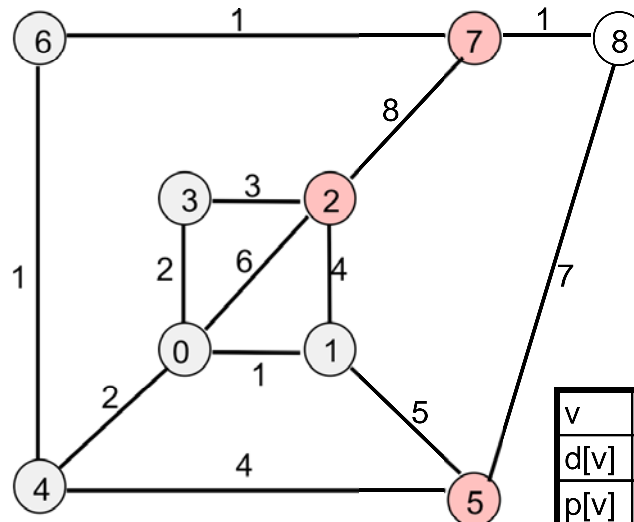
v	0	1	2	3	4	5	6	7	8
d[v]	0	1	5	2	2	6	∞	∞	∞
p[v]	-	0	1	0	0	1	-	-	-

Knoten 4 wird besucht



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

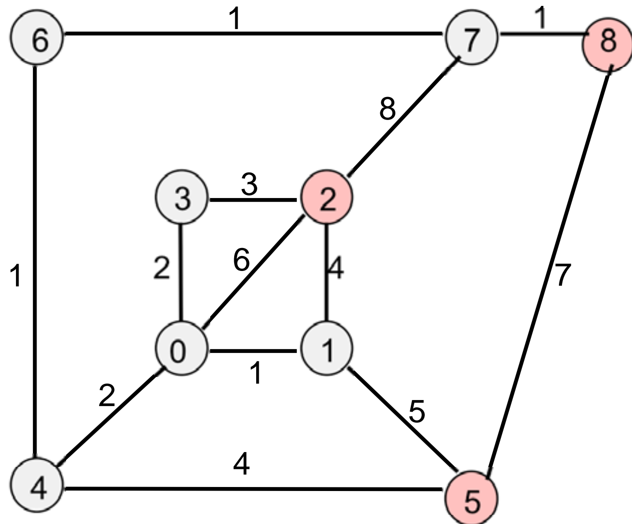
Knoten 6 wird besucht



Knoten 7 wird besucht

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

Beispiel zu Algorithmus von Dijkstra (3)



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

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

A*-Algorithmus (1)

Verbesserung des Algorithmus von Dijkstra:

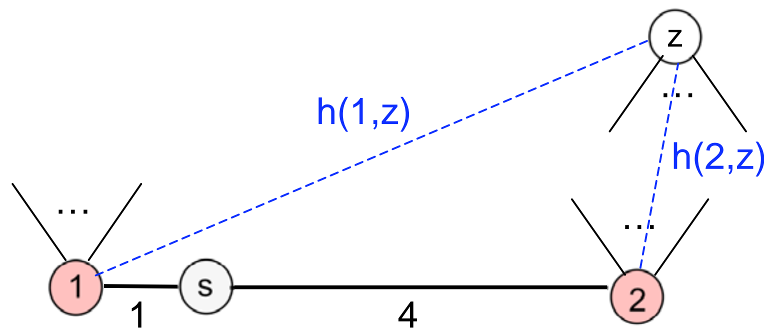
- Ist der kürzeste Weg zu genau einem Zielknoten z gesucht und lässt sich die Distanz von einem Knoten v zum Zielknoten z durch eine Schätzfunktion h(v,z) abschätzen, dann lässt sich der Algorithmus von Dijkstra optimieren.
- Statt den Knoten v mit

  d[v] minimal

  als nächstes zu besuchen zu besuchen, wird der Knoten v besucht mit

  d[v] + h(v,z) minimal.

  Damit werden Knoten v bevorzugt, die näher zum Zielknoten liegen.
- Sind die Knoten des Graphen Punkte in der Euklidischen Ebene und die Gewichte gleich den Euklidischen Abständen („Länge der Luftlinie“), dann lässt sich als Schätzfunktion h(v,z) der Euklidische Abstand zwischen v und z wählen.



Wird der kürzeste Weg von s nach z gesucht, wird zuerst Knoten 2 besucht, weil dieser näher zu z liegt.

A*- Algorithmus (2)

```
bool shortestPath (Vertex s, Vertex z, DistanzGraph G,  int d[ ], Vertex p[ ] )
{
        Set kl;   // Kandidatenliste

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

        while (! kl.empty() ) {
             lösche Knoten v aus kl mit d[v] + h(v,z) minimal;
             if (v == z) // Zielknoten z erreicht
                    return true;
             for (jeden Nachbar w von v ) {
                    if (d[w] == ∞) { // w noch nicht besucht und nicht in Kandidatenliste
                           kl.insert(w);
                    if (d[v] + c(v,w) < d[w]) {
                           p[w] = v;
                           d[w] = d[v] + c(v,w);
                    }
             }
        }
       return false;
}
```

Eingabe: Startknoten s, Zielknoten z und Distanzgraph G

Ausgabe: Distanzfeld d und Vorgängerfeld p

Änderungen gegenüber dem Diskstra-Algorithmus sind blau gekennzeichnet.

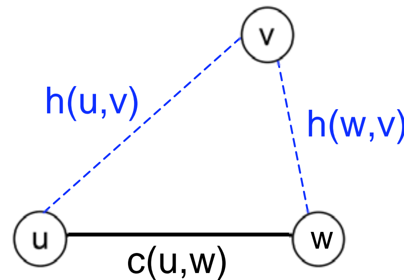
A*- Algorithmus (3)

Korrektheit des 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 optimistisch:
$h(u,v) \leq d(u,v)$ für alle Knoten u, v
(d.h. h unterschätzt die tatsächliche Distanz d)

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



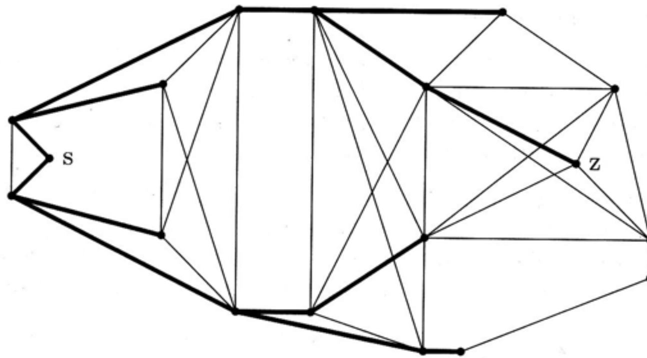
Bemerkungen:

- 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)

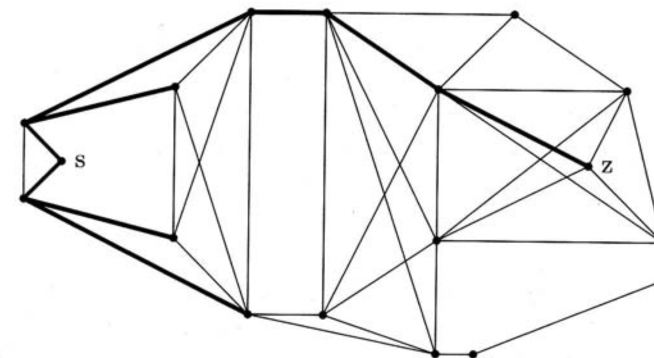
A*- Algorithmus (4)

Beispiel

Aus [Turau 2004]:



Dijkstra-Algorithmus



A*-Algorithmus

Schätzfunktion h und Kantengewicht c sind als Euklidischer Abstand definiert.