2.6 Graphen

2.6.4 Kürzeste Pfade

Kürzeste Pfade

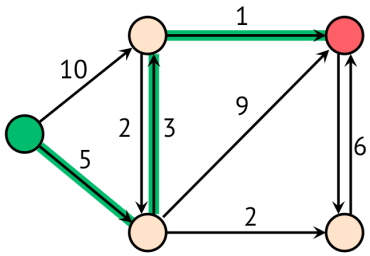
- Gegeben sei ein gerichteter, gewichteter Graph G=(V,E) mit Gewichtsfunktion w:E→R. Das Gewicht w(p) eines Pfades

$$p = v_0, v_1, \ldots, v_k$$

ist definiert als

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Kürzeste Pfade



w(p) = 5 + 3 + 1 = 9

Wenn man die Kantengewichte als "Weglänge" interpretiert, dann wäre der kürzeste Pfad zwischen zwei Knoten u und v der Pfad zwischen
beiden Knoten, der die geringste Wegstrecke aufweist.
In der Praxis können die Kantengewichte für die verschiedensten Kostenfaktoren stehen (Geld-/Zeitkosten, Wartezeiten, etc.).

Kürzeste Pfade

- Das kürzeste Pfad-Gewicht $\delta(u,v)$ von Knoten u zu Knoten v ist definiert als

  $\delta(u,v) = \min\{w(p): p \text{ ist Pfad von } u \text{ nach } v\}$

  wobei $\min \varnothing = \infty$
- Ein kürzester Pfad von Knoten u zu Knoten v ist ein beliebiger Pfad p mit

  $w(p) = \delta(u,v)$

Kürzeste Pfade

- Lemma

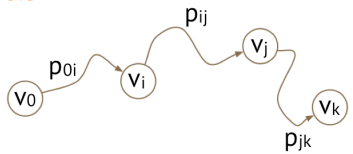
  Teilpfade von kürzesten Pfaden sind ebenfalls kürzeste Pfade.

  Ist $p=v_0,\ldots,v_k$ ein kürzester Pfad von v_0 nach v_k, so ist für $0 \leqslant i < j \leqslant k$ der Teilpfad $p_{ij}=v_i,\ldots,v_j$ ein kürzester Pfad von v_i nach v_j

Kürzeste Pfade sind i.d.R. nicht eindeutig. Beispiel hier wieder: Graph mit homogenen Kantengewichten.

Kürzeste Pfade

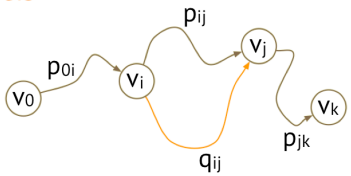
- Beweis



Es ist $p = p_{0i},p_{ij},p_{jk}$ und damit
$w(p) = w(p_{i0}) + w(p_{ij}) + w(p_{jk})$

Kürzeste Pfade

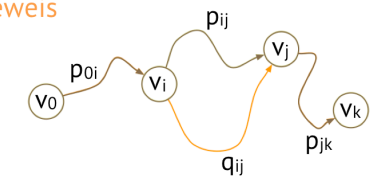
- Beweis



Annahme: Es gibt einen Pfad q_{ij} von v_i nach v_j mit $w(q_{ij})<w(p_{ij})$

Kürzeste Pfade

- Beweis



$$w(p') = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$$
$$< w(p_{0i}) + w(q_{ij}) + w(p_{jk})$$
$$= w(p)$$

- Für den Pfad $p'=p_{0i},q_{ij},p_{jk}$ gilt Widerspruch!

Dieser Satz wird benutzt für den Korrektheitsbeweis des Algorithmus von Dijkstra.
Fazit: Alle Teilpfade eines kürzesten Pfades haben auch minimales Gewicht.

Kürzeste Pfade

- Single-source shortest-paths Problem
  Gegeben sei ein Graph G=(V,E) und ein Startknoten s∈V. Bestimme zu jedem Knoten u∈V einen kürzesten Pfad von s nach u.
- All-pairs shortest-paths Problem
  Gegeben sei ein Graph G=(V,E). Bestimme zu jedem Knotenpaar u∈V und v∈V einen kürzesten Pfad von u nach v.

Oben Dijkstra, unten Floyd-Warshall

Darstellung

- Vorgänger
  Ein Pfad von s nach v wird dargestellt, indem jeder Knoten des Pfades seinen Vorgänger in einem Attribut p speichert.

Darstellung

- Oberschranke
  In jedem Knoten v des Graphen wird ein Attribut d gespeichert, das eine obere Schranke für das Gewicht des kürzesten Pfades von s nach v darstellt, d.h. es gilt immer die Invariante

$$\delta(s,v) \leq d[v]$$

- Ein Knoten heißt final, falls die Oberschranke minimal ist, d.h. falls

$$\delta(s,v) = d[v]$$

Darstellung

- InitializeSingleSource(G,s)
  for each $v \in V$ do
    $d[v] \leftarrow \infty$
    $p[v] \leftarrow NIL$
  $d[s] \leftarrow 0$

Initiialisierung des Algorithmus

Darstellung

- Relaxation einer Kante (u,v)
  Die Oberschranke d[v] kann verbessert werden, wenn eine Kante von u nach v existiert, so dass

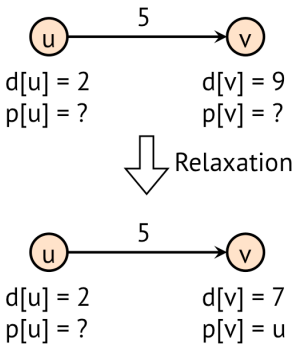
  $d[u] + w(u,v) < d[v]$

  ist. In diesem Fall setzt man

  $d[v] \leftarrow d[u] + w(u,v)$
  $p[v] \leftarrow u$

Relaxation bedeutet, dass man zu einem bestimmten Zeitpunkt einen Pfad zu einem bereits eroberten Knoten finden kann,
der kürzer ist, als der vorher bestimmte. In diesem Falle wird der Pfad und die Oberschranke auf den neuen Wert gesetzt.

Relaxation



Relaxation

- Relax(u,v)

  if d[v] > d[u] + w(u,v) then

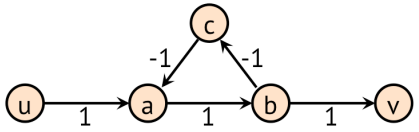
  d[v] ← d[u] + w(u,v)

  p[v] ← u

Implementierung der Relaxation. Sobald ein Pfad gefunden werden kann, der kürzer ist, als die aktuell gültige obere Schranke,
dann wird das Gesamtgewicht (eben diese obere Schranke) auf den neu gefundenen Wert gesetzt.

Negative Gewichte

- Prinzipiell sind gewichtete Graphen mit negativen Gewichten zulässig, dadurch kann aber die Frage nach kürzesten Pfaden unsinnig werden.



Ein Pfad, der einen positiv gewichteten Zyklus enthält, kann nicht der kürzeste Pfad sein.
Manchmal führen negative Gewichte zu unlösbaren Konfigurationen.
In dem obigen Falle wäre das Suchen nach dem kürzesten Pfad sinnlos, da jeder Zyklus zwischen den Knoten
a, b und c die Pfadlänge verringern würde.

2.6.4 Kürzeste Pfade

Dijkstras Algorithmus

- Dijkstras Algorithmus löst das single-source shortest-path Problem für den Fall, dass alle Kantengewichte positiv sind.

Bei Dijkstra wählt man einen Quell-Knoten und berechnet den kürzesten Pfad zu allen anderen Knoten.

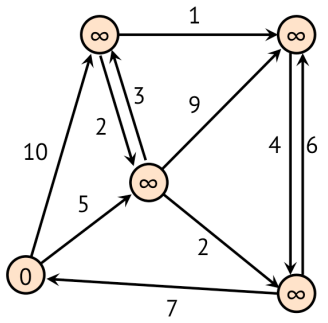
Dijkstras Algorithmus

1. Es sei S eine anfangs leere Menge von Knoten.
2. Wähle einen Knoten $v \in V-S$ mit minimaler Oberschranke $d[v]$.
3. Füge v in S ein und relaxiere alle von v ausgehenden Kanten.
4. Gehe zu Schritt 2 falls $V-S \neq \emptyset$

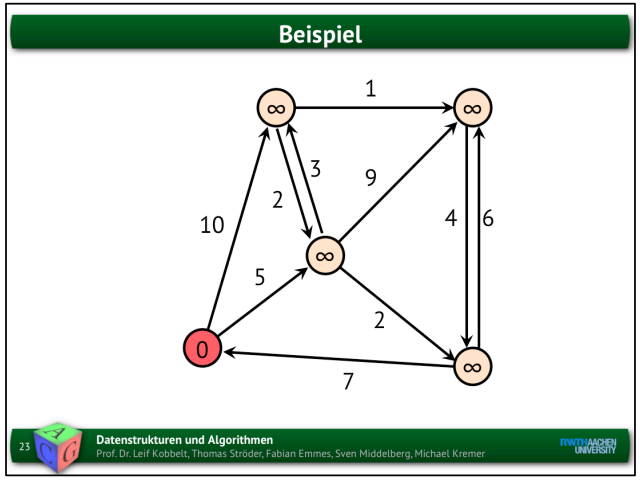
Algorithmus

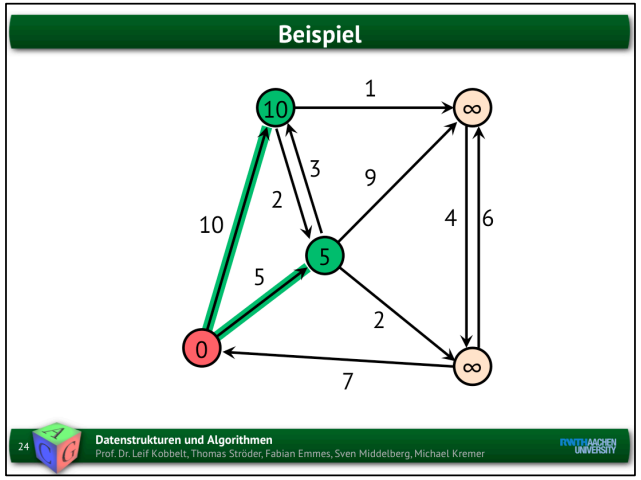
- Dijkstra(G,s)
  - InitializeSingleSource(G,s)
  - S ← ∅
  - Q ← V
  - while Q ≠ ∅ do
    - u ← ExtractMinimum(Q)
    - S ← S ∪ {u}
    - for each v ∈ Adj[u] do
      - Relax(u,v)

Beispiel

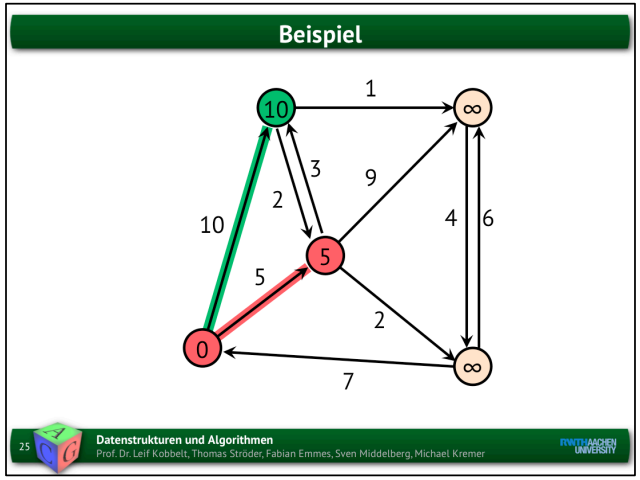


Der Aufbau ist ähnlich zu Prims Algorithmus.

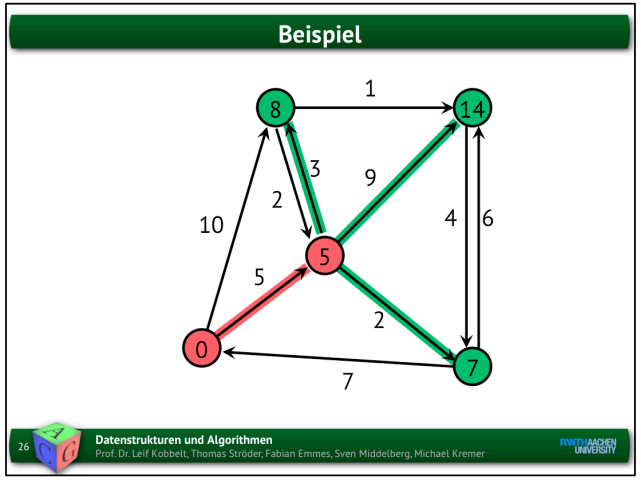




0 + 5 < inf und 0 + 10 < inf, deshalb werden beide Nachbarn erobert.

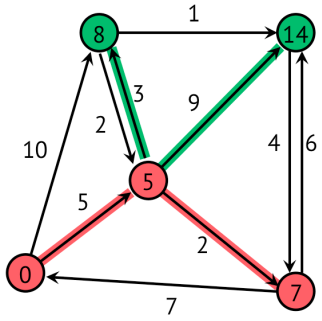


Man wählt nun den kürzesten Pfad, also den, der zum Knoten mit dem Wert 5 führt.

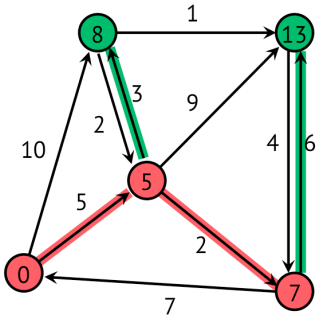


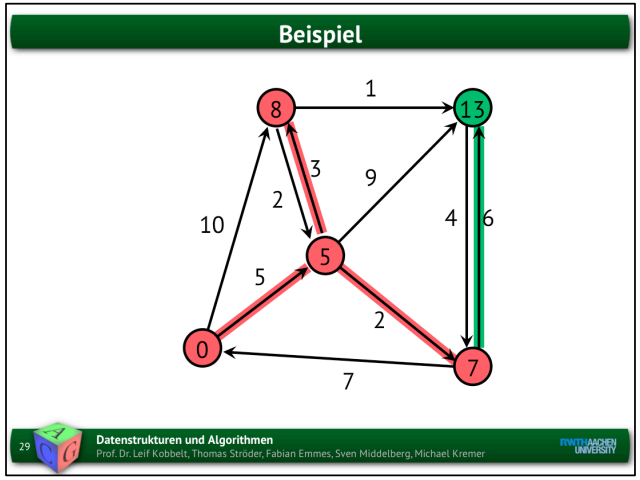
Man macht jetzt überall ein Update. Bei Prims Algorithmus würde der rechte untere Knoten nun den Wert 2 erhalten. Bei Dijkstra
zählt aber der bereits "gegangene" Pfad.

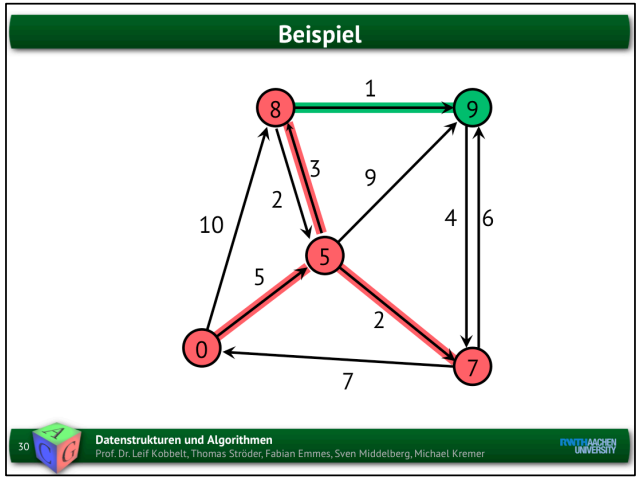
Beispiel



Beispiel

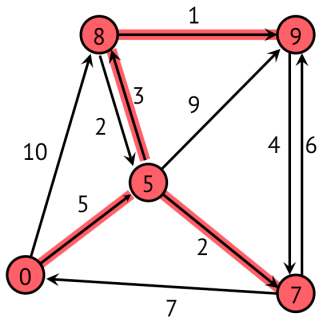






Nun wählt man wieder den Pfad mit dem geringsten Gewicht (von allen Schnittkanten).

Beispiel



Korrektheit

- Korrektheit von Dijkstras Algorithmus
  Es sei $G=(V,E)$ ein gerichteter, gewichteter Graph mit nichtnegativer Gewichtsfunktion w und $s \in V$ ein Startknoten. Dann gilt: Dijkstras Algorithmus angewandt auf G terminiert mit $d[v] = \delta(s,v)$ für alle Knoten $v \in V$.

Korrektheit

- Beweis
  - Schleifeninvariante

    Es gilt d[v]= δ (s,v) für alle Knoten v∈S, d.h. alle Knoten in S sind final.
  - Initialisierung

    Vor Ausführung der Schleife ist S=∅ und die Schleifeninvariante gilt trivialerweise.

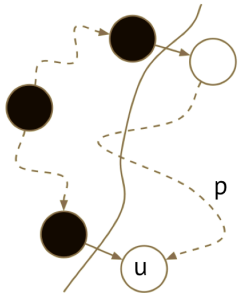
Korrektheit

- Beweis
  - Schleife

    Behauptung: Es gilt d[v]= δ (s,v) für jeden Knoten v, der zu S hinzugefügt wird.
  - Beweis durch Widerspruch: Es sei u der erste Knoten, für den d[u]≠ δ (s,u) ist…

Korrektheit

Es sei u der erste Knoten, für den d[u]≠δ(s,u) ist...

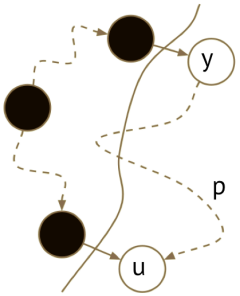


u ist von s erreichbar, es gibt also einen kürzesten Pfad p von s nach u

Schwarz: Die bereits eroberten Knoten. Weiß: Knoten, die im nächsten Schritt erobert werden können.

Korrektheit

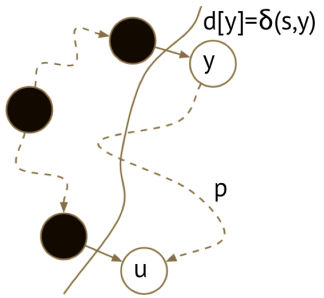
Es sei u der erste Knoten, für den d[u]≠δ(s,u) ist...



Es sei y der erste Knoten auf p, der nicht in S liegt, und x sei sein Vorgänger.

Korrektheit

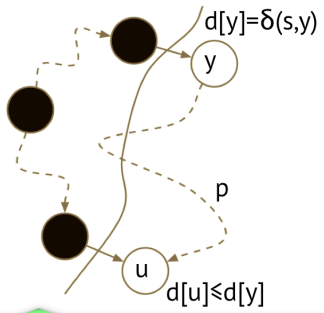
Es sei u der erste Knoten, für den $d[u] \neq \delta(s,u)$ ist...



Teilpfade von kürzesten Pfaden sind kürzeste Pfade, (x,y) wurde bereits relaxiert, also ist $d[y] = \delta(s,y)$

Korrektheit

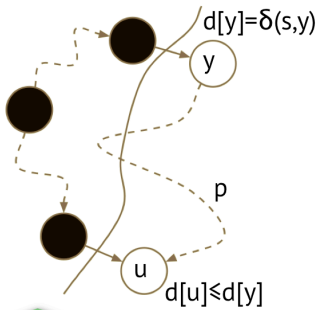
Es sei u der erste Knoten, für den $d[u] \neq \delta(s,u)$ ist...



u wird von Dijkstras Algorithmus vor y ausgewählt, also ist $d[u] \leqslant d[y]$

Korrektheit

Es sei u der erste Knoten, für den $d[u] \neq \delta(s,u)$ ist...



Es gilt
$d[y]=\delta(s,y) \leq \delta(s,u) \leq d[u]$
und
$d[u] \leq d[y]$
also
$d[y]=\delta(s,y)=\delta(s,u)=d[u]$
und insbesondere
$d[u]=\delta(s,u)$

Widerspruch!

Dijkstra würde u vor y auswählen, weil d[u] <= d[y].
D.h., der Algorithmus ist korrekt, weil in jedem Relaxationsschritt des Knoten hinzugefügt wird, der durch
den kürzesten Pfad zugänglich ist.

Korrektheit

- Beweis
  - Terminierung

    Terminierung wenn $Q=\emptyset$, also wegen

    $Q = V-S$, wenn $S=V$

Analyse

- Aufwand

  Implementierung der Priority-Queue durch einen binären Heap.

Analyse

- Dijkstra(G,s)
  - InitializeSingleSource(G,s)
  - $S \leftarrow \varnothing$ — Initialisierung: O(V)
  - $Q \leftarrow V$
  - while $Q \neq \varnothing$ do
    - $u \leftarrow$ ExtractMinimum(Q)
    - $S \leftarrow S \cup \{u\}$
    - for each $v \in$ Adj[u] do
      - Relax(u,v)

Analyse

- Dijkstra(G,s)
  - InitializeSingleSource(G,s)
  - S ← ∅
  - Q ← V
  - while Q ≠ ∅ do
    - u ← ExtractMinimum(Q)
    - S ← S ∪ {u}
    - for each v ∈ Adj[u] do
      - Relax(u,v)

Aufwand für binären Heap

O(V)

O(logV)

Analyse

- Dijkstra(G,s)
  - InitializeSingleSource(G,s)
  - S ← ∅
  - Q ← V
  - while Q ≠ ∅ do
    - u ← ExtractMinimum(Q)
    - S ← S ∪ {u}
    - for each v ∈ Adj[u] do
      - Relax(u,v)

Schleife wird V mal durchlaufen, insgesamt also O(V×logV)

Analyse

- Dijkstra(G,s)
  - InitializeSingleSource(G,s)
  - S ← ∅
  - Q ← V
  - while Q ≠ ∅ do
    - u ← ExtractMinimum(Q)
    - S ← S ∪ {u}
    - for each v ∈ Adj[u] do
      - Relax(u,v)

Jede Kante wird höchstens einmal relaxiert, insgesamt also O(E×logV)

Analyse

- Dijkstra(G,s)
  - InitializeSingleSource(G,s)
  - S ← ∅
  - Q ← V
  - while Q ≠ ∅ do
    - u ← ExtractMinimum(Q)
    - S ← S ∪ {u}
    - for each v ∈ Adj[u] do
      - Relax(u,v)

O(V)

O((E+V)×log V)

Analyse

- Fazit

  Falls G zusammenhängend ist, so hat Dijkstras Algorithmus einen Aufwand von

  $$O(E \times \log V)$$

2.6.4 Kürzeste Pfade

Floyd-Wahrshall: Berechne nicht nur von einem Startknoten die Distanz zu allen anderen, sondern von jedem zu jedem Knoten.

Kürzeste Pfade

- Single-source shortest-paths Problem
  Gegeben sei ein Graph G=(V,E) und ein Startknoten s∈V. Bestimme zu jedem Knoten u∈V einen kürzesten Pfad von s nach u.
- All-pairs shortest-paths Problem
  Gegeben sei ein Graph G=(V,E). Bestimme zu jedem Knotenpaar u∈V und v∈V einen kürzesten Pfad von u nach v.

Kürzeste Pfade

- In diesem Abschnitt gehen wir davon aus, dass …
  - … die Knotenmenge V des Graphen oBdA V={1,2,…,n} ist.
  - … die gewichteten Kanten in einer Adjazenzmatrix W=(w_{ij}) gespeichert sind.

Die Knoten werden nun durchnummeriert und die Kantengewichte in einer einzigen Adjazenzmatrix gespeichert.
Beachte, dass diese Matrix bei gerichteten Graphen i.d.R. nicht symmetrisch ist.

Floyd-Warshall Algorithmus

- Es sei $v_0,v_2,\ldots,v_{m-1},v_m$ ein einfacher Pfad. Die Knoten $v_1,\ldots,v_{m-1}$ heißen Zwischenknoten des Pfades.
- Es sei P_k die Menge der Pfade, deren sämtliche Zwischenknoten in $\{1,\ldots,k\}$ liegen.

$p_{ij} \in P_k$

enthält nur Zwischenknoten v mit $1 \leqslant v \leqslant k$

D.h., P_n wäre die Menge ALLER Pfade im Graphen.
In P_2 sind alle Pfade drin, die als Zwischenknoten nur die Knoten 1 und 2 beinhalten, oder direkt zwei Knoten verbinden (also alle Pfade der Länge 1).

Floyd-Warshall Algorithmus

- Idee: Dynamisches Programmieren
  - Rekursionsformel für die Gewichte der kürzesten Pfade d
  - Rekursionsformel für die Vorgängerattribute p

Idee: Der optimale Pfade in P_k ergibt sich aus optimalen Pfaden aus P_(k-1).

Floyd-Warshall Algorithmus

- Rekursionsformel für die Gewichte

  Es sei $d_k[i,j]$ das Gewicht eines kürzesten Pfades von Knoten i nach Knoten j in P_k. Insbesondere ist also $d_n[i,j]$ das Gewicht eines kürzesten Pfades von i nach j in G.

Floyd-Warshall Algorithmus

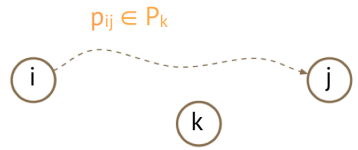
- Initialisierung $k = 0$

  Falls $k = 0$, so dürfen auf den Pfaden in $P_k=P_0$ überhaupt keine Zwischenknoten liegen, d.h. die Länge der Pfade ist höchstens 1 und wir haben $d_0[i,j] = w(i,j)$.

P_0 = Nur direkte Verbindungen zwischen zwei Knoten, d.h. d_0[i,j] ist einfach unsere Adjazenzmatrix.

Floyd-Warshall Algorithmus

- Iteration $k-1 \rightarrow k$
  Es sei p_{ij} ein kürzester Pfad von i nach j in P_k.
  - Fall 1: Ist k kein Zwischenknoten von p_{ij}, so ist p_{ij} auch ein kürzester Pfad von i nach j in P_{k-1} und somit $d_k[i,j] = d_{k-1}[i,j]$



Floyd-Warshall Algorithmus

- Iteration $k-1 \rightarrow k$
  Es sei p_{ij} ein kürzester Pfad von i nach j in P_k.
  - Fall 2: Ist k ein Zwischenknoten von p_{ij}, so ist p_{ik} ein kürzester Pfad von i nach k in P_{k-1} und p_{kj} ein kürzester Pfad von k nach j in P_{k-1} und somit $d_k[i,j] = d_{k-1}[i,k] + d_{k-1}[k,j]$

k kommt in Pfad zwischen i und j vor. Dann setzt sich der kürzeste Pfad von i nach j als die Summe des kürzesten Pfades von i nach k und dem von k nach j zusammen. Diese beiden Teilpfade sind zu diesem Zeitpunkt bereits bekannt, weil weder der Pfad von i nach k nach der von k nach j den Knoten k beinhaltet und wir in aufsteigender Reihenfolge vorgehen.

Floyd-Warshall Algorithmus

- Rekursionsformel

$$d_k[i,j] = \begin{cases} w(i,j) & k = 0 \\ \min(d_{k-1}[i,j], d_{k-1}[i,k]+d_{k-1}[k,j]) & k > 0 \end{cases}$$

Floyd-Warshall Algorithmus

- Rekursionsformel für die Vorgänger

  Es sei $p_k[i,j]$ der Vorgänger von j auf einem kürzesten Pfad von i nach j in P_k.

Floyd-Warshall Algorithmus

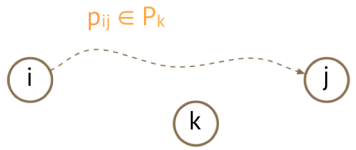
- Initialisierung k = 0
  Möglich sind nur Pfade der Länge 1.

$$p_0[i,j] = \begin{cases} \infty & \text{falls } i = j \text{ oder } w(i,j) = \infty \\ i & \text{falls } i \neq j \text{ und } w(i,j) < \infty \end{cases}$$

Initialwert ist unendlich, falls es keine direkte Verbindung zwischen Knoten i und j gibt.
Ansonsten wird der Wert des Knatengewichts zum Initialisieren verwendet.

Floyd-Warshall Algorithmus

- Iteration k−1 → k
  - Fall 1: Ein kürzester Pfad in P_k enthält den Knoten k nicht, also:
    $d_{k-1}[i,j] \leqslant d_{k-1}[i,k] + d_{k-1}[k,j]$
    Dann ist $p_k[i,j] = p_{k-1}[i,j]$.



Floyd-Warshall Algorithmus

- Iteration $k-1 \rightarrow k$
  - Fall 2: Ein kürzester Pfad in P_k enthält den Knoten k, also:

    $d_{k-1}[i,j] > d_{k-1}[i,k] + d_{k-1}[k,j]$

    Dann ist $p_k[i,j] = p_{k-1}[k,j]$.

Floyd-Warshall Algorithmus

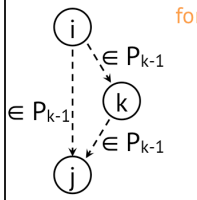
- Rekursionsformel

$$p_k[i,j] = \begin{cases} p_{k-1}[i,j] & \text{falls } d_{k-1}[i,j] \leq d_{k-1}[i,k]+d_{k-1}[k,j] \\ p_{k-1}[k,j] & \text{sonst} \end{cases}$$

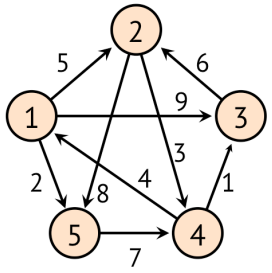
Floyd-Warshall Algorithmus

- FloydWarshall(W)

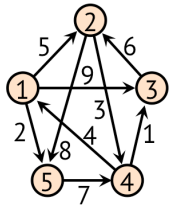
```
D_0 ← W
P_0 ← ...
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            if d_{k-1}[i,j] ≤ d_{k-1}[i,k]+d_{k-1}[k,j] then
                d_k[i,j] ← d_{k-1}[i,j]
                p_k[i,j] ← p_{k-1}[i,j]
            else
                d_k[i,j] ← d_{k-1}[i,k]+d_{k-1}[k,j]
                p_k[i,j] ← p_{k-1}[k,j]
return D_n, P_n
```



Beispiel



Beispiel



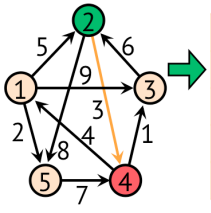
$D_0 =$

0	5	9	∞	2
∞	0	∞	3	8
∞	6	0	∞	∞
4	∞	1	0	∞
∞	∞	∞	7	0

$P_0 =$

∞	1	1	∞	1
∞	∞	∞	2	2
∞	3	∞	∞	∞
4	∞	4	∞	∞
∞	∞	∞	5	∞

Beispiel



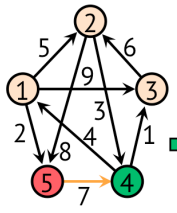
$D_0 =$

0	5	9	∞	2
∞	0	∞	3	8
∞	6	0	∞	∞
4	∞	1	0	∞
∞	∞	∞	7	0

$P_0 =$

∞	1	1	∞	1
∞	∞	∞	2	2
∞	3	∞	∞	∞
4	∞	4	∞	∞
∞	∞	∞	5	∞

Beispiel



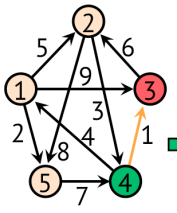
$D_0 =$

0	5	9	∞	2
∞	0	∞	3	8
∞	6	0	∞	∞
4	∞	1	0	∞
∞	∞	∞	7	0

$P_0 =$

∞	1	1	∞	1
∞	∞	∞	2	2
∞	3	∞	∞	∞
4	∞	4	∞	∞
∞	∞	∞	5	∞

Beispiel



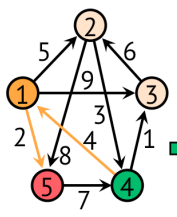
$D_{0\to 1} =$

0	5	9	∞	2
∞	0	∞	3	8
∞	6	0	∞	∞
4	∞	1	0	∞
∞	∞	∞	7	0

$P_{0\to 1} =$

∞	1	1	∞	1
∞	∞	∞	2	2
∞	3	∞	∞	∞
4	∞	4	∞	∞
∞	∞	∞	5	∞

Beispiel



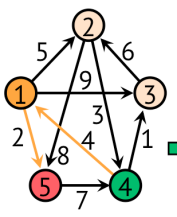
$D_{0\to1}$ =

0	5	9	∞	2
∞	0	∞	3	8
∞	6	0	∞	∞
4	∞	1	0	∞
∞	∞	∞	7	0

$P_{0\to1}$ =

∞	1	1	∞	1
∞	∞	∞	2	2
∞	3	∞	∞	∞
4	∞	4	∞	∞
∞	∞	∞	5	∞

Beispiel



$D_{0\to1}$ =

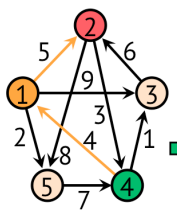
0	5	9	∞	2
∞	0	∞	3	8
∞	6	0	∞	∞
4	∞	1	0	6
∞	∞	∞	7	0

$P_{0\to1}$ =

∞	1	1	∞	1
∞	∞	∞	2	2
∞	3	∞	∞	∞
4	∞	4	∞	1
∞	∞	∞	5	∞

Hier betrachten wir alle Pfade, die als Zwischenknoten entweder Knoten 1 haben, oder eben zwei Knoten direkt miteinander verbinden (D_0).
In P_k werden dann jeweils alle Vorgängerknoten gespeichert.

Beispiel



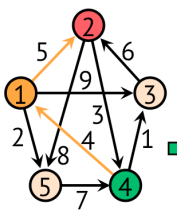
$D_{0\to 1}[F]$

0	5	9	∞	2
∞	0	∞	3	8
∞	6	0	∞	∞
4	∞	1	0	6
∞	∞	∞	7	0

$P_{0\to 1}[F]$

∞	1	1	∞	1
∞	∞	∞	2	2
∞	3	∞	∞	∞
4	∞	4	∞	1
∞	∞	∞	5	∞

Beispiel



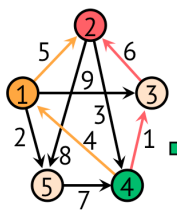
$D_{0\to 1}[F]$

0	5	9	∞	2
∞	0	∞	3	8
∞	6	0	∞	∞
4	9	1	0	6
∞	∞	∞	7	0

$P_{0\to 1}[F]$

∞	1	1	∞	1
∞	∞	∞	2	2
∞	3	∞	∞	∞
4	1	4	∞	1
∞	∞	∞	5	∞

Beispiel



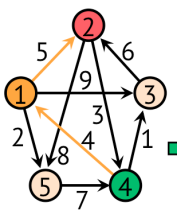
$D_{0\to 1}$ =

0	5	9	∞	2
∞	0	∞	3	8
∞	6	0	∞	∞
4	9	1	0	6
∞	∞	∞	7	0

$P_{0\to 1}$ =

∞	1	1	∞	1
∞	∞	∞	2	2
∞	3	∞	∞	∞
4	1	4	∞	1
∞	∞	∞	5	∞

Beispiel



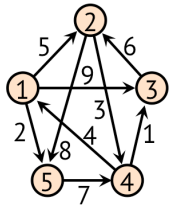
$D_{0\to 1}$ =

0	5	9	∞	2
∞	0	∞	3	8
∞	6	0	∞	∞
4	9	1	0	6
∞	∞	∞	7	0

$P_{0\to 1}$ =

∞	1	1	∞	1
∞	∞	∞	2	2
∞	3	∞	∞	∞
4	1	4	∞	1
∞	∞	∞	5	∞

Beispiel



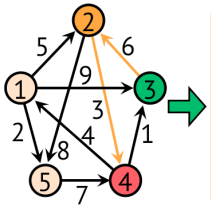
$D_1 =$

0	5	9	∞	2
∞	0	∞	3	8
∞	6	0	∞	∞
4	9	1	0	6
∞	∞	∞	7	0

$P_1 =$

∞	1	1	∞	1
∞	∞	∞	2	2
∞	3	∞	∞	∞
4	1	4	∞	1
∞	∞	∞	5	∞

Beispiel



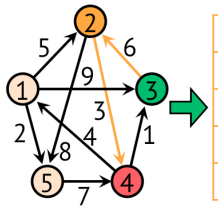
$D_{1\to 2} =$

0	5	9	∞	2
∞	0	∞	3	8
∞	6	0	∞	∞
4	9	1	0	6
∞	∞	∞	7	0

$P_{1\to 2} =$

∞	1	1	∞	1
∞	∞	∞	2	2
∞	3	∞	∞	∞
4	1	4	∞	1
∞	∞	∞	5	∞

Beispiel



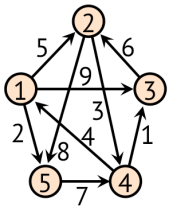
$D_{1\to 2}$ =

0	5	9	∞	2
∞	0	∞	3	8
∞	6	0	9	∞
4	9	1	0	6
∞	∞	∞	7	0

$P_{1\to 2}$ =

∞	1	1	∞	1
∞	∞	∞	2	2
∞	3	∞	2	∞
4	1	4	∞	1
∞	∞	∞	5	∞

Beispiel



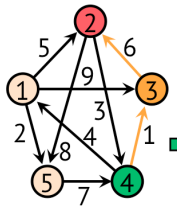
D_2 =

0	5	9	8	2
∞	0	∞	3	8
∞	6	0	9	14
4	9	1	0	6
∞	∞	∞	7	0

P_2 =

∞	1	1	2	1
∞	∞	∞	2	2
∞	3	∞	2	2
4	1	4	∞	1
∞	∞	∞	5	∞

Beispiel



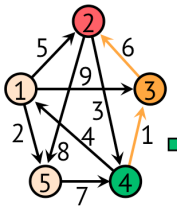
$D_{2\to 3}$

0	5	9	8	2
∞	0	∞	3	8
∞	6	0	9	14
4	9	1	0	6
∞	∞	∞	7	0

$P_{2\to 3}$

∞	1	1	2	1
∞	∞	∞	2	2
∞	3	∞	2	2
4	1	4	∞	1
∞	∞	∞	5	∞

Beispiel



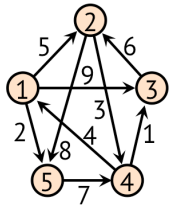
$D_{2\to 3}$

0	5	9	8	2
∞	0	∞	3	8
∞	6	0	9	14
4	7	1	0	6
∞	∞	∞	7	0

$P_{2\to 3}$

∞	1	1	2	1
∞	∞	∞	2	2
∞	3	∞	2	2
4	3	4	∞	1
∞	∞	∞	5	∞

Beispiel



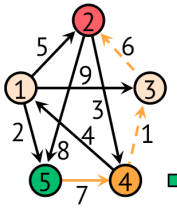
$D_3 =$

0	5	9	8	2
∞	0	∞	3	8
∞	6	0	9	14
4	7	1	0	6
∞	∞	∞	7	0

$P_3 =$

∞	1	1	2	1
∞	∞	∞	2	2
∞	3	∞	2	2
4	3	4	∞	1
∞	∞	∞	5	∞

Beispiel



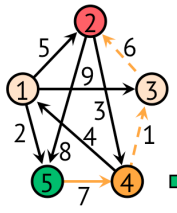
$D_{3\to 4} =$

0	5	9	8	2
∞	0	∞	3	8
∞	6	0	9	14
4	7	1	0	6
∞	∞	∞	7	0

$P_{3\to 4} =$

∞	1	1	2	1
∞	∞	∞	2	2
∞	3	∞	2	2
4	3	4	∞	1
∞	∞	∞	5	∞

Beispiel



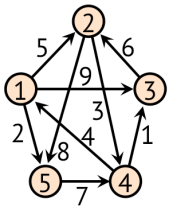
$D_{3\to 4}$ =

0	5	9	8	2
∞	0	∞	3	8
∞	6	0	9	14
4	7	1	0	6
∞	14	∞	7	0

$P_{3\to 4}$ =

∞	1	1	2	1
∞	∞	∞	2	2
∞	3	∞	2	2
4	3	4	∞	1
∞	3	∞	5	∞

Beispiel



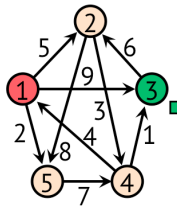
D_5 =

0	5	9	8	2
7	0	4	3	8
13	6	0	9	14
4	7	1	0	6
11	14	8	7	0

P_5 =

∞	1	1	2	1
4	∞	4	2	2
4	3	∞	2	2
4	3	4	∞	1
4	3	4	5	∞

Beispiel



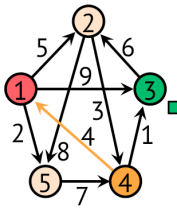
$D^{(1)} =$

0	5	9	8	2
7	0	4	3	8
13	6	0	9	14
4	7	1	0	6
11	14	8	7	0

$P^{(1)} =$

∞	1	1	2	1
4	∞	4	2	2
4	3	∞	2	2
4	3	4	∞	1
4	3	4	5	∞

Beispiel



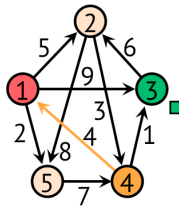
$D^{(1)} =$

0	5	9	8	2
7	0	4	3	8
13	6	0	9	14
4	7	1	0	6
11	14	8	7	0

$P^{(1)} =$

∞	1	1	2	1
4	∞	4	2	2
4	3	∞	2	2
4	3	4	∞	1
4	3	4	5	∞

Beispiel



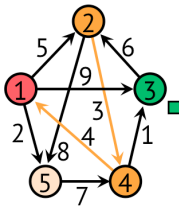
$D_5 =$

0	5	9	8	2
7	0	4	3	8
13	6	0	9	14
4	7	1	0	6
11	14	8	7	0

$P_5 =$

∞	1	1	2	1
4	∞	4	2	2
4	3	∞	2	2
4	3	4	∞	1
4	3	4	5	∞

Beispiel



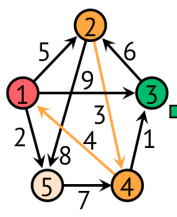
$D_5 =$

0	5	9	8	2
7	0	4	3	8
13	6	0	9	14
4	7	1	0	6
11	14	8	7	0

$P_5 =$

∞	1	1	2	1
4	∞	4	2	2
4	3	∞	2	2
4	3	4	∞	1
4	3	4	5	∞

Beispiel



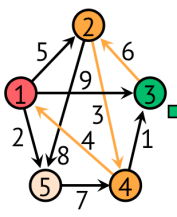
$D_5 =$

0	5	9	8	2
7	0	4	3	8
13	6	0	9	14
4	7	1	0	6
11	14	8	7	0

$P_5 =$

∞	1	1	2	1
4	∞	4	2	2
4	3	∞	2	2
4	3	4	∞	1
4	3	4	5	∞

Beispiel



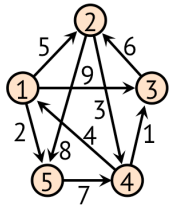
$D_5 =$

0	5	9	8	2
7	0	4	3	8
13	6	0	9	14
4	7	1	0	6
11	14	8	7	0

$P_5 =$

∞	1	1	2	1
4	∞	4	2	2
4	3	∞	2	2
4	3	4	∞	1
4	3	4	5	∞

Beispiel



$D_5 =$

0	5	9	8	2
7	0	4	3	8
13	6	0	9	14
4	7	1	0	6
11	14	8	7	0

$P_5 =$

∞	1	1	2	1
4	∞	4	2	2
4	3	∞	2	2
4	3	4	∞	1
4	3	4	5	∞

Floyd-Warshall Algorithmus

- FloydWarshall(W)

$O(n^2)$
- $D_0 \leftarrow W$
- $P_0 \leftarrow \ldots$

$O(n^3)$
- for k ← 1 to n do
  - for i ← 1 to n do
    - for j ← 1 to n do
      - $d_{k,ij} \leftarrow \min(d_{k-1,ij}, d_{k-1,ik}+d_{k-1,kj})$
      - $p_{k,ij} \leftarrow \ldots$

return D_n

Floyd-Warshall Algorithmus

- Fazit

  Der Floyd-Warshall Algorithmus hat einen Aufwand von

  $O(V^3)$

Man hätte auch Dijkstra auf jeden Knoten anwenden können. Ein Vergleich mit dem Aufwand für Dijkstra ergibt:

Dijkstra O(E log V) <-> Floyd-Warshall O(V^3).

Das heißt, ob die |V|-fache Anwedung von Dijkstra schneller ist, hängt von der Anzahl der Kanten im Graph ab.
E kann bis zu V^2 viele Kanten enthalten. In dem Falle hätten wir einen Aufwand von O(V^3 log V).

2.6.4 Kürzeste Pfade

Transitive Hülle

- Transitive Hülle

  Es sei G=(V,E) mit V={1,…,n} ein Graph. Die transitive Hülle von G ist definiert als

  $$G^*=(V,E^*)$$

  mit

  E^* = { (i,j): es ex. Pfad von i nach j in G }

E* enthält Kanten für alle Knotenpaare, zwischen denen ein Pfad existiert.

Transitive Hülle

- Lösung

  Ordne jeder Kante in E das Gewicht 1 zu und führe den Floyd-Warshall Algorithmus aus. Gibt es einen Weg von i nach j, so erhalten wir d[i,j] < n, andernfalls d[i,j] = ∞.

Transitive Hülle

- In der Praxis schneller

  Ersetze min und + im Floyd-Warshall Algorithmus durch die logischen Operatoren $\vee$ (oder) und $\wedge$ (und).

Man kann reelle Arithmetik durch Boolsche Arithmetik ersetzen, weil man hier nicht an den Gewichten, sondern nur an der Existenz eines Pfades interessiert ist.

Transitive Hülle

- Definition

  Es sei $t_k[i,j] = 1$ falls ein Pfad von i nach j in P_k existiert. Insbesondere ist also $t_n[i,j] = 1$, falls $(i,j) \in E^*$.

Transitive Hülle

- Rekursionsformel

$$t_0[i,j] = \begin{cases} 0 & \text{falls } i \neq j \text{ und } (i,j) \notin E \\ 1 & \text{sonst} \end{cases}$$

$$t_k[i,j] = t_{k-1}[i,j] \vee (t_{k-1}[i,k] \wedge t_{k-1}[k,j])$$

Transitive Hülle

- TransitiveClosure(G)

```
for i ← 1 to n do
    for j ← 1 to n do
        t0[i,j] ← (i = j) ∨ (i,j) ∈ E
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            tk[i,j] = tk-1[i,j] ∨ (tk-1[i,k] ∧ tk-1[k,j])
return Tn
```

Transitive Hülle

- Fazit

  Der Algorithmus zur Berechnung der transitiven Hülle hat den gleichen Aufwand, wie der Floyd-Warshall Algorithmus, nämlich

  $$O(V^3)$$

  In der Praxis können jedoch logische Operatoren schneller bearbeitet werden als arithmetische Operatoren.