Shortest Paths Algorithms

Heike Ripphausen-Lipa
Beuth University of Applied Science
Berlin

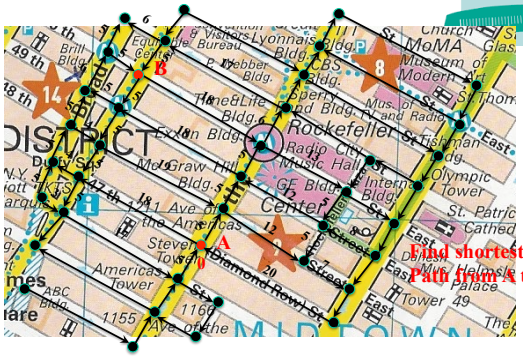
Learning Target

- The students get familiar with the different kind of shortest paths problems
- They get in-sight into the structure of shortest paths which gives the basic idea for several algorithms
- They learn which algorithm can be efficiently applied for which kind of shortest paths problems

Overview

- Introduction
- Dijkstra's Algorithm
- Bellman-Ford Algorithm
- Floyd-Warshall Algorithm

Problem: Find shortest Path



Introduction

Let us consider directed weighted graphs:

- Graph `G = (V,E)` with
- Weight function `w: E -> R` (weights of edges are real-valued)

Definition: Weight of a Path

The ***weight w of a path p*** with

$p = (v_0, v_1, \ldots, v_k)$ and $(v_i, v_{i+1}) \in E$

is defined as:

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

Definition: shortest-path weight

The ***shortest-path weight*** $\delta(u,v)$ between two vertices u and v is defined as

$$\delta(u,v) = \begin{cases} \min\{w(p) : p \text{ is a path in } G \text{ from } u \text{ to } v\}, & \text{if path exists} \\ \infty, & \text{otherwise} \end{cases}$$

Basic Idea for Determining Shortest Paths

The following fact is used in many shortest path algorithms:

„Subpaths of shortest paths are themselves shortest paths“

More precisely:
let $p = (v_0, v_1, \ldots, v_k)$ be a shortest path between v_0 and v_k;
then for i and j with $0 <= i <= j <= k$ the path $p_{ij} = (v_i, v_{i+1}, \ldots, v_j)$ is a shortest path from vertex v_i to vertex v_j.

Basic Idea for Determining Shortest Paths

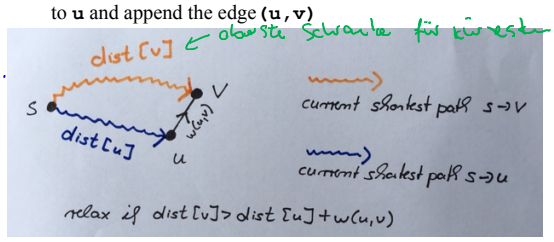
First we concentrate on the ***single-source shortest-paths problem***, that is, the shortests paths with the same start vertex s to all vertices $v \in V$ are determined

Many of these shortest-paths algorithms use the technique ***relaxation***:
→ ① oberste Schranke für kürzesten Weg dyn.
→ ② kürzesten Weg immer kürzer machen

For every vertex v there is an attribute **dist**, which gives an upper bound for the shortest-path-distance from a vertex s (the source) to vertex v.

Relaxation

The method ***relaxation*** of an edge (u,v) consists in the examination, if the so far determined shortest path to v can be improved by taking the current shortest path from s to u and append the edge (u,v)



Initialize-Single-Source

```
// Initialization relaxation algorithm
// for a single-source shortest paths problem
// with start vertex s
Initialize-Single-Source(G, s)
  for (v ∈ V)
     dist[v] = MAXDOUBLE;
     pred[v] = null;

  dist[s] = 0;
```


Relax(u,v,w)

```
Relax(u,v,w)
  // (u,v) edge, w weight function
  if (dist[v] > dist[u] + w(u,v))
     dist[v] = dist[u] + w(u,v);
     pred[v] = u;
```

Overview

- Introduction
- Dijkstra 's Algorithm
- Bellman-Ford Algorithm
- Floyd-Warshall Algorithm

Dijkstra 's Algorithm

Dijkstra 's Algorithm solves the single-source shortest-paths problem with start vertex **s** (source)

Dijkstra-Algorithmus only works for positive edge weights

It works similar to the Breadth-First-Search; however, instead of queue it uses a priority queue

wie Breitensuche: zuerst alle Nachbarn des wsh Knoten, dann Knoten mit nächster geringster Entfernung von dort

Dijkstra 's Algorithm

Dikstra's algorithm maintains a (virtual) vertex set **S**, which contains all vertices, where the shortest paths distance to **s** is already determined;

Step by step the vertex **u** from **V – S** with minimum **dist**-value is taken and all edges **(u,v)** incident to **u** are relaxed

For managing the set **V – S**, those vertices where the shortest path distance is not yet determined, a priority queue **Q** is used.

Dijkstra 's Algorithm

Graph (Klasse vorgegeben) Startknoten

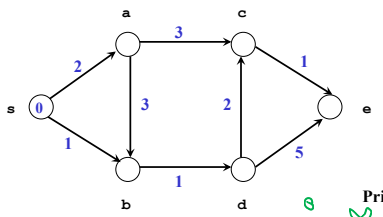
```
Dijkstra(G, w, s)
  // Determines for G with weight function w
  // the shortest paths starting with s
   Initialize-Single-Source(G, s);
   S = ∅; // all vertices reached from s
   Q = V(G);  // initialization: all vertices in Q

   while (Q != ∅)
     u = Extract-Min(Q); // Minimum w.r.t. dist
     add u to S;
     for (v ∈ Adj(u))
       Relax(u,v,w);
```

Java API Klasse

Priority Queue nicht in Q

Notice: Vertex set S is not really needed; it is only introduced for a better illustration of the algorithm!

Example Dijkstra 's Algorithm



i

Dist-value i;

No entry means MAXDOUBLE

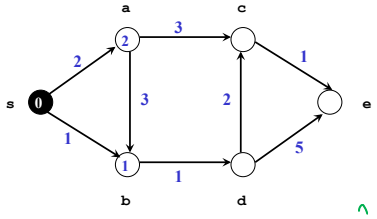
Priority Queue Q

s	a	b	c	d	e
0	∞	∞	∞	∞	∞

Notice : Q is ordered by increasing dist-values !

mit aufsteigendem Dist-Wert

Example Dijkstra 's Algorithm

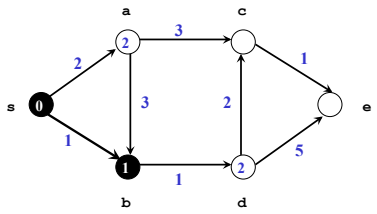


Priority Queue Q

b	a	c	d	e
1	2	∞	∞	∞

Vertices in S are colored black

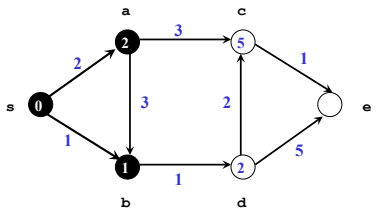
Example Dijkstra ‘s Algorithmus



Priority Queue Q

a	d	c	e

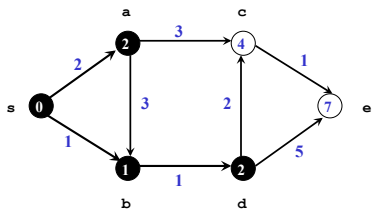
Example Dijkstra ‘s Algorithm



Priority Queue Q

d	c	e

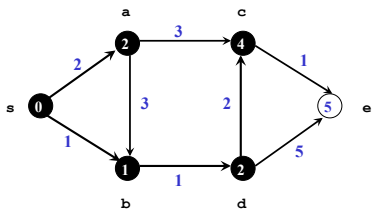
Example Dijkstra ‘s Algorithm



Priority Queue Q

c	e

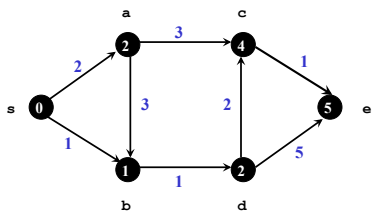
Example Dijkstra ‘s Algorithm



Priority Queue Q

e

Example Dijkstra ‘s Algorithm



Priority Queue Q

Executing Dijkstra in a Table

Initialization

vertex	s	a	b	c	d	e
pred	null	null	null	null	null	null
dist	0	∞	∞	∞	∞	∞

After removing s from Q

vertex	s	a	b	c	d	e
pred	null	s	s	null	null	null
dist	0	2	1	∞	∞	∞

Executing Dijkstra in a Table

After removing b from Q

vertex	s	a	b	c	d	e
pred	null	s	s	null	b	null
dist	0	2	1	∞	2	∞

After removing a from Q

vertex	s	a	b	c	d	e
pred	null	s	s	a	b	null
dist	0	2	1	5	2	∞

Executing Dijkstra in a Table

After removing d from Q

vertex	s	a	b	c	d	e
pred	null	s	s	d	b	d
dist	0	2	1	4	2	7

After removing c from Q

vertex	s	a	b	c	d	e
pred	null	s	s	d	b	c
dist	0	2	1	4	2	5

Executing Dijkstra in a Table

After removing e from Q

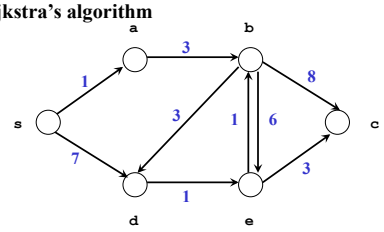
vertex	s	a	b	c	d	e
pred	null	s	s	d	b	c
dist	0	2	1	4	2	5

Determining shortest path from s to e:

e ← c ← d ← b ← s

Exercise: Dijkstra's Algorithm

Determine all shortest paths starting with vertex s with Dijkstra's algorithm



Dijkstra's Algorithm

The correctness of the algorithm is based on the fact „Subpaths of shortest paths are themselves shortest paths"

For a formal proof see: [CLRS]

Running Time Dijkstra's Algorithm

```
Initialize-Single-Source(G, s);

S = Ø;
Q = V(G);

while (Q != Ø)
   u = Extract-Min(Q);
   S = S.add(u);
   for (v e Adj(u))
       Relax(u,v,w);
```

Total Running Time

O(n)

O(1)
O(n) Extract-Min in Priority Queue
O(n)

O(n * log n)
O(n)

O(m)

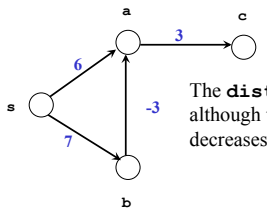
O(n* log n + m) = O(n²)

Exercise: Dijkstra ‘s negative weights

Exercise:

Give an example which shows that Dijkstra‘s algorithm does not work for negative edge weights!

Dijkstra’s Problem with Negative Weights



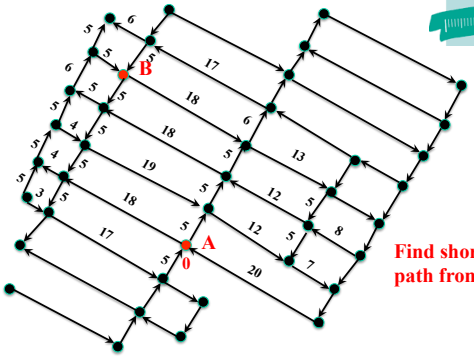
The **dist**-value of **c** remains 9 although the **dist**-value of **a** decreases from 6 to 4

Problem Dijkstra’s Algorithm for finding s-t-Paths

In the case that only one certain s-t-path should be found, Dijkstra’s algorithm is not very target-oriented;

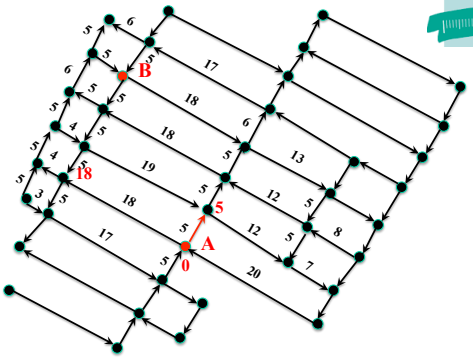
Consider the following example

Dijksta‘s Algorithm

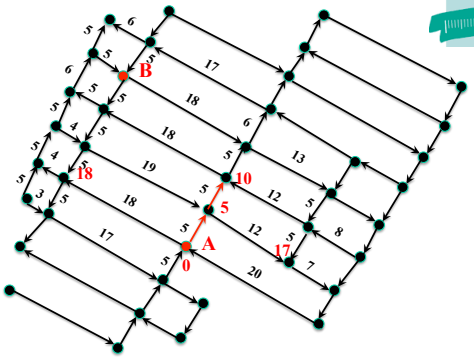


Find shortest path from A to B

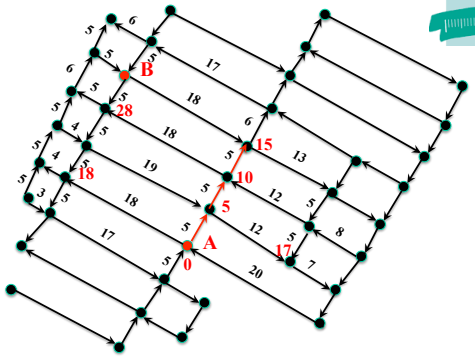
Dijkstra‘s Algorithm



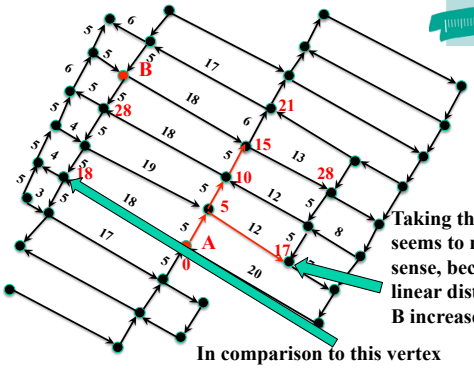
Dijkstra‘s Algorithm



Dijkstra‘s Algorithm



Dijkstra‘s Algorithm



A* Algorithm

The A* Algorithm is an algorithm for finding a path between two vertices s and t which tries to improve the just shown disadvantage of Dikstra’s Algorithm:

For this purpose the A* algorithm needs a funktion h_t which estimates for every vertex u the distance to the target vertex t;

The algorithm only works if $h_t(u) \leq$ the real distance between u and t

For the problem above the $h_t(u)$ could be chosen as the straight-line distance between u and t

A* Algorithm

The A* Algorithm is similar to Dijkstra’s algorithm with the following exception:

The priority queue is sorted with respect to f with

$f(u) = dist[u] + h_t(u)$

Overview

- Introduction
- Dijkstra ‘s Algorithm
- Bellman-Ford Algorithm
- Floyd-Warshall Algorithm

Bellman-Ford Algorithm

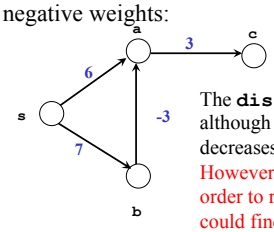
Now we consider the Bellman-Ford Algorithm which solves the single-source shortest-paths problem, where negative edge weights are allowed → löst Problem bei Graphen mit negativen Kantengewichten

However if the graph contains a cycle with negativ weight, no solution exists → gibt "keine Lösung" an wenn es Kanten mit negativen Kantengewichten gibt

The Bellman-Ford Algorithm also uses the method relaxation

Dijkstra's Problem with Negative Weights

Consider the problem of Dijkstra's algorithm with negative weights:



The **dist**-value of **c** remains 9 although the **dist**-value of **a** decreases from 6 to 4;
However, if one would use **a** again in order to relax it's neighbours, one could find the correct solution

Basic Idea of Bellman-Ford Algorithm

This leads to the basic idea of the Bellman-Ford Algorithm;

repeat
relax all edges
until there is no change of a **dist**-value

Bellman-Ford Algorithm

In the first phase the shortest paths are determined, if a solution exists

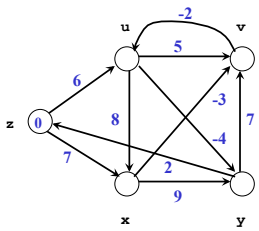
In the second phase the algorithm determines, if there are negative cycles;
In this case the algorithm returns the value **false**, otherwise the value **true**;

Bellman-Ford Algorithm

```
Bellman-Ford(G, w, s)
  // Determines for G with weight function w
  // the shortest paths starting with s
  // returns true, iff a solution exists

  Initialize-Single-Source(G, s);
  // Determine shortest paths
  for (i=1; i < n; i++)
      for (e = (u,v) e E)
            Relax(u,v,w);

  // Examine if there are negative cycles
  for (e = (u,v) e E)
      if (dist[v] > dist[u] + w(u,v))
            return false
  return true;
```

Bellman-Ford Algorithm

Find shortest paths starting with z



i

dist-value i;

No entry means MAXDOUBLE

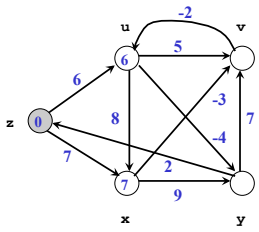
Assumption: the inner for-statement traverses the edges in lexicographical order : (u, v), (u,x), (u,y), (v,u), …

Bellman-Ford Algorithm

First execution of inner **for**-statement:

Only edges incident to **z** improve the corresponding **dist**-values.

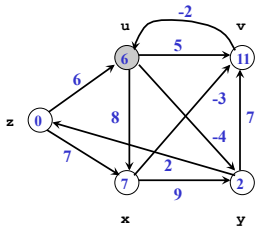
Bellman-Ford Algorithm



Bellman-Ford Algorithm

Second execution of inner **`for`**-statement:

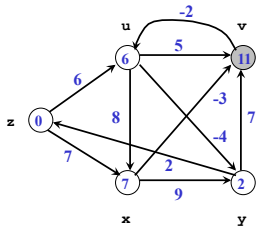
Bellman-Ford Algorithm



Relaxation with edge (u,v);

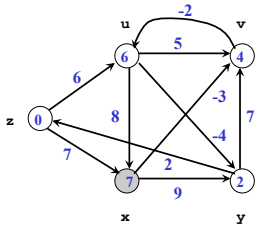
dist-value of v decreases from MAXDOUBLE to 11

Bellman-Ford Algorithm



Edges incident to v don‘t improve the dist-value

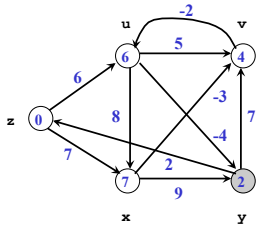
Bellman-Ford-Algorithm



Relaxation with edge (x,v)

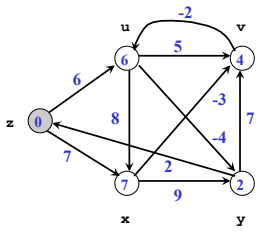
dist-value of v decreases from 11 to 4

Bellman-Ford Algorithm



Edges incident to y don‘t improve the dist-value

Bellman-Ford Algorithm

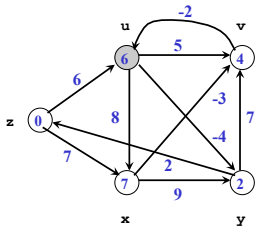


Edges incident to z don‘t improve the dist-value

Bellman-Ford Algorithm

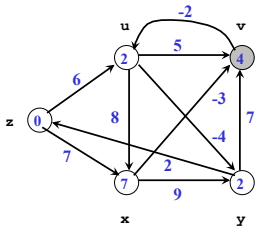
Third execution of inner **`for`**-statement:

Bellman-Ford Algorithm



Edges incident to u don‘t improve the dist-value

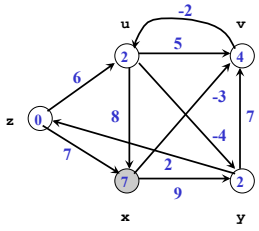
Bellman-Ford Algorithm



Relaxation with edge (v,u);

dist-value of u decreases from 6 to 2

Bellman-Ford Algorithm

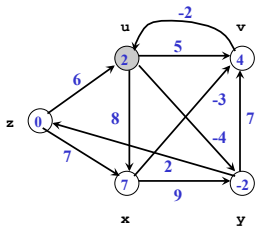


Edges incident to x, y and z don‘t improve the dist-values

Bellman-Ford Algorithm

Fourth execution of inner **`for`**-statement:

Bellman-Ford Algorithm



Relaxation with edge (u,y):

dist-value of y decreases from 2 to -2

Bellman-Ford Algorithm

All subsequent steps do not improve the current solution

Executing Bellman-Ford in a Table

Initialization

vertex	u	v	x	y	z
pred	null	null	null	null	null
dist	∞	∞	∞	∞	0

After first execution of inner for

vertex	u	v	x	y	z
pred	z	null	z	null	null
dist	6	∞	7	∞	0

Executing Bellman-Ford in a Table

After second execution of inner for

vertex	u	v	x	y	z
pred	z	x	z	u	null
dist	6	4	7	2	0

After third execution of inner for

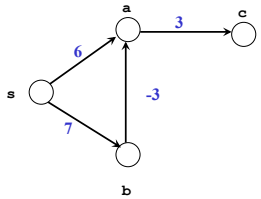
vertex	u	v	x	y	z
pred	v	x	z	u	null
dist	2	4	7	2	0

Executing Bellman-Ford in a Table

After fourth execution of inner for

vertex	u	v	x	y	z
pred	v	x	z	u	null
dist	2	4	7	-2	0

Exercise: Bellman-Ford-Algorithm



Exercise: Bellman-Ford Algorithm

Initialization

vertex	a	b	c	s
pred				
dist				

After first execution of inner for

vertex	a	b	c	s
pred				
dist				

Exercise: Bellman-Ford Algorithm

After second execution of inner for

vertex	a	b	c	s
pred				
dist				

After third execution of inner for

vertex	a	b	c	s
pred				
dist				

Running Time Bellman-Ford Algorithm

```
Bellman-Ford(G, w, s)
  Initialize-Single-Source(G, s);                    O(n)
  // Determine shortest paths
  for (i=1; i < n; i++)                              O(n * m)
      for (e = (u,v) e E)                            O(m)
            Relax(u,v,w);

  // Examine if there are negative cycles
  for (e = (u,v) e E)                                O(m)
      if dist[v] > dist[u] + w(u,v);
            return false
      else
            return true;
```


Running Time Bellman-Ford Algorithm

The total running time of the Bellman-Ford-Algorithm is **`O(n * m)`**

Overview

- Introduction
- Dijkstra ‘s Algorithm
- Bellman-Ford Algorithm
- Floyd-Warshall Algorithm

All-Pair Shortest-Paths

If one wants to find for all vertex pairs the shortest paths, the algorithm described before could be applied for all vertices as start vertex

However the Floyd-Warshall Algorithm is more efficient for this ***all-pair shortest-paths problem***

Floyd-Warshall Algorithm

Input:
A directed graph $G = (V, E)$ with vertex set $V = \{1, \ldots, n\}$ and weights $w(e)$ for all $e \in E$

Output:
A $n \times n$ –matrix $DIST = (dist_{ij})$, such that for $i \neq j$ $dist_{ij}$ is the length of a shortest path from i to j and $dist_{ii}$ is the length of a shortest cycle which contains i.
A $n \times n$ matrix $PRED = (pred_{ij})$, such that $pred_{ij}$ is the next to the last vertex of a shortest (i,j) – path respectively (i,i) –cycle

Floyd-Warshall Algorithm

```
Floyd-Warshall-Algorithm(G, w)
  // Algorithm for determining all-pair shortest paths
  Initialize DIST with MAXDOUBLE, PRED with null;

  // Determination of paths without inner vertices
  for (i = 1; i <= n; i++)
      for (j = 1; j <= n; j++)
          if ((i,j)e E)
                dist_ij = w_ij; // w_ij = w(i,j)
                pred_ij = i;
```

zwischen erster & letztem Knoten kann weiterer Knoten - also mit einer Kante

Floyd-Warshall Algorithm

```
// Continuation Floyd-Warshall Algorithm(G, w)
  // Try to shorten paths with inner vertex k

  for (k = 1; k <= n; k++)
      for (i = 1; i <= n; i++)
          for (j = 1; j <= n; j++)
              if (dist_ij > dist_ik + dist_kj)
              dist_ij = dist_ik + dist_kj;
              pred_ij = pred_kj;
```

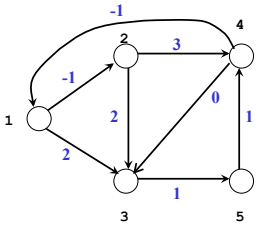

Floyd-Warshall Algorithm

If during the algorithm there is a i with $dist_{ii} < 0$ the algorithm can stop, since there is a cycle with negative weight and thus there exists no solution

Floyd-Warshall Algorithmus

Remark:
The calculation formula of the Floyd-Warshall Algorithm corresponds to the calculation formula of matrix multiplication

Floyd-Warshall Algorithm



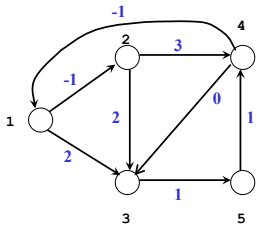
DIST⁰

	-1	2		
		2	3	
				1
-1		0		
			1	

DIST¹

	-1	2		
		2	3	
				1
-1	-2	0		
			1	

Floyd-Warshall Algorithm



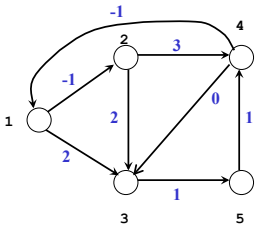
DIST¹

	-1	2		
		2	3	
				1
-1	-2	0		
			1	

DIST²

	-1	1	2	
		2	3	
				1
-1	-2	0	1	
			1	

Floyd-Warshall Algorithm



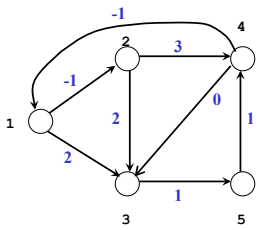
DIST²

	-1	1	2	
		2	3	
				1
-1	-2	0	1	
			1	

DIST³

	-1	1	2	2
		2	3	3
				1
-1	-2	0	1	1
			1	

Floyd-Warshall Algorithm



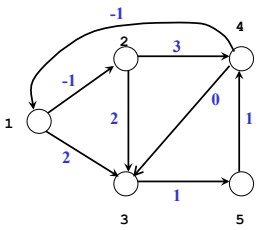
DIST³

	-1	1	2	2
		2	3	3
				1
-1	-2	0	1	1
			1	

DIST⁴

1	-1	1	2	2
2	1	2	3	3
				1
-1	-2	0	1	1
0	-1	1	1	2

Floyd-Warshall Algorithm



DIST⁴

1	-1	1	2	2
2	1	2	3	3
				1
-1	-2	0	1	1
0	-1	1	1	2

DIST⁵

1	-1	1	2	2
2	1	2	3	3
1	0	2	2	1
-1	-2	0	1	1
0	-1	1	1	2

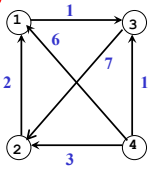
Floyd-Warshall Algorithmus

Notice:

In the former shown example the PRED-matrix is not involved

Klausuraufgabe

Exercise: Floyd-Warshall Algorithm



DIST⁰

	1	2	3	4
1				
2				
3				
4				

PRED⁰

	1	2	3	4
1				
2				
3				
4				

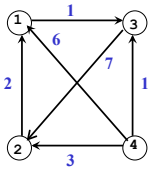
DIST¹

	1	2	3	4
1				
2				
3				
4				

PRED¹

	1	2	3	4
1				
2				
3				
4				

Exercise: Floyd-Warshall Algorithm



$DIST^1$	1	2	3	4
1				
2				
3				
4				

$PRED^1$	1	2	3	4
1				
2				
3				
4				

$DIST^2$	1	2	3	4
1				
2				
3				
4				

$PRED^2$	1	2	3	4
1				
2				
3				
4				

Running Time Floyd-Warshall Algorithm

Running time is dominated by the second part

```
Floyd-Warshall-Algorithm(G, w)
// Determination of paths without inner vertices

for (k = 1; k <= n; k++)
  for (i = 1; i <= n; i++)
      for (j = 1; j <= n; j++)
            if (dist_ij > dist_ik + dist_kj)
                  dist_ij = dist_ik + dist_kj;
                  p_ij = p_kj;
```

O(n) O(n²) O(n³)

Correctness Floyd-Warshall Algorithm

Correctness can be shown by complete induction

Let $\mathtt{Dist}^k$ be the Dist-matrix after the **k**-th execution of the outer for statement

Show:

For every entry in this matrix, $\mathtt{dist}^k_{ij}$ is the length of a shortest **(i,j)**-path only with inner vertices from **{1, …, k}**