

# ICPC Teilnehmervortrag: Graphenalgorithmen II

Markus Schneckenburger, Moritz Uehling,  
Florian Weber, Cora Weidner

KIT  
ICPC-Teilnehmervortrag

28.05.15

- 1 Minimum Spanning Tree (MST)
  - Problem
  - Lösung: Kruskal
- 2 SSSP (Single Source Shortest Path)
- 3 SSSP mit negativen Kanten
- 4 All Pairs Shortest Paths (APSP)
  - Idee
  - Code
  - Beurteilung
- 5 Zusammenfassung

Den kompletten Code (inklusive dem der Folien) findet ihr unter:  
<https://github.com/Florianjw/ICPC-Graphen>

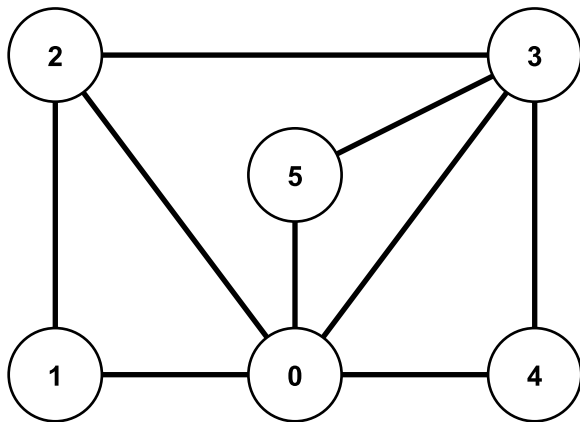
## Minimum Spanning Tree (MST)

# Minimum Spanning Tree (MST)

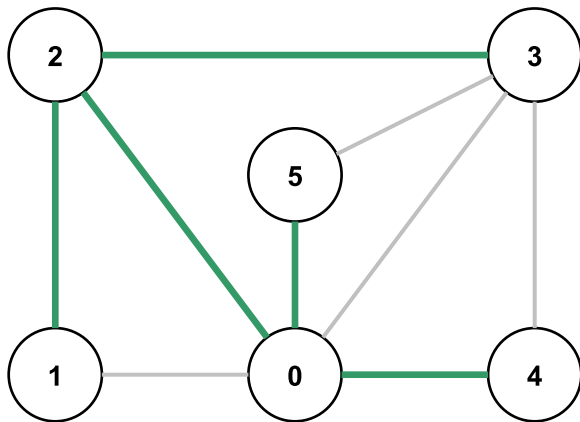
## Problemstellung

- „finde das billigste Netzwerk“
- genau:  
Gegeben sei ein zusammenhängender ungerichteter gewichteter Graph, gesucht ist ein Spannbaum mit geringstem Gesamtgewicht.

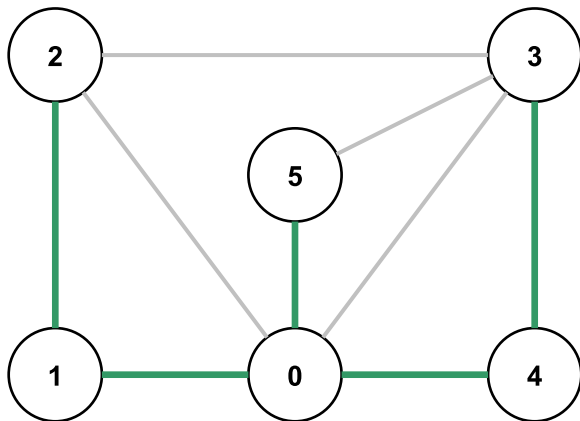
# Minimum Spanning Tree (MST)



# Minimum Spanning Tree (MST)

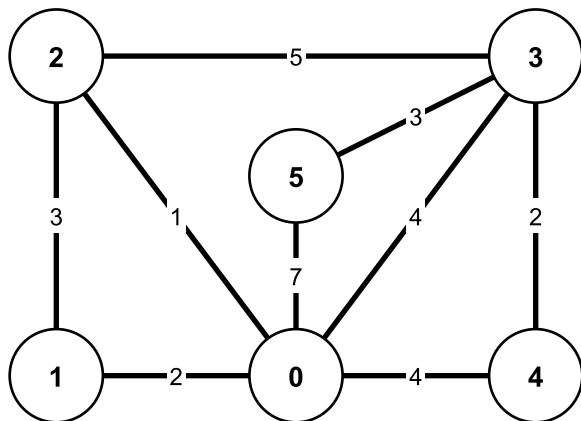


# Minimum Spanning Tree (MST)

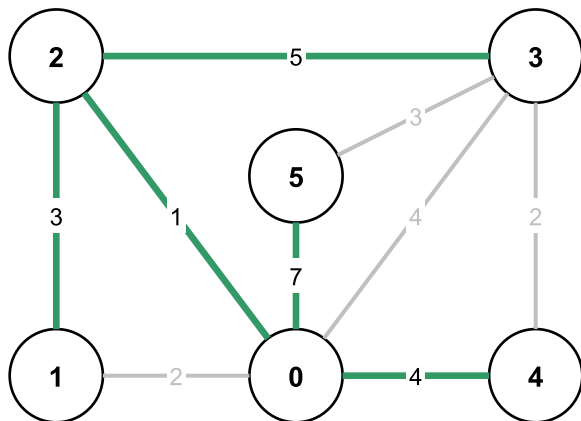




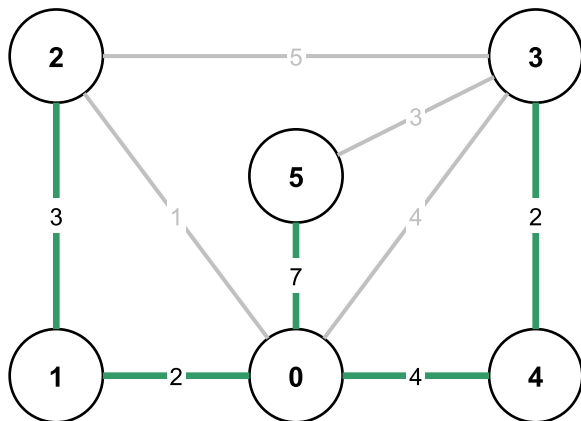
# Minimum Spanning Tree (MST)



# Minimum Spanning Tree (MST)



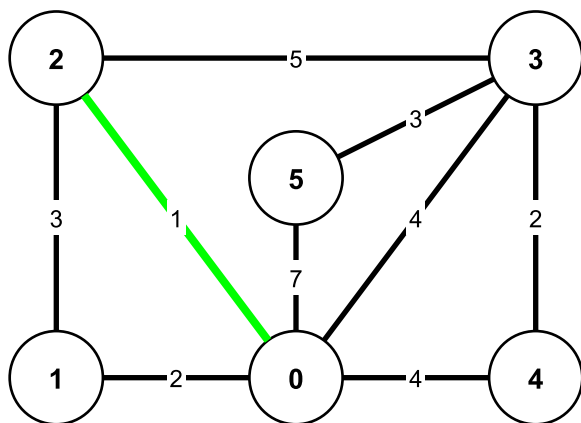
# Minimum Spanning Tree (MST)

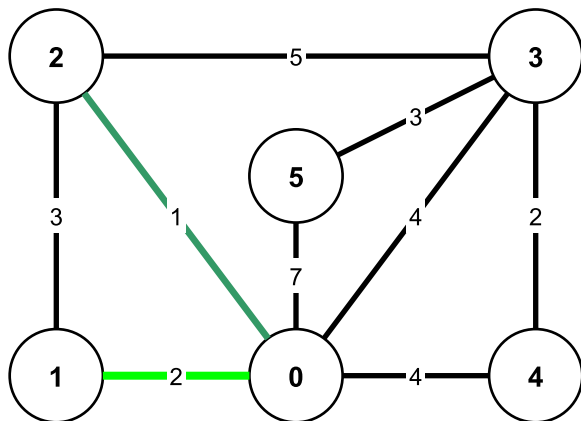


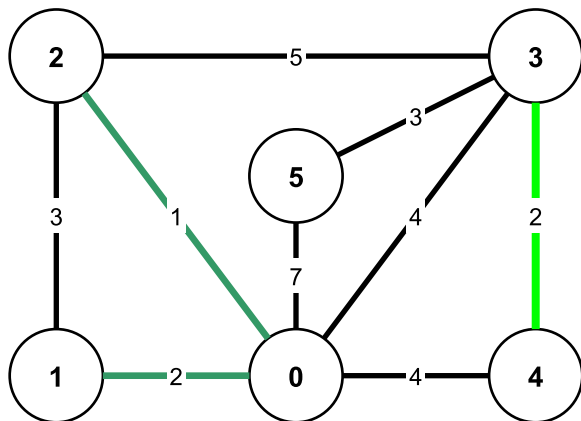
# Minimum Spanning Tree (MST)

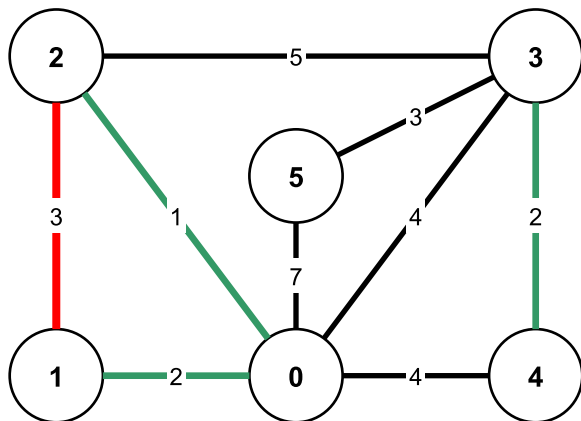
## Lösung

- Ansatz: baue einen Baum mit greedy Algorithmus:
  - 1 betrachte Kante mit niedrigstem Gewicht
  - 2 untersuche: führt hinzufügen der Kante zu einem Zyklus?
    - Ja: verwirfe Kante
    - Nein: füge Kante zum Baum hinzu
  - 3 starte bei 1. mit restlichen Kanten bis alle abgearbeitet sind
  - 4  $\implies$  Baum ist ein MST

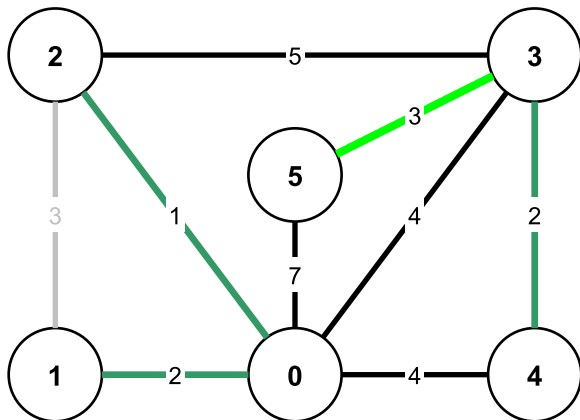


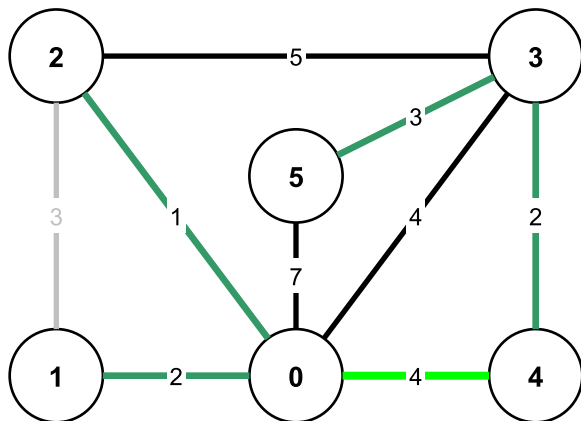


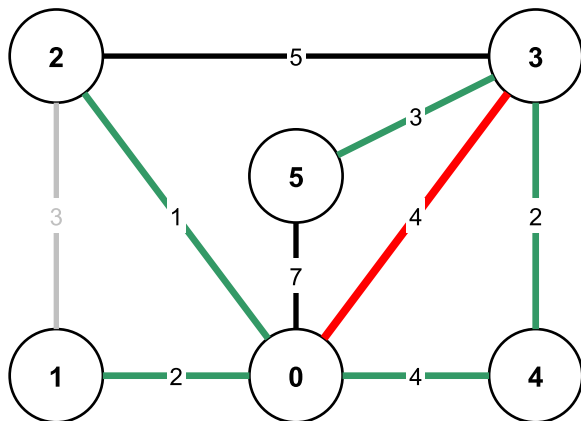


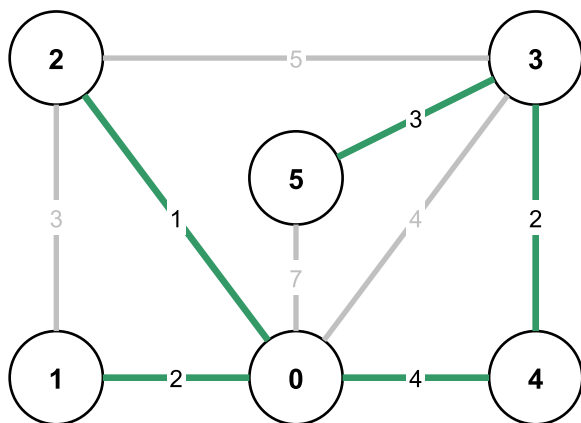












## Implementierung - Algorithmus von Kruskal

- sortiere Kanten nach Gewicht
- benutze Union-Find um Zyklen zu detektieren

# Kruskal

```
int kruskal(std::vector<edge>& edges, int maxnode) {  
    int fullweight = 0;  
    UnionFind ufind(maxnode + 1);  
    std::sort(edges.begin(), edges.end());  
    for (const auto& e : edges) {  
        if (!ufind.sameSet(e.from, e.to)) {  
            ufind.unify(e.from, e.to);  
            fullweight += e.weight;  
        }  
    }  
    return fullweight;  
}
```

Für Laufzeiten:  $n := |V|$ ,  $m := |E|$

## Laufzeit

$$\begin{aligned} O(m \log(m) + m \cdot \alpha(n)) &= O(m \log(m)) \\ &= O(m \log(n^2)) = O(2m \log(n)) = O(m \log(n)) \end{aligned}$$

## Weitere lösbare Probleme

- Maximum Spanning Tree
- „MST“ finden, wenn Kanten vorgegeben sind
- Minimum Spanning Forest: mehrere getrennte Bäume
- Minimax-Problem:
  - Finden des Pfades zwischen zwei Knoten mit dem kleinsten Maximalgewicht einer Kante



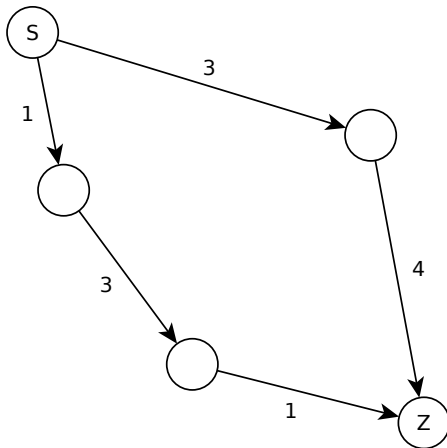
SSSP (Single Source Shortest Path)

# Das Problem

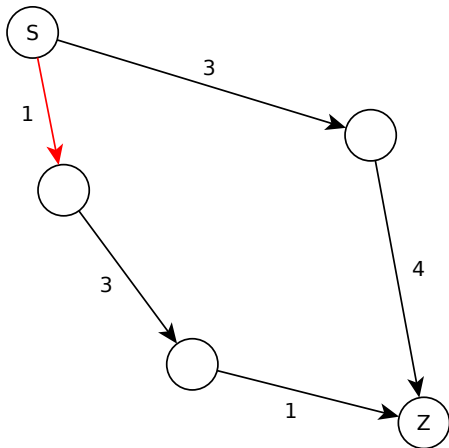
## Das Problem

Breitensuche schlägt bei gewichteten Graphen fehl.

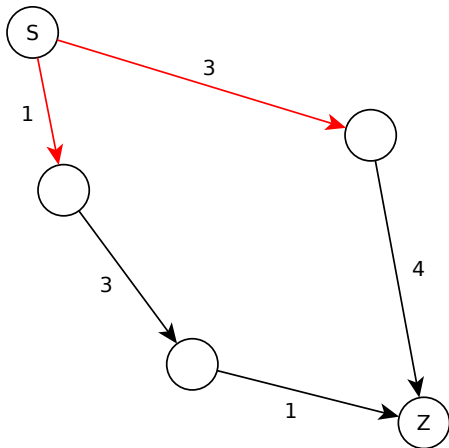
# Das Problem



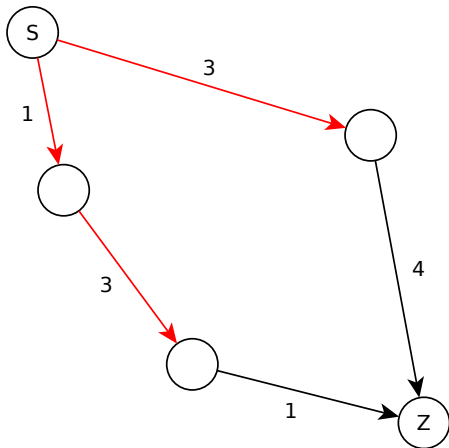
# Das Problem



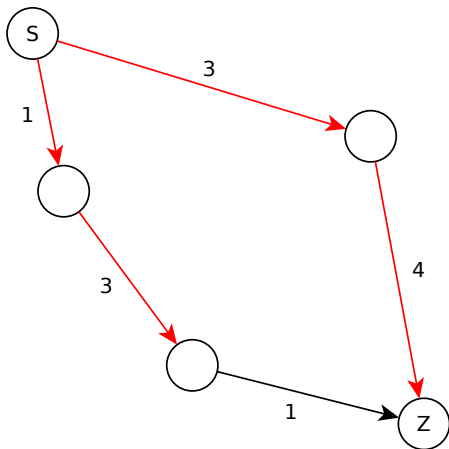
# Das Problem



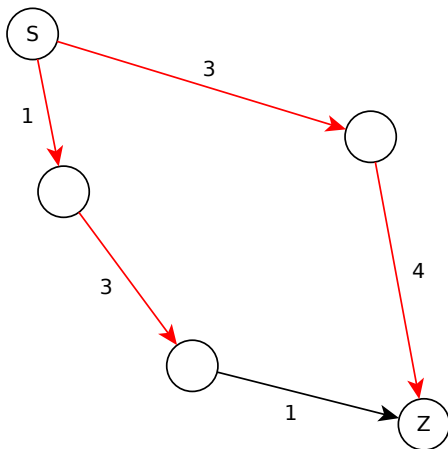
# Das Problem



# Das Problem



# Das Problem



## Problem

Es wird ein Weg der Länge 7 gefunden, obwohl 5 das Optimum ist



# Dijkstras Algorithmus

- Grundsätzliche Idee: Breitensuche mit Priority-Queue (so dass "nähere" Knoten zuerst behandelt werden)
- `std::priority_queue` verwendet binären Heap
- $\implies$  Laufzeit von Dijkstra ist  $\Theta((n + m) \log n)$
- Nachteil: Funktioniert nicht bei negativen Kantengewichten

## Header:

```
#include<vector>
#include<algorithm>
#include<queue> // not priority_queue!
#include<iostream>

using namespace std;

struct arrival_event {
    int to;
    int weight;
};

bool operator < (const arrival_event& e1, const arrival_event& e2) {
    // inversed
    return e1.weight > e2.weight;
}
```

# Code

```
vector<int> dijkstra(vector<vector<arrival_event>>& nodes, int startnode) {
    vector<int> distances (nodes.size(), 2000000000);

    priority_queue<arrival_event> todo;

    todo.push({startnode, 0});

    while(!todo.empty()) {
        auto current = todo.top();
        todo.pop();

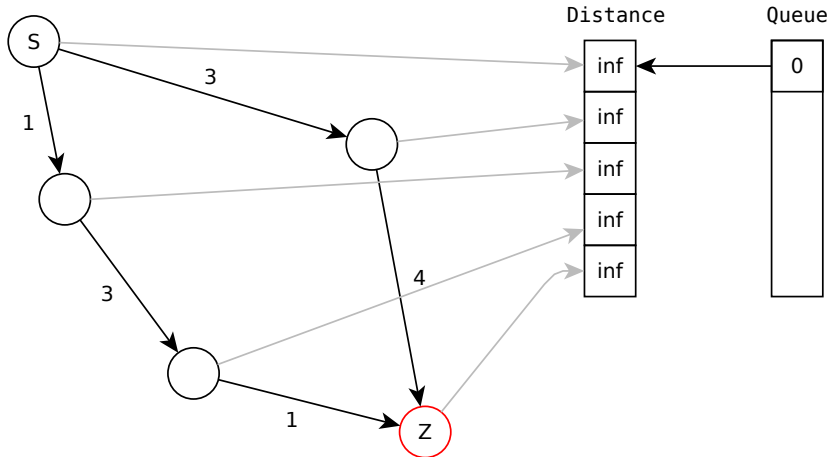
        if(current.weight < distances[current.to]) {
            distances[current.to] = current.weight;

            for(int i = 0; i < nodes[current.to].size(); i++) {
                arrival_event next = nodes[current.to][i];
                next.weight += current.weight;

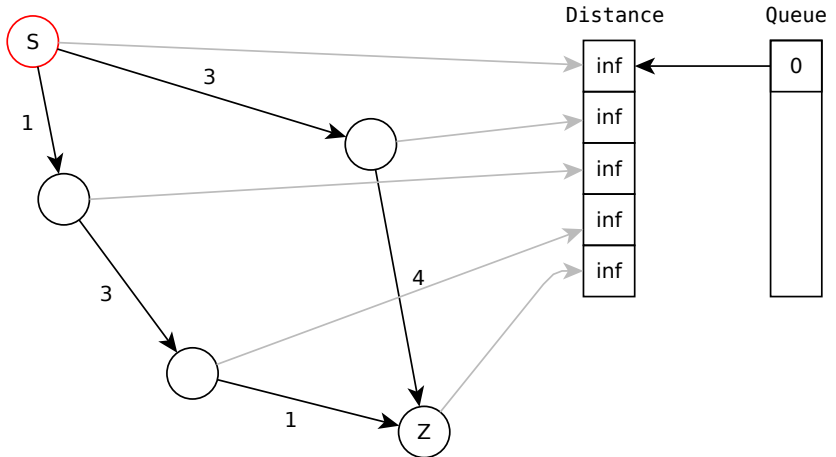
                todo.push(next);
            }
        }
    }

    return distances;
}
```

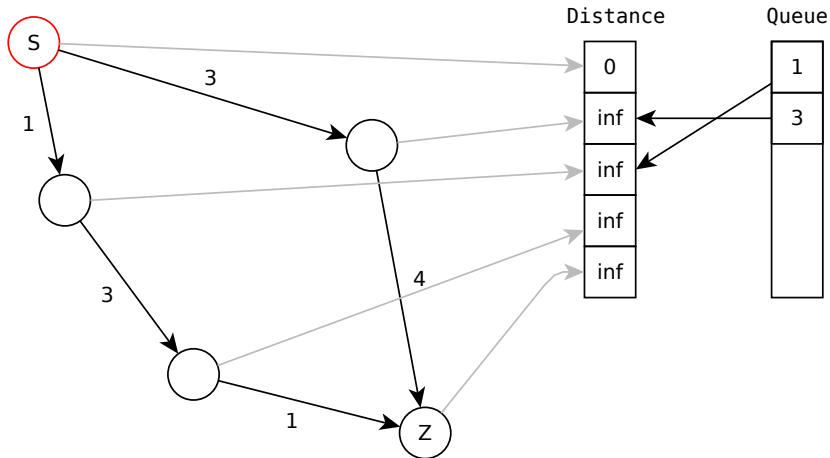
# Beispiel



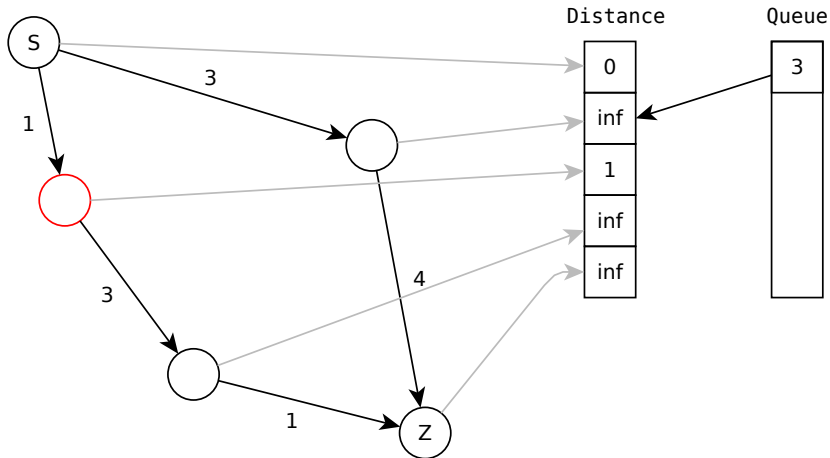
# Beispiel



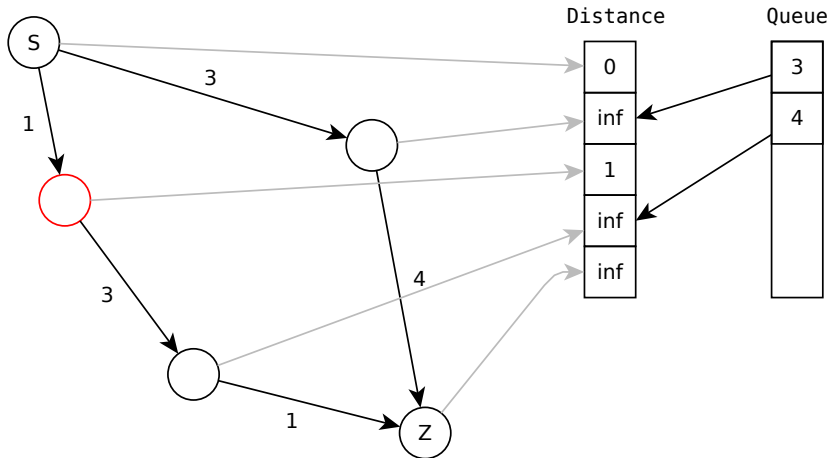
# Beispiel



# Beispiel

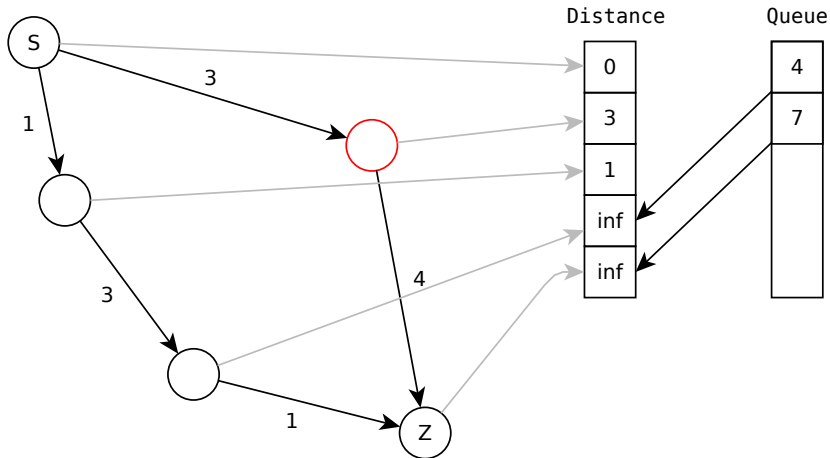


# Beispiel

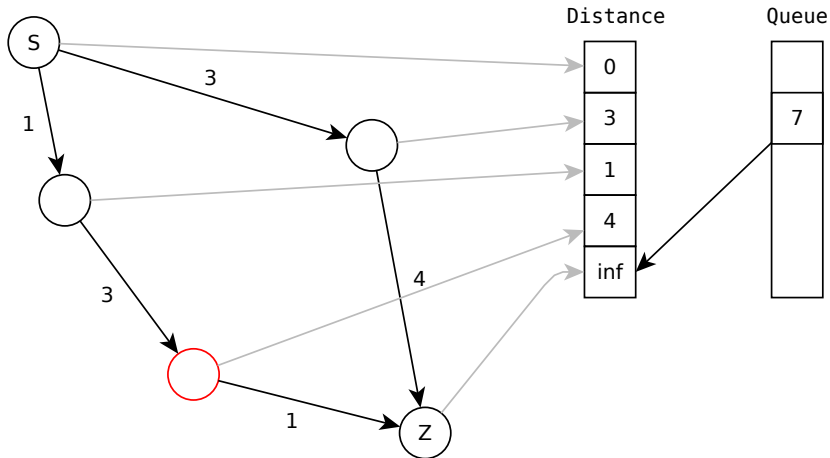




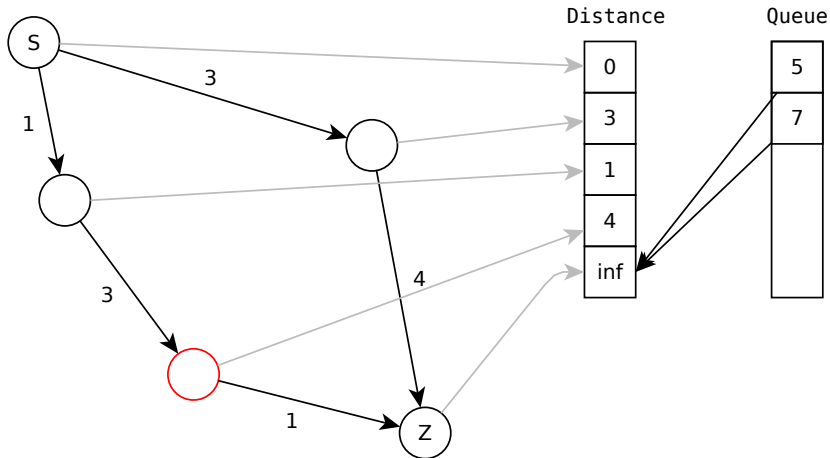
# Beispiel



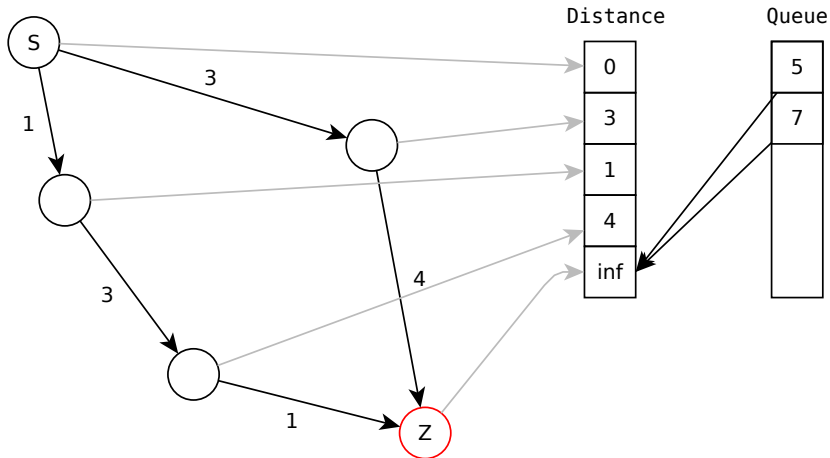
# Beispiel



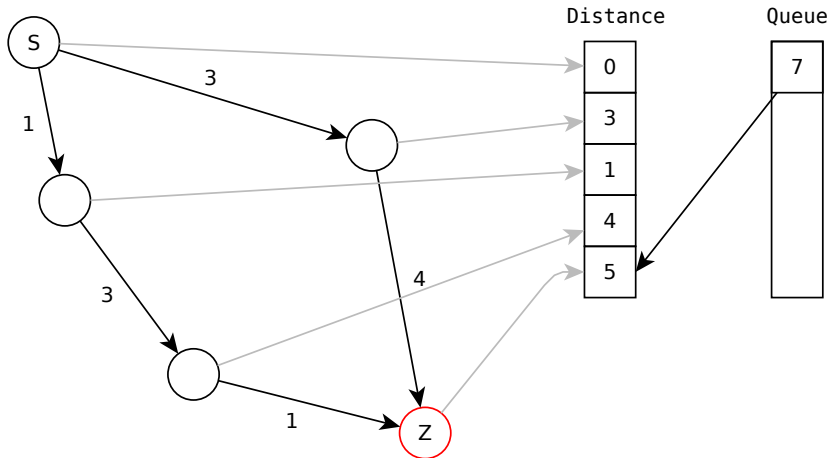
# Beispiel



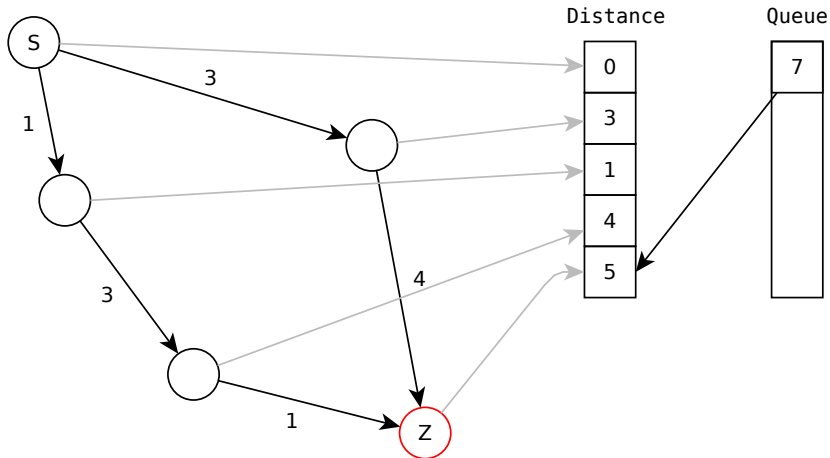
# Beispiel



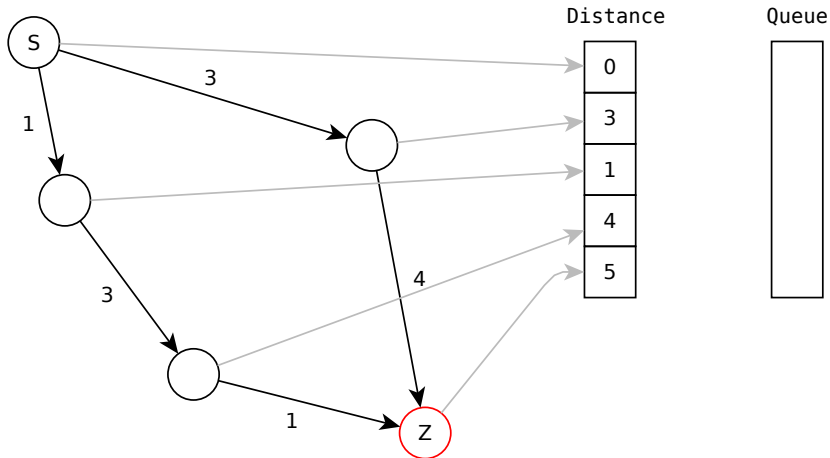
# Beispiel



# Beispiel

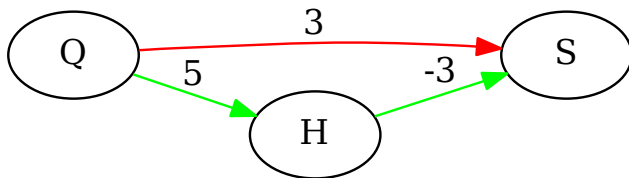


# Beispiel



SSSP mit negativen Kanten





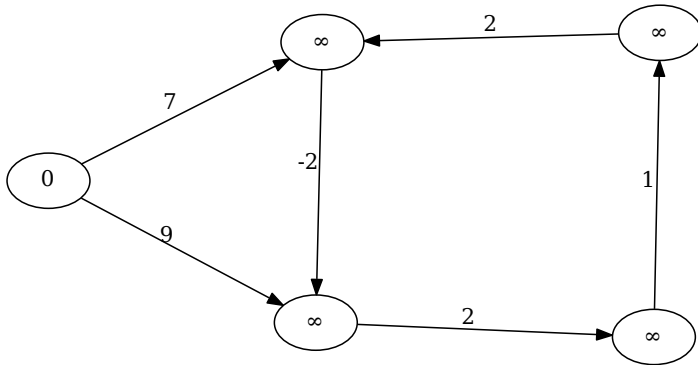
## Problem

Dijkstra kommt nicht mit negativen Kanten zurecht

- Lösung: rohe Rechenleistung
- Wichtige Einschränkung: negative Kreise auf irgendeinem Pfad von Q zu S bedeuten Nichtexistenz eines kürzesten Pfades
- Idee 1: vollständige Tiefensuche.
  - selbst für Brute-Force-Verhältnisse zu langsam (exponentielle Laufzeit)

- Idee 2:
  - kürzester Pfad enthält maximal  $|V| - 1$  Kanten
  - Enthalte der kürzeste Pfad  $i$  Kanten. Falls wir alle kürzesten Pfade mit bis zu  $i - 1$  Knoten kennen:
    - Zu den kürzesten Pfaden mit bis zu  $i$  Kanten fehlt höchstens eine Kante.
    - Probiere für alle Kanten, ob sie irgendwo einen kürzeren Pfad erzeugen
  - Für  $i = 0$  ist die Distanz der Quelle zu sich selbst 0, und die zu allen anderen Knoten inf
- Idee 2 ist offensichtlich vielversprechender, sie führt zum Algorithmus von Bellman und Ford.

# Initialisierung

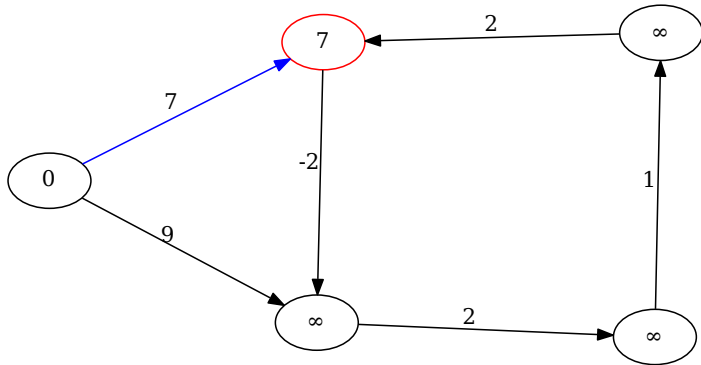


# Runde 1

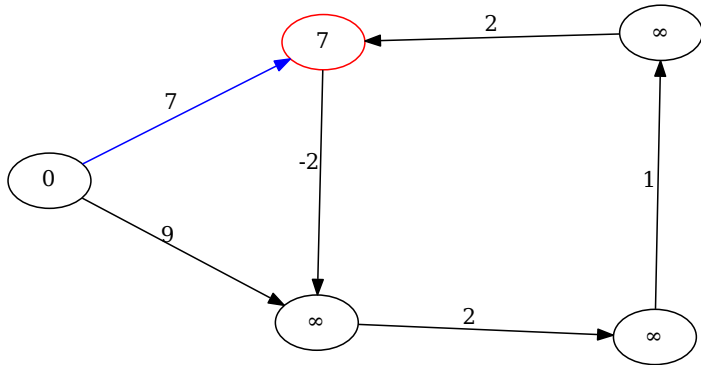
## Runde 1

Relaxiere in *beliebiger Reihenfolge* jede Kante einmal.

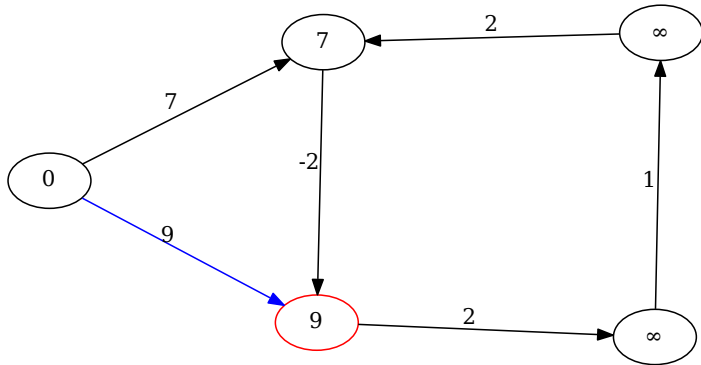
# Runde 1



# Runde 1

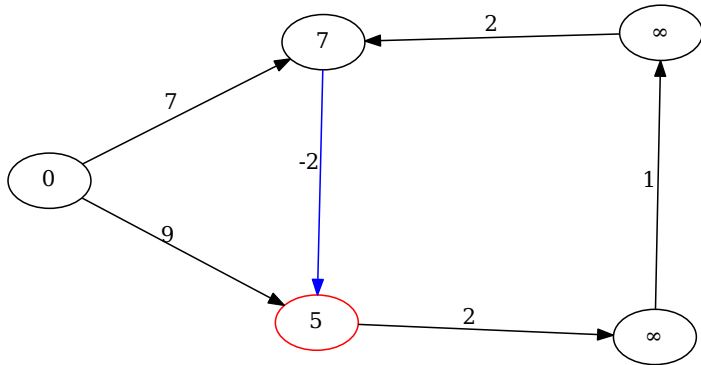


# Runde 1

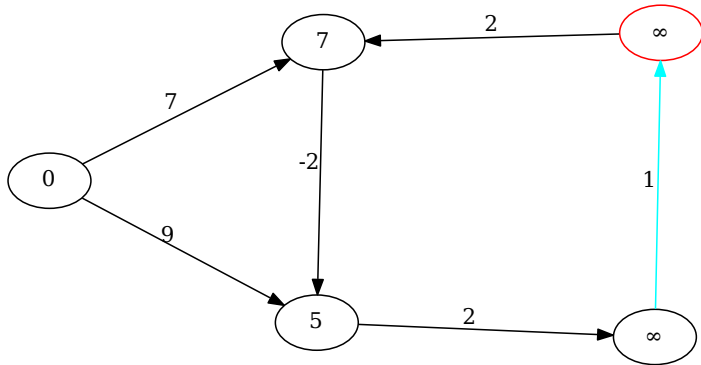




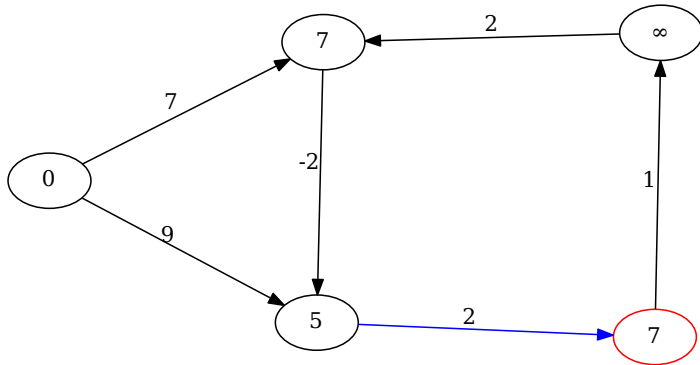
# Runde 1



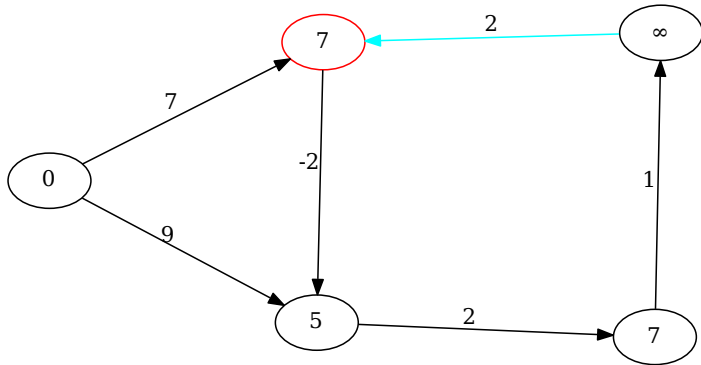
# Runde 1



# Runde 1



# Runde 1

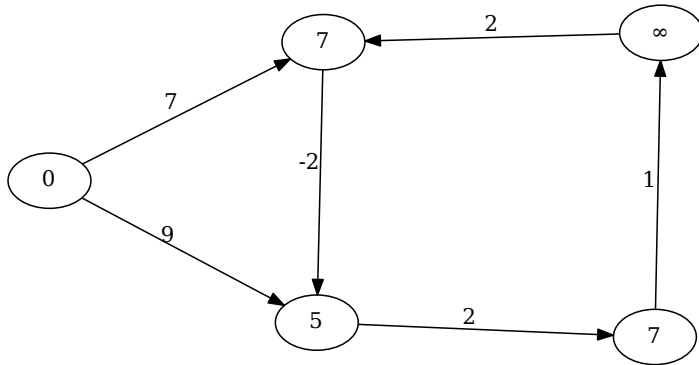


# Runde 2

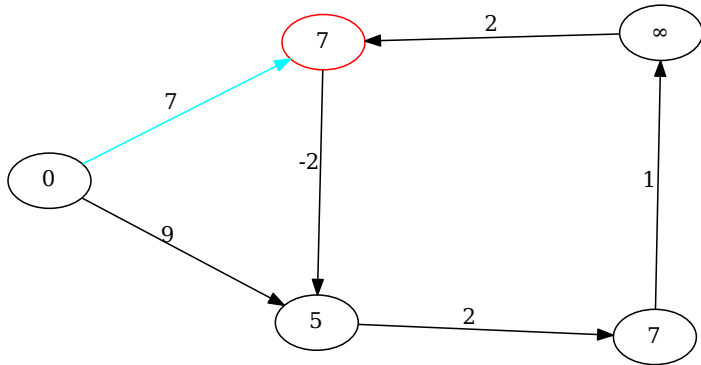
## Runde 2

Wiederhole, da es Änderungen gab.

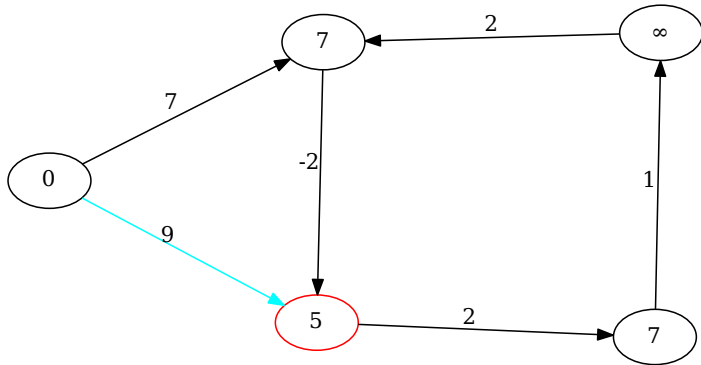
## Runde 2



## Runde 2

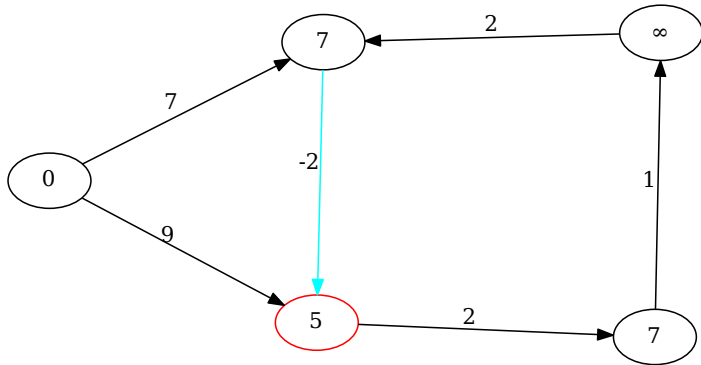


## Runde 2

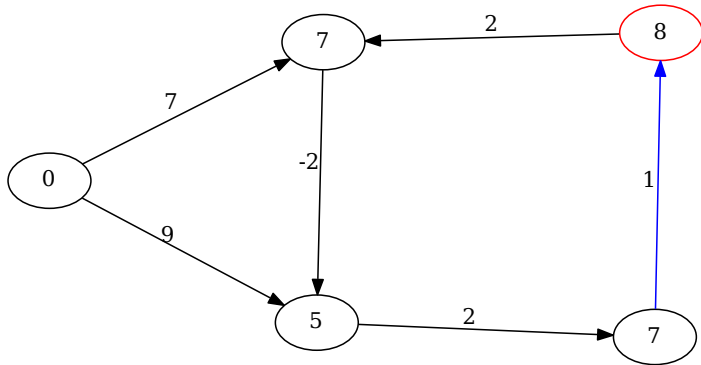




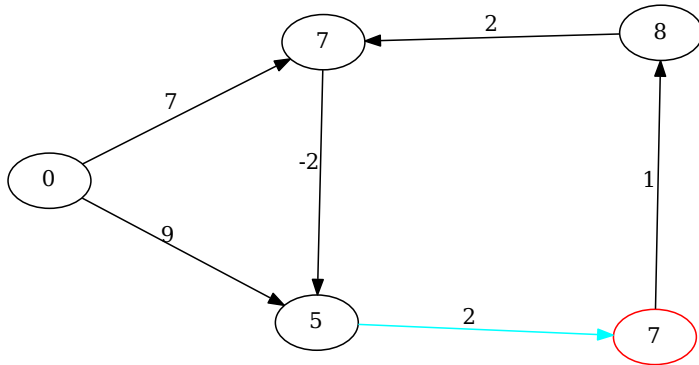
## Runde 2



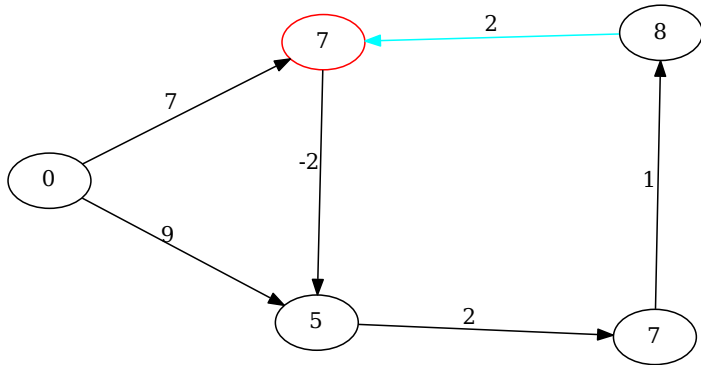
## Runde 2



## Runde 2



## Runde 2



# Runde 3

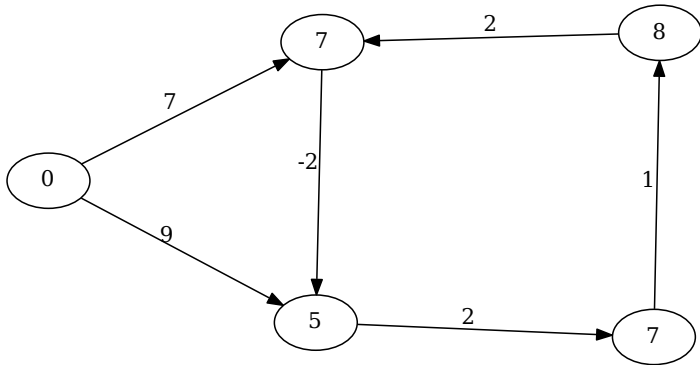
## Runde 3

Wiederhole, da es Änderungen gab.

## Keine Änderungen

In der dritten Runde finden keine Relaxierungen mehr statt → fertig

# Endergebnis



# Code

```
using node = std::size_t;
const auto infinity = std::numeric_limits<double>::infinity();

struct edge {
    node from;
    node to;
    double dist;
};

std::vector<dist> bellman_ford(const std::size_t node_count,
                             const std::vector<edge>& edges,
                             node source) {
    {
        std::vector<double> min_dists(node_count, infinity);
        min_dists[source] = 0;
        for (std::size_t i = 0; i < node_count + 1; ++i) {
            bool changes = false;
            for(const auto& e: edges) {
                const double old_dist = min_dists[e.to];
                const double new_dist = min_dists[e.from] + e.dist;
                if (new_dist < old_dist) {
                    min_dists[e.to] = new_dist;
                    changes = true;
                }
            }
            if (!changes) { break; }
            if (i == node_count) {
                throw std::runtime_error{"negative cycle"};
            }
        }
        return min_dists;
    }
}
```

- Negative Kreise lassen sich durch eine weitere Anwendung detektieren
- Die Entfernung von nicht über negative Kreise erreichbare Knoten wird immer korrekt ermittelt.
- In jedem negativen Kreis ändert sich im  $n + 1$ -ten Schritt mindestens eine Entfernung, die Detektion aller Knoten ohne Minimaldistanz ist somit leicht per Breitensuche möglich.



- Asymptotische Komplexität  $\in O(n \cdot m)$
- Profitiert nicht von kurzen Distanzen zwischen Quelle und Senke
- Relativ leicht zu implementieren

*Kann man schon so machen, meistens will man das aber nicht*

## All Pairs Shortest Paths (APSP)

## Problemstellung

Man hat einen Graphen gegeben, der gewichtet ist. Nun möchte man den kürzesten Pfad zwischen allen Knoten  $i$  zu allen Knoten  $j$  herausfinden.

## Lösungsansatz

Man verwendet den bereits bekannten SSSP-Algorithmus, und führe diesen nach bedarf aus, d.h. in diesem Fall  $n$ -mal.

## Laufzeit

$$n \cdot O(m + n \cdot \log(n))$$

$$= n \cdot O(n^2 + n \cdot \log(n))$$

$$= n \cdot O(n^2 + n^2)$$

$$= O(n^3)$$

$\implies$  geht es einfacher in etwa der selben Zeit?

## Lösungsansatz

Wir wissen: jeder Pfad zwischen zwei Knoten ist entweder bereits der kürzeste, oder es gibt einen Kürzeren Pfad als zwei Verknüpfung anderer Pfade über mindestens einen dritten Knoten.

## Genauer

Systematisch in einer Adjazenzmatrix  $A$ : Nehme für jeden Pfad  $A[i][j] = \min(A[i][j], A[i][k] + A[k][j])$ , d.h. entweder der Pfad ist bereits minimal, oder ein Pfad über Knoten  $k$  ist kürzer und wird als neues Minimum übernommen. Wenn man nun richtig iteriert, erhält man alle minimalen Pfade.

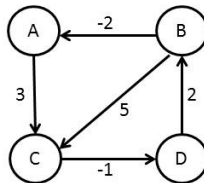
```
for (int k = 0; k < V; k++)  
    for (int i = 0; i < V; i++)  
        for (int j = 0; j < V; j++)  
            A[i][j] = min(  
                A[i][j],  
                A[i][k] + A[k][j]  
            );
```

⇒ der Aufwand liegt in  $O(n^3)$

# Beispiel - Urzustand

Anfang

ij	A	B	C	D
A	$\infty$	$\infty$	3	$\infty$
B	-2	$\infty$	5	$\infty$
C	$\infty$	$\infty$	$\infty$	-1
D	$\infty$	2	$\infty$	$\infty$





# Beispiel - Über Knoten A

$K = A$  (i über A nach j)

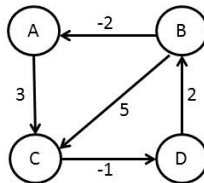
ij	A	B	C	D
A	$\infty$	$\infty$	3	$\infty$
B	-2	$\infty$	1	$\infty$
C	$\infty$	$\infty$	$\infty$	-1
D	$\infty$	2	$\infty$	$\infty$



# Beispiel - Über Knoten B

$K = B$  (i über B nach j)

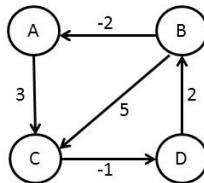
ij	A	B	C	D
A	$\infty$	$\infty$	3	$\infty$
B	-2	$\infty$	1	$\infty$
C	$\infty$	$\infty$	$\infty$	-1
D	0	2	3	$\infty$



# Beispiel - Über Knoten C

$K = C$  (i über C nach j)

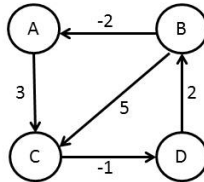
ij	A	B	C	D
A	$\infty$	$\infty$	3	2
B	-2	$\infty$	1	0
C	$\infty$	$\infty$	$\infty$	-1
D	0	2	3	2



# Beispiel - Über Knoten D (1)

(1)  $K = D$  (i über D nach j)

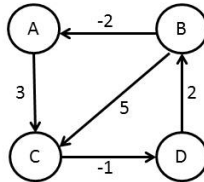
ij	A	B	C	D
A	2	$\infty$	3	2
B	-2	$\infty$	1	0
C	$\infty$	$\infty$	$\infty$	-1
D	0	2	3	2



# Beispiel - Über Knoten D (2)

(2)  $K = D$  (i über D nach j)

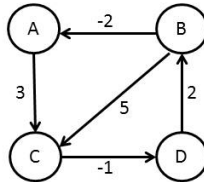
ij	A	B	C	D
A	2	4	3	2
B	-2	$\infty$	1	0
C	$\infty$	$\infty$	$\infty$	-1
D	0	2	3	2



# Beispiel - Über Knoten D (3)

(3)  $K = D$  (i über D nach j)

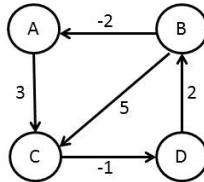
ij	A	B	C	D
A	2	4	3	2
B	-2	2	1	0
C	$\infty$	$\infty$	$\infty$	-1
D	0	2	3	2



# Beispiel - Über Knoten D (4)

(4)  $K = D$  (i über D nach j)

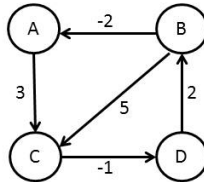
ij	A	B	C	D
A	2	4	3	2
B	-2	2	1	0
C	-1	$\infty$	$\infty$	-1
D	0	2	3	2



# Beispiel - Über Knoten D (5)

(5)  $K = D$  (i über D nach j)

ij	A	B	C	D
A	2	4	3	2
B	-2	2	1	0
C	-1	1	$\infty$	-1
D	0	2	3	2

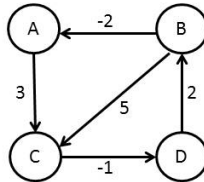




# Beispiel - Über Knoten D (6)

(6)  $K = D$  (i über D nach j)

ij	A	B	C	D
A	2	4	3	2
B	-2	2	1	0
C	-1	1	2	-1
D	0	2	3	2



- Auch für SSSP - Probleme anwendbar (wenn  $|V| < 400$ )
- Detektion von negativen oder günstigsten Zyklen möglich  
     $\implies$  ein negativer Zyklus existiert genau dann, wenn ein  
    Diagonaleintrag negativ ist
- Finden des Durchmessers eines Graphen (der längste der  
    kürzesten Pfade)
- Minimax, Maximin

- + Asymptotische Komplexität  $\in O(n^3)$  und mit Speicher  $\in O(n^2)$
- + Sehr leicht zu implementieren (Vierzeiler)
- + Für andere Probleme günstig anzuwenden, wenn  $|V| < 400$
- - Für andere Probleme **nur** günstig anzuwenden, wenn  $|V| < 400$
- $\implies$  Gut für das ursprüngliche Problem
- $\implies$  Auch nützlich für andere Probleme, solange  $|V| < 400$

# Zusammenfassung

# Zusammenfassung

Kriterium	Dijkstra	Bellman Ford	Floyd Warshall
Laufzeit	$O((n + m) \log(n))$	$O(n \cdot m)$	$O(n^3)$
Max. Größe	$n, m \leq 300K$	$n \cdot m \leq 10M$	$n \leq 400$
Ungewichtet	Ja	Ja (schlecht)	Ja (kleine Graphen)
Gewichtet	Ja	Ok	Ja (kleine Graphen)
Neg. Kanten	Nein	Ja	Ja
Neg. Zyklen	Nein	Ja	Ja
Kleine Graphen	Overkill	Overkill	Bestes
Kann SSSP?	Ja	Ja	Eingeschränkt

Tabelle: Übersicht

Erinnerung:  $n := |V|$  und  $m := |E|$