ICPC Teilnehmervortrag: Graphenalgorithmen II

Markus Schneckenburger, Moritz Uehling, Florian Weber, Cora Weidner

KIT
ICPC-Teilnehmervortrag

28.05.15

Übersicht

Code und mehr

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

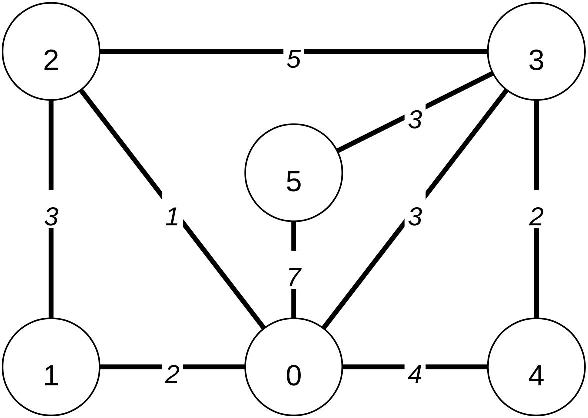
- „finde das billigste Netzwerk“
- genau:
  Gegeben sei ein zusammenhängender ungerichteter gewichteter Graph, gesucht ist ein zusammenhgängender Teilgraph mit geringstem Gesamtgewicht.

Minimum Spanning Tree (MST)

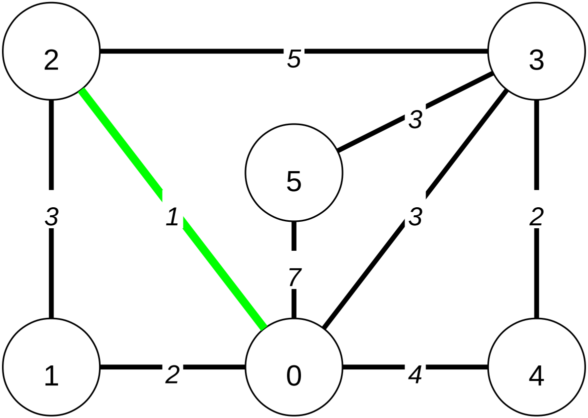
Lösung

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

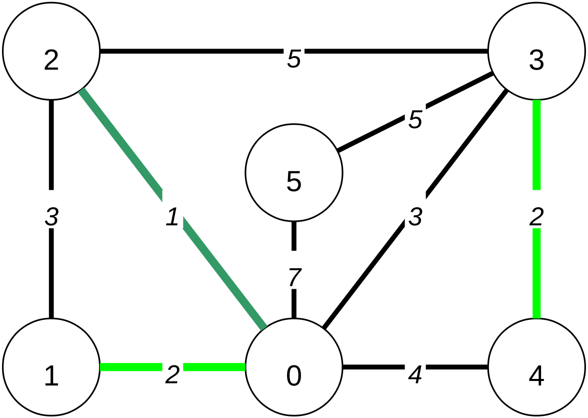
Lösung



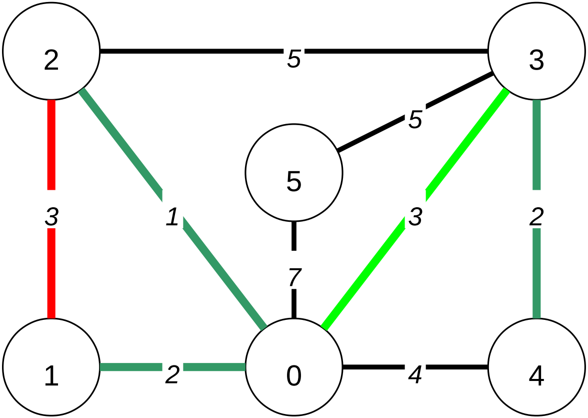
Lösung



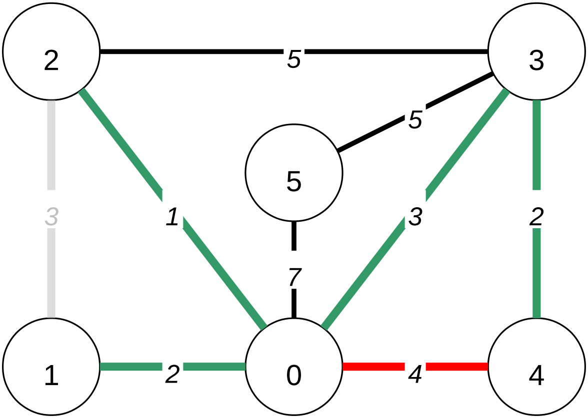
Lösung



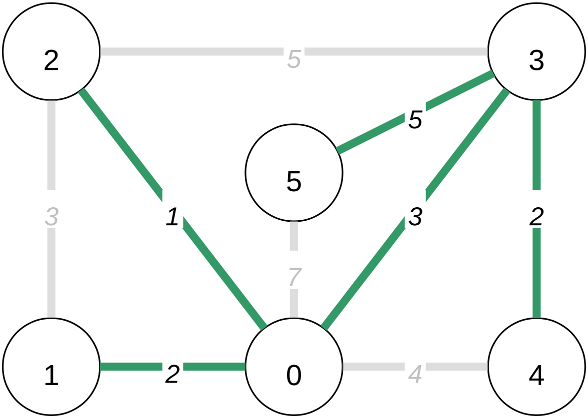
Lösung



Lösung



Lösung



Kruskal

Implementierung - Algorithmus von Kruskal

- sortiere Kanten nach Gewicht
- benutze Union-Find um Zyklen zu detektieren
- Laufzeit:
  $O(E\log(E) + E \cdot \alpha) = O(E\log(E)) = O(E\log(V^2)) = O(2E\log(V)) = O(E\log(V))$

Kruskal

```cpp
double kruskal(std::vector<edge>& edges, int maxnode) {
    double 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;
}
```

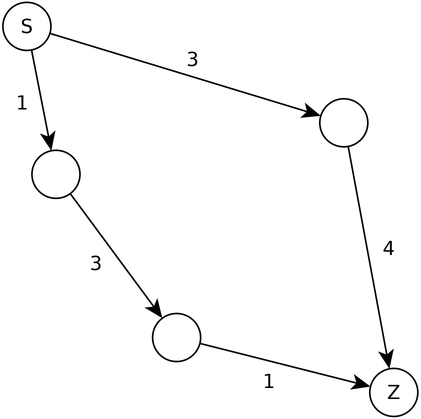
Weitere lösbare Probleme

- "MST" finden, wenn Kanten vorgegeben sind
- Minimum Spanning Forest: mehrere getrennte Bäume
- min-max Problem

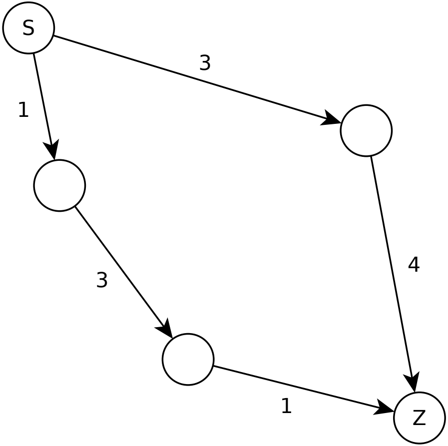
SSSP (Single Source Shortest Path)

Das Problem

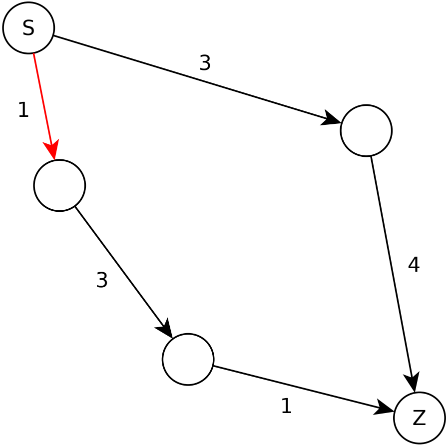
Breitensuche schlägt bei gewichteten Graphen fehl:



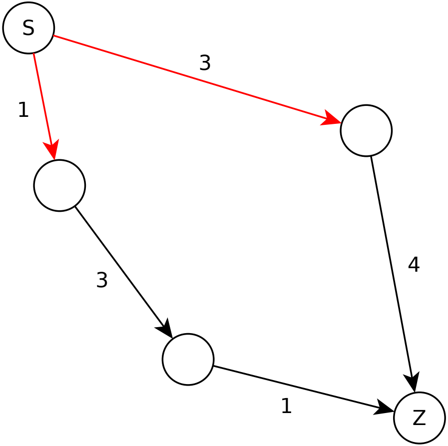
Das Problem



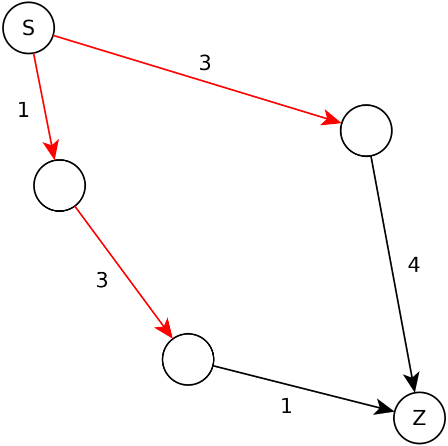
Das Problem



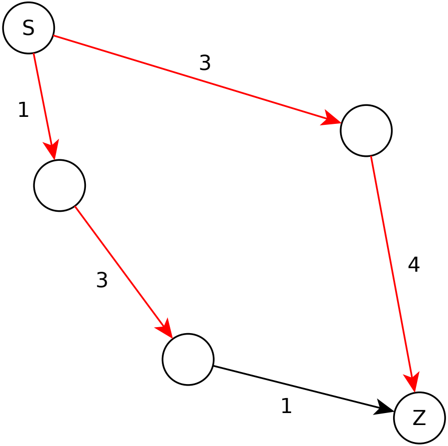
Das Problem



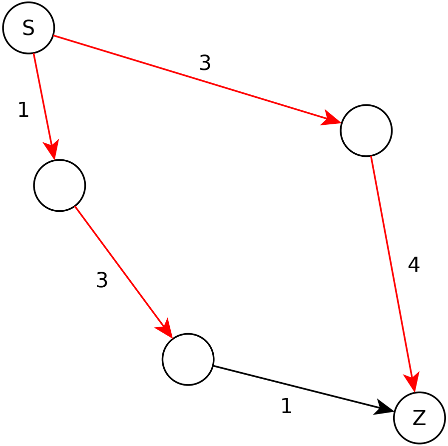
Das Problem



Das Problem



Das Problem



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

Djikstras Algorithmus

- Grundsätzliche Idee: Breitensuche mit Priortiy-Queue (so dass "nähere" Knoten zuerst behandelt werden)
- Dynamic Programming um doppeltes Untersuchen von Knoten zu vermeiden.
- std :: priority_queue verwendet binären Heap
- $\implies$ Laufzeit von Dijkstra ist $\Theta((|E| + |V|) \log |V|)$
- Theoretisch mit Fibonacci-Heap bessere Laufzeit, aber
  - Praktisch langsamer für ICPC-Problemgrößen
  - Und dann auch nur bei sehr dichten Graphen
  - Nicht in der C++-Standardbibliothek

Code

Header:

```
#include<vector>
#include<algorithm>
#include<queue>
#include<iostream>

using namespace std;

struct edge {
    size_t to;
    double weight;
};

bool operator < (const edge& e1, const edge& e2) {
    // inversed, because priority_queue returns biggest element
    return e1.weight > e2.weight;
}

using node = vector<edge>;
```

Code

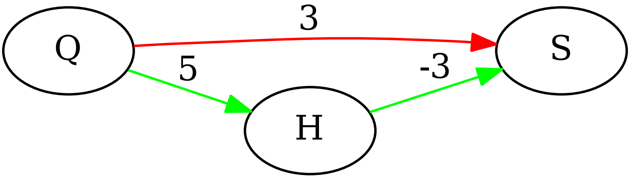
```
vector<double> dijkstra(vector<node>& nodes, size_t startnode) {
  // initialize all distances with infinity
  vector<double> distances (nodes.size(), 100000000000);

  priority_queue<edge> 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(size_t i = 0; i < nodes[current.to].size(); i++) {
        edge next = nodes[current.to][i];
        next.weight += current.weight;

        todo.push(next);
      }
    }
  }

  return distances;
}
```

Code

```cpp
int dijkstra_to_target(vector<node>& nodes, size_t startnode, size_t target) {
  vector<double> distances (nodes.size(), 1000000000000);

  priority_queue<edge> todo;

  todo.push({startnode, 0});

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

    // Early return
    if(current.to == target) return current.weight;

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

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

        todo.push(next);
      }
    }
  }

  // target not found
  return -1;
}
```

Übersicht

- Problem: Dijkstra kommt nicht mit negativen Kanten zurecht



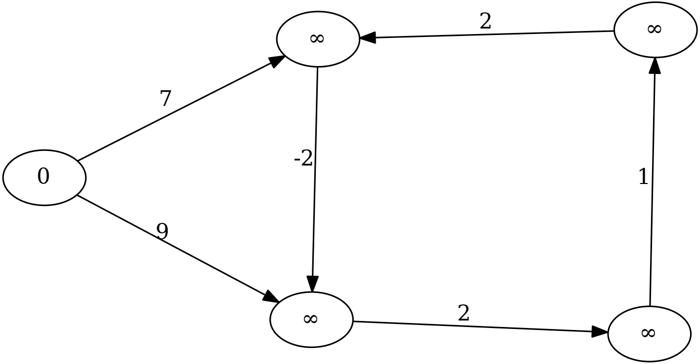
Ansätze

- 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)

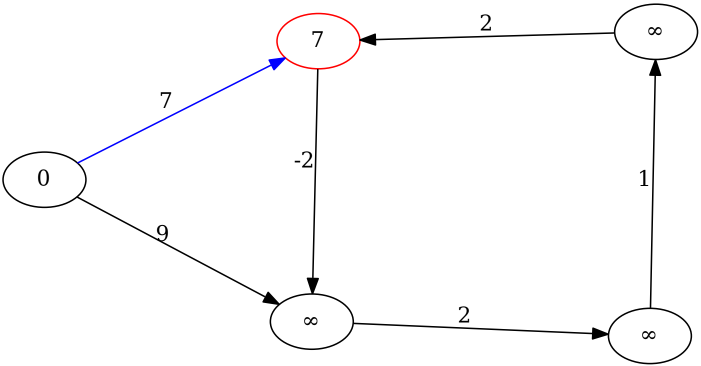
Ansätze

- 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 Belllman und Ford.

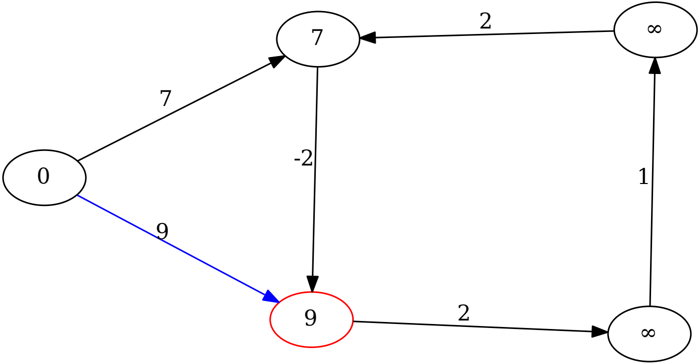
Initialisierung



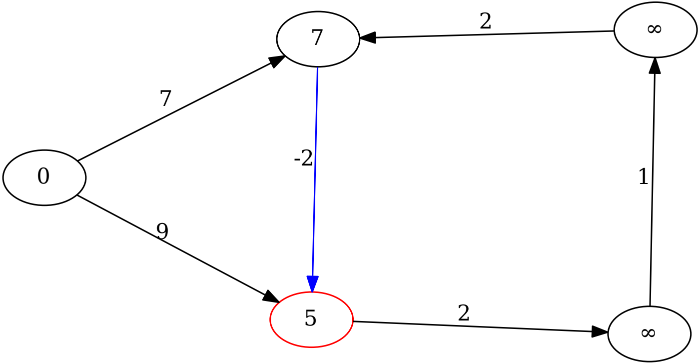
Runde 1



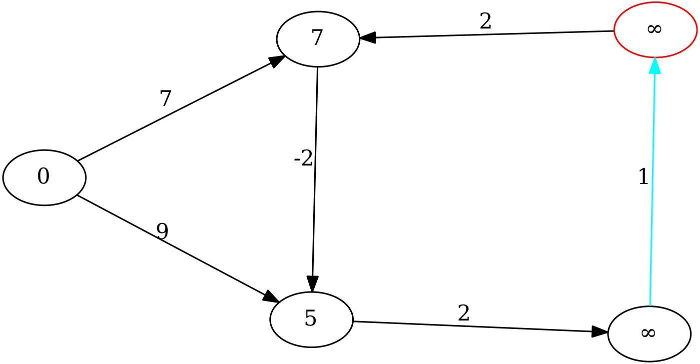
Runde 1



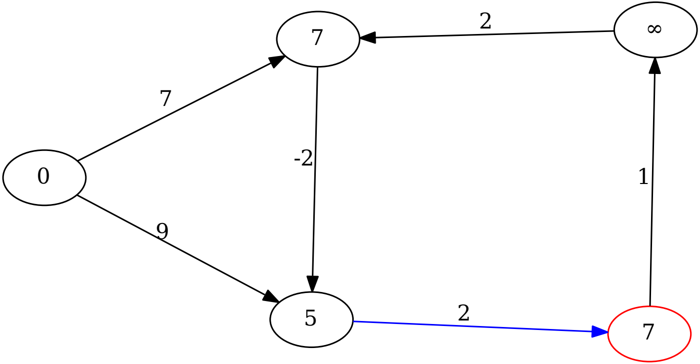
Runde 1



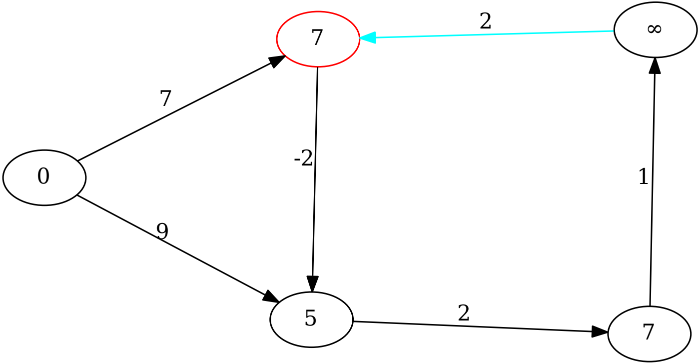
Runde 1



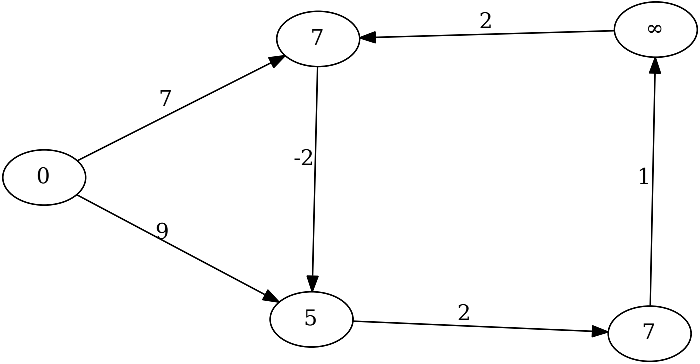
Runde 1



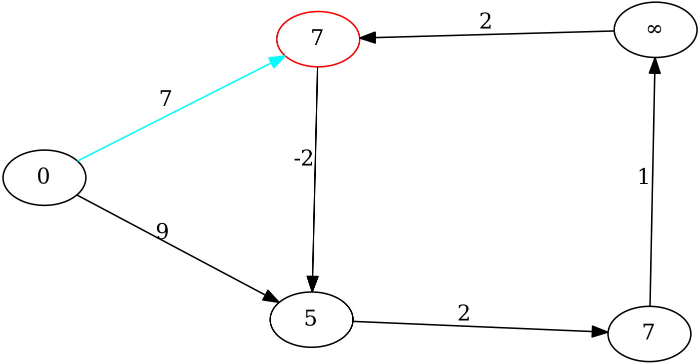
Runde 1



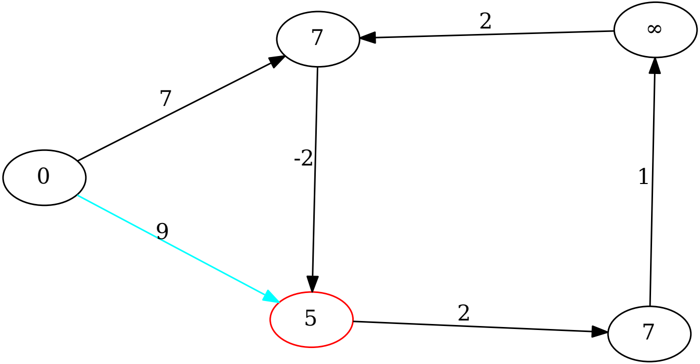
Runde 2



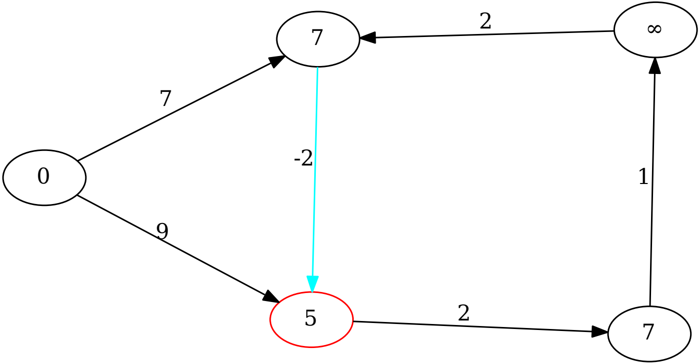
Runde 2



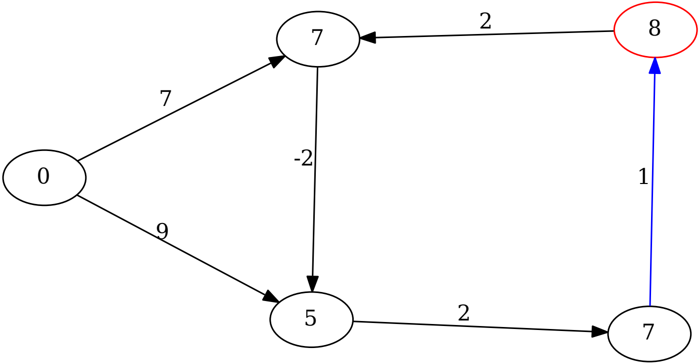
Runde 2



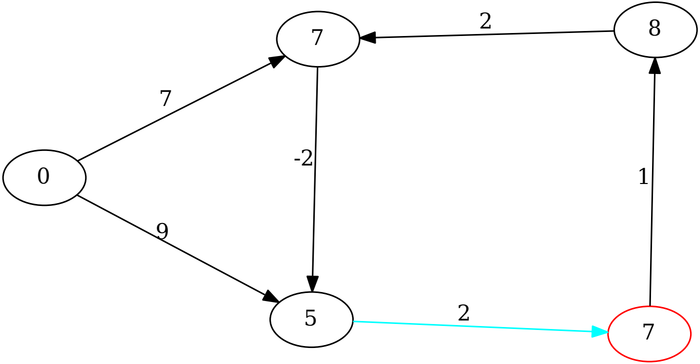
Runde 2



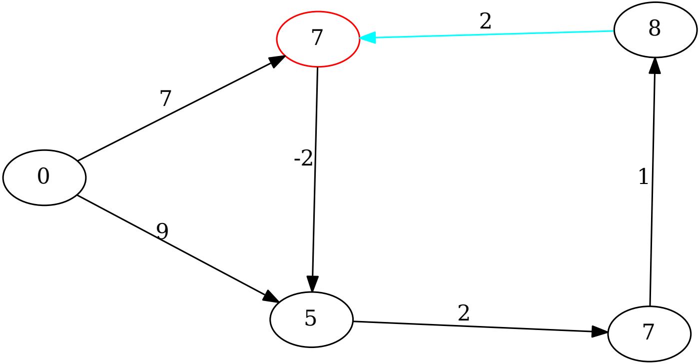
Runde 2



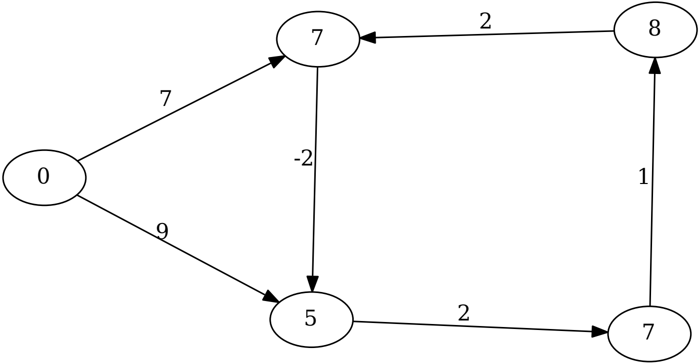
Runde 2



Runde 2



Runde 3 (keine Änderungen → fertig)



Code

```
using node = std::size_t;
using dist = double;

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

const auto inf_dist = std::numeric_limits<dist>::
    infinity();
```

Code

```
std::vector<dist> bellman_ford(
        std::size_t node_count,
        const std::vector<edge>& edges,
        node source
) {
    std::vector<dist> min_dists(node_count, inf_dist);
    min_dists[source] = 0;
    for (std::size_t i = 0; i < node_count + 1; ++i) {
        auto changes = false;
        for(const auto& e: edges) {
            const auto old_dist = min_dists[e.to];
            const auto new_dist = min_dists[e.from]
                                + e.weight;
            if (new_dist < old_dist) {
                min_dists[e.to] = new_dist;
                changes = true;
            }
        }
        // ...
```

Code

```
            // ...
            if (!changes) { break; }
            if (i == node_count) {
                throw std::runtime_error{
                    "negative cycle"};
            }
        }
        return min_dists;
    }
```

Code

```cpp
int main() try {
    const auto edges = std::vector<edge>{
        {0, 1,  7}, {0, 4,  -1},
        {1, 0, 10}, {1, 3,  -4},
        {2, 4,  1},
        {3, 0,  0}, {3, 2, 2.5},
        {4, 1, 23}
    };
    const auto min_dists = bellman_ford(5, edges, 0);
    std::copy(min_dists.begin(), min_dists.end(),
        std::ostream_iterator<dist>{std::cout, "\n"});
} catch (std::runtime_error& e) {
    std::cerr << "Error: " << e.what() << '\n';
}
```

Weitere Eigenschaften

- Negative Kreise lassen sich durch eine weitere Anwendung detektieren
- Negative Kreise die nicht auf dem Weg zum Ziel liegen, verfälschen das Ergebnis nicht
  - Die Detektion aller problemlosen Knoten ist mit $V - 1$ weiteren Anwendungen möglich

Beurteilung

- Assymptotische Komplexität $\in O(|V| \cdot |E|)$
- Profitiert nicht von kurzen Distanzen zwischen Quelle und Senke
- Relativ leicht zu implementieren

Fazit

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, wobei i, j aus V sind.

APSP - Erster Ansatz

Lösungsansatz

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

Laufzeit

$|V| \cdot O((|E| + |V|) \cdot log(|V|))$
$= |V| \cdot O((|V|^2 + |V|) \cdot log(|V|))$
$= |V| \cdot O((|V|^2) \cdot log(|V|))$
$= O((|V|^3) \cdot log(|V|))$
$\Longrightarrow$ geht es schneller?

APSP - Zweiter Ansatz

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 $A[i][j] = \min(A[i][j], A[i][k] + A[i][k])$, 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.

Code

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

$\Longrightarrow$ der Aufwand liegt in $O(|V|^3)$

Weitere Anwendungen

- Auch für SSSP - Probleme anwendbar (wenn $|V| < 400$)
- Detektion von negativen oder günstigsten Zyklen möglich
  $\Longrightarrow$ setze die Diagonale auf Unendlich (hohen Wert)
- Finden des Durchmessers eines Graphen (der längste der kürzesten Pfade)
- Minimax, Maximin
- Transitive Hülle berechnen (wer ist mit wen verbunden $\rightarrow$ bits)
- Finden von starken Zusammenhangskomponenten
- evtl. weitere Anwendungen in konkreten Fällen

Beurteilung

- $+$ Asymptotische Komplexität $\in O(|V|^3)$ und mit Speicher $\in O(|V|^2)$
- $+$ Sehr leicht zu implementieren (Vierzeiler)
- $+$ Für andere Probleme günstig anzuwenden, wenn $|V| < 400$
- $--$ Für andere Probleme **nur** günstig anzuwenden, wenn $|V| < 400$

$\Longrightarrow$ Gut für das ursprüngliche Problem

$\Longrightarrow$ Auch nützlich für andere Probleme, solange $|V| < 400$

Zusammenfassung

Zusammenfassung

Kriterium	**Dijkstra**	**Bellman Ford**	**Floyd Warshall**
Laufzeit	$O(V+E)\log(V)$	$O(V \cdot E)$	$O(V^3)$
Max. Größe	$V, E \leq 300K$	$V, E \leq 10M$	$V, E \leq 400$
Ungewichtet	Ok	Schlecht	I.A. Schlecht
Gewichtet	Bestes	Ok	I.A. Schlecht
Neg. Gewichte	Ok	Ok	I.A. Schlecht
Neg. Zyklen	Nein	Aufspürbar	Aufspürbar
Kleine Graphen	Overkill	Overkill	Bestes

Tabelle: Übersicht

Anmerkung: Lese V als $|V|$ und E als $|E|$