Problem G

Patent Claims

Michael Bendel, David Wolz

Problemstellung

Wir befinden uns im Jahr 1902. Albert Einstein arbeitet im Patentamt in Bern. Viele Patentanträge enthalten grobe Fehler, manche verletzen sogar das Gesetz der Energieerhaltung.

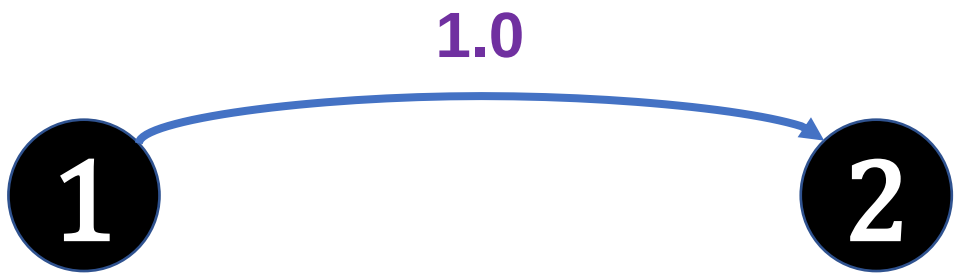
Jeder Patentantrag hat folgenden Aufbau:

- Enthält eine feste Anzahl von Energiewandlern
- Jeder dieser Energiewandler hat einen unbekannten Energie-Input
- Wenn der Output eines Energiewandlers von anderen Energiewandlern als Input genutzt werden kann, besteht eine Verbindung zwischen diesen Wandlern
- Jede dieser Verbindungen hat einen festen Umwandlungsfaktor

Modellierung

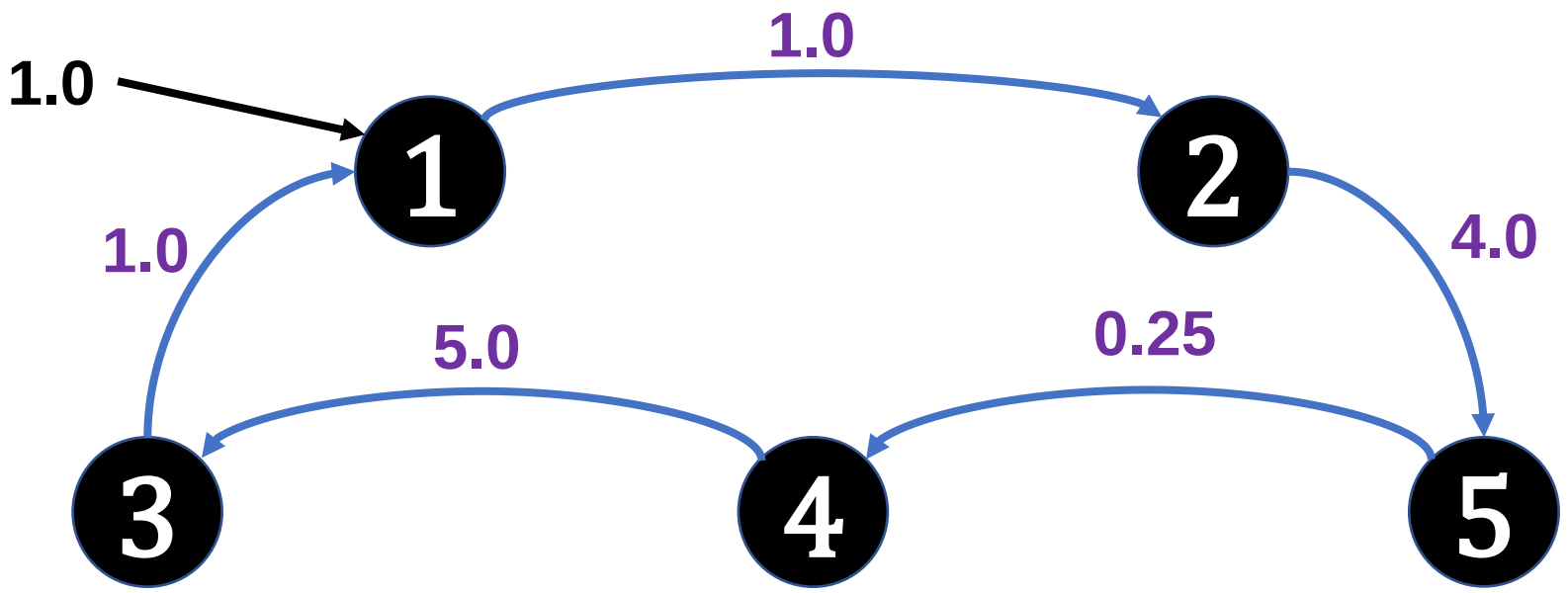
Problem kann als gewichteter, gerichteter Graph dargestellt werden:

- Die Energiewandler sind die Knoten des Graphen
- Die Verbindungen zwischen den Energiewandlern sind dessen Kanten
- Der Umwandlungsfaktor ist das Gewicht der Kanten



Problemstellung

Einstein möchte alle Patentanträge, die den Energieerhaltungssatz verletzen, aussortieren.



Wandler	Input
1	1.0
2	1.0
5	4.0
4	1.0
3	5.0
1	5.0

Einsteins Assistenten sortieren bereits die schwierigsten Fälle aus. So erreichen Einstein nur Patentanträge, in denen das Produkt der Kanten eines Kreises ≤0.9 oder ≥1.1 ist (≤0.9 ist zulässig und ≥1.1 ist unzulässig).

Input

```
3 4
1 2 0.5
2 1 2.3
3 1 0.3
1 3 0.8
```

Die erste Zeile besteht aus zwei Zahlen, wobei

n = Anzahl der Knoten $(2 \leq n \leq 800)$

m = Anzahl der Kanten $(0 \leq m \leq 4000)$

Die nachfolgenden m Zeilen enthalten jeweils Kanten der Form $a_i\ b_i\ c_i$ mit

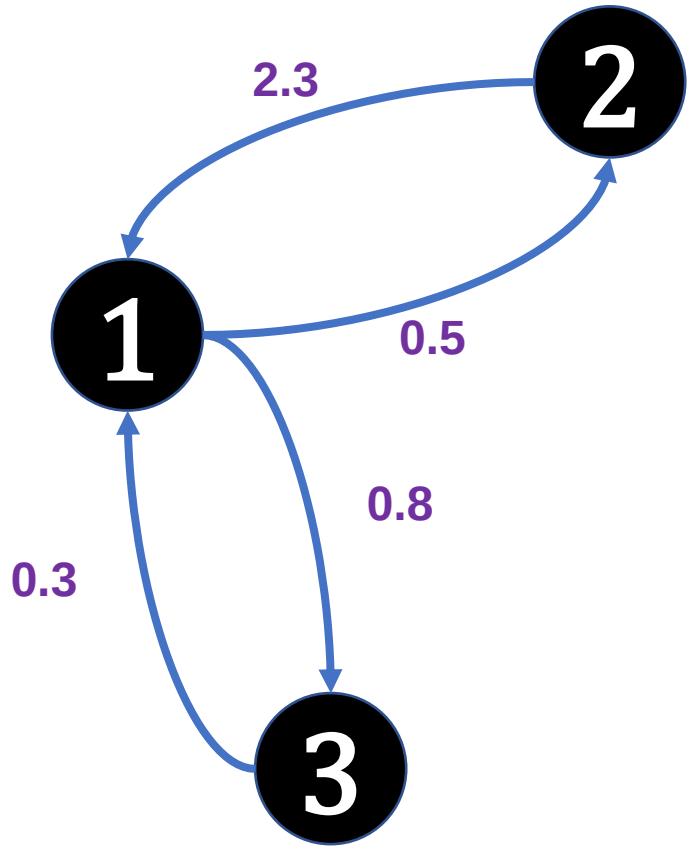
a_i = Startknoten $(1 \leq a_i \leq n)$

b_i = Zielknoten $(1 \leq b_i \leq n)$

c_i = Kantengewicht $(0 < c_i \leq 5.0)$

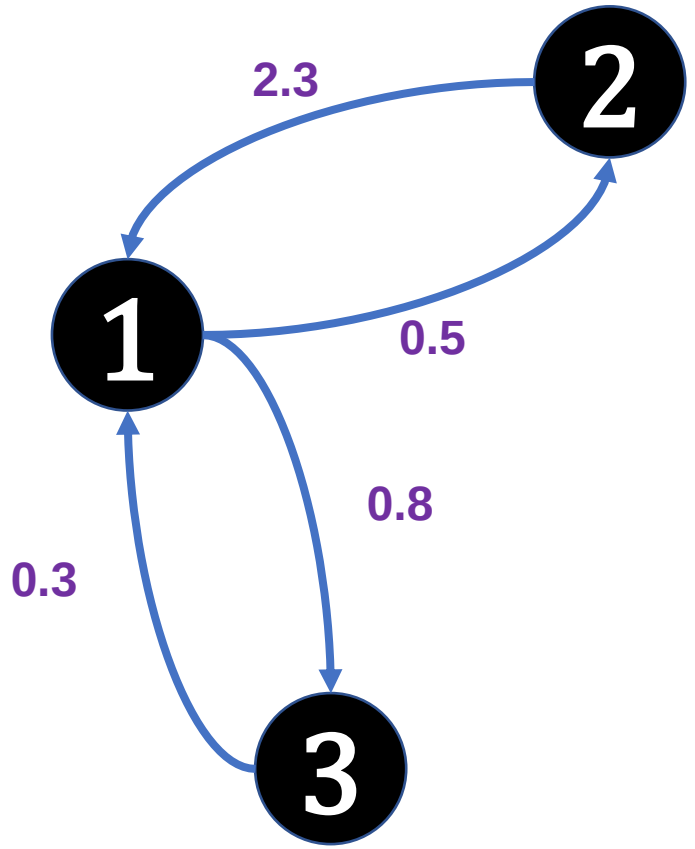
Input

```
3 4
1 2 0.5
2 1 2.3
3 1 0.3
1 3 0.8
```



Output

3 4
1 2 0.5
2 1 2.3
3 1 0.3
1 3 0.8

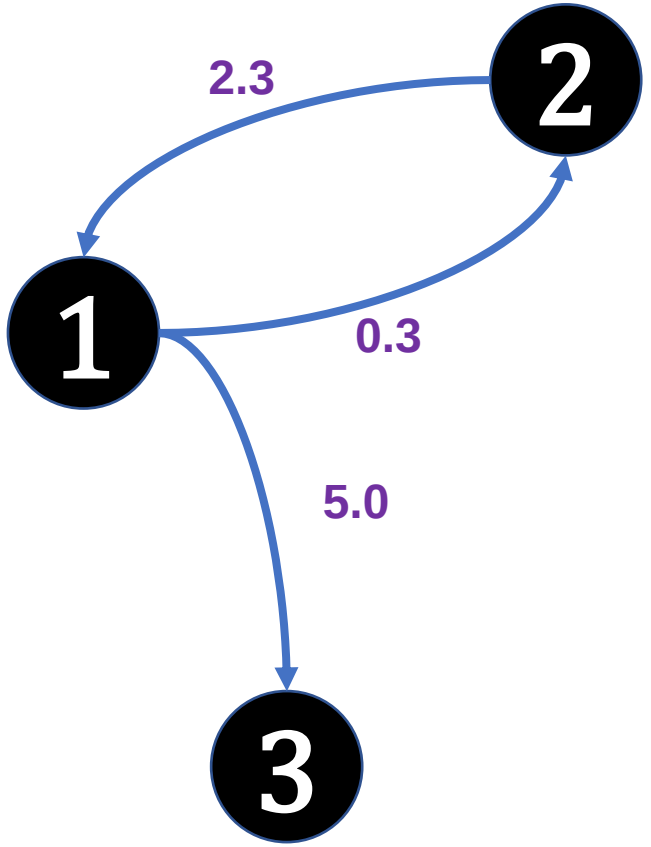


$0.5 \times 2.3 = 1.15 \geq 1.1$

inadmissible

Output

3 3
1 2 0.3
2 1 2.3
1 3 5.0



$$0.3 \times 2.3 = 0.69 \leq 0.9$$

admissible

Idee

- Es reicht zu prüfen, ob das Kantengewichtprodukt größer oder kleiner 1 ist

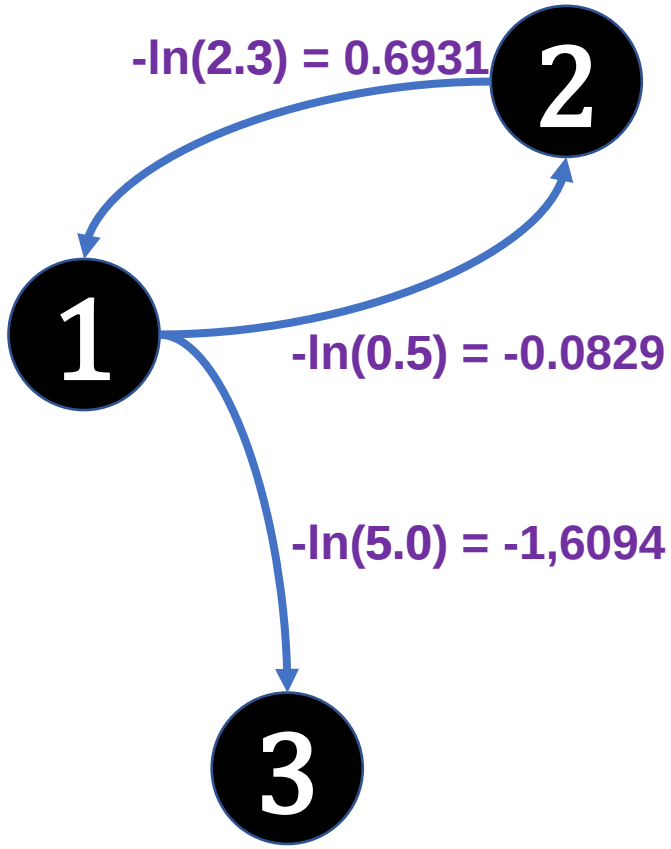
$$\prod_{\forall i \in K} c_i > 1 \qquad | \ln()$$

$$\ln\left(\prod_{\forall i \in K} c_i\right) > \ln(1)$$

$$\sum_{\forall i \in K} \ln(c_i) > 0 \qquad | \cdot (-1)$$

$$-\sum_{\forall i \in K} \ln(c_i) < 0$$

$$\sum_{\forall i \in K} -\ln(c_i) < 0$$



- Es reicht also einen Kreis mit negativer Summe der modifizierten Gewichte zu finden
- Angepasster Bellman-Ford Algorithmus findet genau diese Kreise

Funktionsweise

- Generell berechnet der Bellman-Ford Algorithmus kürzeste Wege, mit Möglichkeit zur Verwendung negativer Kantengewichte
- Statt angegebener Gewichte deren negative natürliche Logarithmen verwenden
- Dadurch, dass ein Kreis maximal 800 Knoten hat und wir 0.1 Ungenauigkeit tolerieren können, kann die Logarithmusberechnung mit einer Genauigkeit von vier Nachkommastellen arbeiten
- i-ter Schleifendurchlauf berechnet kürzesten Weg von 1 zu jedem anderen Knoten mit maximal i Kanten
- Wenn es keine Zyklen mit negativer Kantensumme gibt, enthält jeder kürzeste Pfad jeden Knoten maximal einmal
- Wenn also nach ($|V| - 1$) Durchläufen eine weitere Verbesserung vorgenommen werden kann, muss es einen Zyklus negativer Kantensummen geben

Algorithmus

BellmanFord(V, E)

dist[] $=$ neues Array mit Länge $|V|$, gefüllt mit ∞
dist[1] $=$ 0

repeat $(|V| - 1)$ times
  foreach $(u, v, c) \in E$ do
    if dist[u]$\neq \infty$ && dist[u]$+c <$ dist[v] then
      dist[v]= dist[u]$+c$

foreach $(u, v, c) \in E$ do
  if dist[u]$\neq \infty$ && dist[u]$+c <$ dist[v] then
    return true

return false

Algorithmus

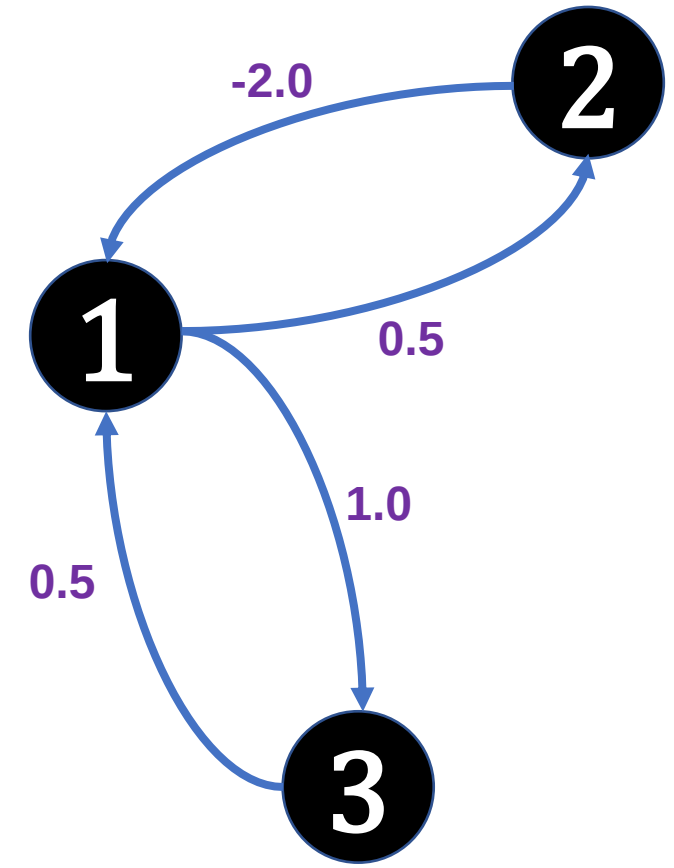
BellmanFord(V, E)

dist[] $=$ neues Array mit Länge $|V|$, gefüllt mit ∞
dist[1] $=$ 0

repeat $(|V| - 1)$ times
  foreach $(u, v, c) \in E$ do
    if dist[u]$\neq \infty$ && dist[u]$+c <$ dist[v] then
      dist[v]= dist[u]$+c$

foreach $(u, v, c) \in E$ do
  if dist[u]$\neq \infty$ && dist[u]$+c <$ dist[v] then
    return true

return false



0	∞	∞
1	2	3

Algorithmus

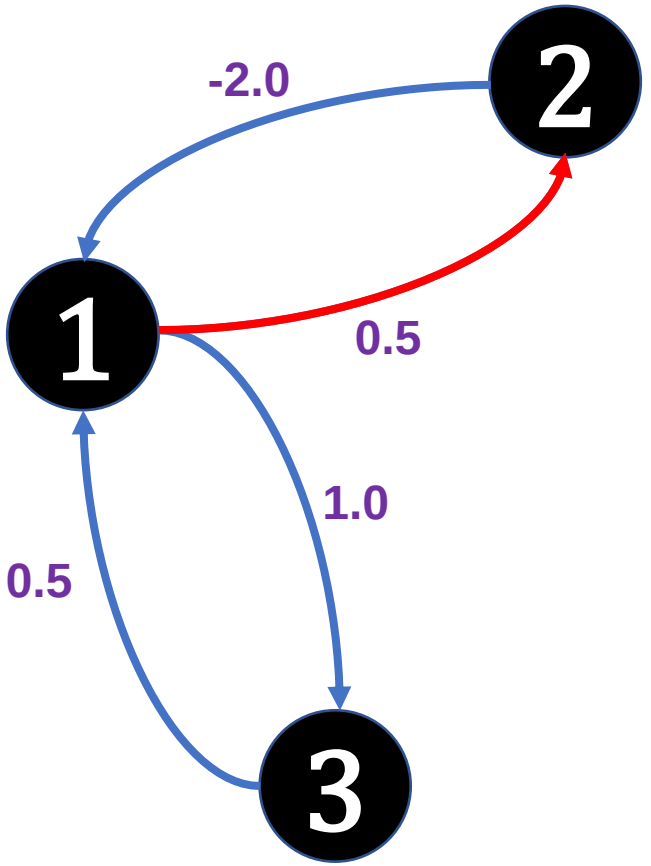
BellmanFord(V, E)

dist[] $=$ neues Array mit Länge $|V|$, gefüllt mit ∞
dist[1] $=$ 0

repeat $(|V| - 1)$ times
  foreach $(u, v, c) \in E$ do
    if dist[u]$\neq \infty$ && dist[u]$+c <$ dist[v] then
      dist[v]= dist[u]$+c$

foreach $(u, v, c) \in E$ do
  if dist[u]$\neq \infty$ && dist[u]$+c <$ dist[v] then
    return true

return false



0	0.5	∞
1	2	3

Algorithmus

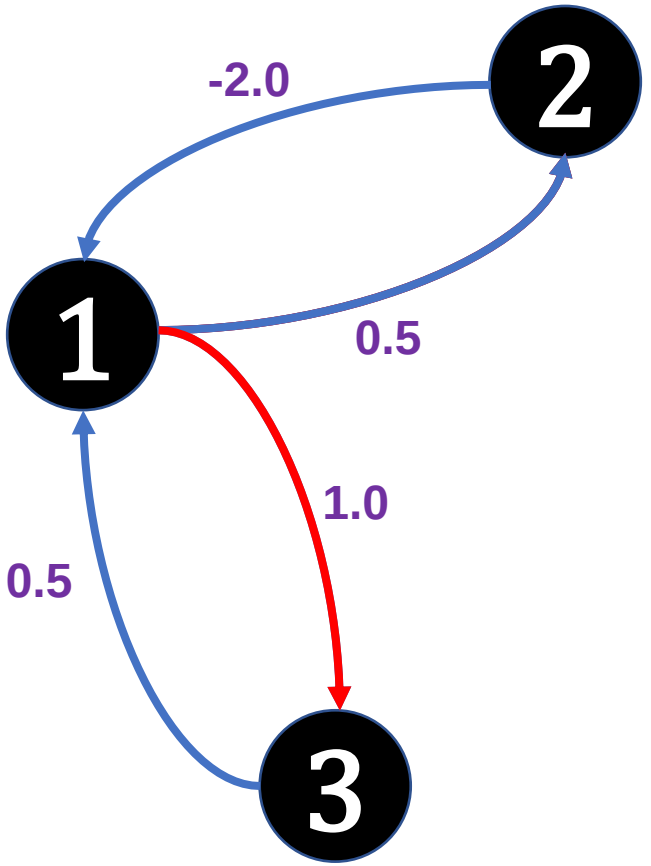
BellmanFord(V, E)

dist[] $=$ neues Array mit Länge $|V|$, gefüllt mit ∞
dist[1] $=$ 0

repeat $(|V| - 1)$ times
  foreach $(u, v, c) \in E$ do
    if dist[u]$\neq \infty$ && dist[u]$+c <$ dist[v] then
      dist[v]= dist[u]$+c$

foreach $(u, v, c) \in E$ do
  if dist[u]$\neq \infty$ && dist[u]$+c <$ dist[v] then
    return true

return false



0	0.5	1
1	2	3

Algorithmus

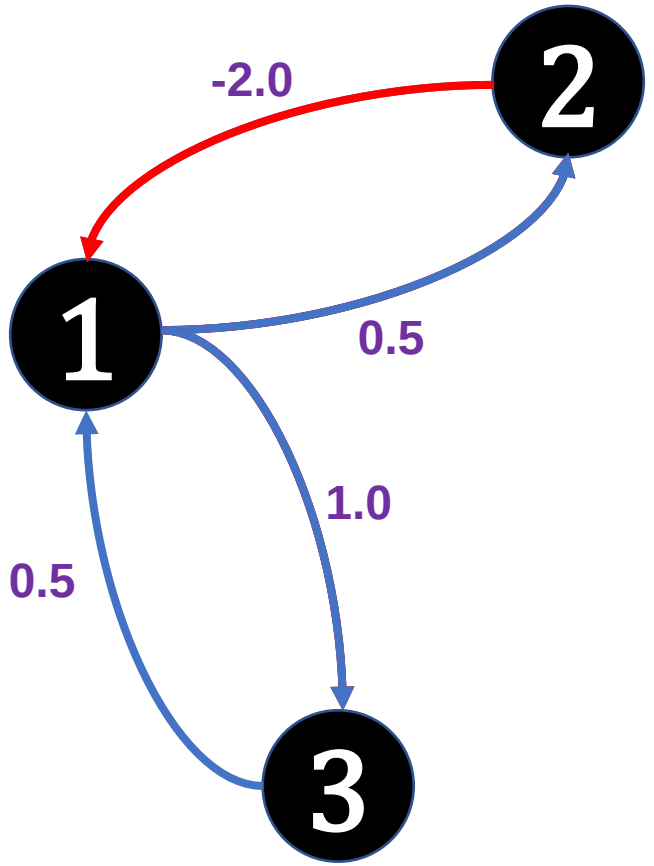
BellmanFord(V, E)

dist[] $=$ neues Array mit Länge $|V|$, gefüllt mit ∞
dist[1] $=$ 0

repeat $(|V| - 1)$ times
  foreach $(u, v, c) \in E$ do
    if dist[u]$\neq \infty$ && dist[u]$+c <$ dist[v] then
      dist[v]= dist[u]$+c$

foreach $(u, v, c) \in E$ do
  if dist[u]$\neq \infty$ && dist[u]$+c <$ dist[v] then
    return true

return false



-1.5	0.5	1
1	2	3

Algorithmus

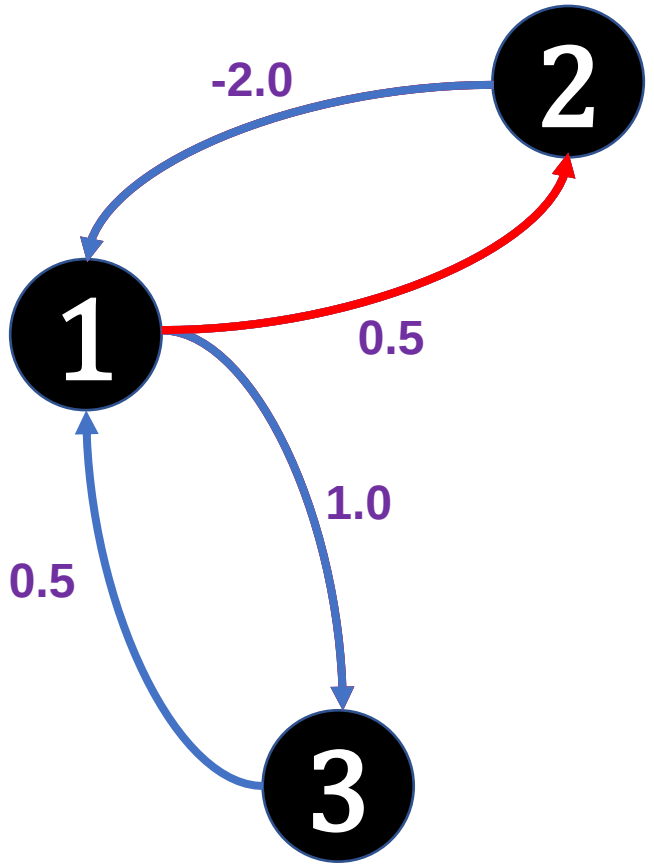
BellmanFord(V, E)

dist[] $=$ neues Array mit Länge $|V|$, gefüllt mit ∞
dist[1] $=$ 0

repeat $(|V| - 1)$ times
  foreach $(u, v, c) \in E$ do
    if dist[u]$\neq \infty$ && dist[u]$+c <$ dist[v] then
      dist[v]= dist[u]$+c$

foreach $(u, v, c) \in E$ do
  if dist[u]$\neq \infty$ && dist[u]$+c <$ dist[v] then
    return true

return false



-1.5	-1	1
1	2	3

Algorithmus

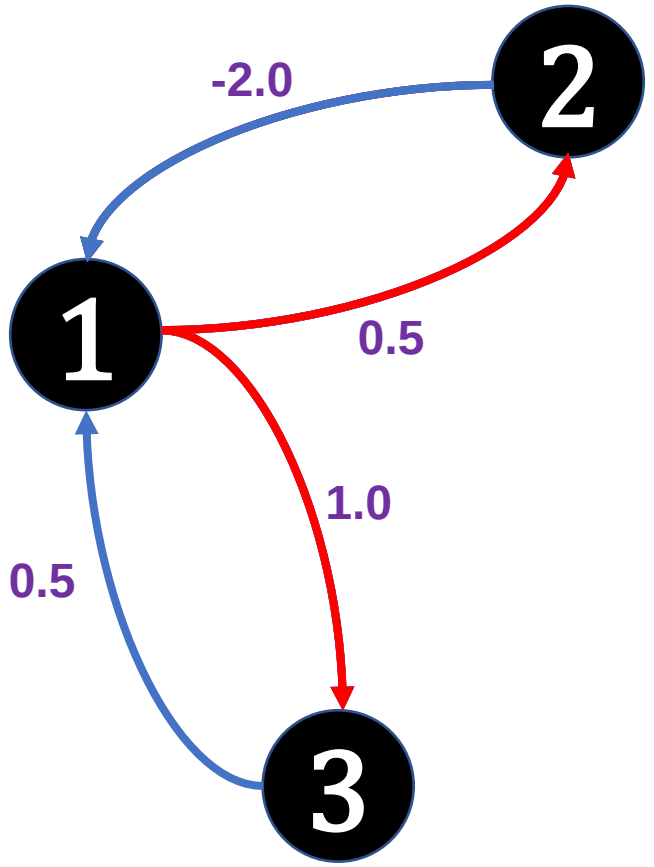
BellmanFord(V, E)

dist[] $=$ neues Array mit Länge $|V|$, gefüllt mit ∞
dist[1] $=$ 0

repeat $(|V| - 1)$ times
  foreach $(u, v, c) \in E$ do
    if dist[u]$\neq \infty$ && dist[u]$+c <$ dist[v] then
      dist[v]= dist[u]$+c$

foreach $(u, v, c) \in E$ do
  if dist[u]$\neq \infty$ && dist[u]$+c <$ dist[v] then
    return true

return false



-1.5	-1	-0.5
1	2	3

Laufzeitanalyse

BellmanFord(V, E)

dist[] $=$ neues Array mit Länge $|V|$, gefüllt mit ∞ — $O(|V|)$
dist[1] $=$ 0

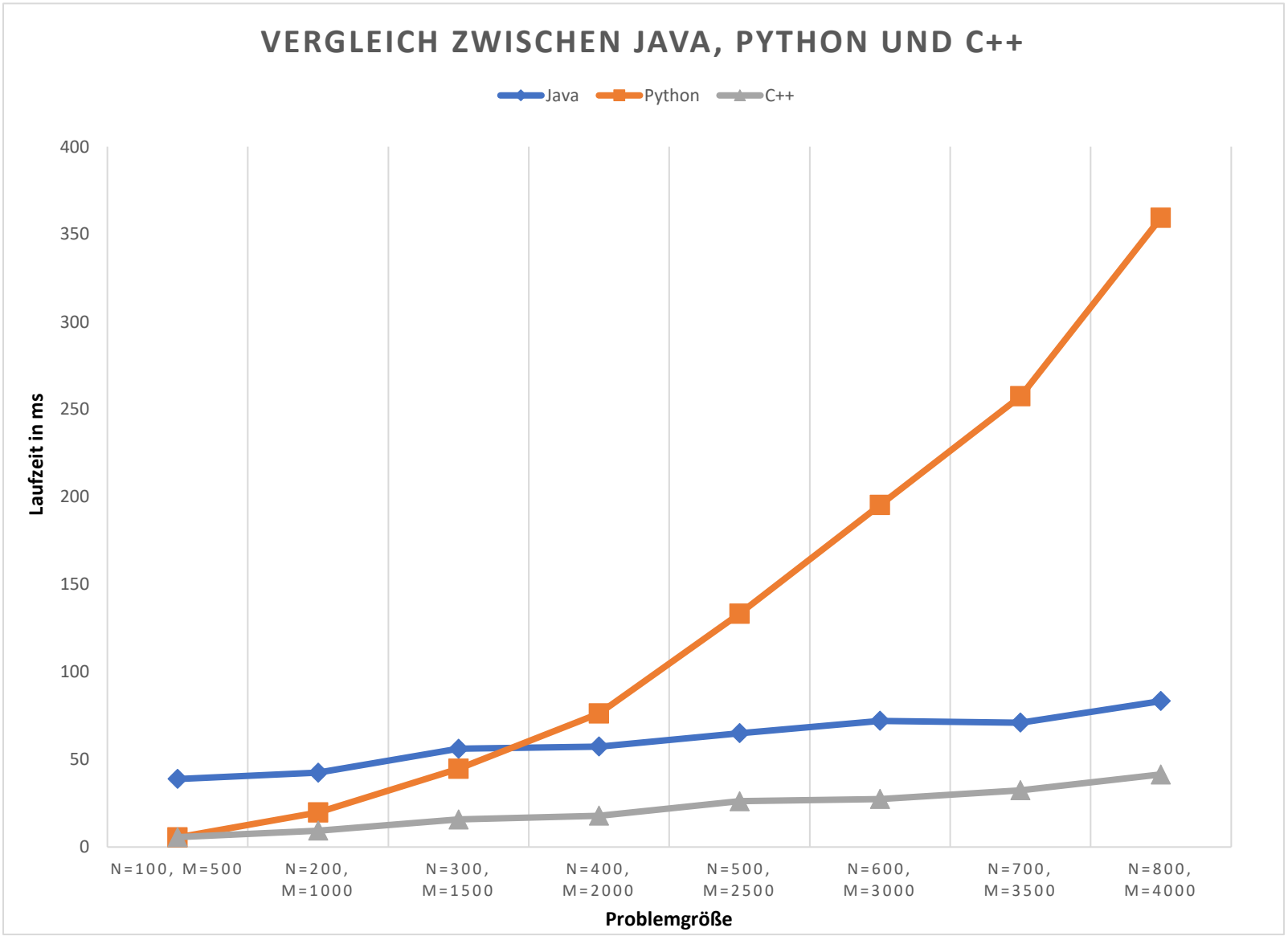
repeat $(|V| - 1)$ times — $O(|V| \cdot |E|)$
  foreach $(u, v, c) \in E$ do
    if dist[u]$\neq \infty$ && dist[u]$+c <$ dist[v] then
      dist[v]= dist[u]$+c$

foreach $(u, v, c) \in E$ do — $O(|E|)$
  if dist[u]$\neq \infty$ && dist[u]$+c <$ dist[v] then
    return true

return false

$$O(|V| + |V| \cdot |E| + |E|) \subseteq O(|V| \cdot |E|)$$

Laufzeitmessungen



- Probleme waren für alle Programmiersprachen stets identische, „admissible“-Beispiele, die randomisiert generiert wurden
- Kurven sehen (außer bei Python) nicht quadratisch aus, das ist auf die vergleichsweise kleine Problemgröße zurückzuführen, wenn die maximale Größe n=8000, m=40000 wären, hätte C++ eine Zeit von 2650ms, Java von 6500ms und Python bräuchte 42000ms