Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Problem N:
Is this a detour?

Valentin Baral, Maximilian Neb

Inhalt

- Problembeschreibung
- Input
- Lösungsansatz
- Laufzeitanalyse

Problem

“Every day you drive to work using the same roads as it is the **shortest way**. This is efficient, but over time you have grown increasingly bored of seeing the same buildings and junctions every day. So you decide to look for different routes. Of course you do not want to sacrifice time, so the new way should be **as short as the old one**. Is there another way that **differs** from the old one in **at least one street**?”

“Jeden Tag fährst du zur Arbeit und benutzt immer den selben **kürzesten Weg**. Das ist zwar effizient, aber im Laufe der Zeit wird es immer langweiliger, jeden Tag dieselben Gebäude und Kreuzungen zu sehen. Also entscheidest du dich nach verschiedenen Routen zu suchen. Natürlich möchtest du keine Zeit opfern, daher sollte der neue Weg **so kurz sein wie der Alte**. Gibt es einen anderen Weg, der sich in **mindestens einer Straße** vom alten **unterscheidet**?”

Input

3 3 3

1 2 3

1 2 1

2 3 2

1 3 3

Input

```
3 3 3

1 2 3

1 2 1

2 3 2

1 3 3
```

Die erste Zeile gibt folgendes an:

- Kreuzungen (Knoten) **N** (1 ≤ N ≤ 10.000)
- Straßen (Kanten) **M** (0 ≤ M ≤ 1.000.000)
- Knoten des Arbeitswegs **K** (1 ≤ K ≤ 10.000)

Input

3 3 3

1 2 3

1 2 1

2 3 2

1 3 3

Die erste Zeile gibt folgendes an:

- Kreuzungen (Knoten) **N** (1 ≤ N ≤ 10.000)
- Straßen (Kanten) **M** (0 ≤ M ≤ 1.000.000)
- Knoten des Arbeitswegs **K** (1 ≤ K ≤ 10.000)

Die zweite Zeile:

- **K** Kreuzungen (**Arbeitsweg**)
- Beginnt mit 1 und endet mit N

Input

```
3 3 3

1 2 3

1 2 1

2 3 2

1 3 3
```

Die erste Zeile gibt folgendes an:

- Kreuzungen (Knoten) **N** (1 ≤ N ≤ 10.000)
- Straßen (Kanten) **M** (0 ≤ M ≤ 1.000.000)
- Knoten des Arbeitswegs **K** (1 ≤ K ≤ 10.000)

Die zweite Zeile:

- **K** Kreuzungen (**Arbeitsweg**)
- Beginnt mit 1 und endet mit N

Folgende **M** Zeilen:

- ungerichtete **Straßen**
- Start- und Zielkreuzung
- Fahrzeit (Kanten Gewicht)

Beispiel:

4 5 3

1 2 4

1 2 1

1 3 2

1 4 4

2 4 2

3 4 1

Beispiel:

4 5 3

1 2 4

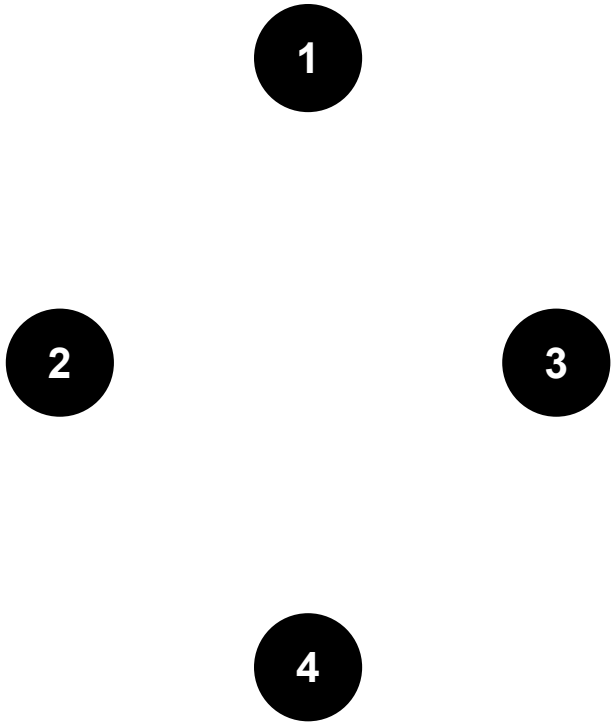
1 2 1

1 3 2

1 4 4

2 4 2

3 4 1



Beispiel:

4 5 3

1 2 4

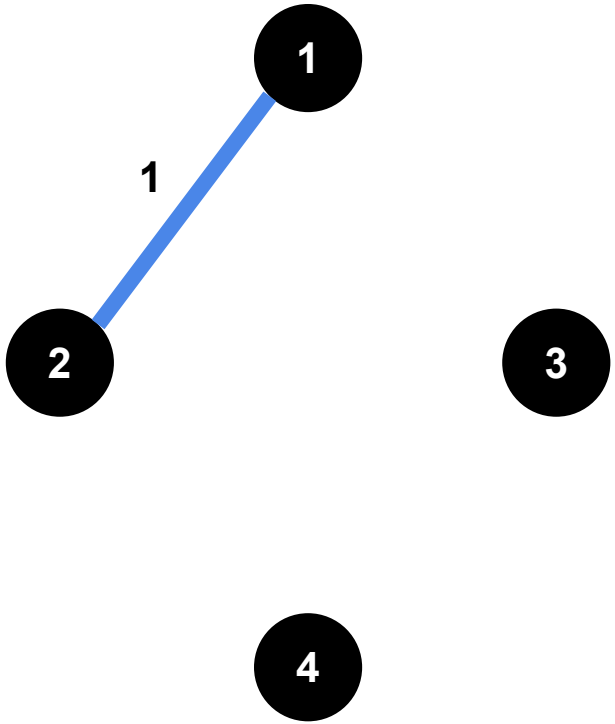
1 2 1

1 3 2

1 4 4

2 4 2

3 4 1



Beispiel:

4 5 3

1 2 4

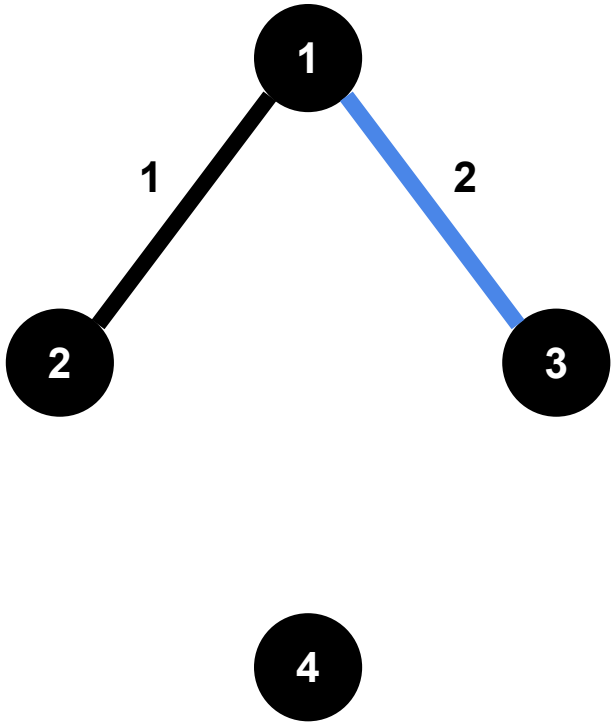
1 2 1

1 3 2

1 4 4

2 4 2

3 4 1



Beispiel:

4 5 3

1 2 4

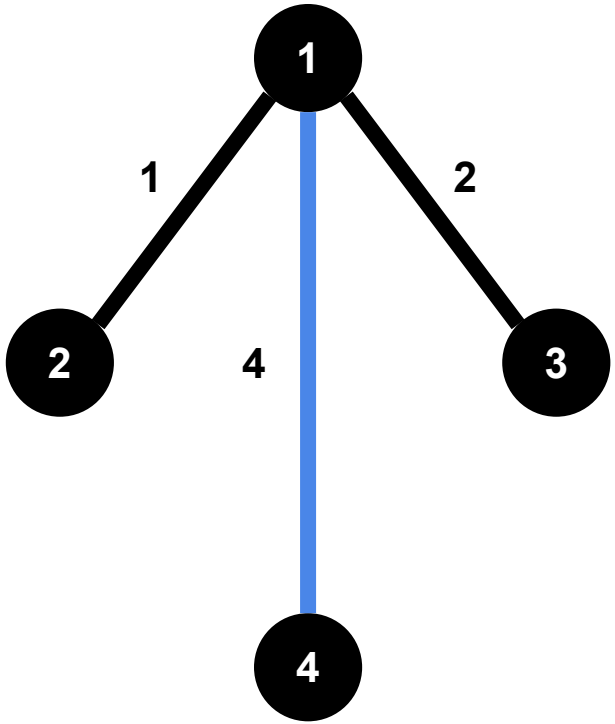
1 2 1

1 3 2

1 4 4

2 4 2

3 4 1



Beispiel:

4 5 3

1 2 4

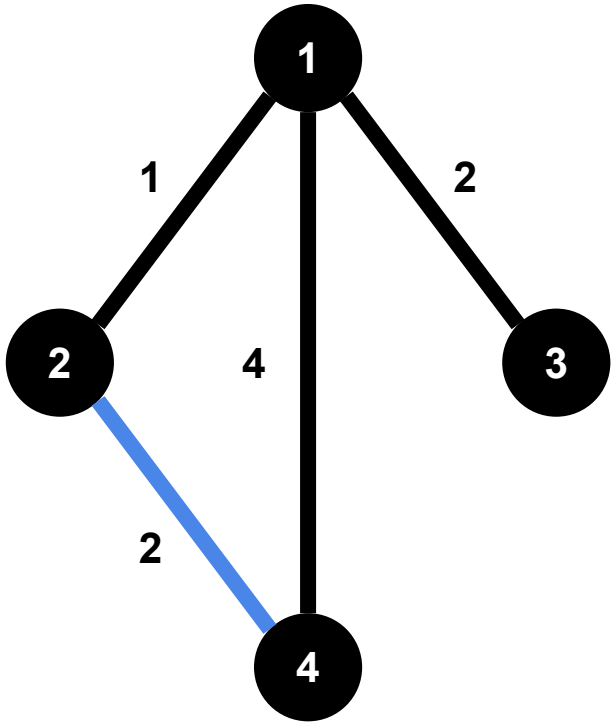
1 2 1

1 3 2

1 4 4

2 4 2

3 4 1



Beispiel:

4 5 3

1 2 4

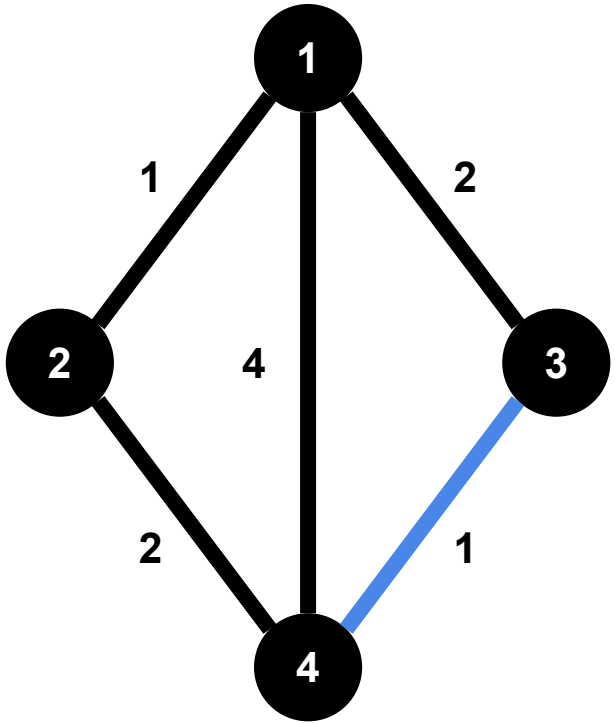
1 2 1

1 3 2

1 4 4

2 4 2

3 4 1



Beispiel:

4 5 3

1 2 4

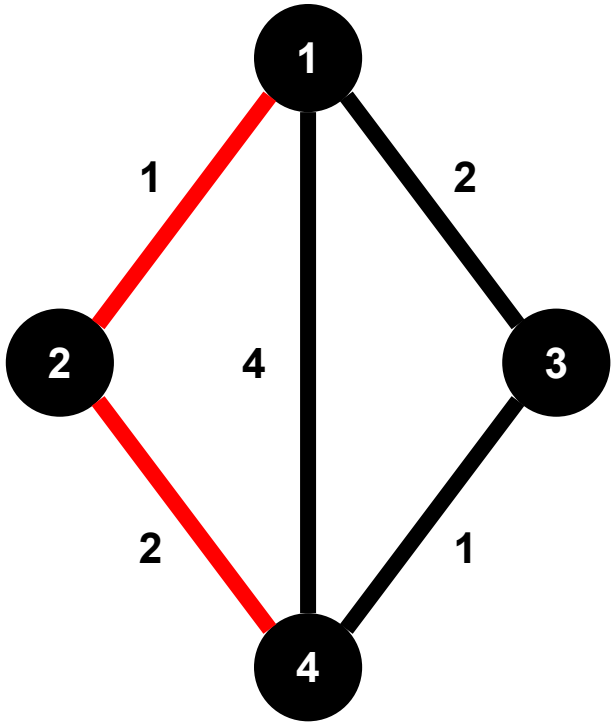
1 2 1

1 3 2

1 4 4

2 4 2

3 4 1



Beispiel:

4 5 3

1 2 4

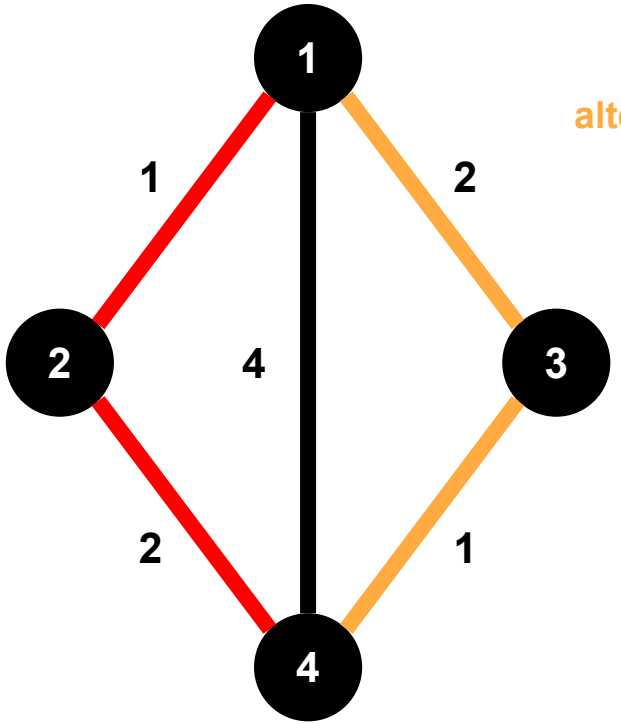
1 2 1

1 3 2

1 4 4

2 4 2

3 4 1



alternativer Pfad

Lösungsansatz

- **Beobachtung 1:**

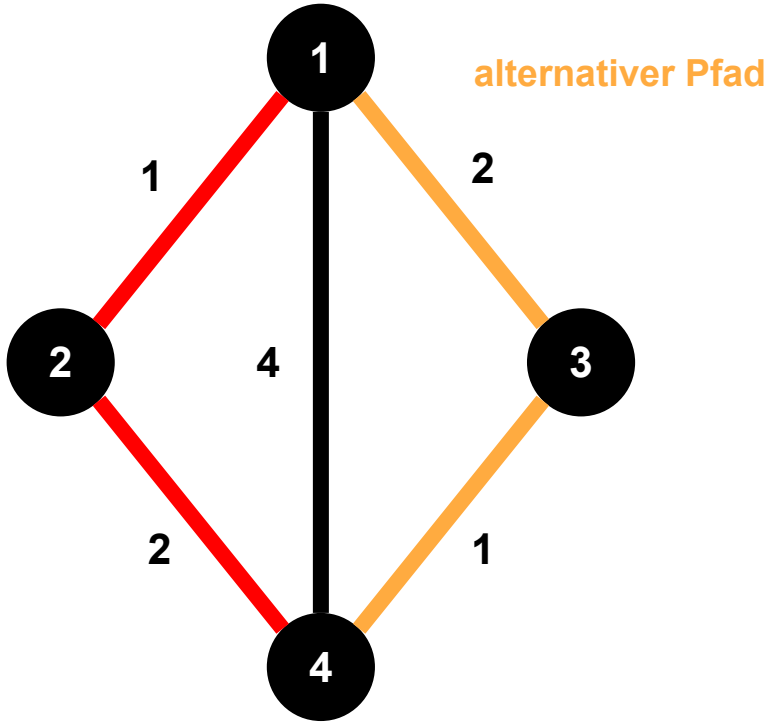
  Umweg kann über komplett andere Kreuzungen gehen

Lösungsansatz

- **Beobachtung 1:**

  Umweg kann über komplett andere Kreuzungen gehen

  ➢ Es reicht nicht nur die Arbeitsweg Knoten zu betrachten



Lösungsansatz

- **Beobachtung 2:**

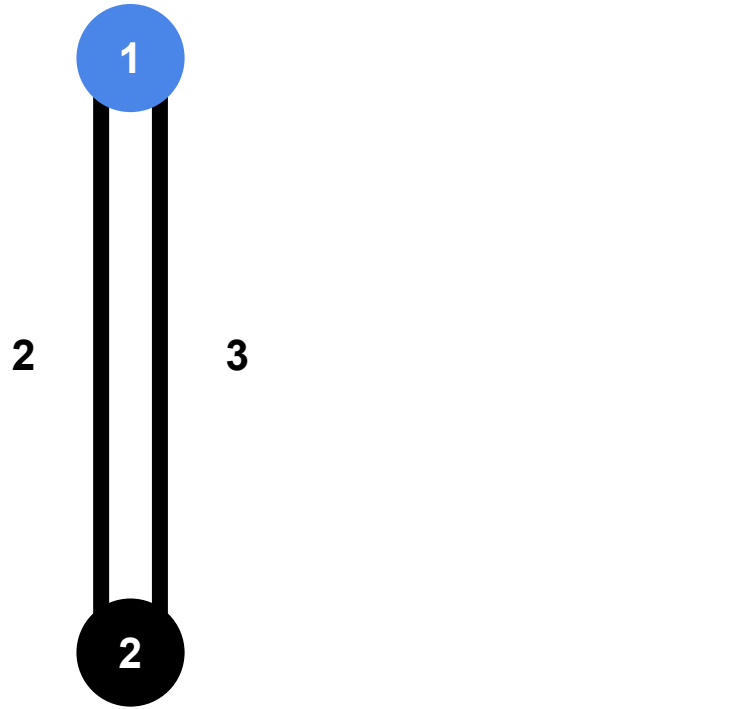
  Kürzester Weg nimmt immer die kürzesten Straßen

Lösungsansatz

- **Beobachtung 2:**

  Kürzester Weg nimmt immer die kürzesten Straßen

  ➢ Nur Straßen mit geringstem Gewicht sind relevant

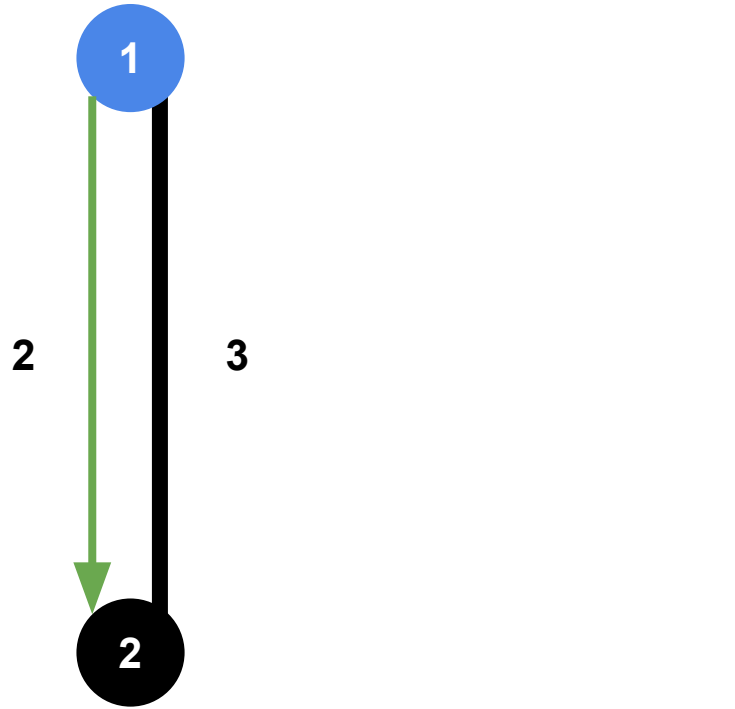


Lösungsansatz

- **Beobachtung 2:**

  Kürzester Weg nimmt immer die kürzesten Straßen

  ➢ Nur Straßen mit geringstem Gewicht sind relevant

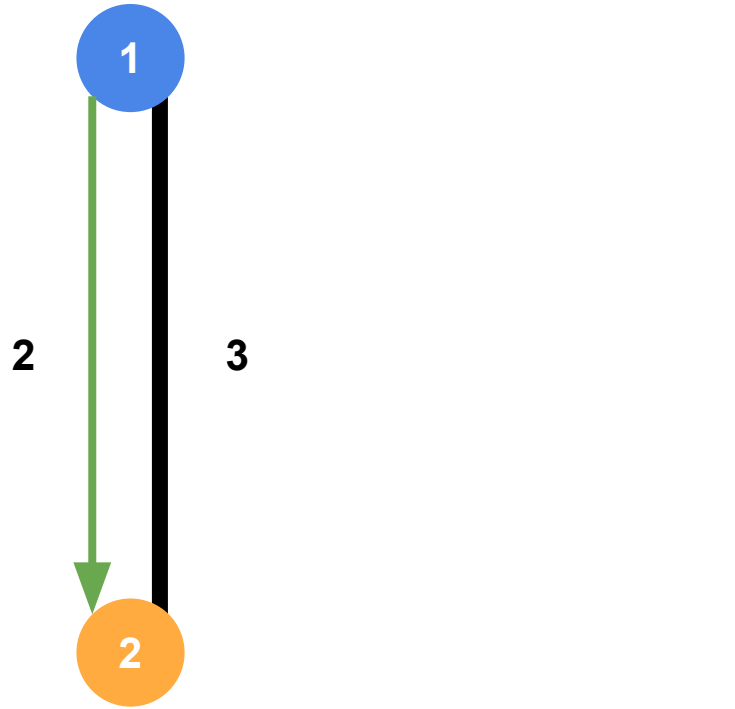


Lösungsansatz

- **Beobachtung 2:**

  Kürzester Weg nimmt immer die kürzesten Straßen

  ➢ Nur Straßen mit geringstem Gewicht sind relevant

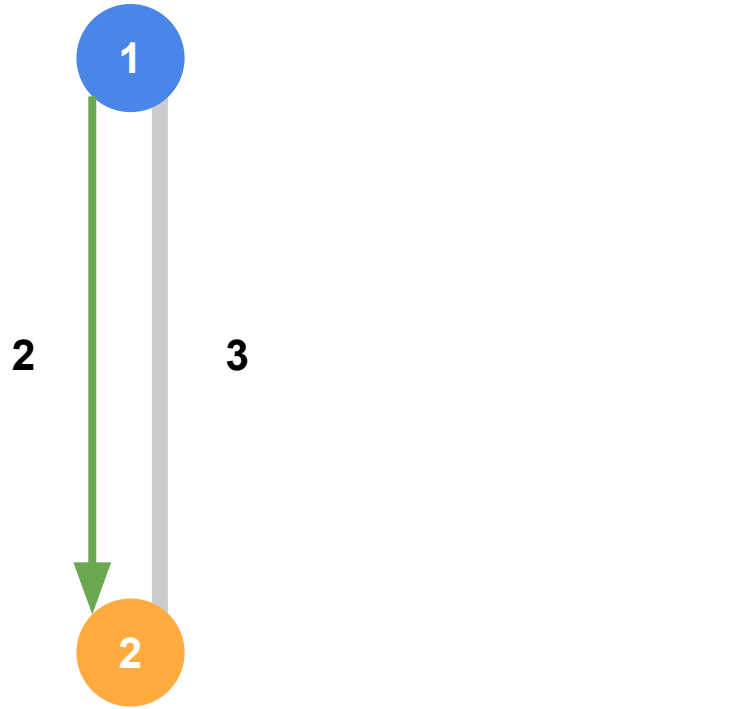


Lösungsansatz

- **Beobachtung 2:**

  Kürzester Weg nimmt immer die kürzesten Straßen

  ➢ Nur Straßen mit geringstem Gewicht sind relevant



Lösungsansatz

- **Beobachtung 3:**

  Mehrere kürzeste Straßen können vorhanden sein

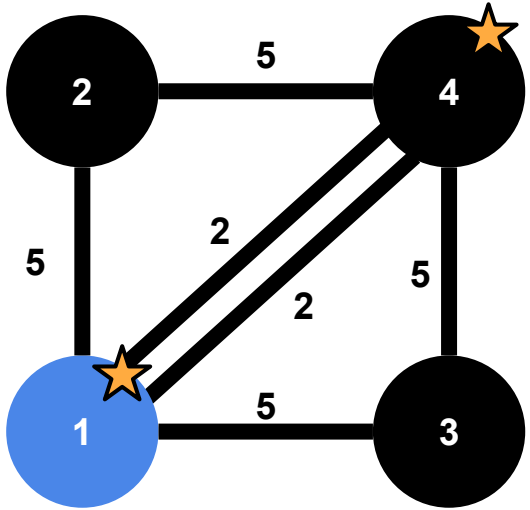
= Arbeitsweg Knoten

Lösungsansatz

- **Beobachtung 3:**

  Mehrere kürzeste Straßen können vorhanden sein

  ➢ Umweg existiert, wenn zwischen Knoten im Arbeitsweg mehrere minimale Straßen existieren



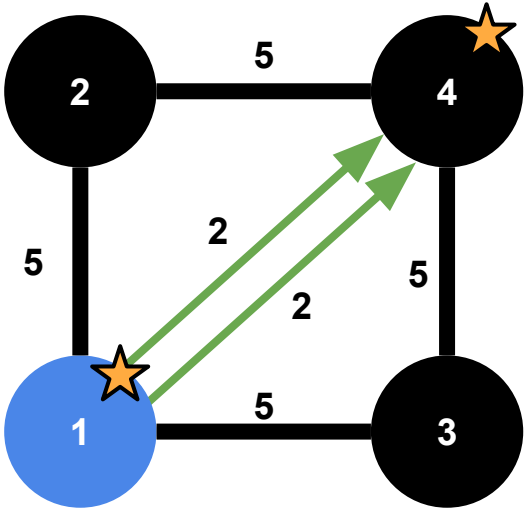
= Arbeitsweg Knoten

Lösungsansatz

- **Beobachtung 3:**

  Mehrere kürzeste Straßen können vorhanden sein

  ➢ Umweg existiert, wenn zwischen Knoten im Arbeitsweg mehrere minimale Straßen existieren



Folgerung

- **Beobachtung 1:**

  Umweg kann über komplett andere Kreuzungen gehen

- **Beobachtung 2:**

  Kürzester Weg nimmt immer die kürzesten Straßen

- **Beobachtung 3:**

  Mehrere kürzeste Straßen können vorhanden sein

Algorithmus:

- Nutze **Dijkstras Algorithmus** um **kürzesten Weg in Graphen** zu finden
- Anpassungen notwendig

Dijkstra kürzeste Wege

G	= ungerichteter Graph
V	= Knotenmenge
A	= Arbeitsweg Knoten
s	= Startknoten
Adj[]	= Adjazenzliste

DijkstraMod(G, s, A)

return “no”

Dijkstra kürzeste Wege

G	= ungerichteter Graph
V	= Knotenmenge
A	= Arbeitsweg Knoten
s	= Startknoten
Adj[]	= Adjazenzliste

```
DijkstraMod(G, s, A)
    Initialize(G, s)
    return “no”
```

Dijkstra kürzeste Wege

G	= ungerichteter Graph
V	= Knotenmenge
A	= Arbeitsweg Knoten
s	= Startknoten
Adj[]	= Adjazenzliste

```
DijkstraMod(G, s, A)
    Initialize(G, s)
    return “no”
```

```
Initialize(G, s)
    foreach Knoten u in V do
        u.distance = ∞
    s.distance = 0
```

Dijkstra kürzeste Wege

G	= ungerichteter Graph
V	= Knotenmenge
A	= Arbeitsweg Knoten
s	= Startknoten
Adj[]	= Adjazenzliste

```
DijkstraMod(G, s, A)
    Initialize(G, s)
    InitializeArbeitsweg(G, A)
    return “no”
```

```
Initialize(G, s)
    foreach Knoten u in V do
        u.distance = ∞
    s.distance = 0
```

Arbeitsweg initialisieren

- Multikanten auf Arbeitsweg prüfen (**Beobachtung 3**)
- Parallel redundante Kanten entfernen (**Beobachtung 2**)
- Vorgänger setzen
- Distanz gleich der Summe der Vorgänger und der minimalen Straße setzen

Dijkstra kürzeste Wege

G	= ungerichteter Graph
V	= Knotenmenge
A	= Arbeitsweg Knoten
s	= Startknoten
Adj[]	= Adjazenzliste

```
DijkstraMod(G, s, A)
    Initialize(G, s)
    InitializeArbeitsweg(G, A)
    return “no”
```

```
Initialize(G, s)
    foreach Knoten u in V do
        u.distance = ∞
    s.distance = 0
```

Dijkstra kürzeste Wege

```
DijkstraMod(G, s, A)
    Initialize(G, s)
    InitializeArbeitsweg(G, A)
    Q = new PriorityQueue(V)
    while not Q.empty() do
        Knoten u = Q.pop()
        foreach Knoten v in Adj[u] do
            if v.distance ≥ u.distance + weight(u,v) then
                if v.★ && v.pred != u then
                    return “ja”
                Q.update(v, v.distance)
    return “no”
```

G	= ungerichteter Graph
V	= Knotenmenge
A	= Arbeitsweg Knoten
s	= Startknoten
Adj[]	= Adjazenzliste
pop()	= Nimmt erstes Element aus Warteschlange
weight()	= Gewichtung
distance	= Distanz zu s
pred	= Vorgängerknoten
★	= Arbeitsweg Knoten

```
Initialize(G, s)
    foreach Knoten u in V do
        u.distance = ∞
    s.distance = 0
```

Dijkstra kürzeste Wege

```
DijkstraMod(G, s, A)
    Initialize(G, s)
    InitializeArbeitsweg(G, A)
    Q = new PriorityQueue(V)
    while not Q.empty() do
        Knoten u = Q.pop()
        foreach Knoten v in Adj[u] do
            if v.distance ≥ u.distance + weight(u,v) then
                if v.★ && v.pred != u then
                    return “ja”
            Q.update(v, v.distance)
    return “no”
```

G	= ungerichteter Graph
V	= Knotenmenge
A	= Arbeitsweg Knoten
s	= Startknoten
Adj[]	= Adjazenzliste
pop()	= Nimmt erstes Element aus Warteschlange
weight()	= Gewichtung
distance	= Distanz zu s
pred	= Vorgängerknoten
★	= Arbeitsweg Knoten

```
Initialize(G, s)
    foreach Knoten u in V do
        u.distance = ∞
    s.distance = 0
```

Dijkstra kürzeste Wege

```
DijkstraMod(G, s, A)
    Initialize(G, s)
    InitializeArbeitsweg(G, A)
    Q = new PriorityQueue(V)
    while not Q.empty() do
        Knoten u = Q.pop()
        foreach Knoten v in Adj[u] do
            if v.distance ≥ u.distance + weight(u,v) then
                if v.★ && v.pred != u then
                    return "ja"
            Q.update(v, v.distance)
    return "no"
```

Betrachte auch gleiche Fälle

G	= ungerichteter Graph
V	= Knotenmenge
A	= Arbeitsweg Knoten
s	= Startknoten
Adj[]	= Adjazenzliste
pop()	= Nimmt erstes Element aus Warteschlange
weight()	= Gewichtung
distance	= Distanz zu s
pred	= Vorgängerknoten
★	= Arbeitsweg Knoten

Dijkstra kürzeste Wege

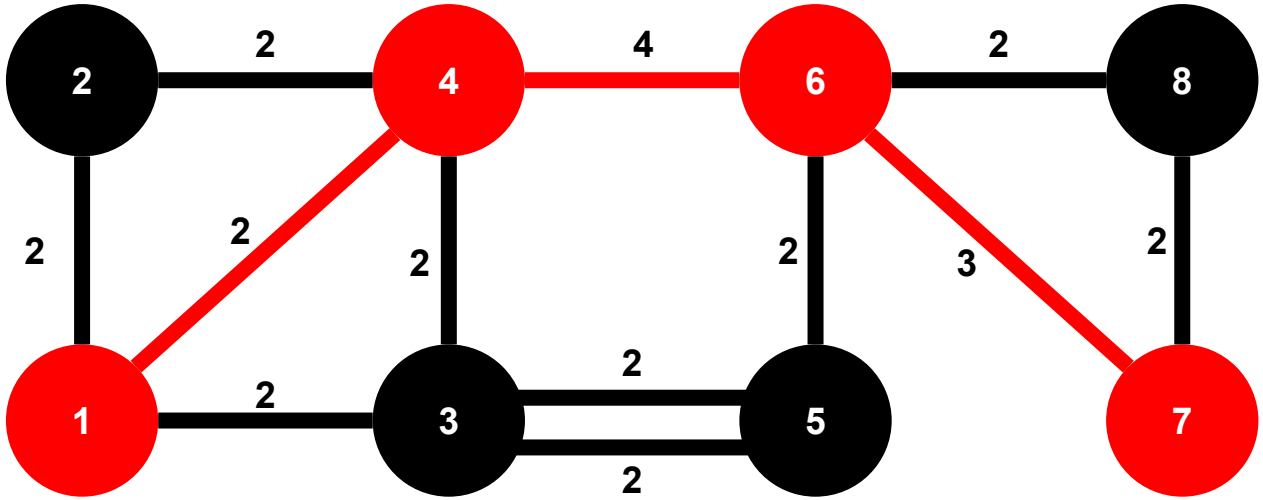
```
DijkstraMod(G, s, A)
    Initialize(G, s)
    InitializeArbeitsweg(G, A)
    Q = new PriorityQueue(V)
    while not Q.empty() do
        Knoten u = Q.pop()
        foreach Knoten v in Adj[u] do
            if v.distance ≥ u.distance + weight(u,v) then
                if v.★ && v.pred != u then
                    return "ja"
            Q.update(v, v.distance)
    return "no"
```

Betrachte auch gleiche Fälle

Abbruchbedingung, neuer Pfad wurde gefunden

G	= ungerichteter Graph
V	= Knotenmenge
A	= Arbeitsweg Knoten
s	= Startknoten
Adj[]	= Adjazenzliste
pop()	= Nimmt erstes Element aus Warteschlange
weight()	= Gewichtung
distance	= Distanz zu s
pred	= Vorgängerknoten
★	= Arbeitsweg Knoten

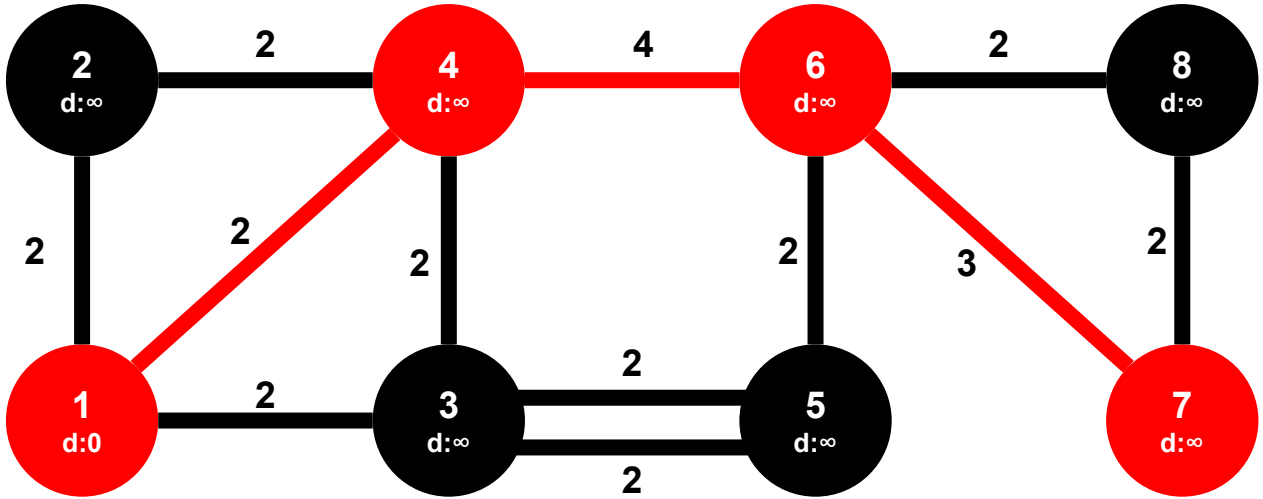
Anwendungsbeispiel



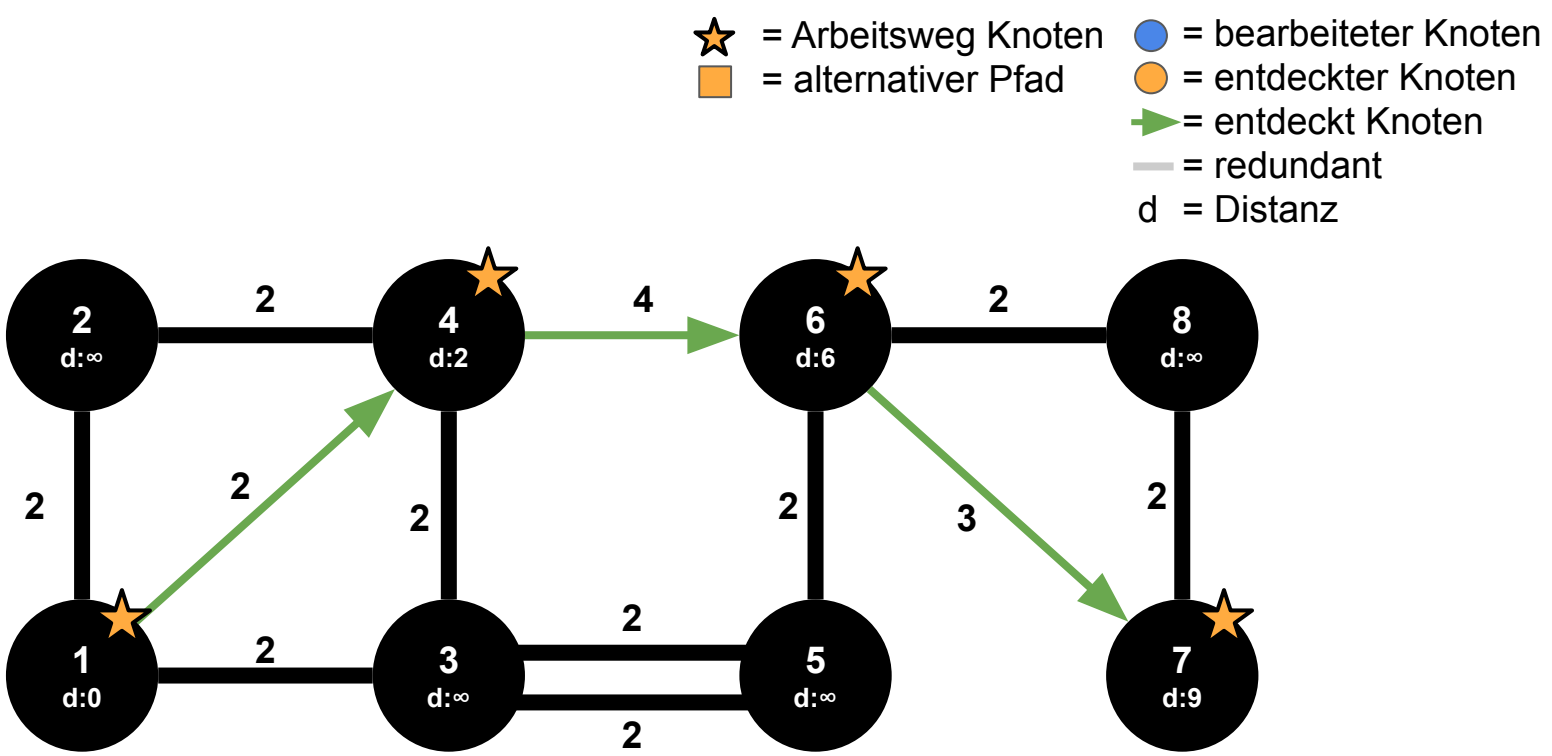
Arbeitsweg: 1,4,6,7.

d = Distanz

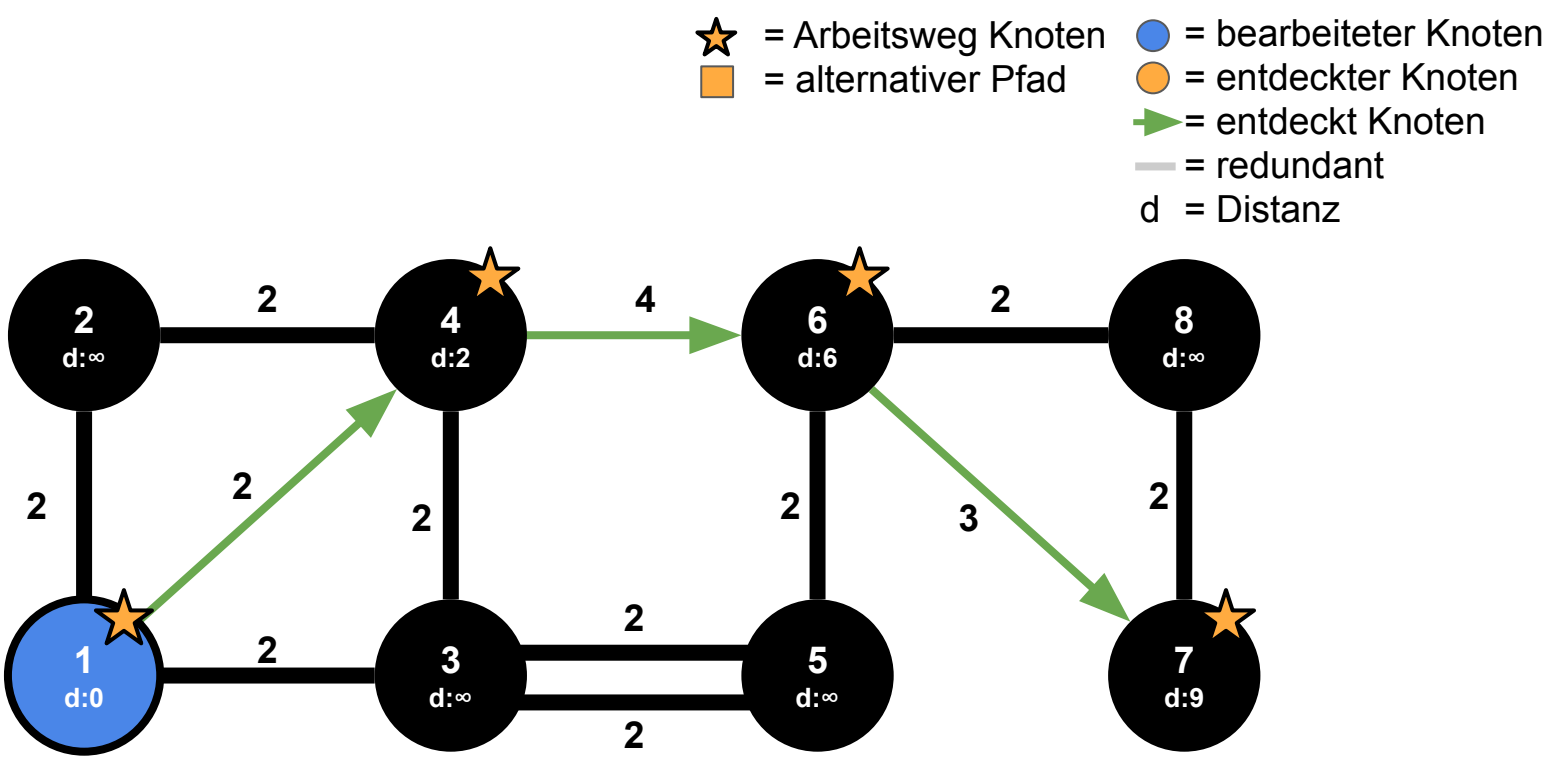
Anwendungsbeispiel



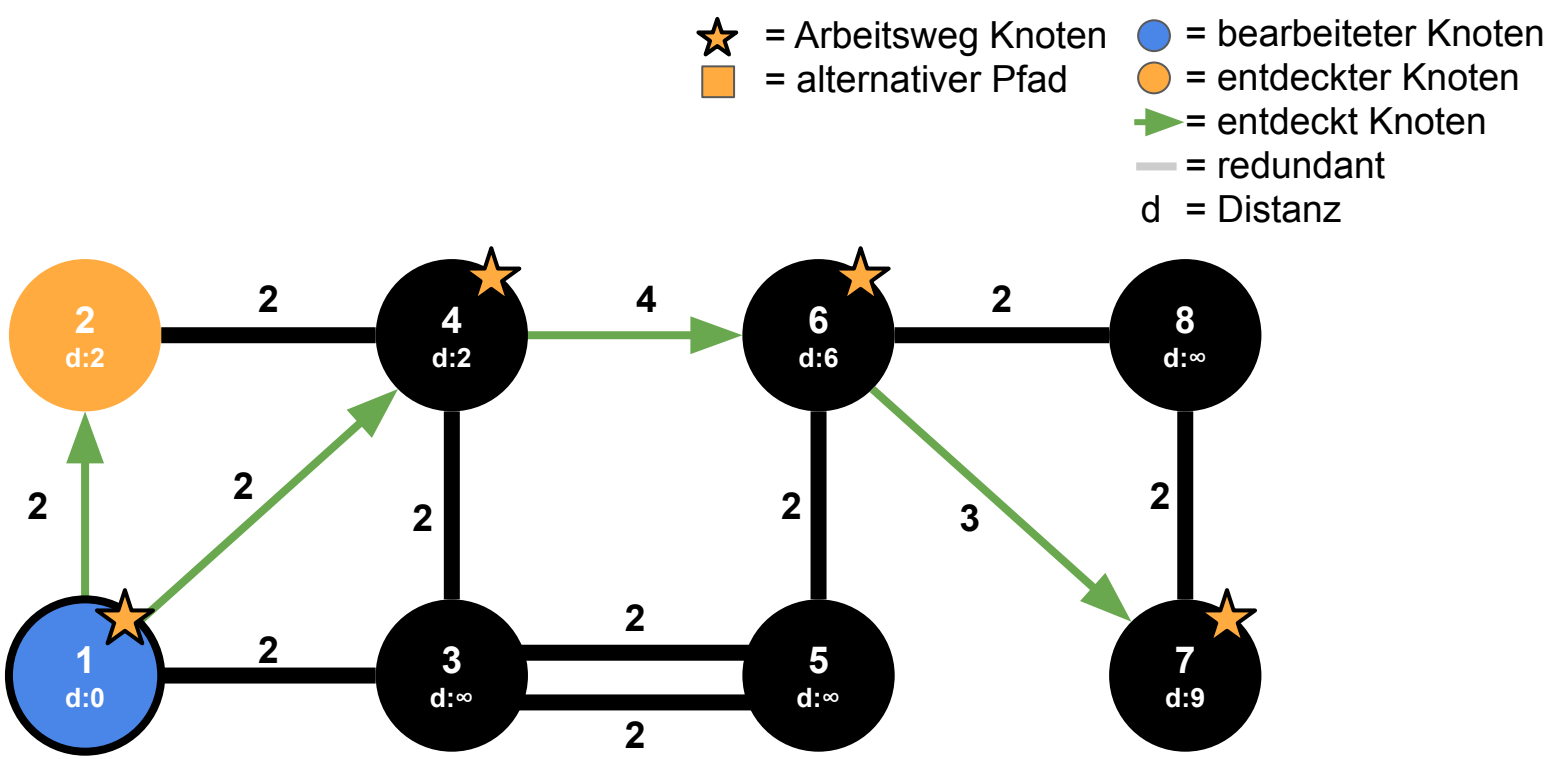
Initialize(G, 1)



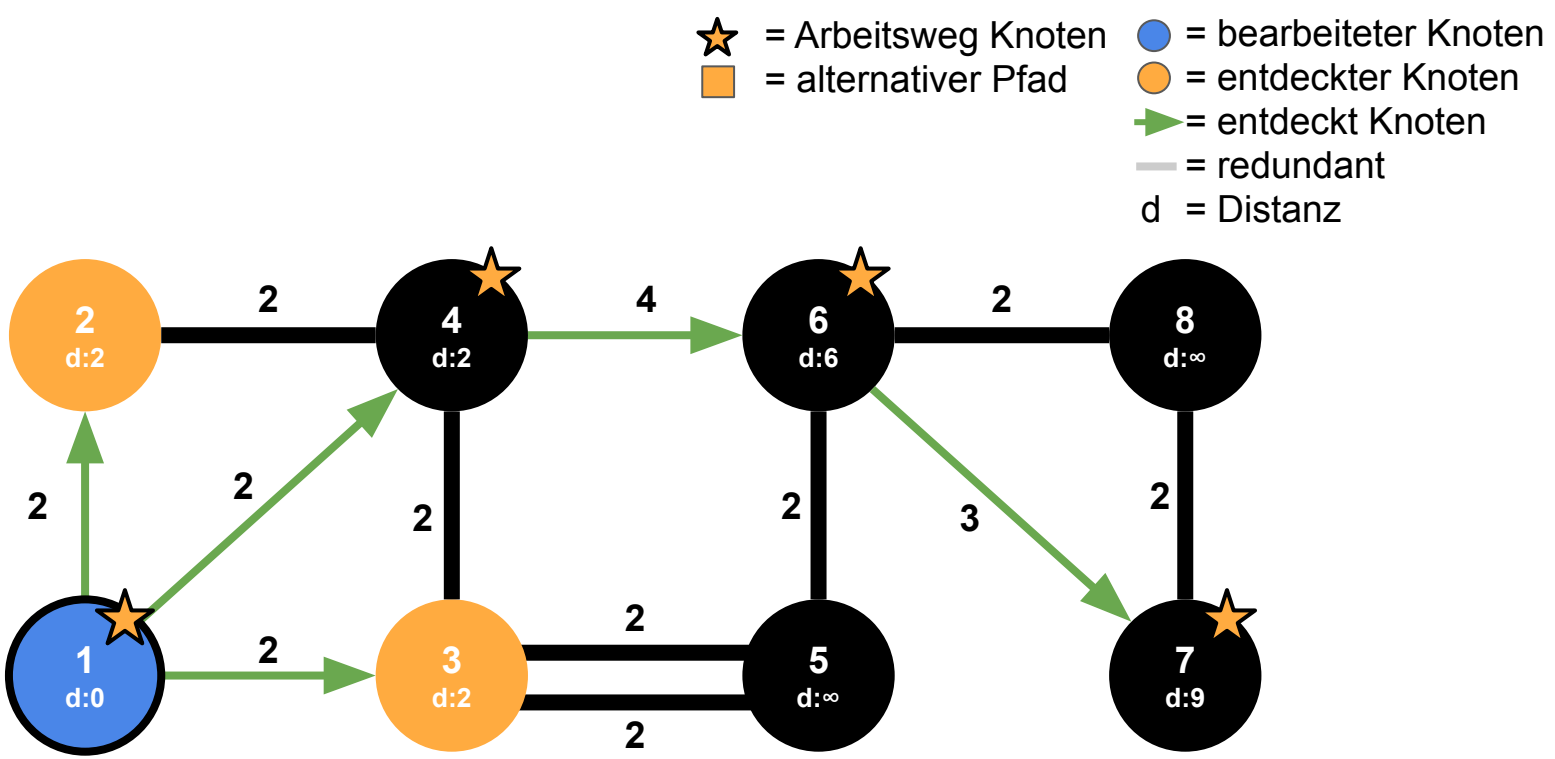
InitializeArbeitsweg(G, {1, 4, 6, 7})



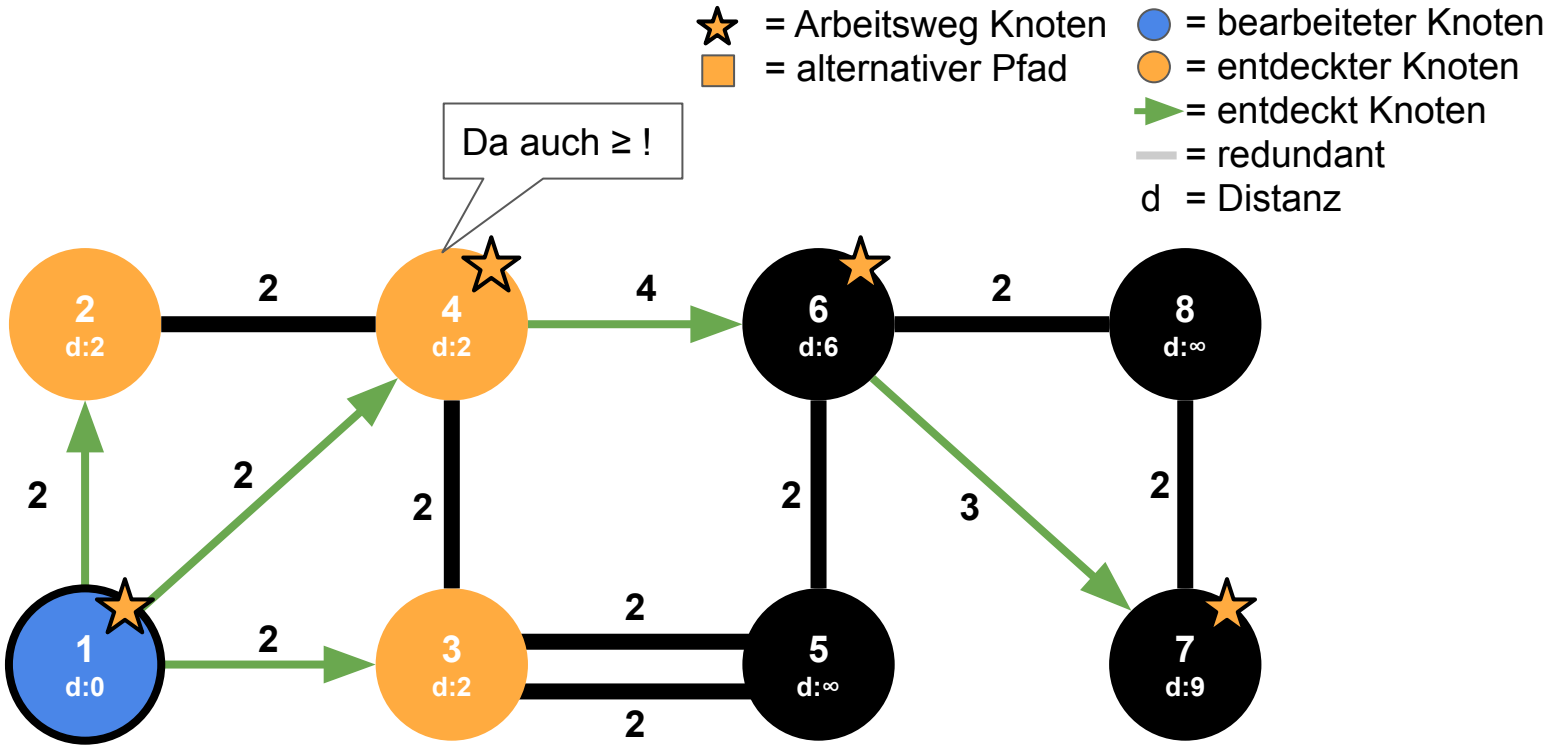
Bearbeite Knoten 1



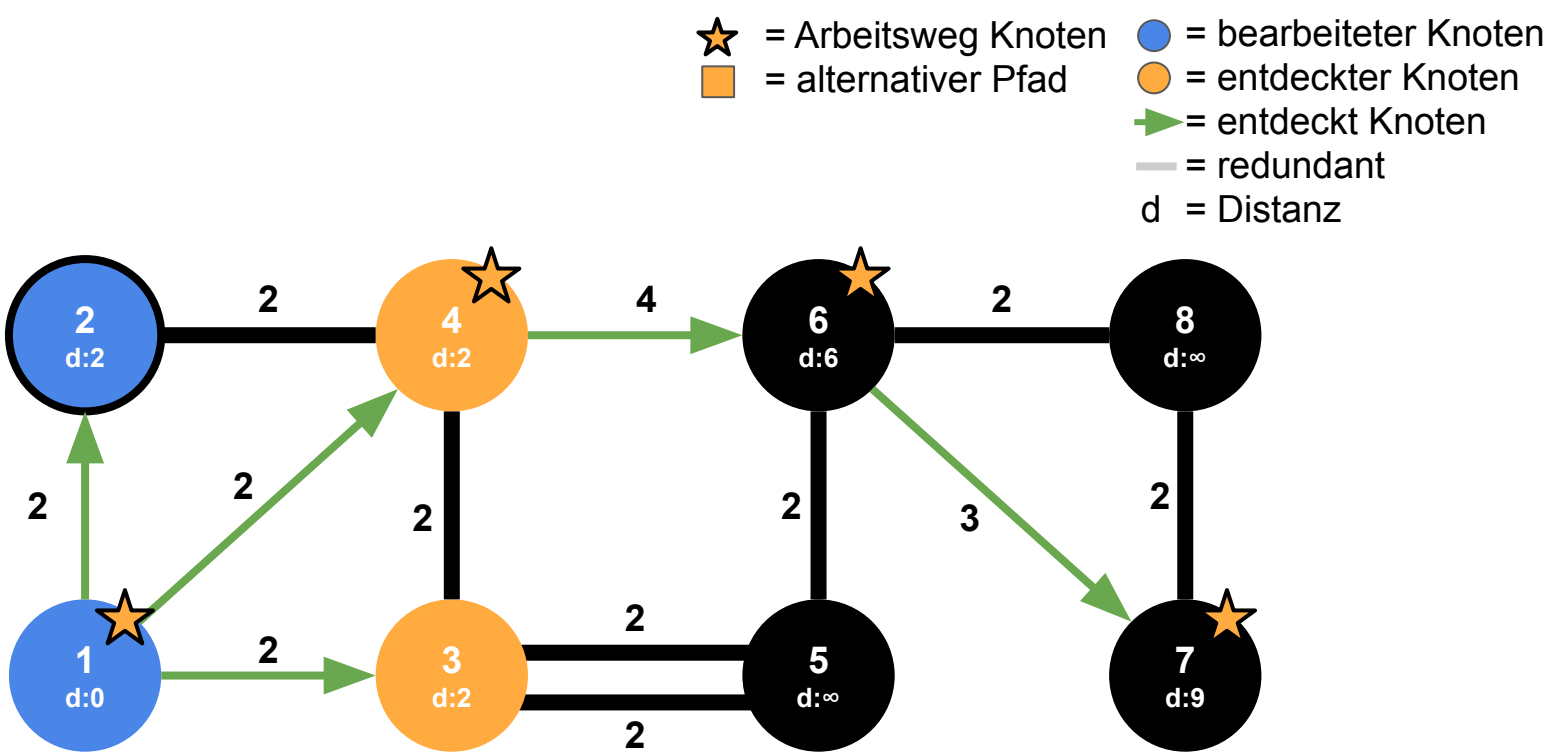
Knoten 2 entdeckt



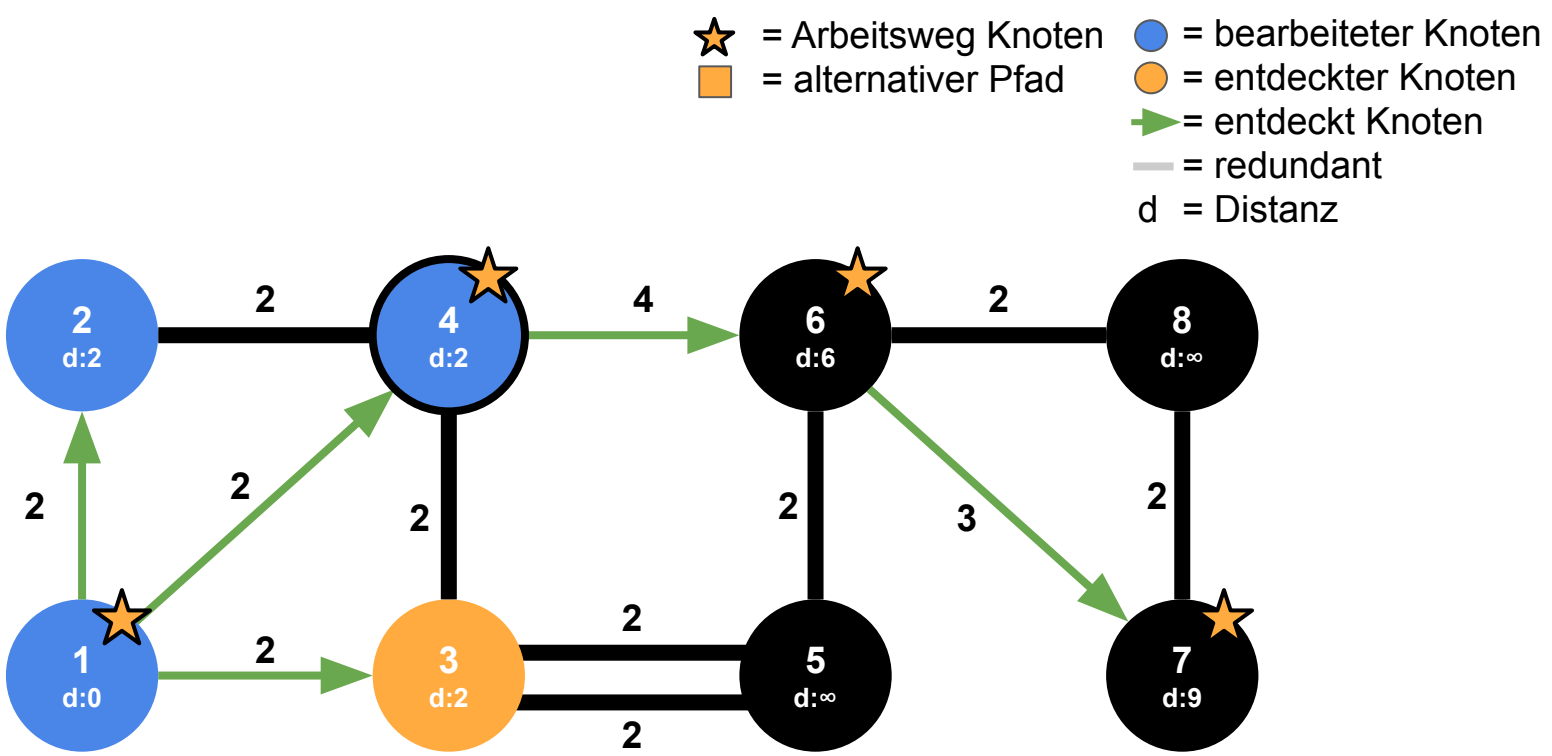
Knoten 3 entdeckt



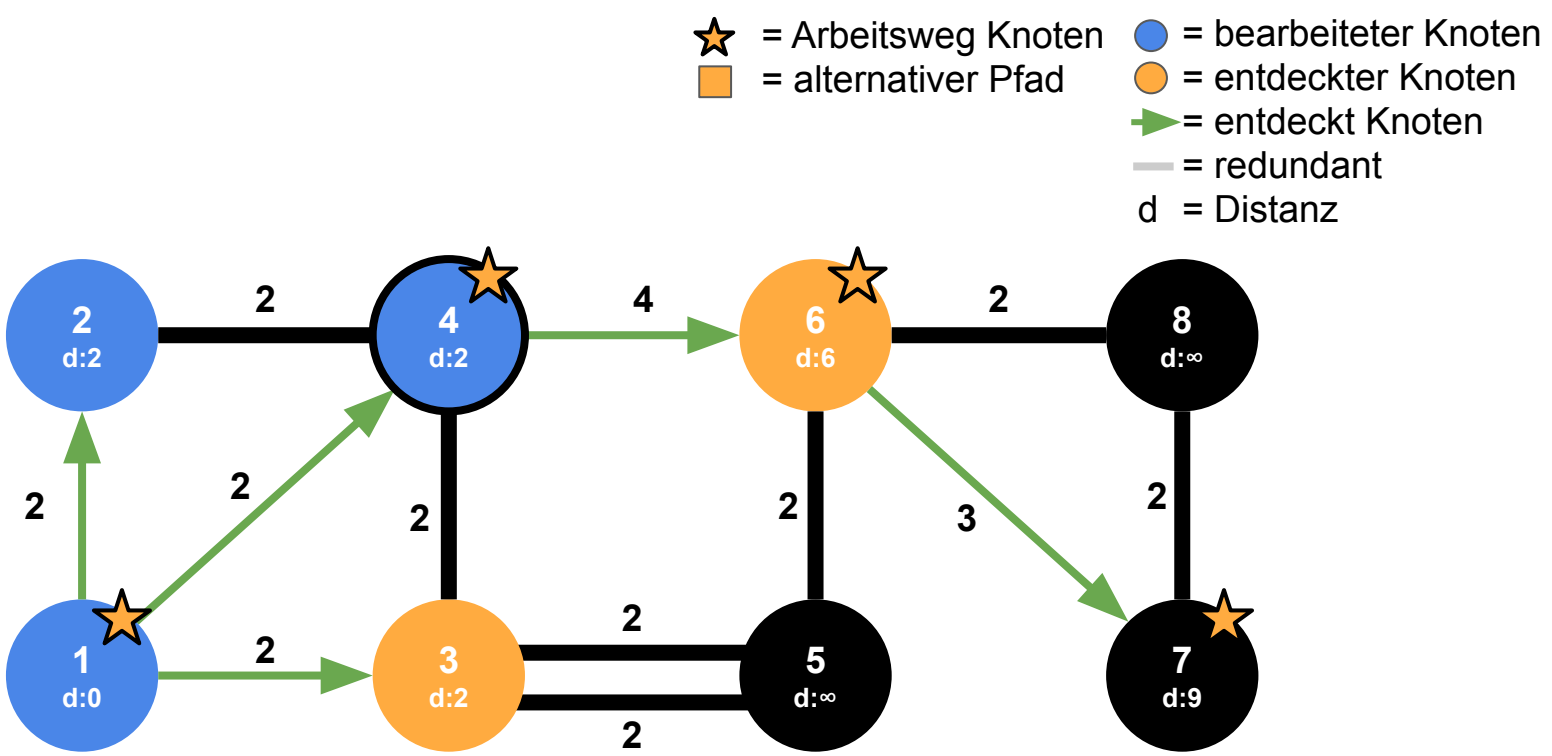
Knoten 4 entdeckt



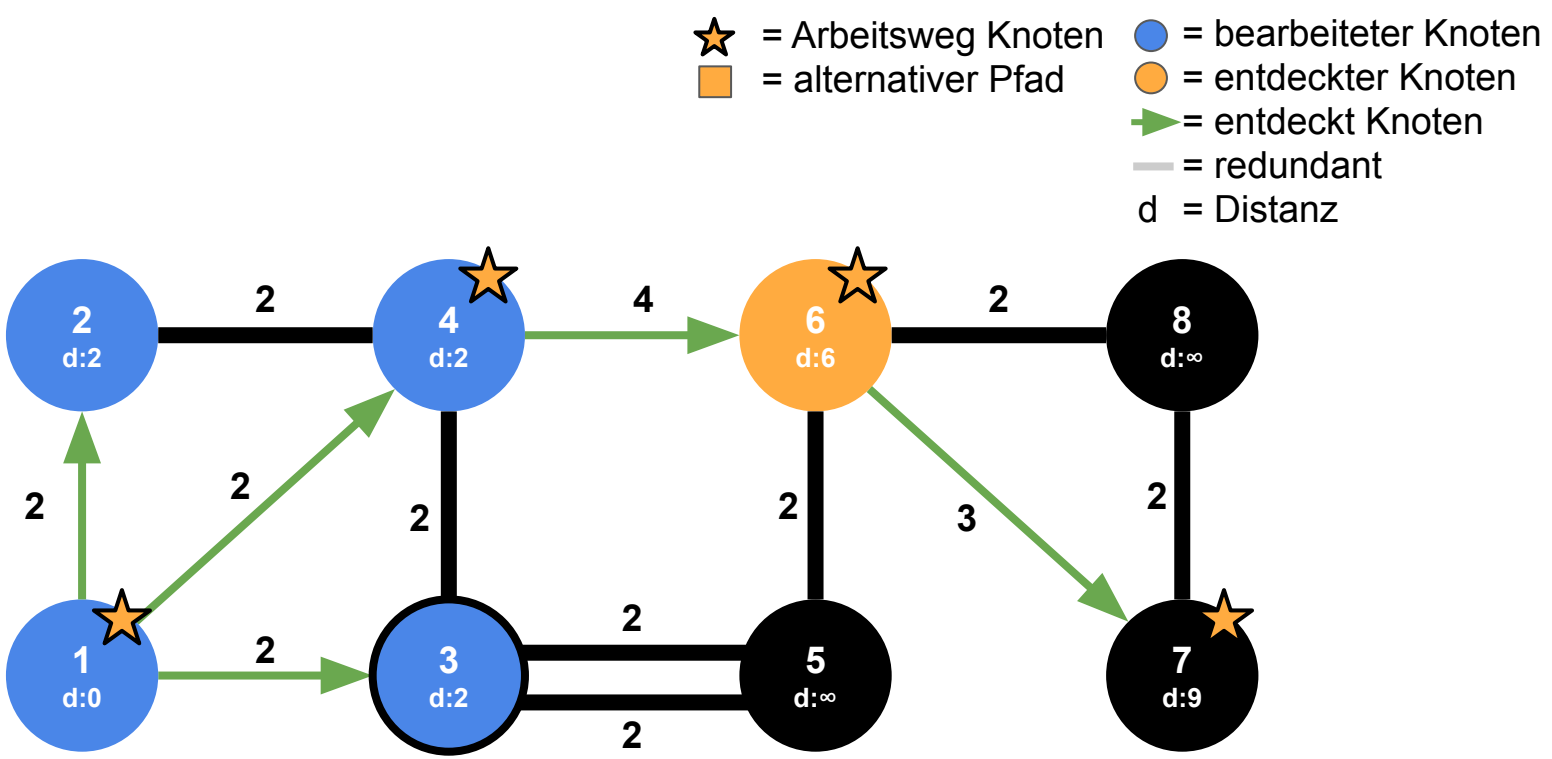
Bearbeite Knoten 2



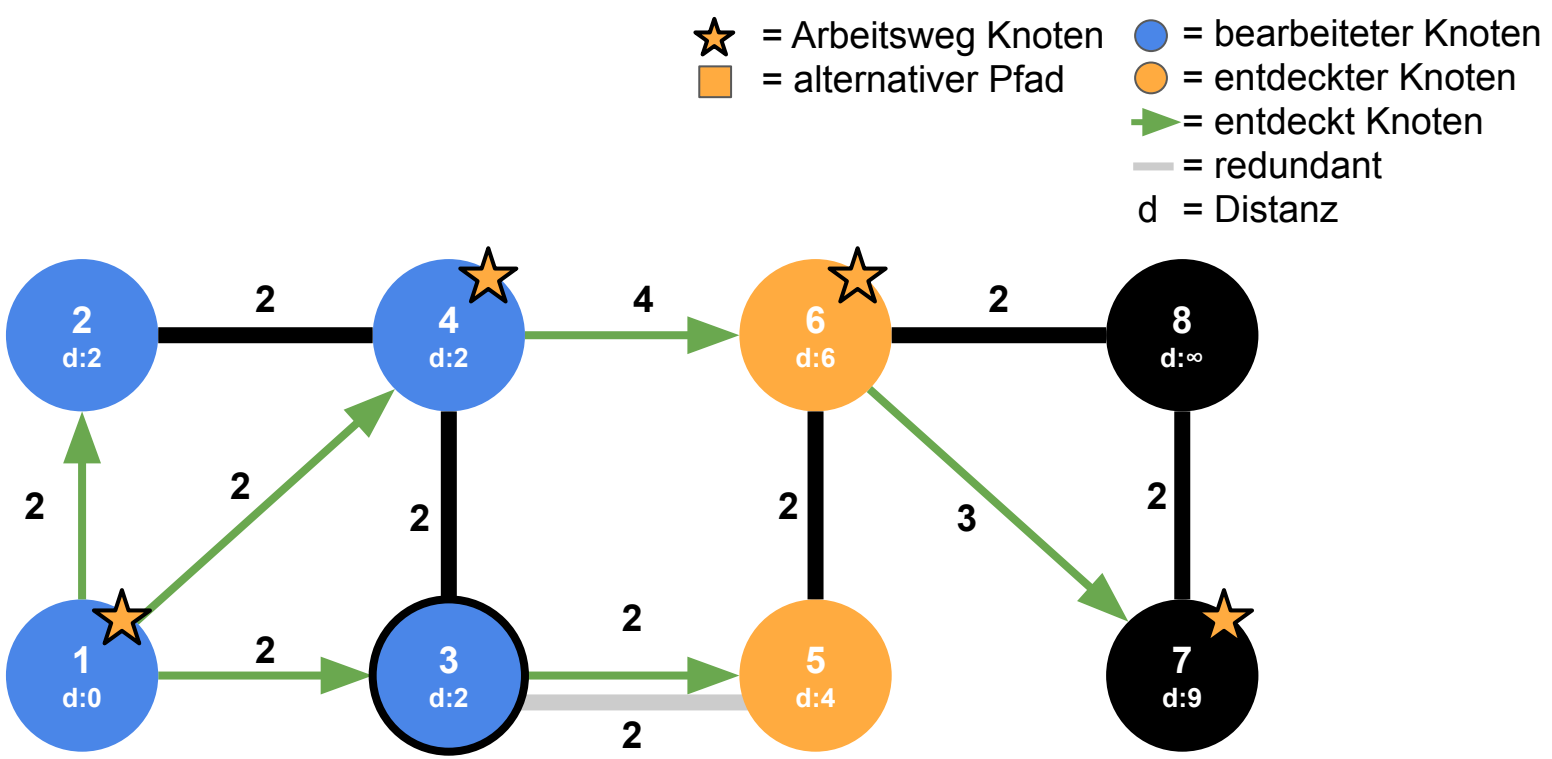
Bearbeite Knoten 4



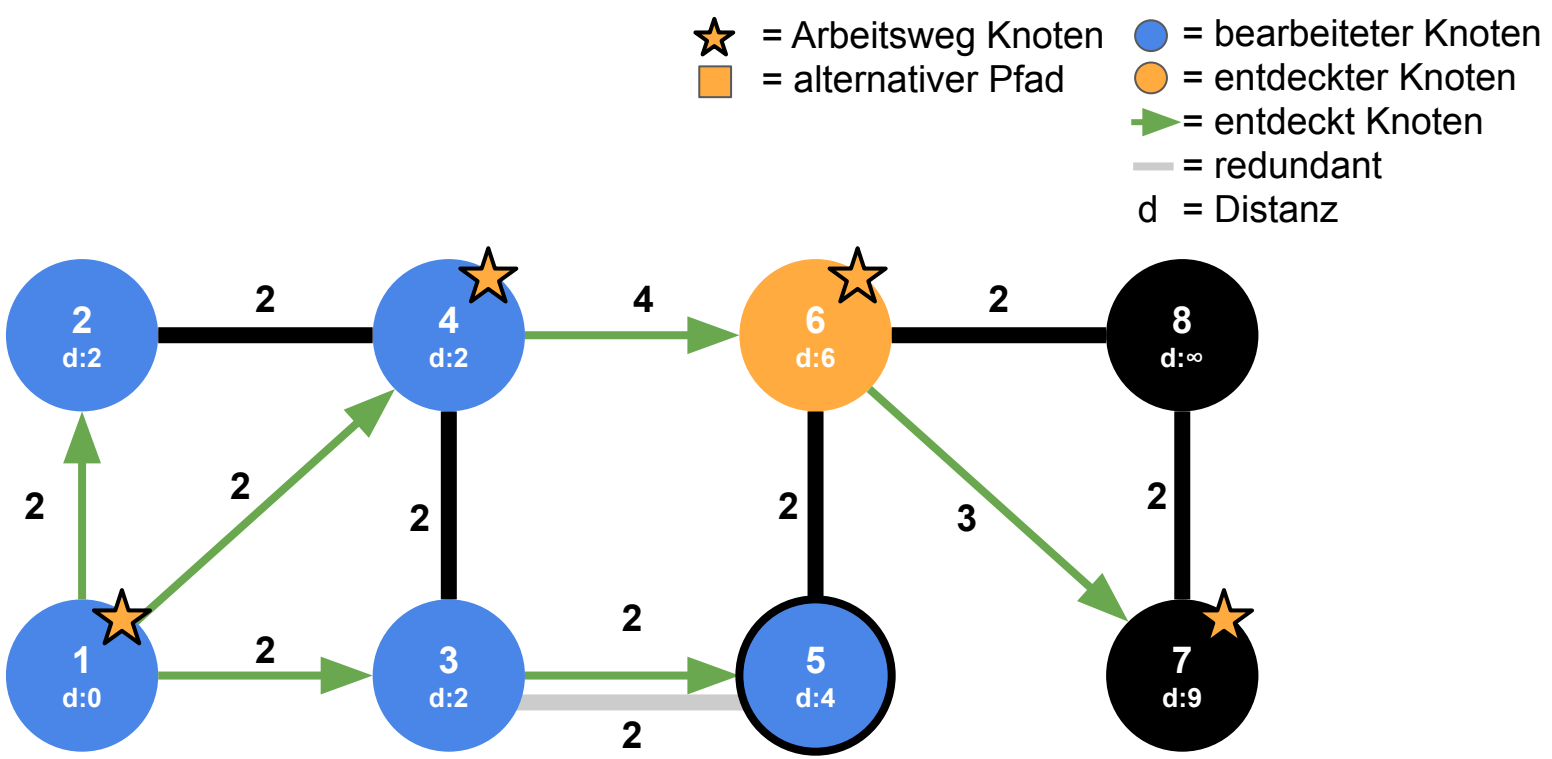
Knoten 6 entdeckt



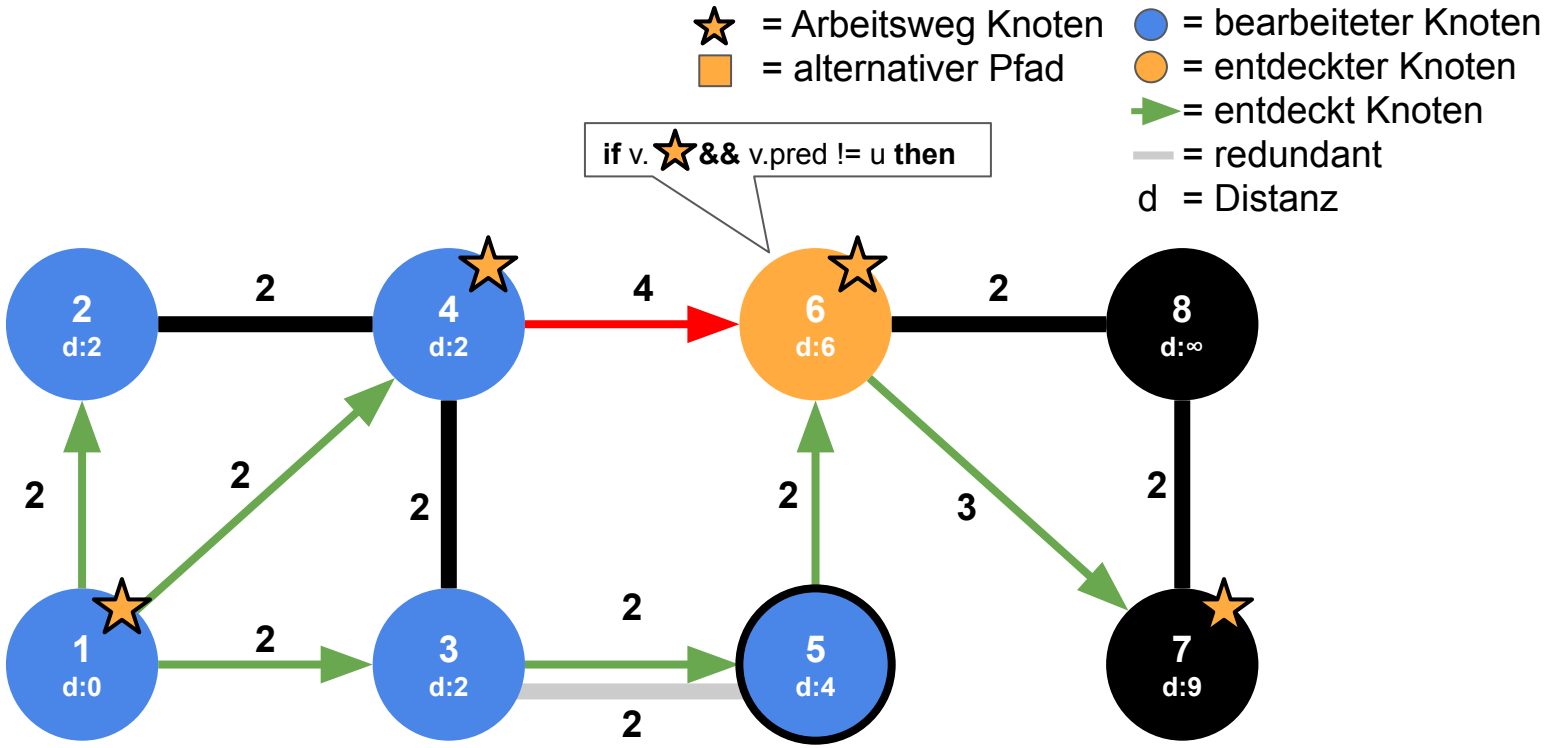
Bearbeite Knoten 3



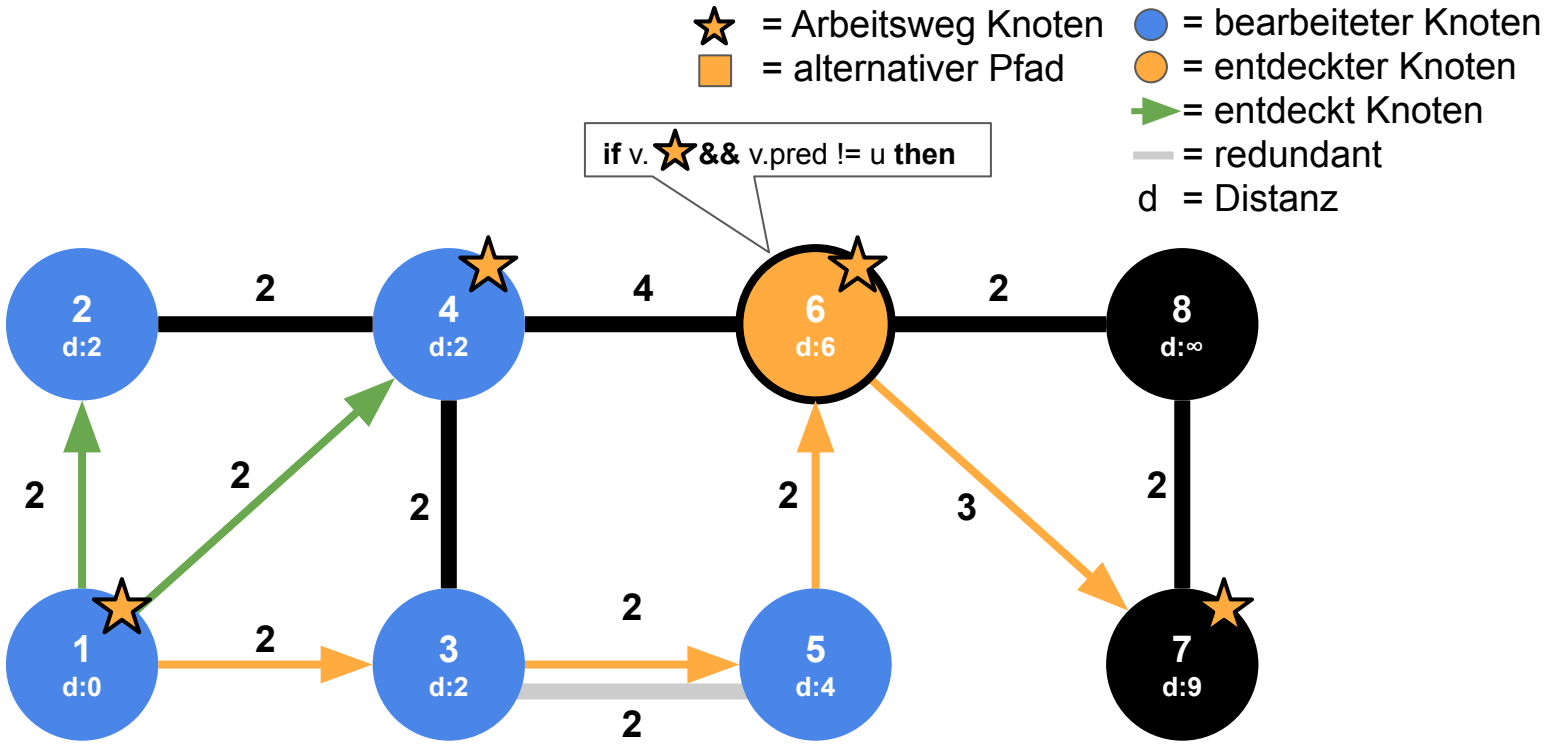
Knoten 5 entdeckt



Bearbeite Knoten 5



Knoten 6 entdeckt



Alternativer Pfad gefunden

Tipps zur Implementierung

Einlesen des Inputs

- Scanner.readLine() und String.splitt(“ “)
- sorgt für Timeout

➢ Scanner.nextInt()!

Tipps zur Implementierung

Einlesen des Inputs

- Scanner.readLine() und String.splitt(“ “)
- sorgt für Timeout

➢ Scanner.nextInt()!

Kanten

- Adjazenzmatrix nicht effizient

➢ Adjazenzliste

Laufzeitanalyse

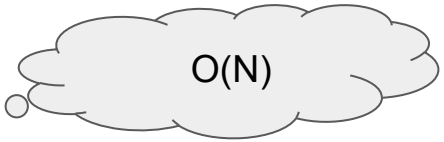
Anzahl Kreuzungen: 1 <= **N** <= 10.000
Anzahl Abreitsweg Knoten: 1 <= **K** <= 10.000
Anzahl Straßen: 0 <= **M** <= 1.000.000

```
DijkstraMod(G, s, A)
    Initialize(G, s)
    InitializeArbeitsweg(G, A)
    Q = new PriorityQueue(V)
    while not Q.empty() do
        Knoten u = Q.pop()
        foreach Knoten v in Adj[u] do
            if v.distance ≥ u.distance + weight(u,v) then
                if v.★ && v.pred != u then
                    return “ja”
                Q.update(v, v.distance)
    return “no”
```

```
Initialize(G, s)
    foreach Knoten u in V do
        u.distance = ∞
    s.distance = 0
```

Laufzeitanalyse

Anzahl Kreuzungen: 1 <= **N** <= 10.000
Anzahl Abreitsweg Knoten: 1 <= **K** <= 10.000
Anzahl Straßen: 0 <= **M** <= 1.000.000




```
DijkstraMod(G, s, A)
    Initialize(G, s)
    InitializeArbeitsweg(G, A)
    Q = new PriorityQueue(V)
    while not Q.empty() do
        Knoten u = Q.pop()
        foreach Knoten v in Adj[u] do
            if v.distance ≥ u.distance + weight(u,v) then
                if v.★ && v.pred != u then
                    return “ja”
                Q.update(v, v.distance)
    return “no”
```

```
Initialize(G, s)
    foreach Knoten u in V do
        u.distance = ∞
    s.distance = 0
```

Laufzeitanalyse

Anzahl Kreuzungen: 1 <= **N** <= 10.000
Anzahl Abreitsweg Knoten: 1 <= **K** <= 10.000
Anzahl Straßen: 0 <= **M** <= 1.000.000

```
DijkstraMod(G, s, A)
    Initialize(G, s)
    InitializeArbeitsweg(G, A)
    Q = new PriorityQueue(V)
    while not Q.empty() do
        Knoten u = Q.pop()
        foreach Knoten v in Adj[u] do
            if v.distance ≥ u.distance + weight(u,v) then
                if v.★ && v.pred != u then
                    return “ja”
                Q.update(v, v.distance)
    return “no”
```


```
Initialize(G, s)
    foreach Knoten u in V do
        u.distance = ∞
    s.distance = 0
```

InitializeArbeitsweg

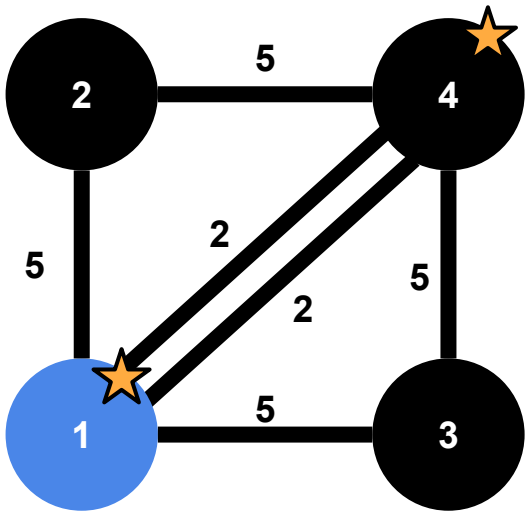
Anzahl Kreuzungen: 1 <= **N** <= 10.000
Anzahl Abreitsweg Knoten: 1 <= **K** <= 10.000
Anzahl Straßen: 0 <= **M** <= 1.000.000

- maximaler |Arbeitsweg| ist **N**
- maximal **M** Kanten

InitializeArbeitsweg

Anzahl Kreuzungen: 1 <= **N** <= 10.000
Anzahl Abreitsweg Knoten: 1 <= **K** <= 10.000
Anzahl Straßen: 0 <= **M** <= 1.000.000

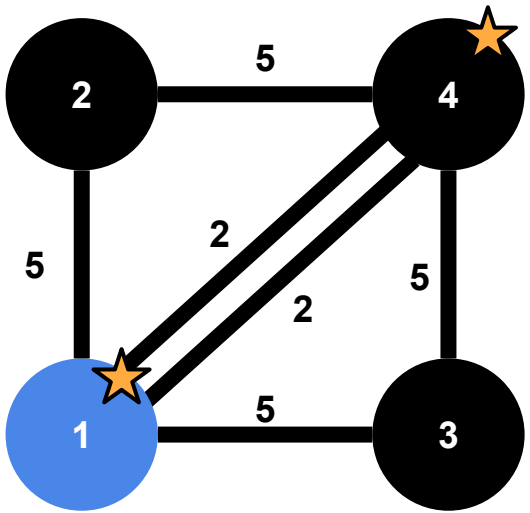
- maximaler |Arbeitsweg| ist **N**
- maximal **M** Kanten



InitializeArbeitsweg

Anzahl Kreuzungen: 1 <= **N** <= 10.000
Anzahl Abreitsweg Knoten: 1 <= **K** <= 10.000
Anzahl Straßen: 0 <= **M** <= 1.000.000

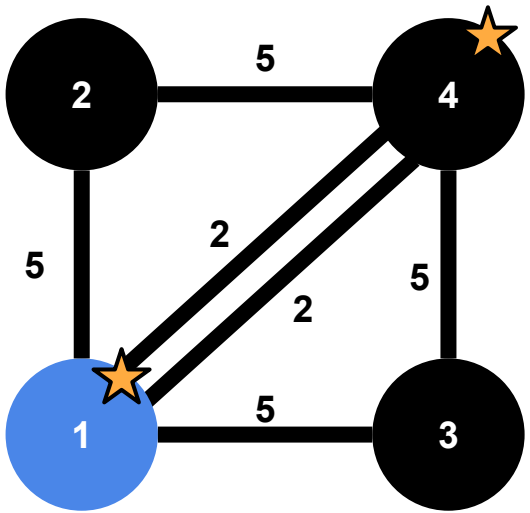
- maximaler |Arbeitsweg| ist **N**
- maximal **M** Kanten
  - ➢ **O(M)**



InitializeArbeitsweg

Anzahl Kreuzungen: 1 <= **N** <= 10.000
Anzahl Abreitsweg Knoten: 1 <= **K** <= 10.000
Anzahl Straßen: 0 <= **M** <= 1.000.000

- maximaler |Arbeitsweg| ist **N**
- maximal **M** Kanten
  - ➢ **O(M)**



- Alle redundanten Kanten entfernen
  - O(max{N², M})
  - nicht effizient

Laufzeitanalyse

Anzahl Kreuzungen: 1 <= **N** <= 10.000
Anzahl Abreitsweg Knoten: 1 <= **K** <= 10.000
Anzahl Straßen: 0 <= **M** <= 1.000.000

```
DijkstraMod(G, s, A)
    Initialize(G, s)
    InitializeArbeitsweg(G, A)
    Q = new PriorityQueue(V)
    while not Q.empty() do
        Knoten u = Q.pop()
        foreach Knoten v in Adj[u] do
            if v.distance ≥ u.distance + weight(u,v) then
                if v.★ && v.pred != u then
                    return “ja”
                Q.update(v, v.distance)
    return “no”
```

O(M log N)

```
Initialize(G, s)
    foreach Knoten u in V do
        u.distance = ∞
    s.distance = 0
```

Laufzeit

- Input als Graph einlesen:
  - O(M)
- InitializeArbeitsweg(G, A):
  - O(M)
- DijkstraMod(G, s, A):
  - O(M log N) //bei unserer Implementierung
  - Fibonacci-Heap O(M + N log N)

Anzahl Kreuzungen: 1 <= N <= 10.000

Anzahl Abreitsweg Knoten: 1 <= K <= 10.000

Anzahl Straßen: 0 <= M <= 1.000.000