

# Praktikum Routenplanung

Vorbesprechung, Wintersemester 2022/2023

Adrian Feilhauer, Jonas Sauer, Michael Zündorf | 26. Oktober 2022



# Organisatorisches

## Praktikum

- Erste Phase: 1 Übungsblatt mit 4 Aufgaben lösen
- Zweite Phase: Große Aufgabe in 3er-Gruppen
- Betreuer: Adrian Feilhauer, Jonas Sauer, Michael Zündorf
- Email: {adrian.feilhauer, jonas.sauer2, michael.zuendorf}@kit.edu
- 6 LP/ECTS
- Bei Fragen einfach vorbei kommen

Homepage: <https://i11www.itl.kit.edu/teaching/winter2022/algorithmengineeringpraktikum/>

# Organisatorisches

## Voraussetzungen

- Ihr seid im Master Informatik, **alle anderen bitte melden**

# Organisatorisches

## Voraussetzungen

- Ihr seid im Master Informatik, **alle anderen bitte melden**
- Ihr habt Interesse an algorithmischen Fragestellungen
- Ihr mögt Algorithmen implementieren

# Organisatorisches

## Voraussetzungen

- Ihr seid im Master Informatik, **alle anderen bitte melden**
- Ihr habt Interesse an algorithmischen Fragestellungen
- Ihr mögt Algorithmen implementieren

## Übersicht

- Übungsaufgabe
- Anfangsvortrag
- Gruppenaufgabe
- Abschlussvortrag
- Ausarbeitung

# Organisatorisches

## Voraussetzungen

- Ihr seid im Master Informatik, **alle anderen bitte melden**
- Ihr habt Interesse an algorithmischen Fragestellungen
- Ihr mögt Algorithmen implementieren

## Übersicht

- Übungsaufgabe
- Anfangsvortrag
- Gruppenaufgabe
- Abschlussvortrag
- Ausarbeitung

## Implementierung

- C++
- oder Rust

# Organisatorisches

## Übungsblatt

- Es werden Punkte vergeben
  - Punkte gehen nicht in die Endnote ein
  - 2,5 Mio. Punkte müssen erreicht werden, um zu bestehen
- Gruppen werden nach Punktzahl gebildet
- Gruppe mit den meisten Punkten darf sich Gruppenarbeitsthema zuerst aussuchen
- Nach Übungsblatt: formale Prüfungsanmeldung  
d.h. ab da: nichts gemacht → durchgefallen
- Gruppenarbeit ist schwerer als Übungsblatt

# Benotung Übungsblätter

- Pro Aufgabe: Liste an Start- und Zielknotenpaaren
- Ihr soll die Pfadlänge berechnen
- Punkte einer Aufgabe = #Korrekt berechnete Pfadlängen

## Bestehen

Es müssen 2,5 Mio. Punkte erreicht werden, um zu bestehen!

# Hilfestellung Übungsblätter

- Bei Fragen oder Problemen könnt ihr euch gerne an uns wenden
- Falls ihr danach fragt, können wir gerne Feedback zu eurem Code geben

# Hilfestellung Übungsblätter

- Bei Fragen oder Problemen könnt ihr euch gerne an uns wenden
- Falls ihr danach fragt, können wir gerne Feedback zu eurem Code geben
  
- Allerdings: **Eigeninitiative erwünscht!**
- Es ist eure Aufgabe, bei Problemen auf einen der Betreuer zuzugehen
- Wer nicht fragt, der kriegt keine Hilfe

# Organisatorisches

## Gruppenarbeit

- Bearbeitung in 3er-Gruppe
- Aufgabe: Nachimplementieren eines Forschungspapers
  - Jede Gruppe hat ein anderes Paper
- Visualisierung der Ergebnisse
- Einige Experimente aus dem Paper wiederholen
- Einige neue Experimente entwerfen und durchführen

## Einteilung und Themen

- Gruppeneinteilung nach Übungsblatt
- Themenvorstellung bei Gruppeneinteilung

# Organisatorisches

## Anfangsvortrag

- 10 Minuten
- Problemstellung und den Kernansatz erklären

## Ausarbeitung

- Alles, was ihr implementiert habt, in eigenen Worten beschreiben
- Experimente und Ergebnisse dokumentieren

## Abschlussvortrag

- 20–30 Minuten
- Inhalte der Ausarbeitung vorstellen

# Aufwand

## Aufwand

- 6 ECTS/LP
- $6 \times 30\text{h} = 180\text{h}$
- Bei 20 Wochen: 9h pro Woche,  
also etwas mehr als 1 Tag Vollzeit pro Woche

## Grobe Verteilung

- 35h Übungsblatt
- 95h Gruppenaufgabe, inklusive
  - Einarbeitung ins Thema
  - Implementierung
- 5h Kurzvortrag
- 20h Abschlussvortrag
- 20h Ausarbeitung
- 5h Anwesenheit

	<b>Wann?</b>	<b>Wo?</b>	<b>Was?</b>
Heute	26.10. um 14:00	SR -120	Vorbesprechung
	22.11. 23:59	—	Abgabe Übungsblatt
	23.11.	—	Punktevergabe per E-Mail
	23.11. um 14:00	SR -120	Themen & Gruppeneinteilung
	7.12. um 14:00	SR -120	Anfangsvorträge
	23.–27.1.	—	Zwischentreffen
	8.3.	—	Abgabe Ausarbeitung
	22.3. um 14:00	SR -120	Abschlussvorträge

Es gilt Anwesenheitspflicht. Wer nicht kommen kann, muss sich mit Begründung abmelden.

# Problemstellung

## Gesucht:

- Finde die **beste** Verbindung in einem Transportnetzwerk

## Idee:

- Netzwerk als Graph  $G = (V, E)$
- Pfad durch Graph entspricht Route
- klassisches Problem (Dijkstra)

## Probleme:

- Transportnetzwerke sind **groß**
- Dijkstra zu **langsam** ( $> 1$  Sekunde)



# Problemstellung

## Gesucht:

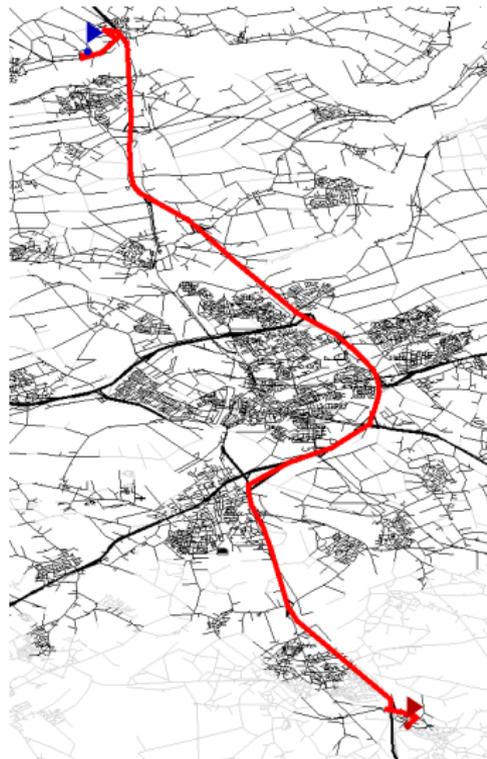
- Finde die **beste** Verbindung in einem Transportnetzwerk

## Idee:

- Netzwerk als Graph  $G = (V, E)$
- Pfad durch Graph entspricht Route
- klassisches Problem (Dijkstra)

## Probleme:

- Transportnetzwerke sind **groß**
- Dijkstra zu **langsam** ( $> 1$  Sekunde)



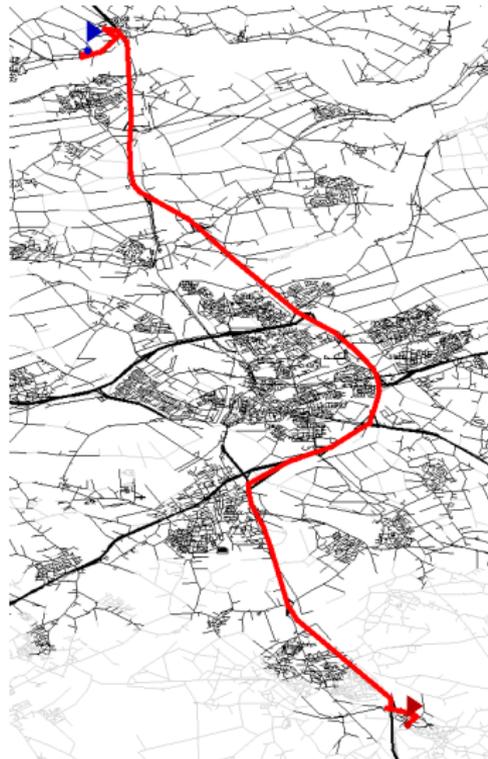
# Beschleunigungstechniken

## Beobachtungen:

- viele Anfragen in (statischem) Netzwerk
- manche Berechnungen scheinen **unnötig**

## Idee:

- Zwei-Phasen-Algorithmus:
  - offline: berechne Zusatzinformation während **Vorbereitung**
  - online: **beschleunige** Berechnung mit diesen Zusatzinformationen
- drei Kriterien:
  - wenig Zusatzinformation
  - kurze Vorbereitung (im Bereich Stunden/Minuten)
  - hohe Beschleunigung





# Modellierung (Straßengraphen)

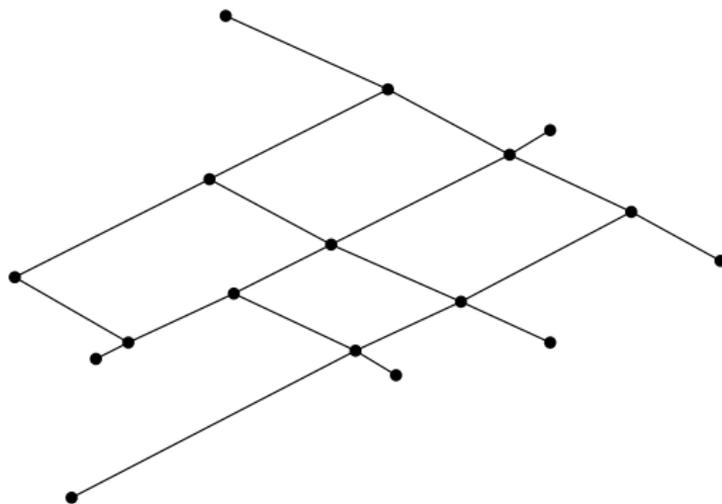


# Modellierung (Straßengraphen)



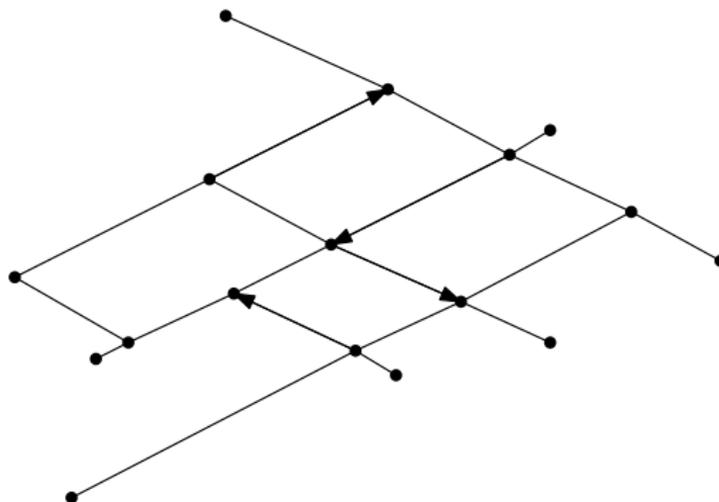
# Modellierung (Straßengraphen)

- Knoten sind Kreuzungen
- Kanten sind Straßen



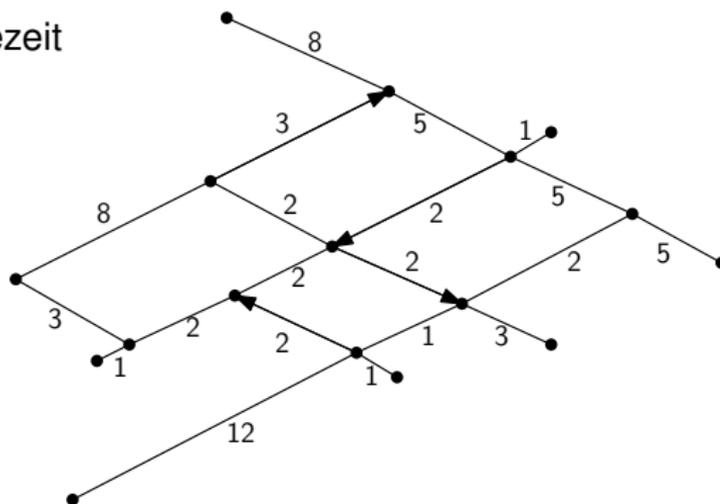
# Modellierung (Straßengraphen)

- Knoten sind Kreuzungen
- Kanten sind Straßen
- Einbahnstraßen

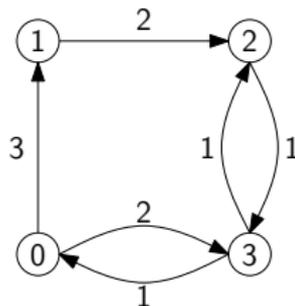


# Modellierung (Straßengraphen)

- Knoten sind Kreuzungen
- Kanten sind Straßen
- Einbahnstraßen
- Metrik ist Reisezeit



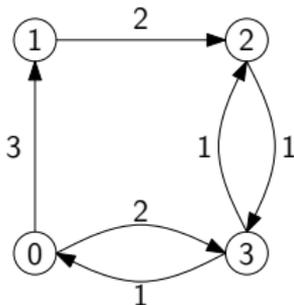
# Graph-Repräsentationen



# Graph-Repräsentationen

## Drei klassische Ansätze:

- Adjazenzmatrix

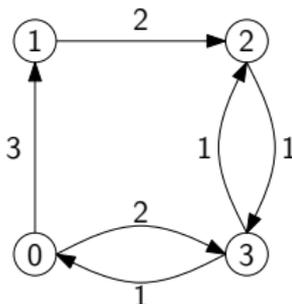


	0	1	2	3
0	—	3	—	2
1	—	—	2	—
2	—	—	—	1
3	1	—	1	—

# Graph-Repräsentationen

## Drei klassische Ansätze:

- (statisches) Adjazenzarray

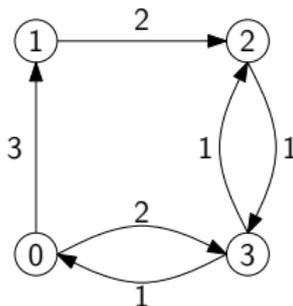


first_out	0	2	3	4	6
head	1	3	2	3	2
weight	3	2	2	1	1

# Graph-Repräsentationen

Drei klassische Ansätze:

- Kantenarray



tail	1	0	3	0	2	3
head	2	1	2	3	3	0
weight	2	3	1	2	1	1

# Was benutzen wir?

Adjazenzmatrix:

- Braucht  $O(n^2)$  Speicher
- $n = 18 \cdot 10^6$
- Speicher  $\geq 1/4$  Terabyte
- Impraktikabel

# Was benutzen wir?

Adjazenzmatrix:

- Braucht  $O(n^2)$  Speicher
- $n = 18 \cdot 10^6$
- Speicher  $\geq 1/4$  Terabyte
- Impraktikabel

Kantenarray:

- Perfekt für einfache Transformationen (z.B. Graph umdrehen)
- Traversieren (d. h. Pfadsuche) geht nicht

# Was benutzen wir?

Adjazenzmatrix:

- Braucht  $O(n^2)$  Speicher
- $n = 18 \cdot 10^6$
- Speicher  $\geq 1/4$  Terabyte
- Impraktikabel

Kantenarray:

- Perfekt für einfache Transformationen (z.B. Graph umdrehen)
- Traversieren (d. h. Pfadsuche) geht nicht

Adjazenzarray:

- Gut wenn man Pfade suchen will

# Konvertierung Kantenarray $\rightarrow$ Adjazenzarray

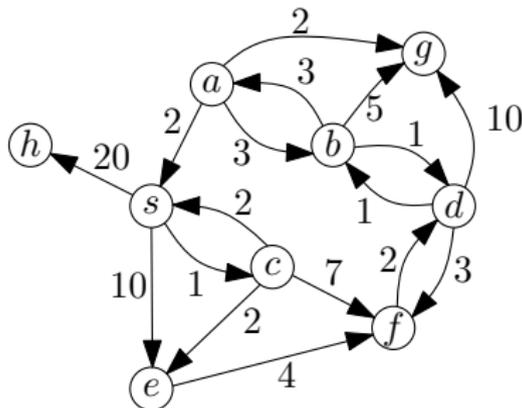
- Nach tail sortieren
- Ausgangsgrad jedes Knotens berechnen
- `first_out` = Präfixsumme über Array der Ausgangsgrade

# Dijkstras Algorithmus

---

```
1 forall nodes  $v \in V$  do
2    $d[v] = \infty$ 
3  $d[s] = 0$ 
4  $q.clear()$ 
5  $q.insert(s, 0)$ 
6 while  $!q.empty()$  do
7    $x \leftarrow q.pop()$ 
8   forall edges  $(x, y) \in E$  do
9     if  $d[x] + \text{len}(x, y) < d[y]$  then
10       $d[y] \leftarrow d[x] + \text{len}(x, y)$ 
11      if  $y \in q$  then  $q.decreaseKey(y, d[y])$ 
12      else  $q.insert(y, d[y])$ 
```

# Dijkstras Algorithmus



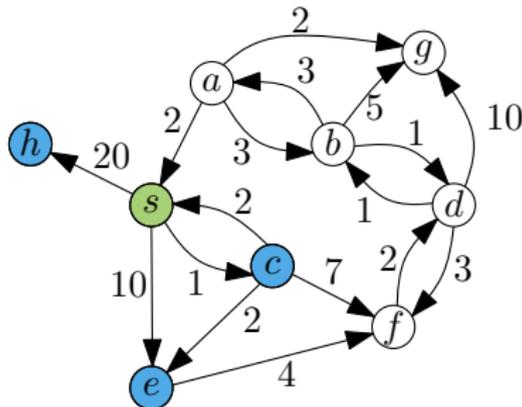
tentative distance  $d$ :

ID	Dist.
s	0
a	$\infty$
b	$\infty$
c	$\infty$
d	$\infty$
e	$\infty$
f	$\infty$
g	$\infty$
h	$\infty$

queue  $q$ :

ID	Key
s	0

# Dijkstras Algorithmus



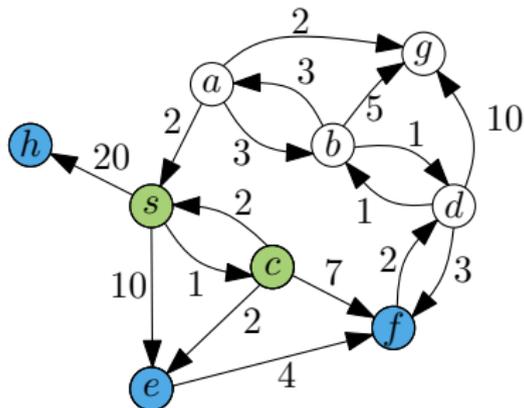
tentative distance  $d$ :

ID	Dist.
s	0
a	$\infty$
b	$\infty$
c	1
d	$\infty$
e	10
f	$\infty$
g	$\infty$
h	20

queue  $q$ :

ID	Key
c	1
e	10
h	20

# Dijkstras Algorithmus



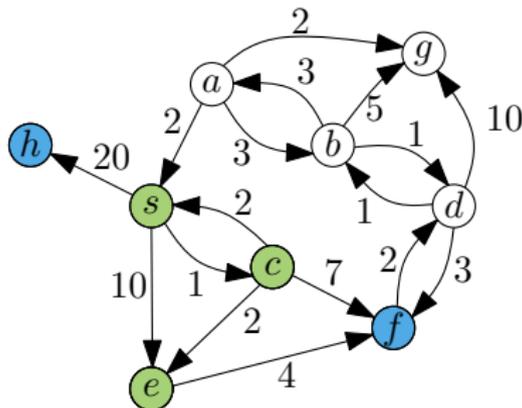
tentative distance  $d$ :

ID	Dist.
s	0
a	$\infty$
b	$\infty$
c	1
d	$\infty$
e	3
f	8
g	$\infty$
h	20

queue  $q$ :

ID	Key
e	3
f	8
h	20

# Dijkstras Algorithmus



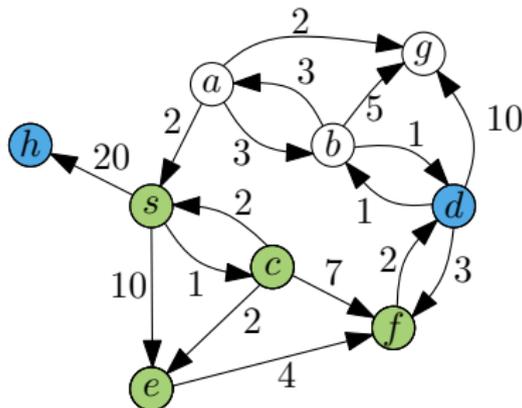
tentative distance  $d$ :

ID	Dist.
$s$	0
$a$	$\infty$
$b$	$\infty$
$c$	1
$d$	$\infty$
$e$	3
$f$	7
$g$	$\infty$
$h$	20

queue  $q$ :

ID	Key
$f$	7
$h$	20

# Dijkstras Algorithmus



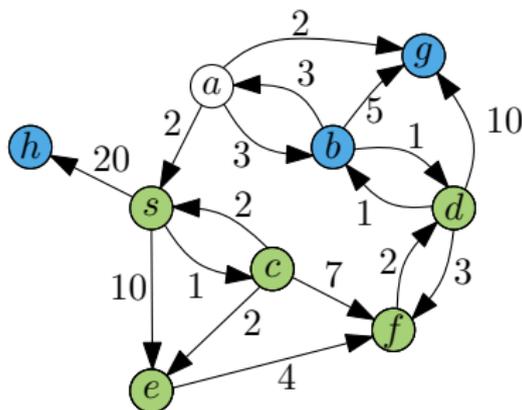
tentative distance  $d$ :

ID	Dist.
$s$	0
$a$	$\infty$
$b$	$\infty$
$c$	1
$d$	9
$e$	3
$f$	7
$g$	$\infty$
$h$	20

queue  $q$ :

ID	Key
$d$	9
$h$	20

# Dijkstras Algorithmus



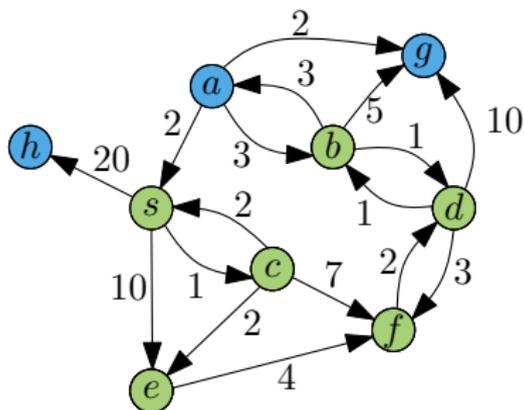
tentative distance  $d$ :

ID	Dist.
s	0
a	$\infty$
b	10
c	1
d	9
e	3
f	7
g	19
h	20

queue  $q$ :

ID	Key
b	10
g	19
h	20

# Dijkstras Algorithmus



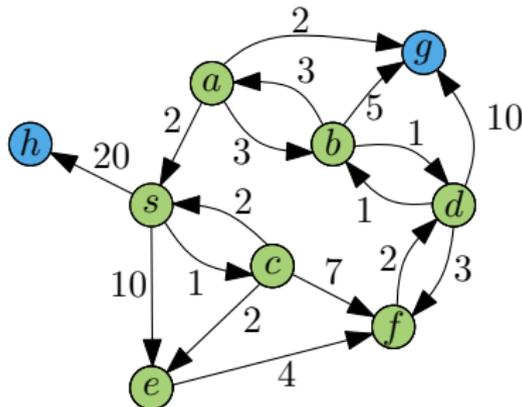
tentative distance  $d$ :

ID	Dist.
s	0
a	13
b	10
c	1
d	9
e	3
f	7
g	15
h	20

queue  $q$ :

ID	Key
a	13
g	15
h	20

# Dijkstras Algorithmus



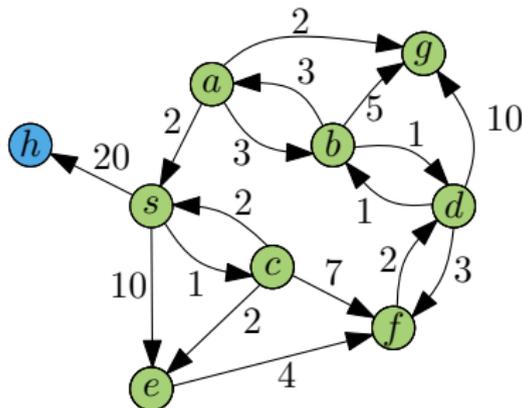
tentative distance  $d$ :

ID	Dist.
s	0
a	13
b	10
c	1
d	9
e	3
f	7
g	15
h	20

queue  $q$ :

ID	Key
g	15
h	20

# Dijkstras Algorithmus



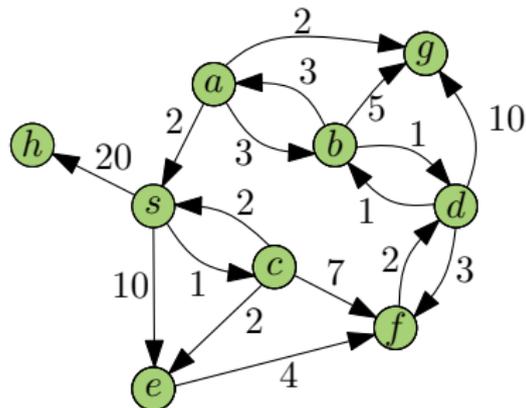
tentative distance  $d$ :

ID	Dist.
$s$	0
$a$	13
$b$	10
$c$	1
$d$	9
$e$	3
$f$	7
$g$	15
$h$	20

queue  $q$ :

ID	Key
$h$	20

# Dijkstras Algorithmus



tentative distance  $d$ :

ID	Dist.
s	0
a	13
b	10
c	1
d	9
e	3
f	7
g	15
h	20

queue  $q$ :

ID	Key

# Dijkstras Algorithmus

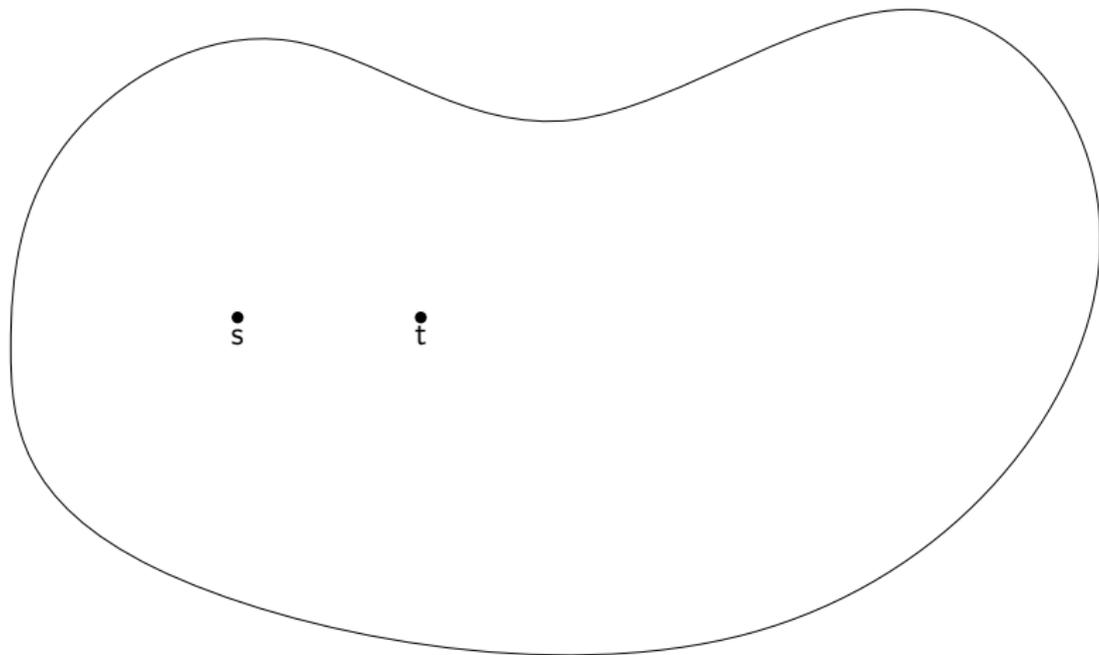
## Resultat

- Nach der Ausführung gilt:  $\forall v : d[v] = \text{dist}_G(s, v)$

## Stopkriterium

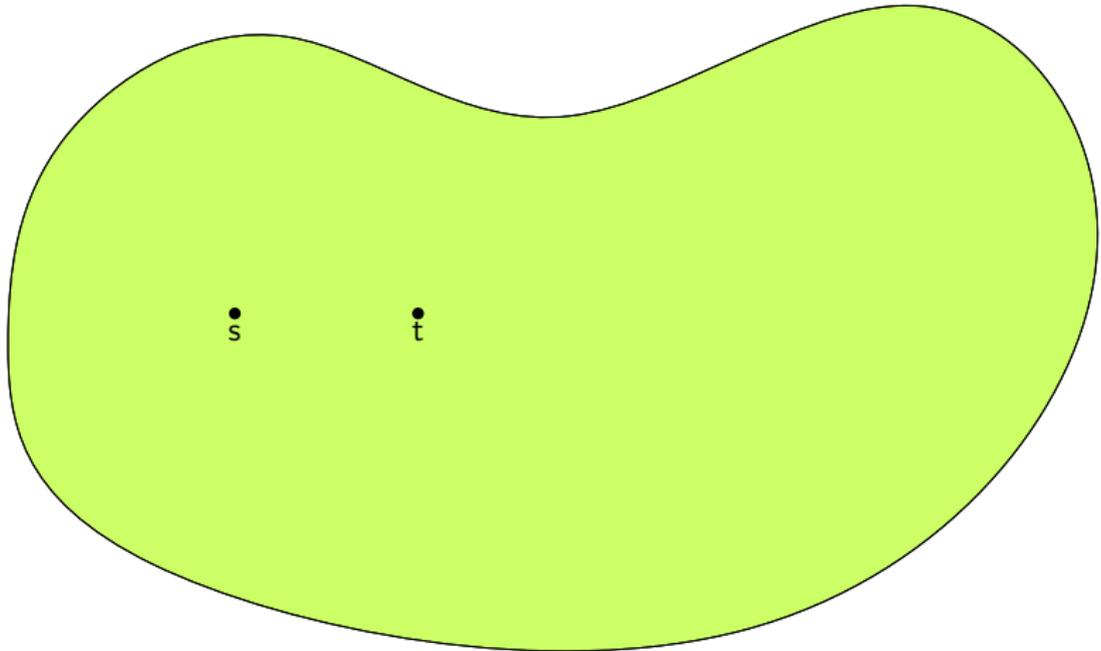
- Geht es schneller, wenn wir  $\text{dist}_G(s, t)$  nur für ein  $t$  bestimmen müssen?
- Ja: Breche Schleife ab, sobald  $t$  aus der Queue genommen wird

# Schematischer Suchraum



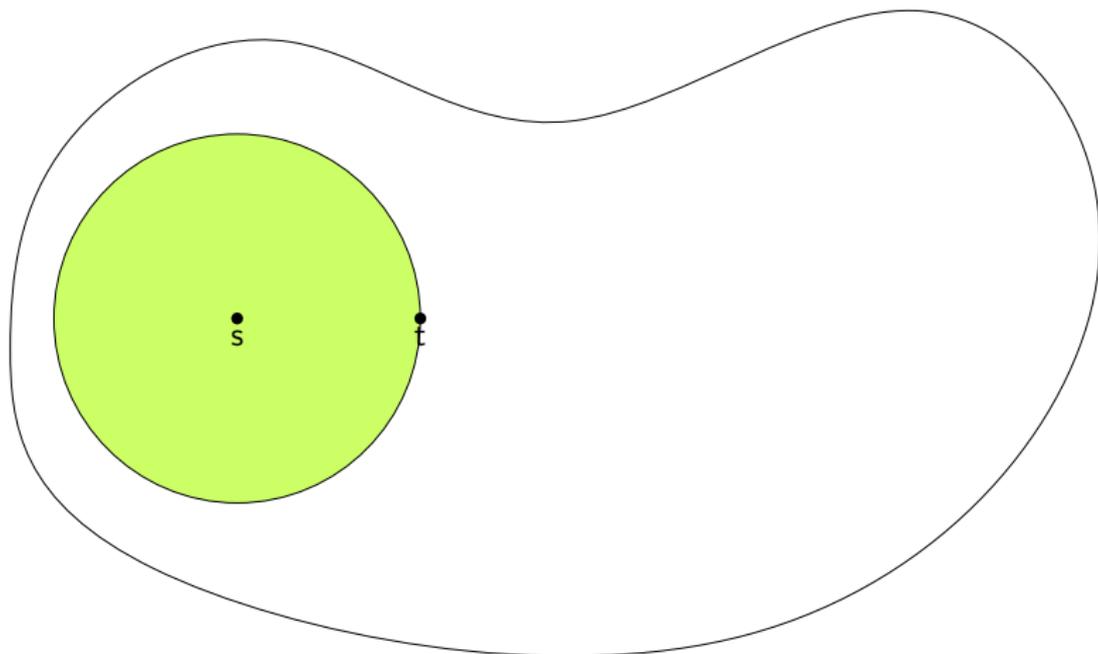
Ein Graph

# Schematischer Suchraum



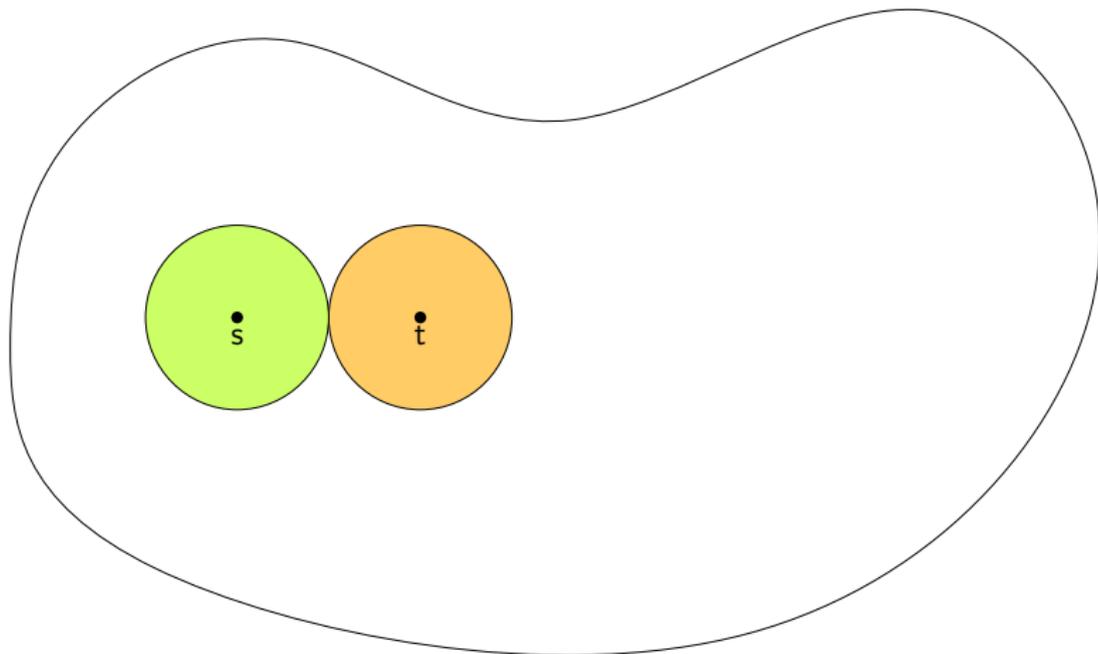
Suchraum ohne Stopkriterium

# Schematischer Suchraum



## Suchraum mit Stopkriterium

# Schematischer Suchraum

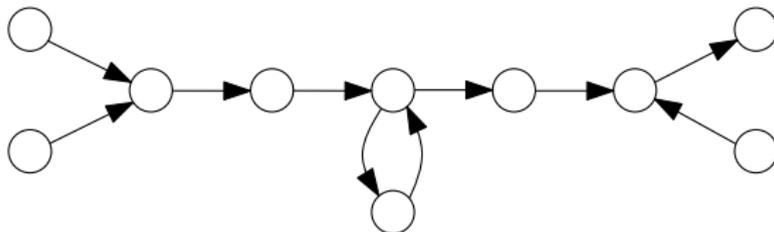


## Bidirektionale Variante von Dijkstras Algorithmus

# Bidirektionale Variante von Dijkstras Algorithmus

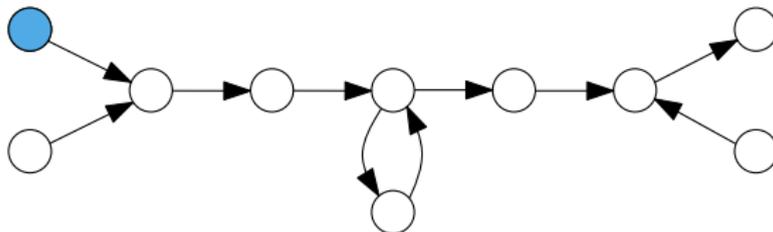
- Bidirektionale Variante von Dijkstras Algorithmus
- Mit zwei Queues und zwei tentativen Distanzarrays
- Arbeite die Seite mit den wenigsten Elementen in der Queue als nächstes ab
- Abbruch, wenn die Summe der min-keys beider Queues größer ist als der kürzeste bisher gefundene Pfad

# Shortcut



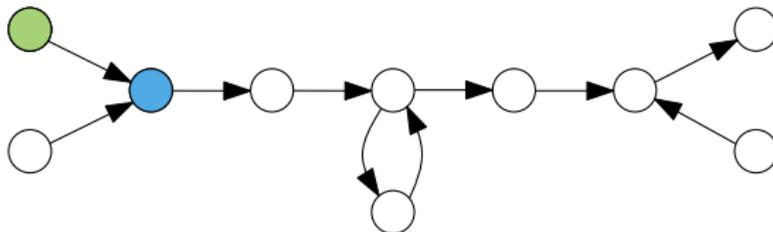
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.  
Das dauert.

# Shortcut



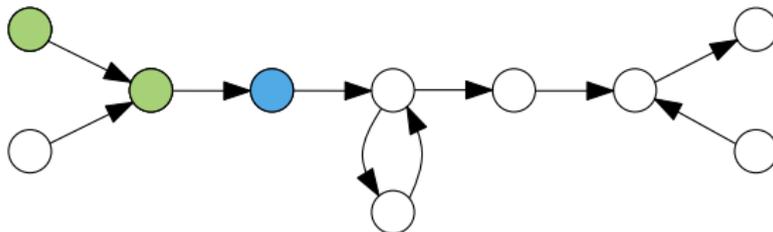
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.  
Das dauert.

# Shortcut



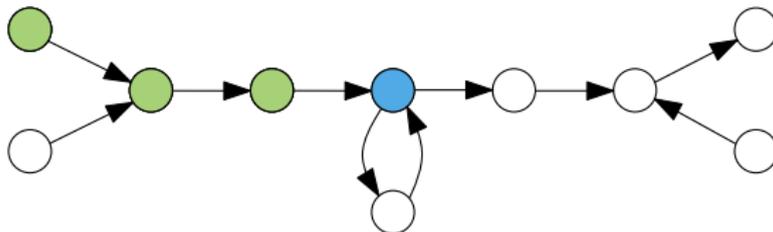
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.  
Das dauert.

# Shortcut



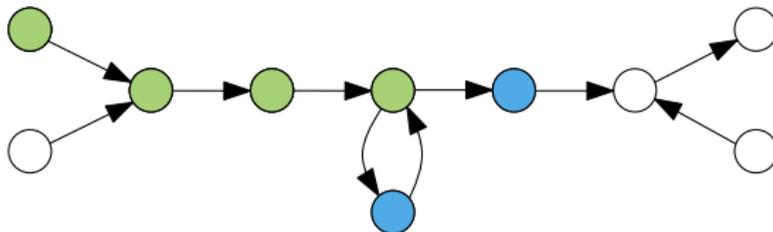
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.  
Das dauert.

# Shortcut



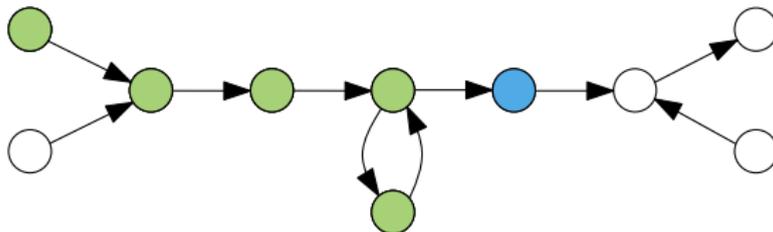
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.  
Das dauert.

# Shortcut



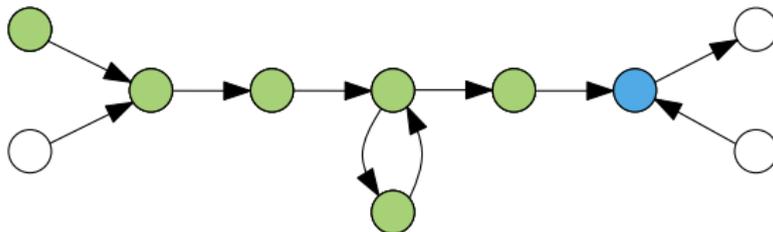
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.  
Das dauert.

# Shortcut



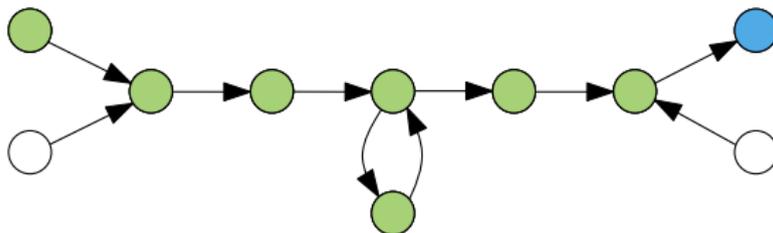
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.  
Das dauert.

# Shortcut



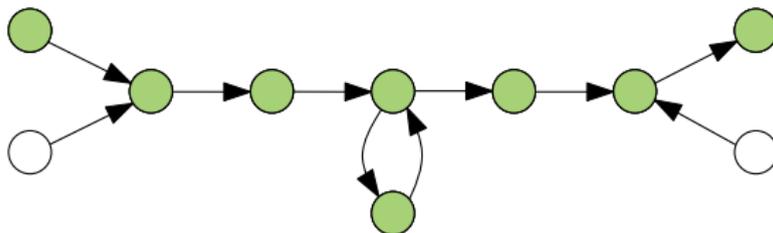
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.  
Das dauert.

# Shortcut



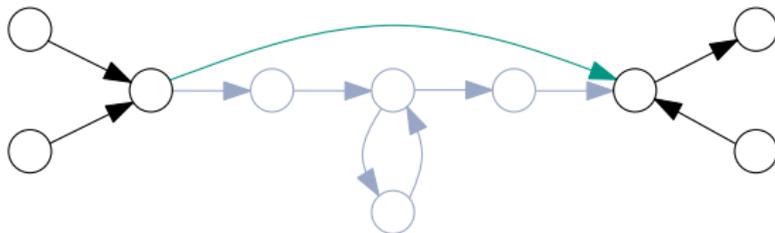
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.  
Das dauert.

# Shortcut



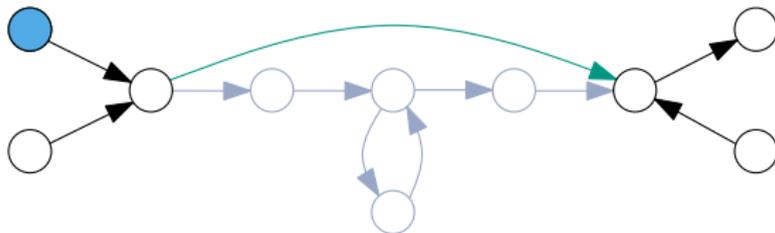
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.  
Das dauert.

# Shortcut



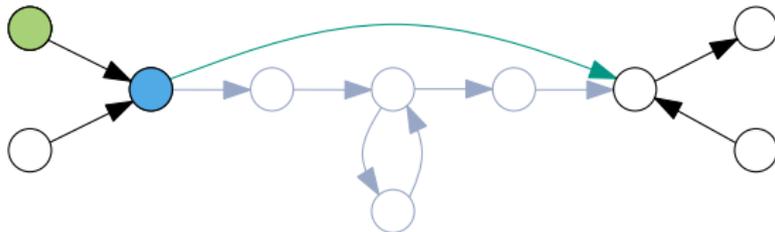
**Idee:** Füge Shortcut-Kante ein. Grauer Teilgraph muss nur angeschaut werden, wenn  $s$  oder  $t$  drin liegt.

# Shortcut



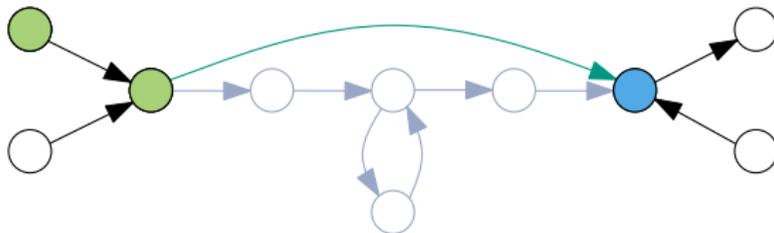
**Idee:** Füge Shortcut-Kante ein. Grauer Teilgraph muss nur angeschaut werden, wenn  $s$  oder  $t$  drin liegt.

# Shortcut



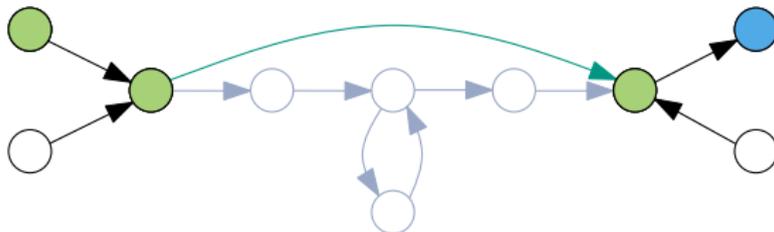
**Idee:** Füge Shortcut-Kante ein. Grauer Teilgraph muss nur angeschaut werden, wenn  $s$  oder  $t$  drin liegt.

# Shortcut



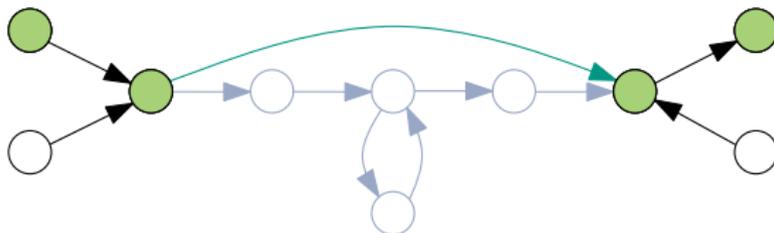
**Idee:** Füge Shortcut-Kante ein. Grauer Teilgraph muss nur angeschaut werden, wenn  $s$  oder  $t$  drin liegt.

# Shortcut



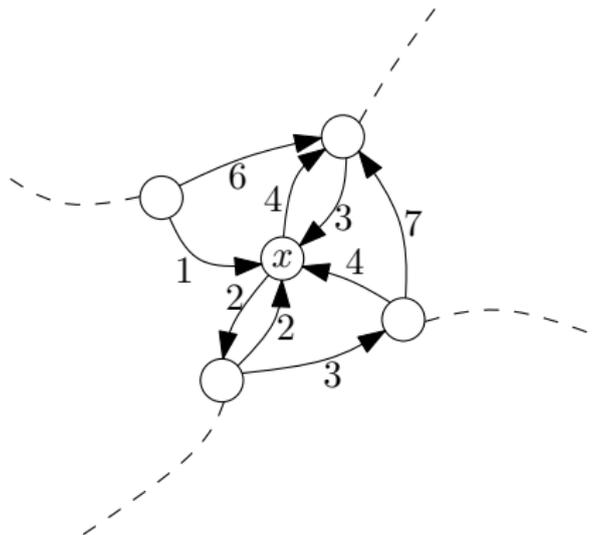
**Idee:** Füge Shortcut-Kante ein. Grauer Teilgraph muss nur angeschaut werden, wenn  $s$  oder  $t$  drin liegt.

# Shortcut



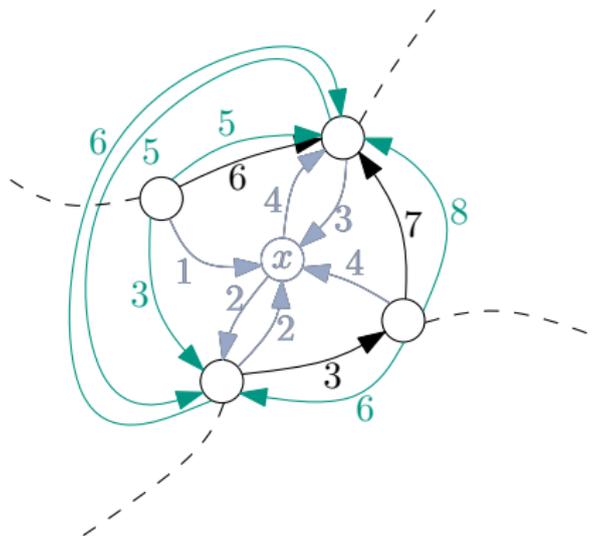
**Idee:** Füge Shortcut-Kante ein. Grauer Teilgraph muss nur angeschaut werden, wenn  $s$  oder  $t$  drin liegt.

# Knotenkontraktion von $x$



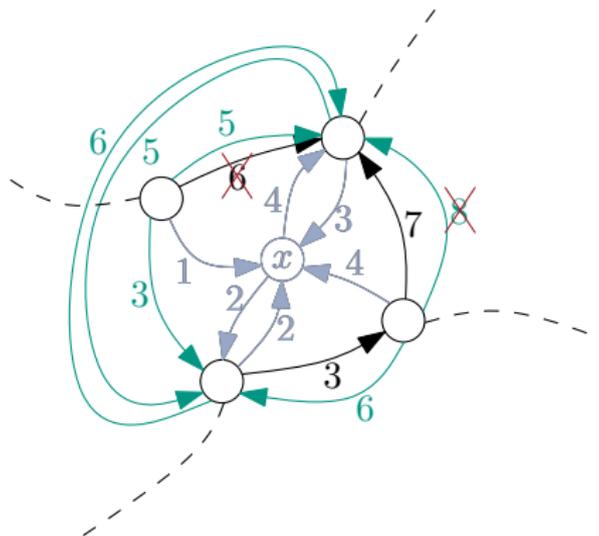
Kontraktion von  $x$ : Lösche  $x$  und füge Shortcuts zwischen Nachbarn ein, um die Distanzen zwischen allen Knoten zu erhalten

# Knotenkontraktion von $x$



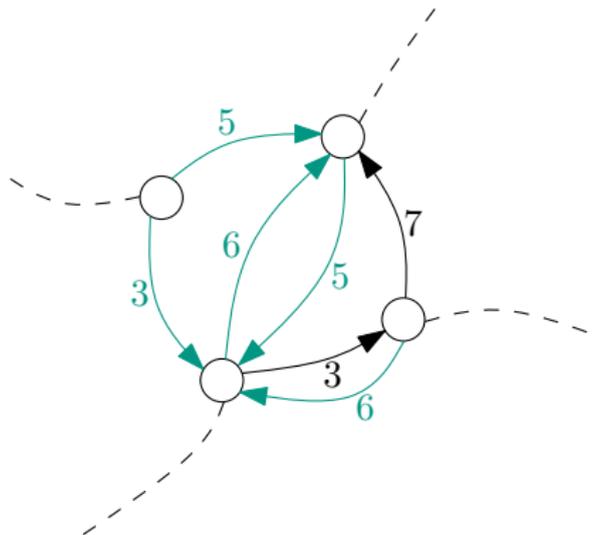
Kontraktion von  $x$ : Lösche  $x$  und füge Shortcuts zwischen Nachbarn ein, um die Distanzen zwischen allen Knoten zu erhalten

# Knotenkontraktion von $x$

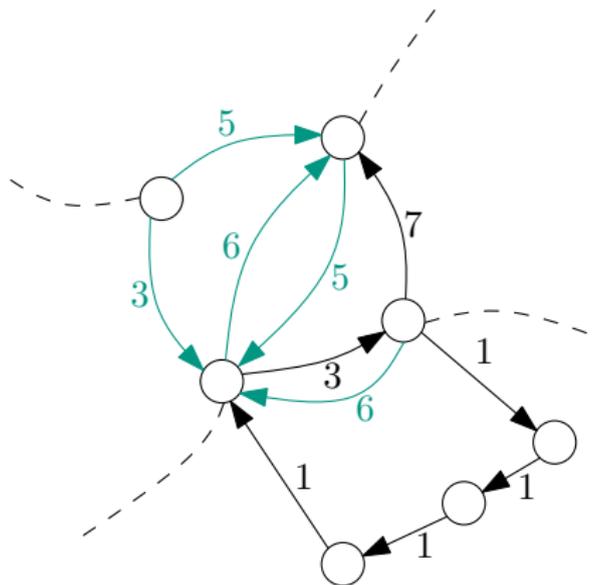


Bei Mehrfachkanten: Längere Kanten verwerfen

# Knotenkontraktion von $x$



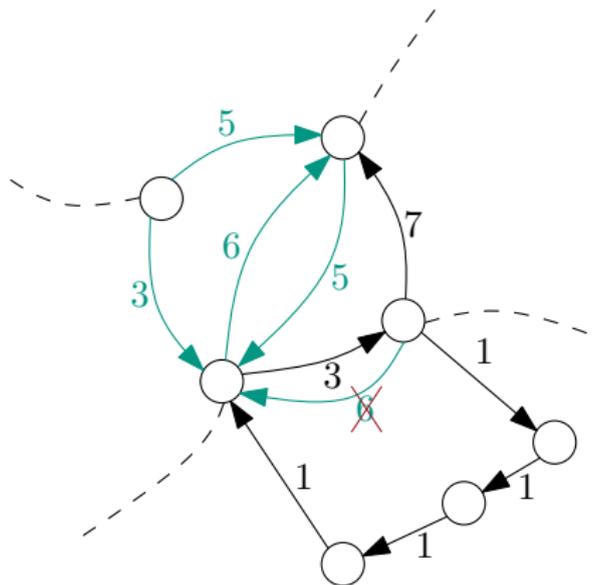
# Knotenkontraktion von $x$



Falls es einen kürzeren Pfad durch den Restgraphen gibt, dann kann man einen Shortcut auch verwerfen.

Suche nach solchem Pfad heißt Zeugensuche/Witness Search

# Knotenkontraktion von $x$



Falls es einen kürzeren Pfad durch den Restgraphen gibt, dann kann man einen Shortcut auch verwerfen.

Suche nach solchem Pfad heißt Zeugensuche/Witness Search

# Zeugensuche

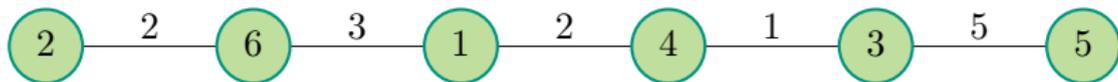
- Es seien  $y$  und  $z$  zwei Nachbarn des kontrahierten Knoten  $x$
- Wir fügen einen Shortcut  $(y, z)$  mit Gewicht  $\text{len}(y, x) + \text{len}(x, z)$  ein, wenn  $y \rightarrow x \rightarrow z$  der einzige kürzeste  $y$ - $z$ -Weg ist
- Zum Überprüfen, ob es einen kürzeren Weg gibt, startet man einen Dijkstra von  $y$  aus nach  $z$ . Diese Suche kann teuer sein. Mögliche Optimierungen:
  - Suche darf nicht über den Knoten  $x$  gehen
  - Bidirektionale Variante von Dijkstras Algorithmus
  - Wenn die Suchen sich treffen, kann man abbrechen
  - Wenn die Suchfront größer wird als  $\text{len}(y, x) + \text{len}(x, z)$ , kann man abbrechen
- Wenn das immer noch zu langsam ist: Suche nach  $k$  Schritten abbrechen. Eventuell gibt es einen Pfad, den wir nicht finden. Das führt zu zusätzlichen Shortcuts, aber das ist kein Problem bzgl. der Korrektheit.

# Contraction Hierarchy

## Grundidee

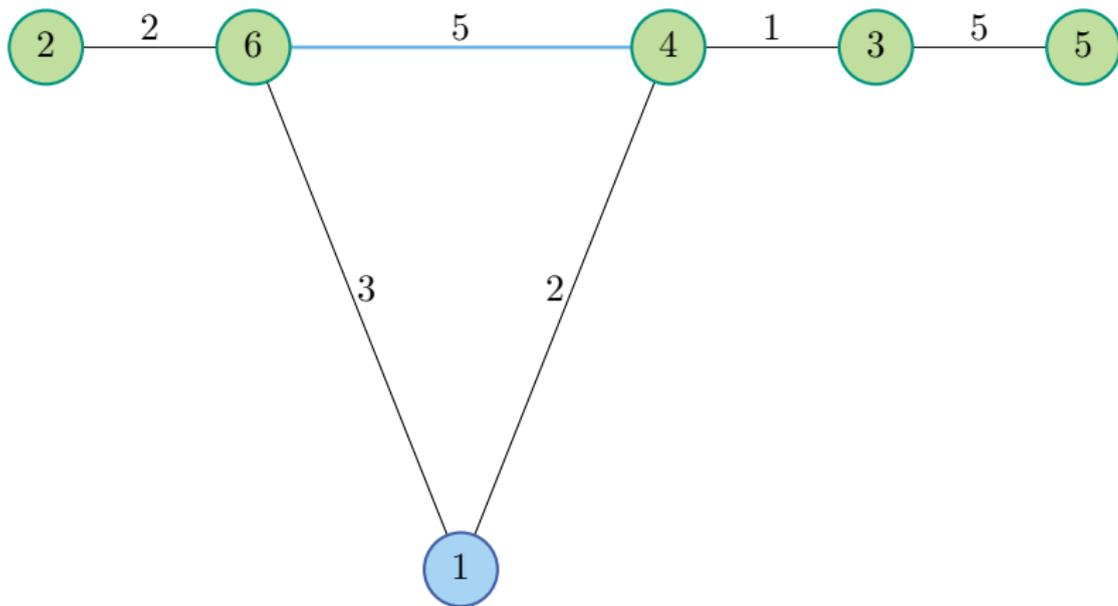
- Eingabegraph  $G$
- Ordne Knoten von  $G$  nach “Wichtigkeit”:  $v_1 \dots v_n$
- Kontrahiere Knoten iterativ aus  $G$  heraus
  - zuerst den “unwichtigsten” Knoten  $v_1$
  - den “wichtigsten” Knoten  $v_n$  als letztes
- Graph mit Shortcuts heißt augmentierter Graph

# Contraction Hierarchy



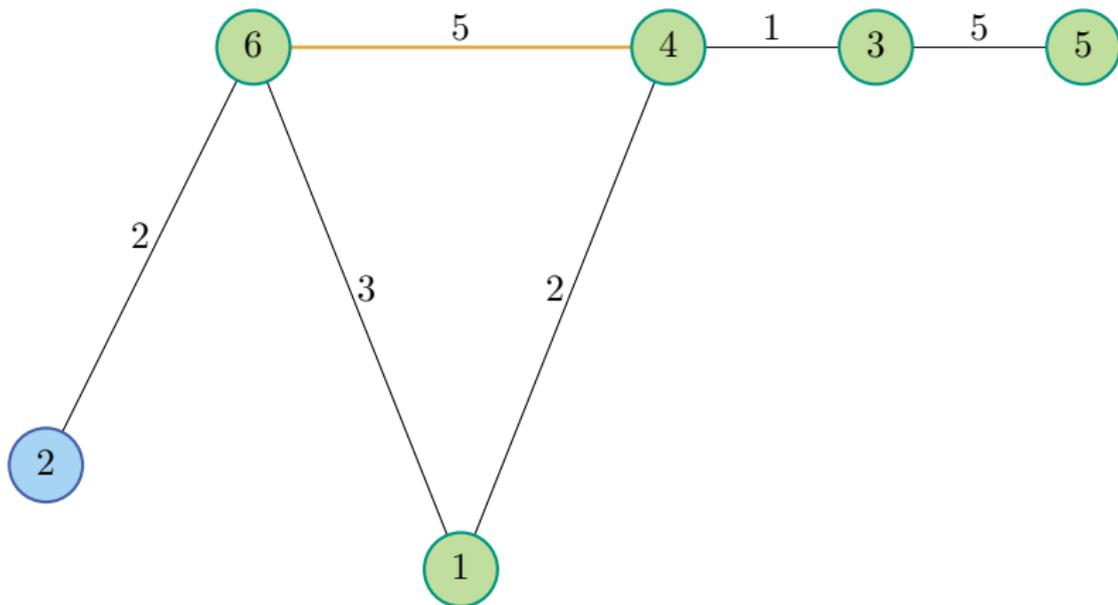
Knoten nummeriert nach “Wichtigkeit”

# Contraction Hierarchy



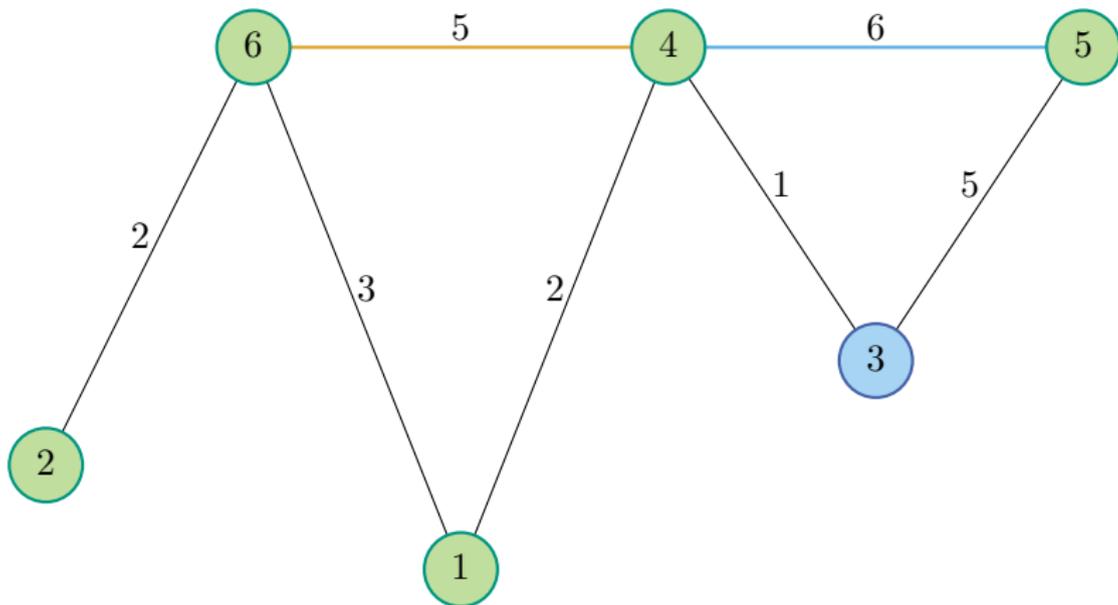
Knoten nummeriert nach "Wichtigkeit"

# Contraction Hierarchy



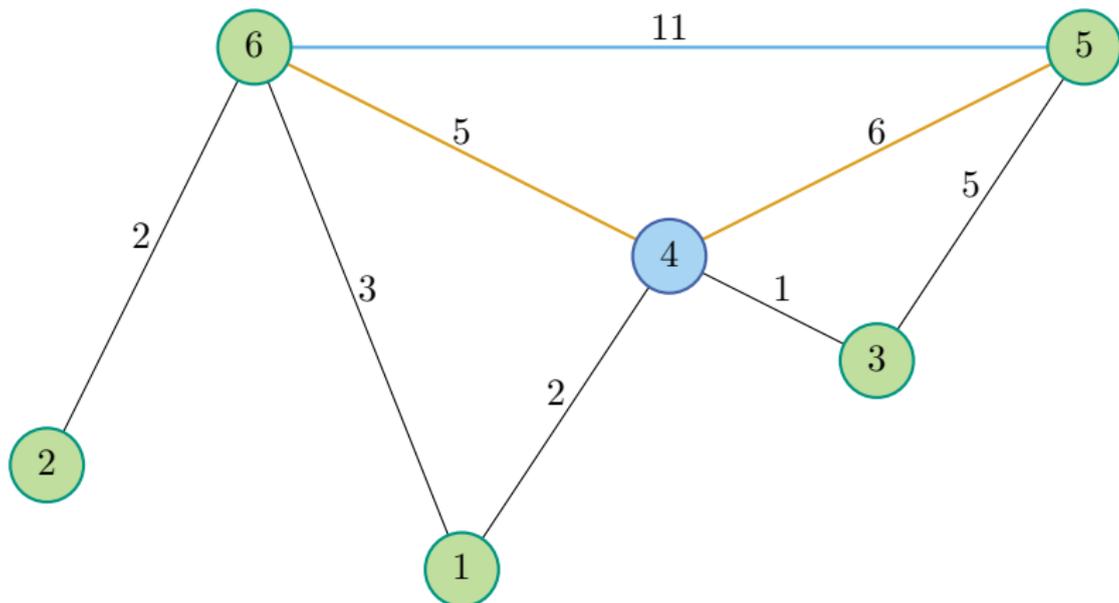
Knoten nummeriert nach "Wichtigkeit"

# Contraction Hierarchy



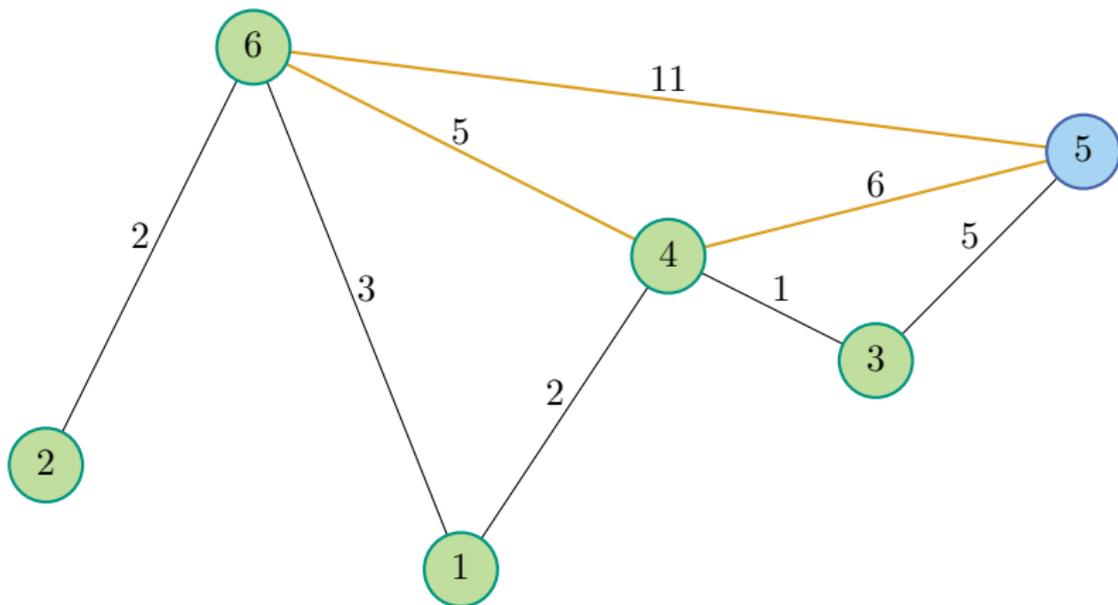
Knoten nummeriert nach "Wichtigkeit"

# Contraction Hierarchy



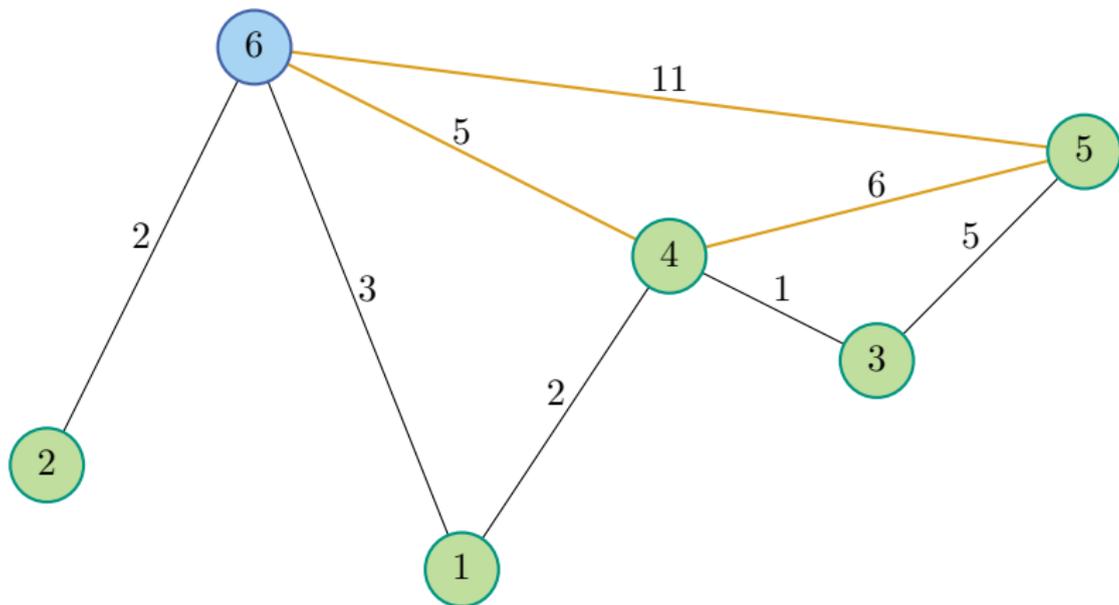
Knoten nummeriert nach "Wichtigkeit"

# Contraction Hierarchy



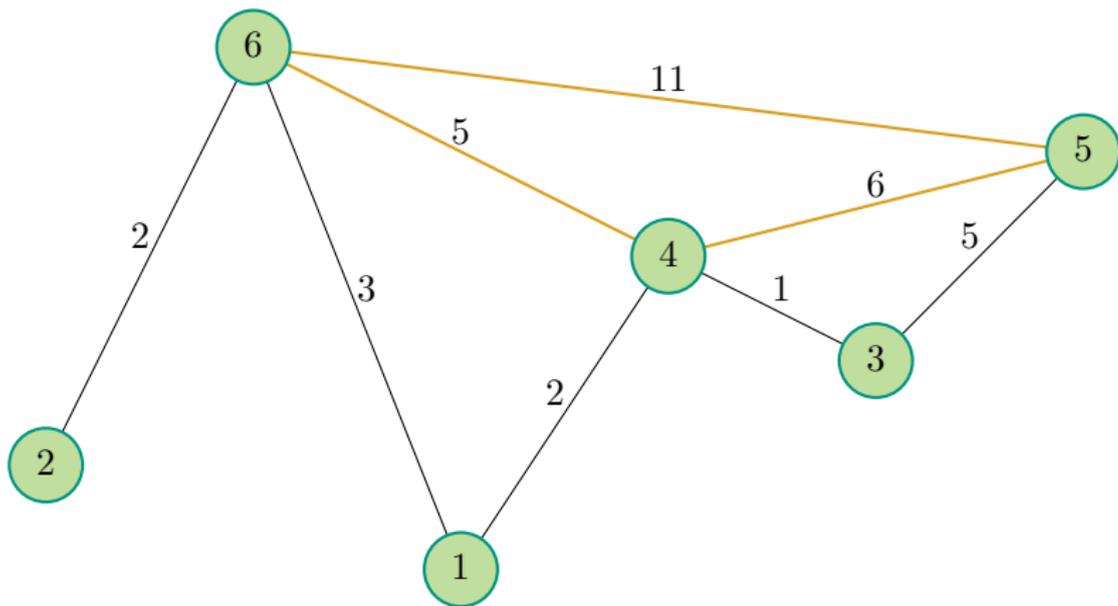
Knoten nummeriert nach "Wichtigkeit"

# Contraction Hierarchy



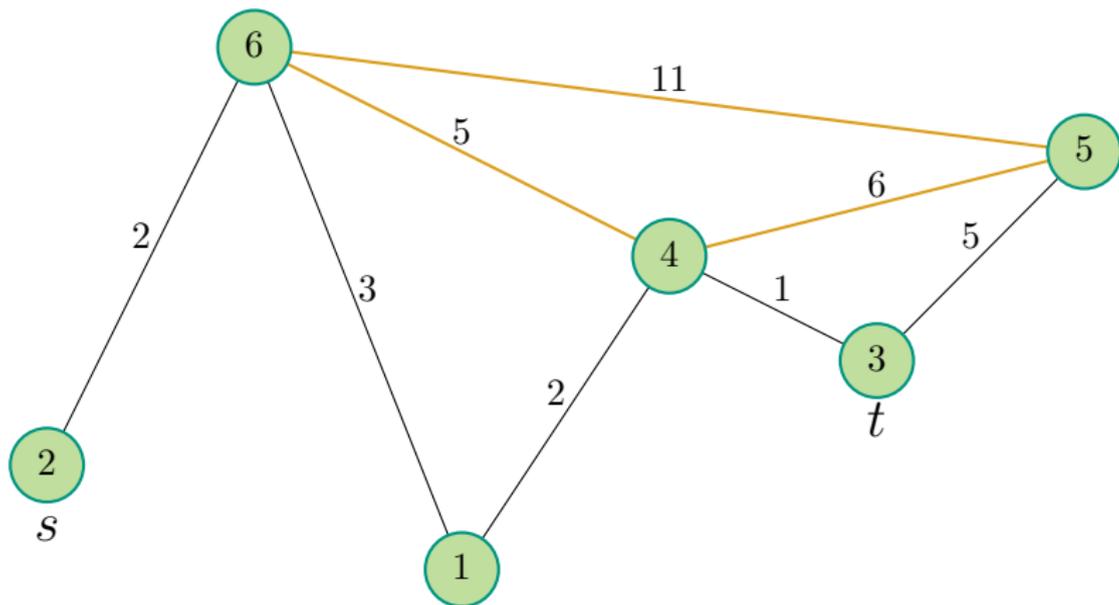
Knoten nummeriert nach "Wichtigkeit"

# Contraction Hierarchy



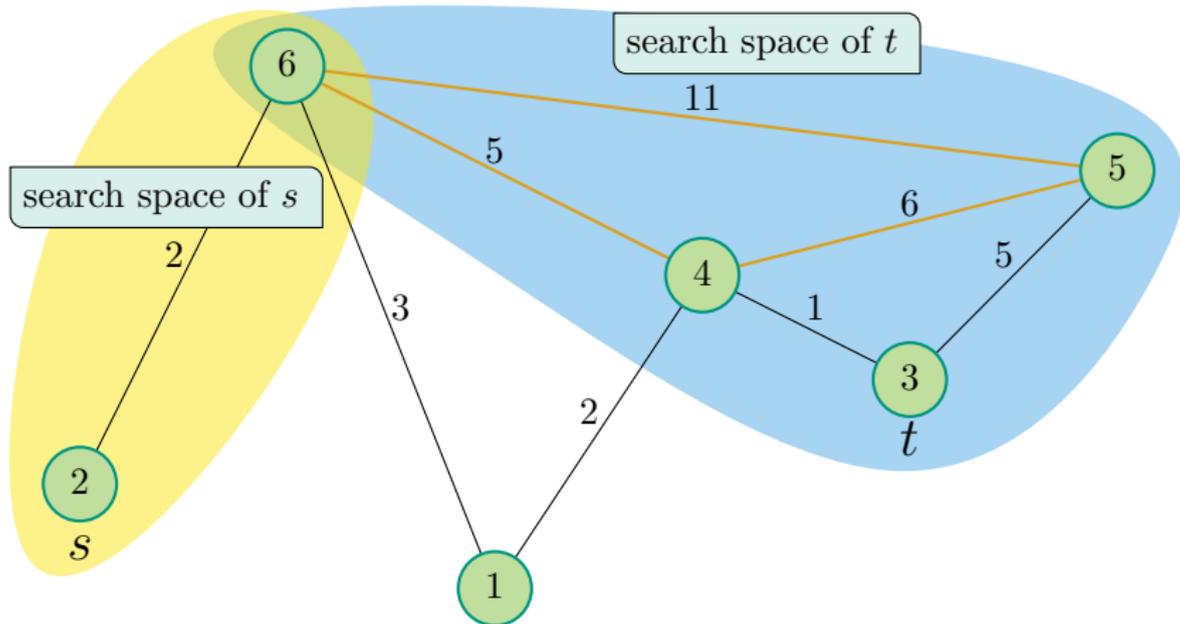
Knoten nummeriert nach "Wichtigkeit"

# Contraction Hierarchy



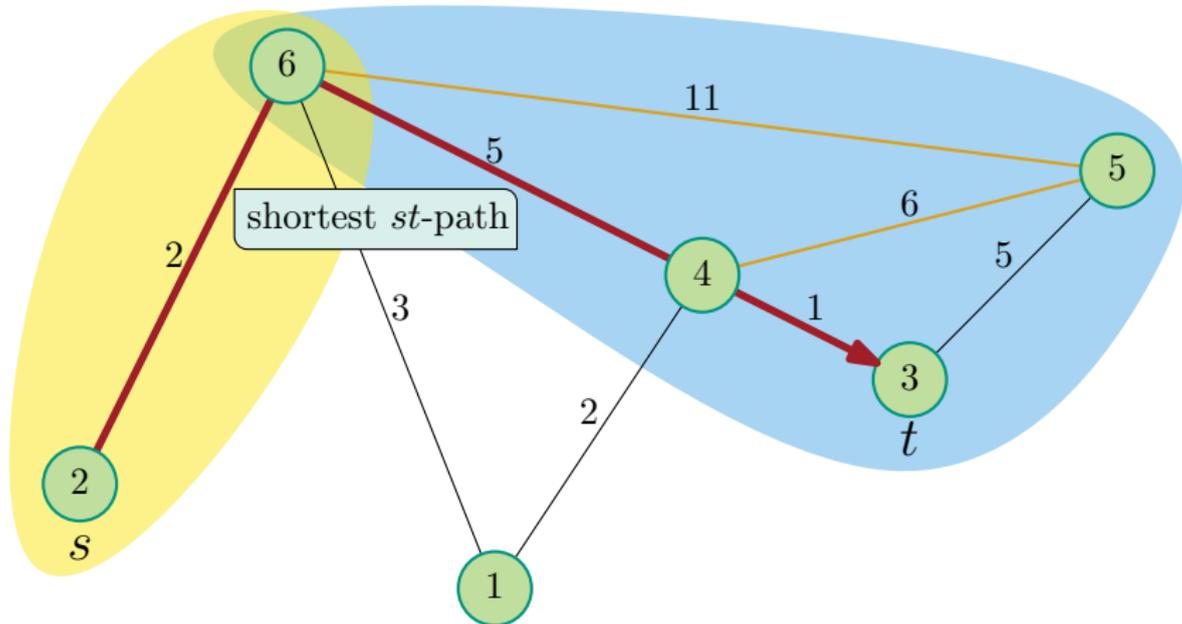
Knoten nummeriert nach "Wichtigkeit"

# Contraction Hierarchy



Knoten nummeriert nach “Wichtigkeit”

# Contraction Hierarchy



Für jeden ursprünglichen kürzesten Weg gibt es einen hoch-runter-Pfad

- Bidirektionale Variante von Dijkstras Algorithmus
- Verfolge nur Kanten zu wichtigeren Knoten
- Vorwärtssuche findet den “hoch”-Teil des Pfads
- Rückwärtssuche findet den “runter”-Teil des Pfads
- Abbruch, wenn der min-key beider Queues größer ist als der bisher kürzeste gefundene Pfad

# Nach “Wichtigkeit” Ordnen

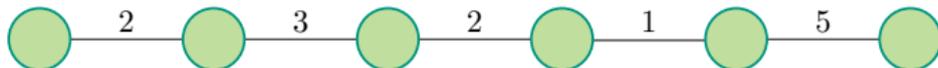
## Grund-Idee:

- Wir wollen wenig Shortcuts
- Ein Knoten ist “unwichtig”, wenn er wenig Shortcuts erzeugt
- → simulierte Knotenkontraktion, um Knoten zu gewichten

## Algorithmus:

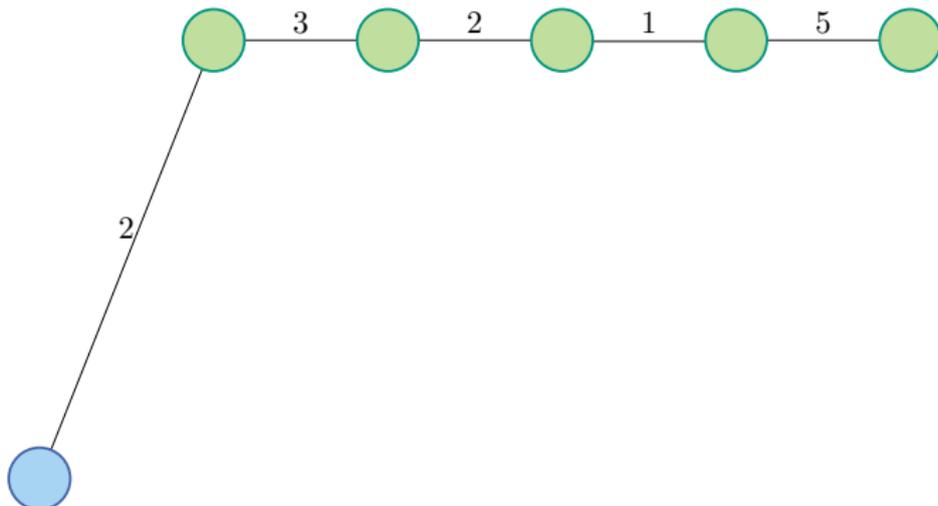
- Baue eine große Warteschlange mit allen Knoten sortiert nach ihrer “Wichtigkeit”
- Kontrahiere iterativ unwichtigsten Knoten
- Kontraktion eines Knotens kann “Wichtigkeit” der Nachbarn beeinflussen
- → “Wichtigkeit” der Nachbarn neu berechnen

## Problemfall: Pfad



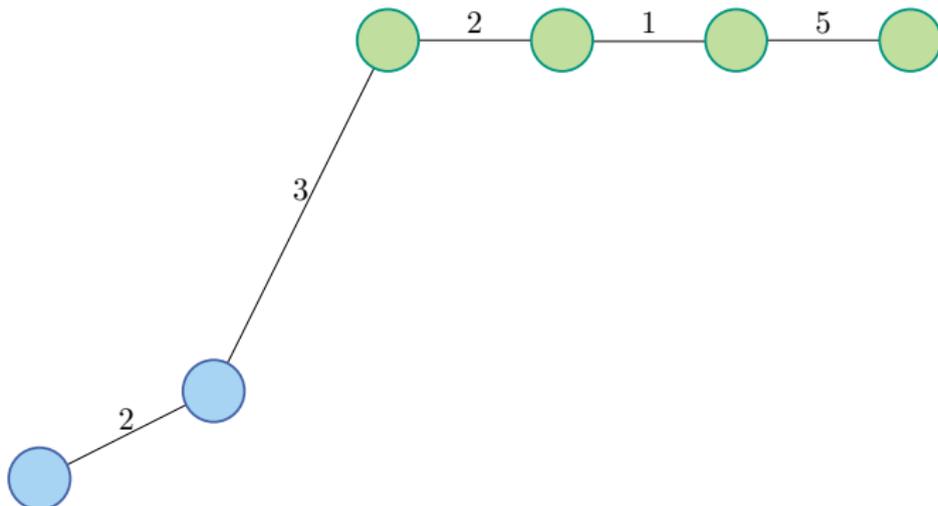
Linken Knoten kontrahieren erzeugt keine Shortcuts

## Problemfall: Pfad



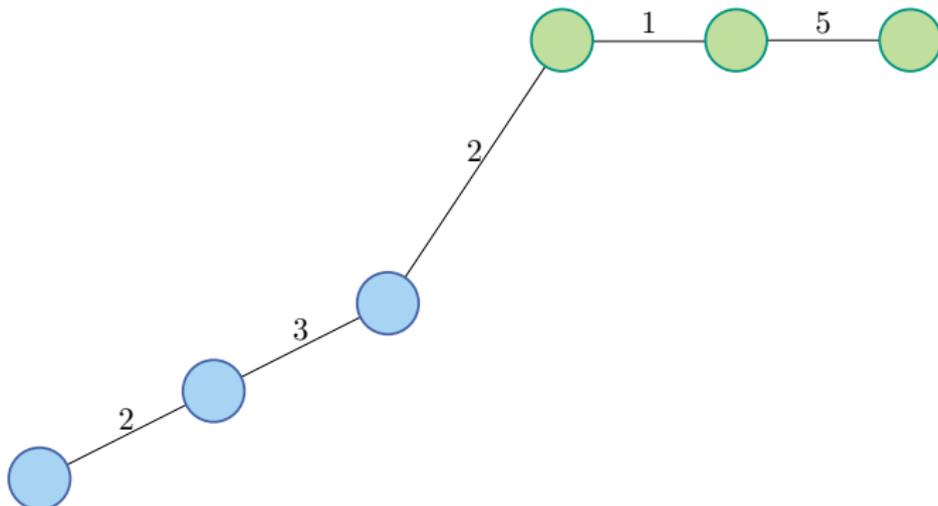
Linken Knoten kontrahieren erzeugt keine Shortcuts

# Problemfall: Pfad



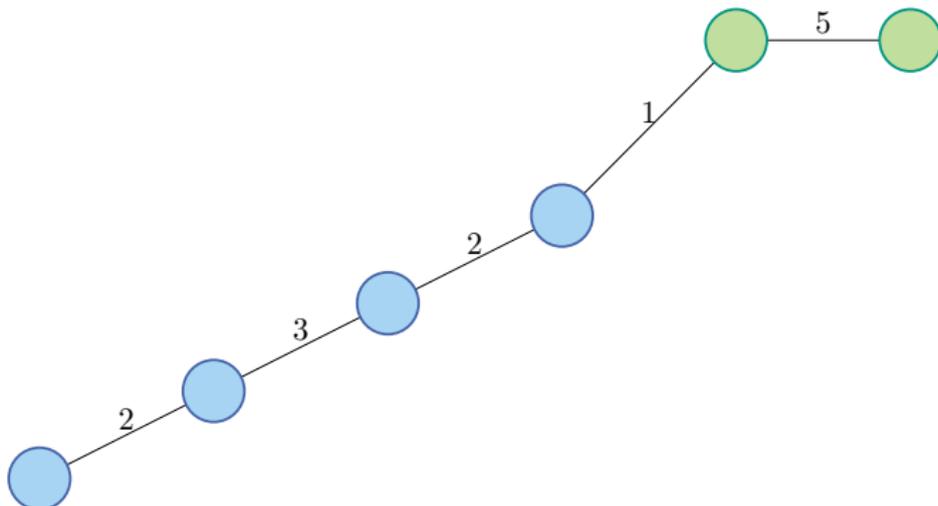
Linken Knoten kontrahieren erzeugt keine Shortcuts

# Problemfall: Pfad



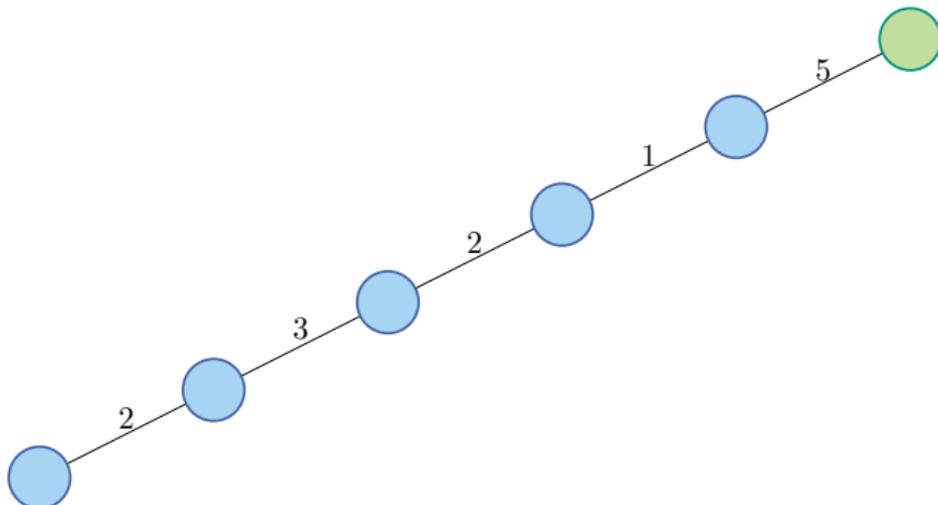
Linken Knoten kontrahieren erzeugt keine Shortcuts

# Problemfall: Pfad



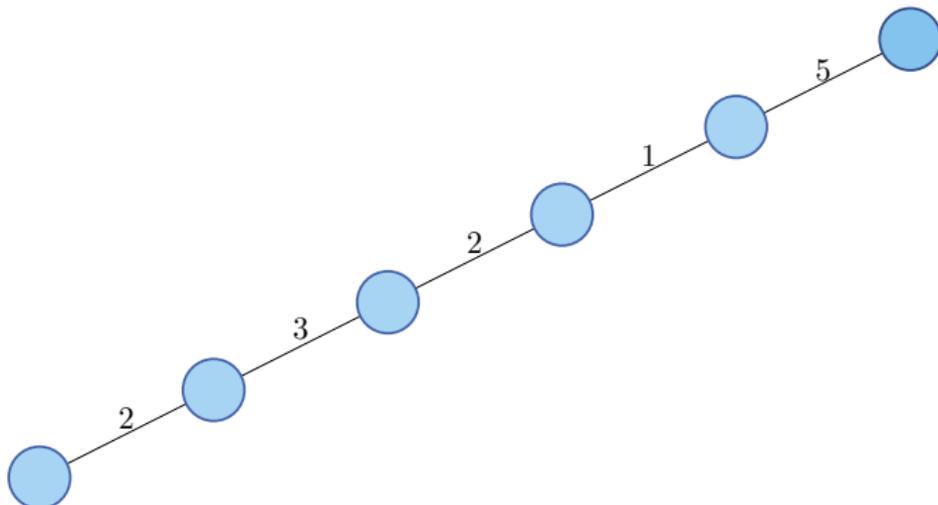
Linken Knoten kontrahieren erzeugt keine Shortcuts

# Problemfall: Pfad



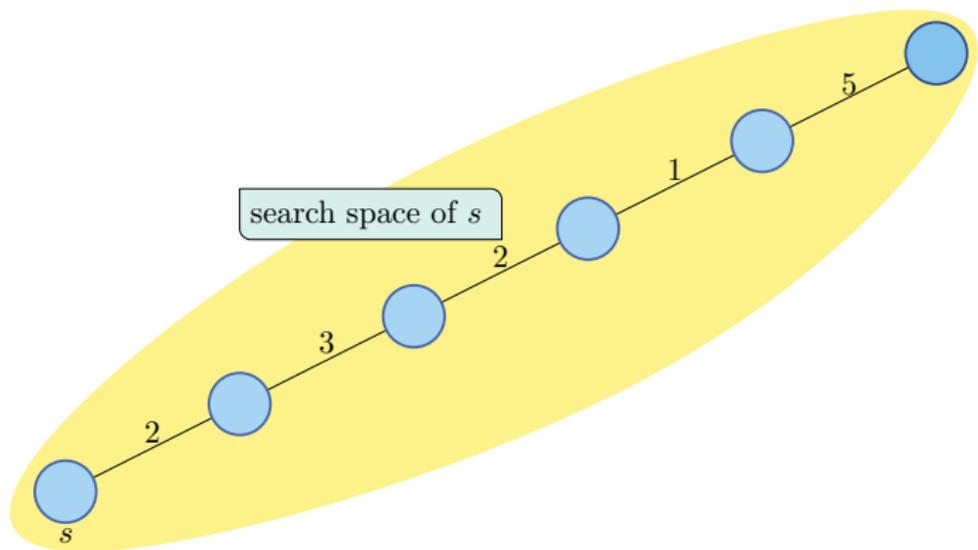
Linken Knoten kontrahieren erzeugt keine Shortcuts

# Problemfall: Pfad



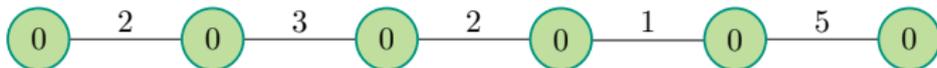
Linken Knoten kontrahieren erzeugt keine Shortcuts

# Problemfall: Pfad



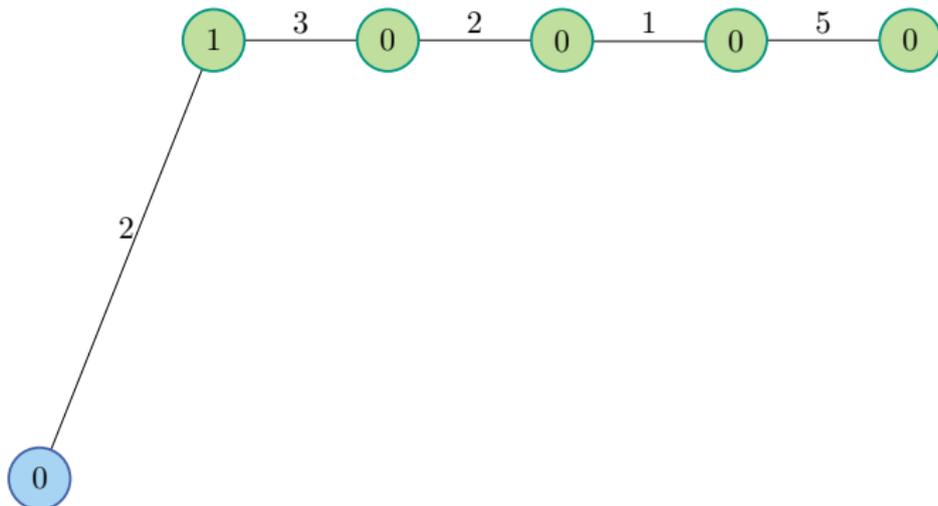
Suchraum von  $s$  ist der ganze Graph  $\rightarrow$  keine Beschleunigung

## Problemfall: Pfad



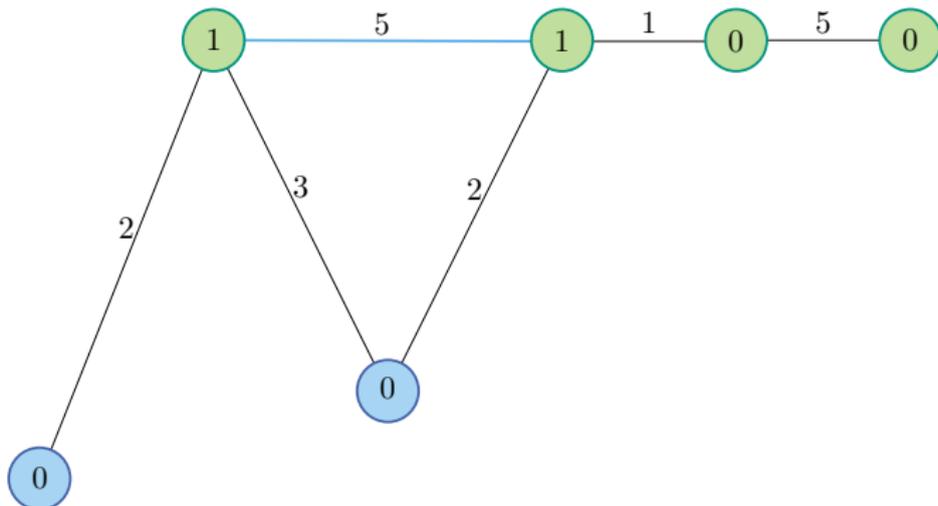
2-tes Kriterium: Das geschätzte Level  $\ell(x)$  eines Knotens  $x$  kontrahiert  $\rightarrow$  für alle Nachbarn  $y$ :  $\ell(y) \leftarrow \max\{\ell(y), \ell(x) + 1\}$

# Problemfall: Pfad



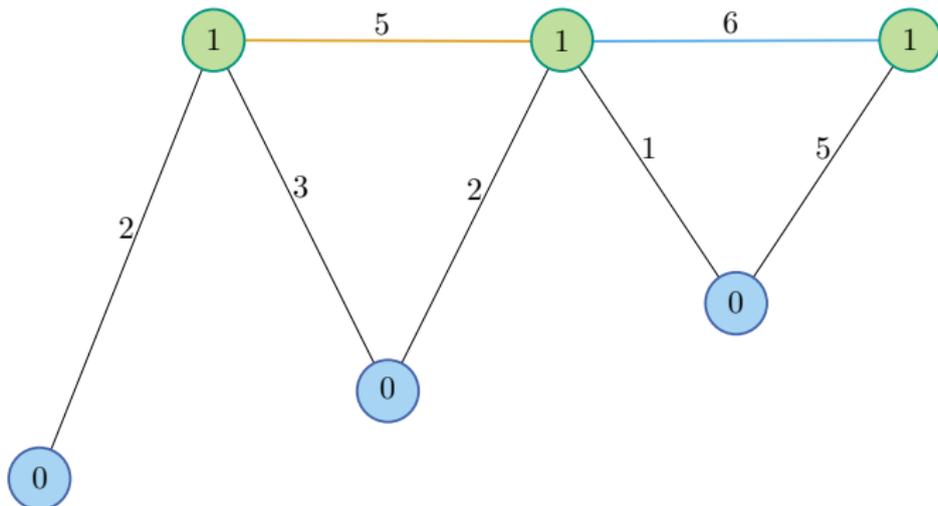
2-tes Kriterium: Das geschätzte Level  $\ell(x)$  eines Knotens  $x$  kontrahiert  $\rightarrow$  für alle Nachbarn  $y$ :  $\ell(y) \leftarrow \max\{\ell(y), \ell(x) + 1\}$

# Problemfall: Pfad



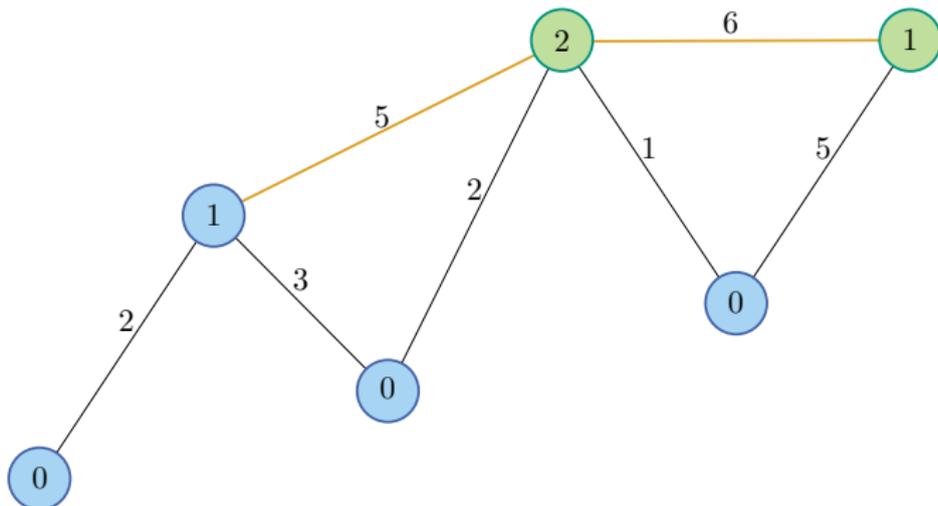
2-tes Kriterium: Das geschätzte Level  $\ell(x)$  eines Knotens  $x$  kontrahiert  $\rightarrow$  für alle Nachbarn  $y$ :  $\ell(y) \leftarrow \max\{\ell(y), \ell(x) + 1\}$

# Problemfall: Pfad



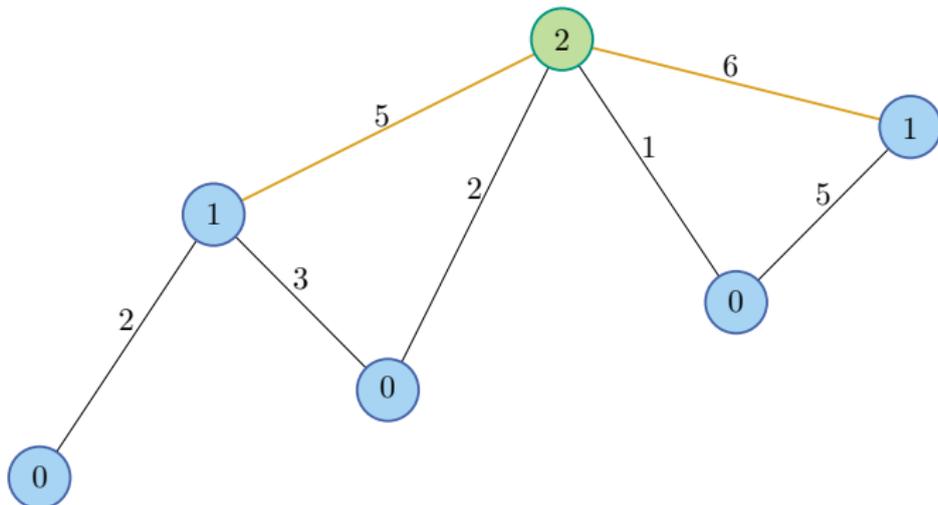
2-tes Kriterium: Das geschätzte Level  $\ell(x)$  eines Knotens  $x$  kontrahiert  $\rightarrow$  für alle Nachbarn  $y$ :  $\ell(y) \leftarrow \max\{\ell(y), \ell(x) + 1\}$

# Problemfall: Pfad



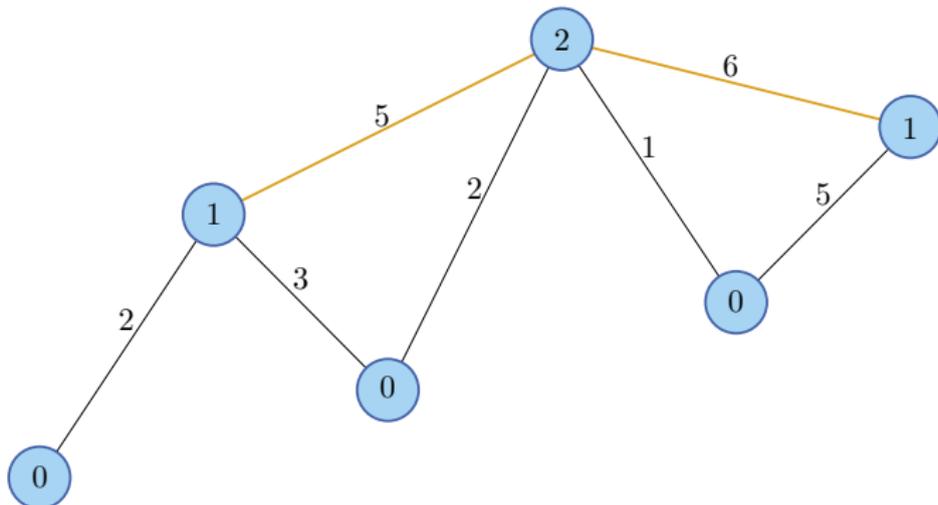
2-tes Kriterium: Das geschätzte Level  $\ell(x)$  eines Knotens  $x$  kontrahiert  $\rightarrow$  für alle Nachbarn  $y$ :  $\ell(y) \leftarrow \max\{\ell(y), \ell(x) + 1\}$

# Problemfall: Pfad



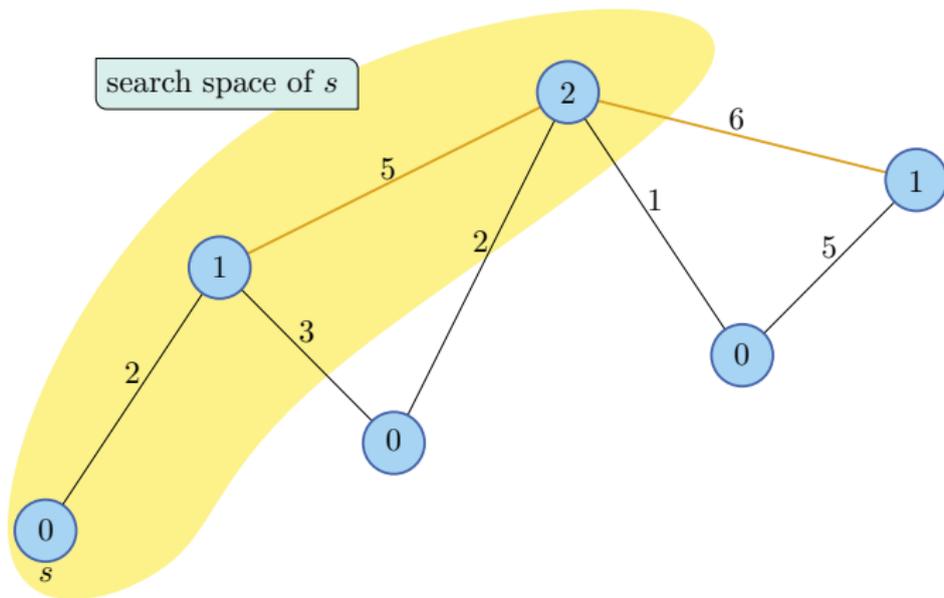
2-tes Kriterium: Das geschätzte Level  $\ell(x)$  eines Knotens  $x$  kontrahiert  $\rightarrow$  für alle Nachbarn  $y$ :  $\ell(y) \leftarrow \max\{\ell(y), \ell(x) + 1\}$

# Problemfall: Pfad



2-tes Kriterium: Das geschätzte Level  $\ell(x)$  eines Knotens  $x$  kontrahiert  $\rightarrow$  für alle Nachbarn  $y$ :  $\ell(y) \leftarrow \max\{\ell(y), \ell(x) + 1\}$

# Problemfall: Pfad



2-tes Kriterium: Das geschätzte Level  $\ell(x)$  eines Knotens  $x$  kontrahiert  $\rightarrow$  für alle Nachbarn  $y$ :  $\ell(y) \leftarrow \max\{\ell(y), \ell(x) + 1\}$

# Kombination mehrerer Kriterien

- Speichere für jede Kante  $e$  die Anzahl  $h(e)$  der Originalkanten, aus denen sie besteht
- Es sei  $A(x)$  die Menge der eingefügten Shortcuts, wenn  $x$  kontrahiert werden würde
- Analog:  $D(x)$  die Menge der gelöschten Kanten
- Es sei  $I(x)$  die “Wichtigkeit” von  $x$

Eine funktionierende Definition von  $I(x)$  ist

$$I(x) := \ell(x) + \frac{|A(x)|}{|D(x)|} + \frac{\sum_{e \in A(x)} h(e)}{\sum_{e \in D(x)} h(e)}$$

**Hinweis:** Es gibt sehr viele unterschiedliche Definitionen für  $I$ . Das ist nur ein Kochrezept, das sich bewährt hat und jeder würzt leicht anders.

- Graph mit Shortcuts heißt augmentierter Graph
- Eine Ordnung  $\pi$  ist eine Permutation der Knoten, so dass die Knoten in der Reihe  $\pi(0), \pi(1) \dots \pi(n-1)$  kontrahiert werden.
- Die inverse Permutation  $\pi^{-1}$  heißt Rank. Der Rank entspricht der “Höhe” eines Knotens in der CH.