Karlsruher Institut für Technologie

Algorithmen für Routenplanung

6. Vorlesung, Sommersemester 2022

Adrian Feilhauer | 11. Mai 2022

Kürzeste Wege in Straßennetzwerken

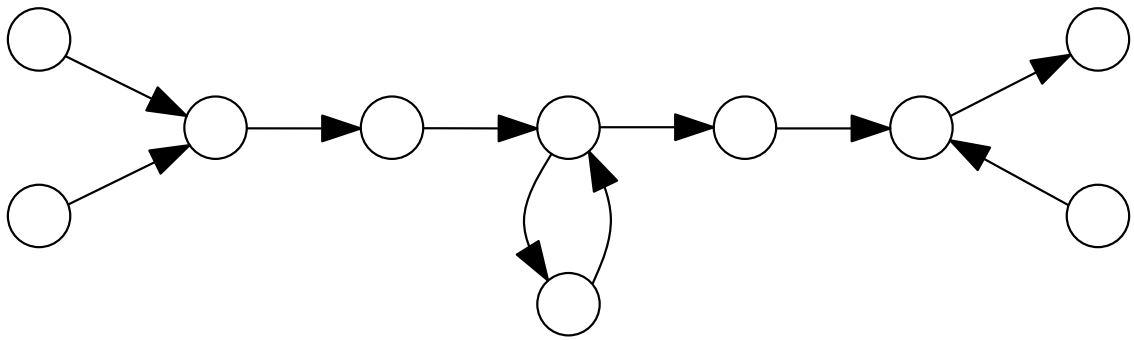
Beschleunigungstechniken (Fortsetzung)

Thema: Contraction Hierarchies (CH)

- CH-Basisvariante
  - Knotenkontraktion
  - Suche und Pfadentpackung
  - Bottom-Up-Knotenordnungen
- CH im Detail
  - Stall-on-demand
  - Top-Down-Knotenordnungen

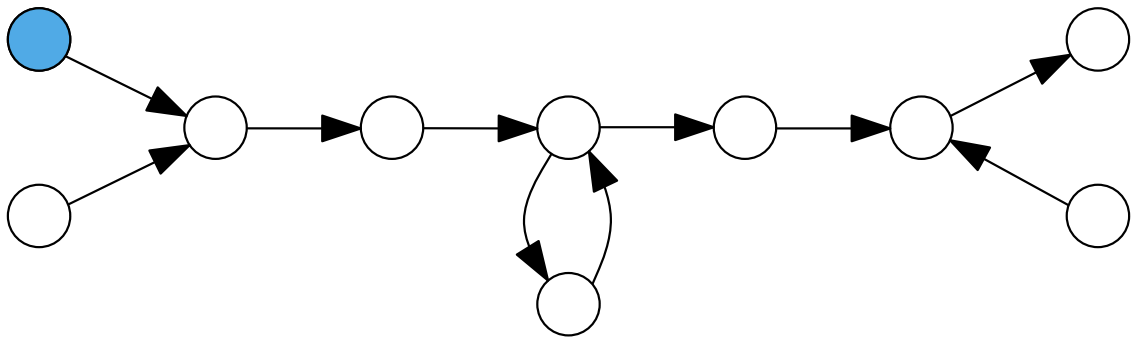
CH-Basisvariante

Shortcut



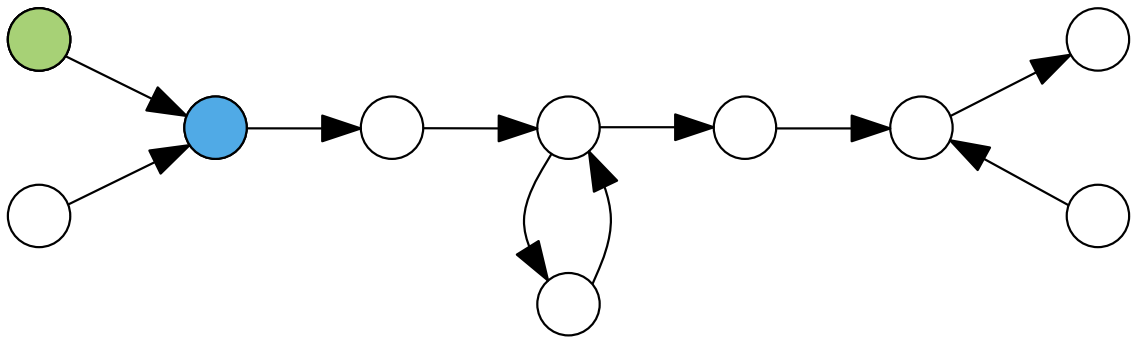
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.
Das dauert.

Shortcut



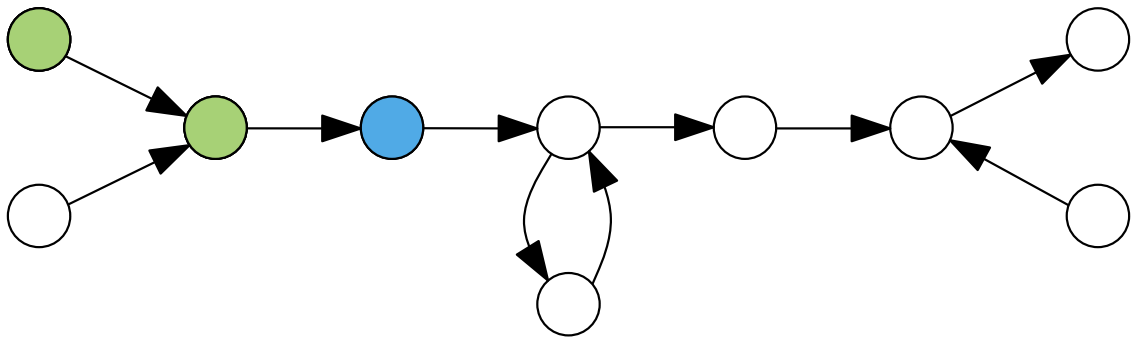
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.
Das dauert.

Shortcut



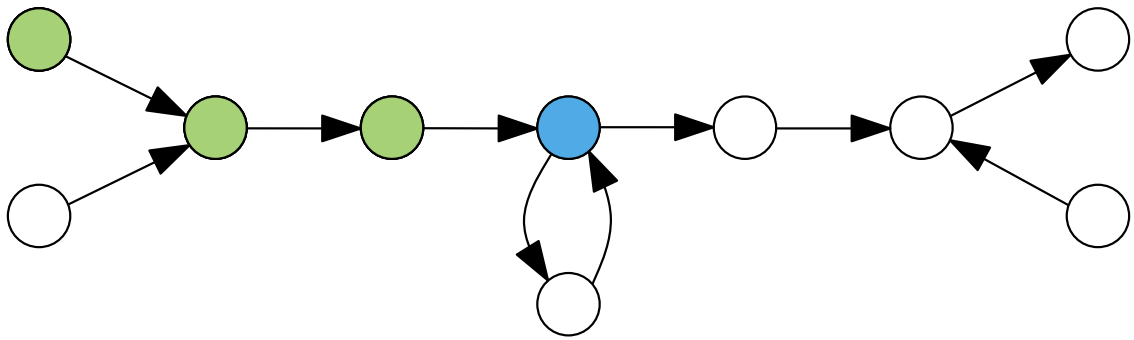
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.
Das dauert.

Shortcut



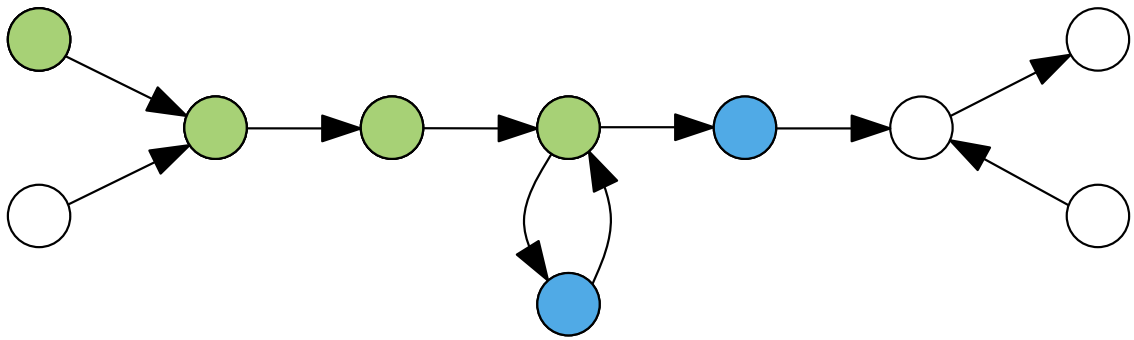
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.
Das dauert.

Shortcut



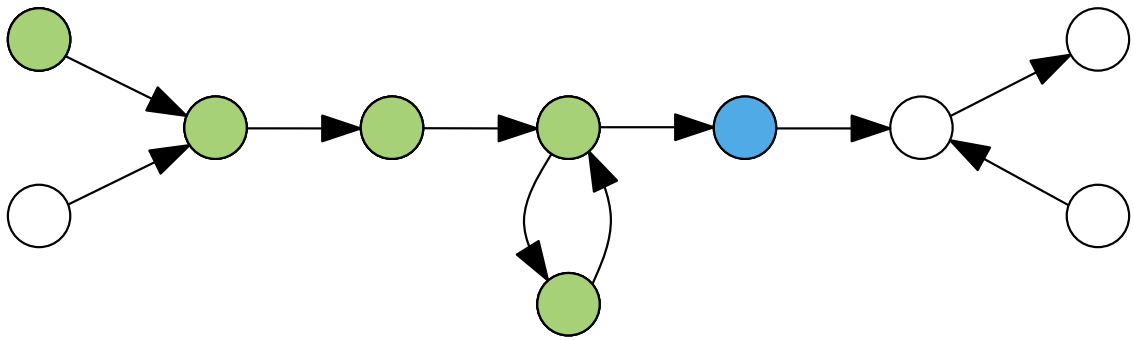
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.
Das dauert.

Shortcut



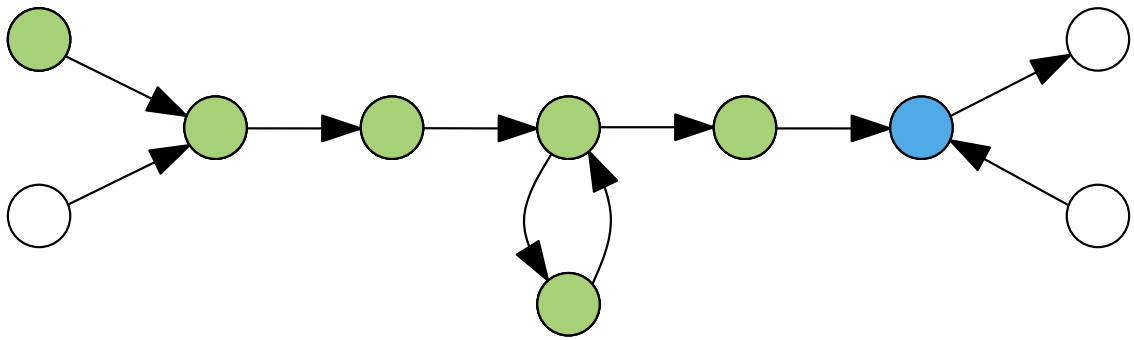
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.
Das dauert.

Shortcut



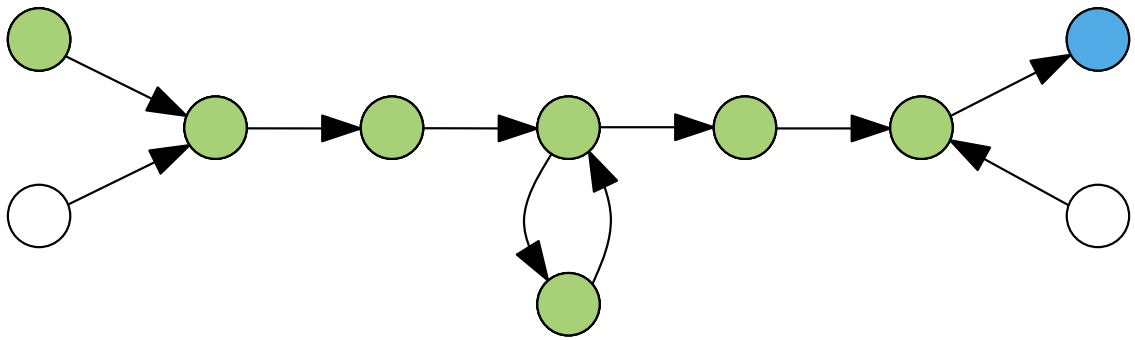
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.
Das dauert.

Shortcut



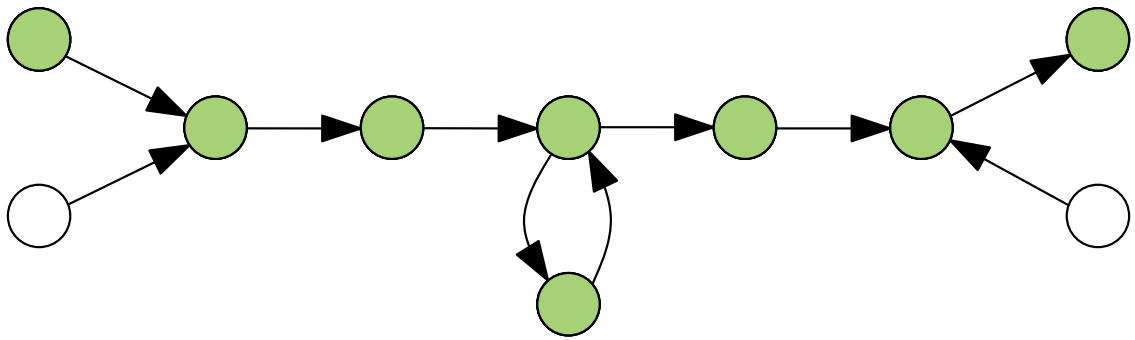
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.
Das dauert.

Shortcut



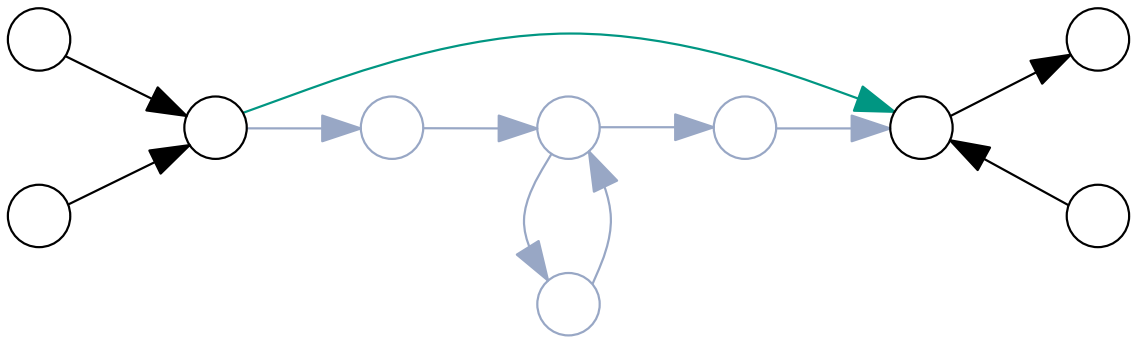
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.
Das dauert.

Shortcut



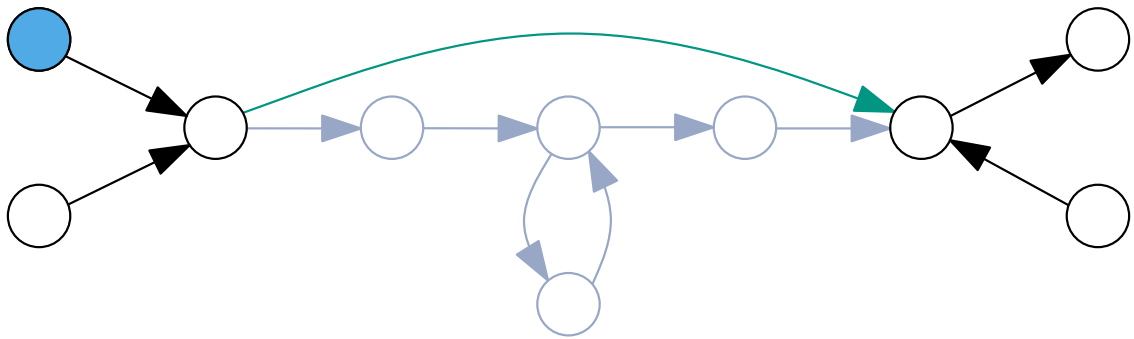
Dijkstras Algorithmus schaut sich alle Zwischenknoten an.
Das dauert.

Shortcut



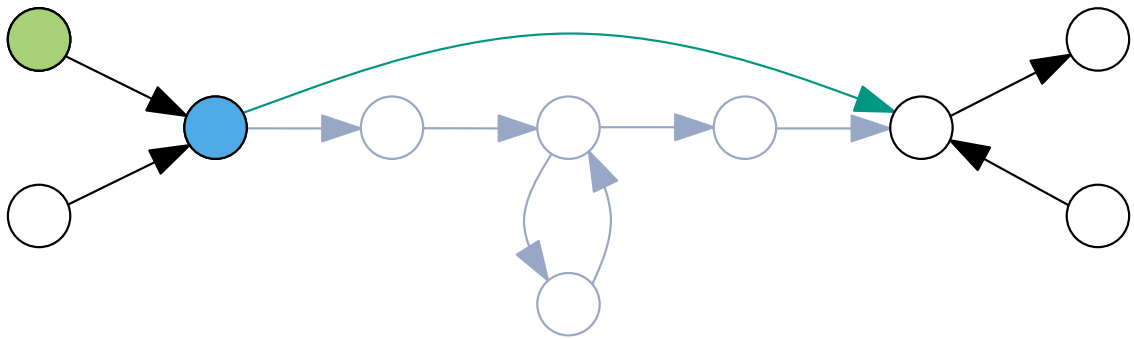
Idee: Füge Shortcut-Kante ein. Grauer Teilgraph muss nur angeschaut werden, wenn s oder t drin liegt.

Shortcut



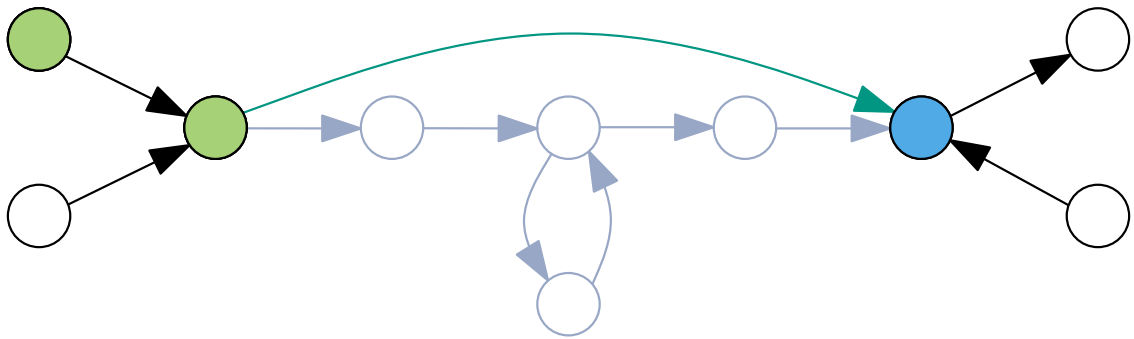
Idee: Füge Shortcut-Kante ein. Grauer Teilgraph muss nur angeschaut werden, wenn s oder t drin liegt.

Shortcut



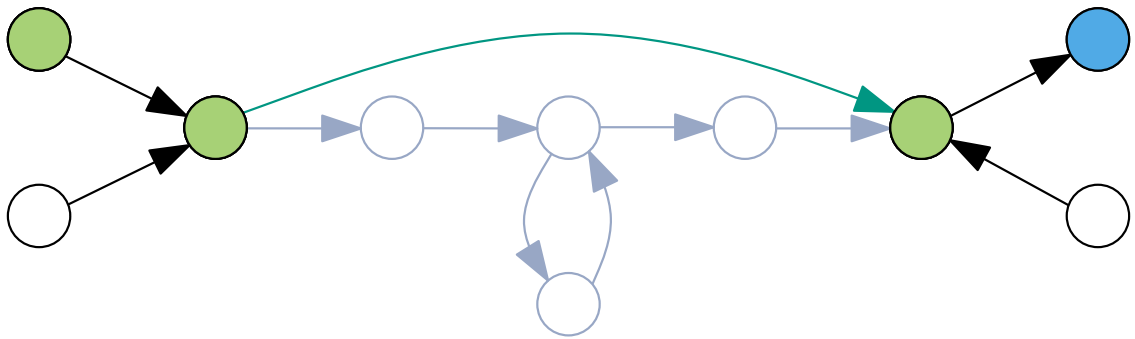
Idee: Füge Shortcut-Kante ein. Grauer Teilgraph muss nur angeschaut werden, wenn s oder t drin liegt.

Shortcut



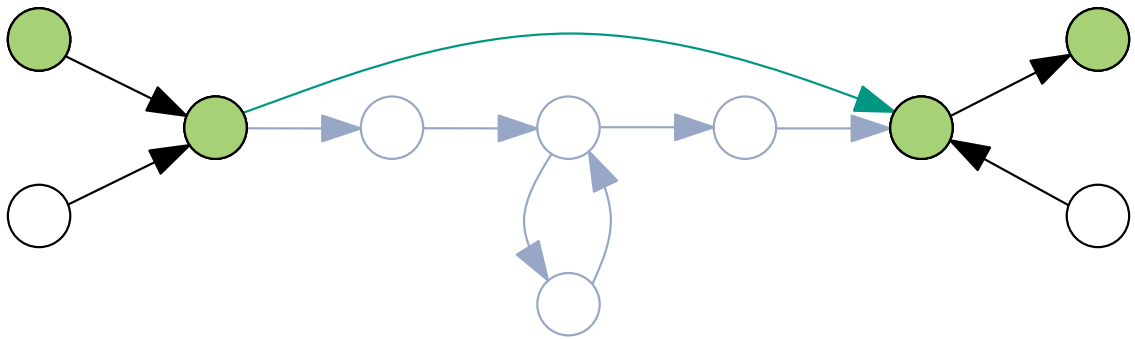
Idee: Füge Shortcut-Kante ein. Grauer Teilgraph muss nur angeschaut werden, wenn s oder t drin liegt.

Shortcut



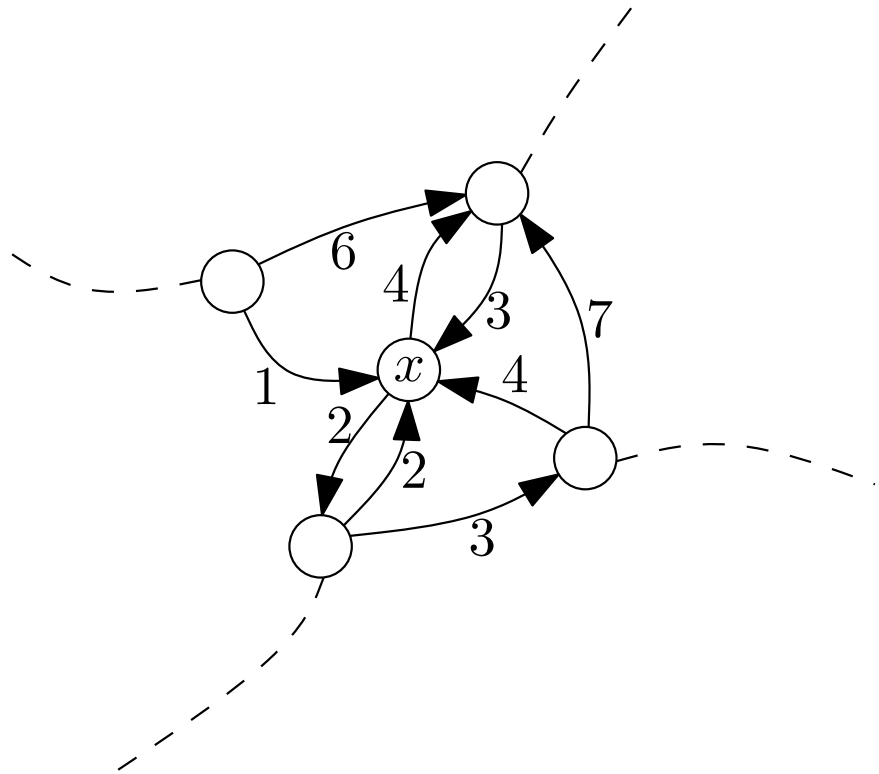
Idee: Füge Shortcut-Kante ein. Grauer Teilgraph muss nur angeschaut werden, wenn s oder t drin liegt.

Shortcut



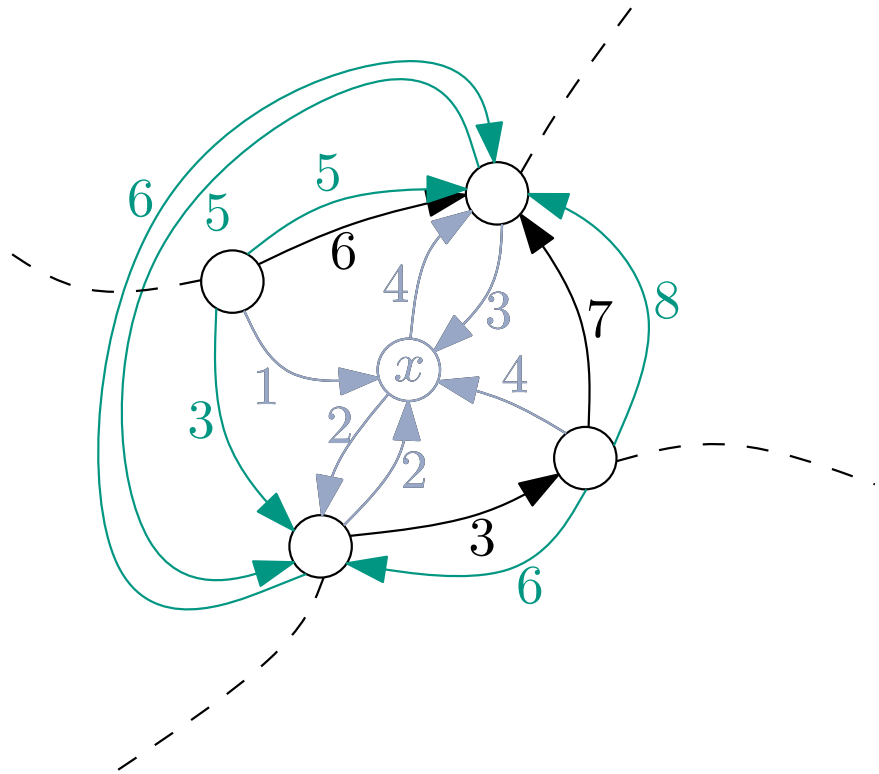
Idee: Füge Shortcut-Kante ein. Grauer Teilgraph muss nur angeschaut werden, wenn s oder t drin liegt.

Knotenkontraktion von x



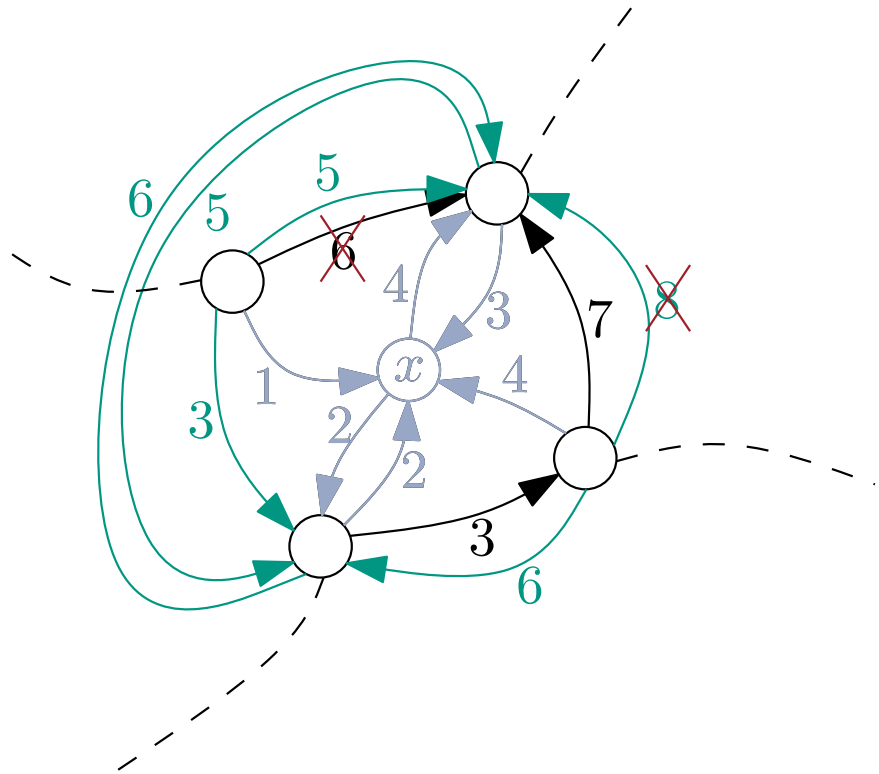
Kontraktion von x: Lösche x und füge Shortcuts zwischen Nachbarn ein, um die Distanzen zwischen allen Knoten zu erhalten

Knotenkontraktion von x



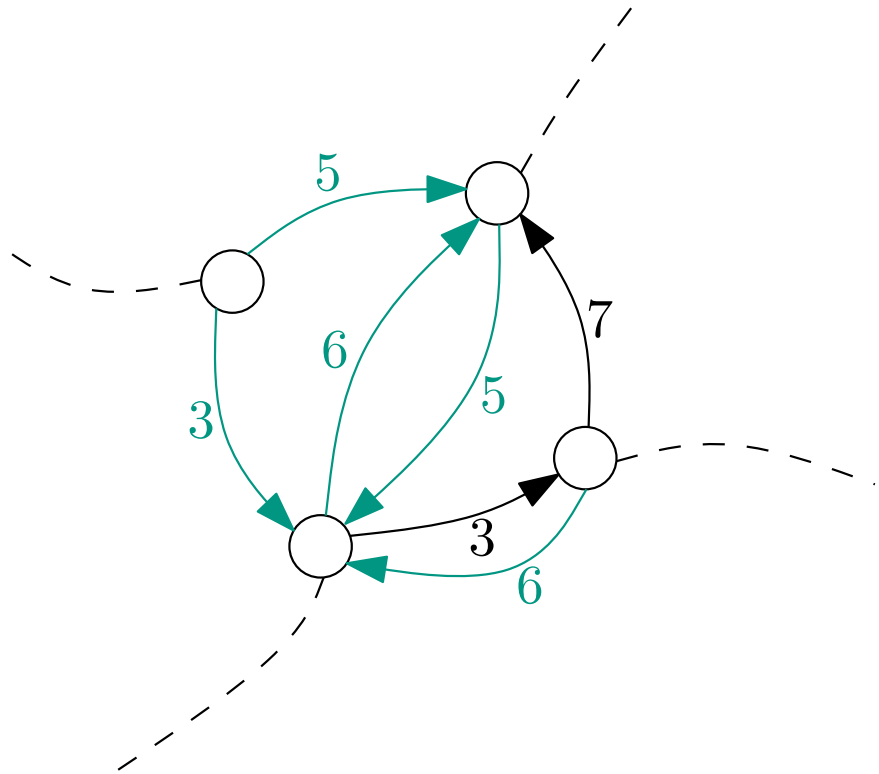
Kontraktion von x: Lösche x und füge Shortcuts zwischen Nachbarn ein, um die Distanzen zwischen allen Knoten zu erhalten

Knotenkontraktion von x

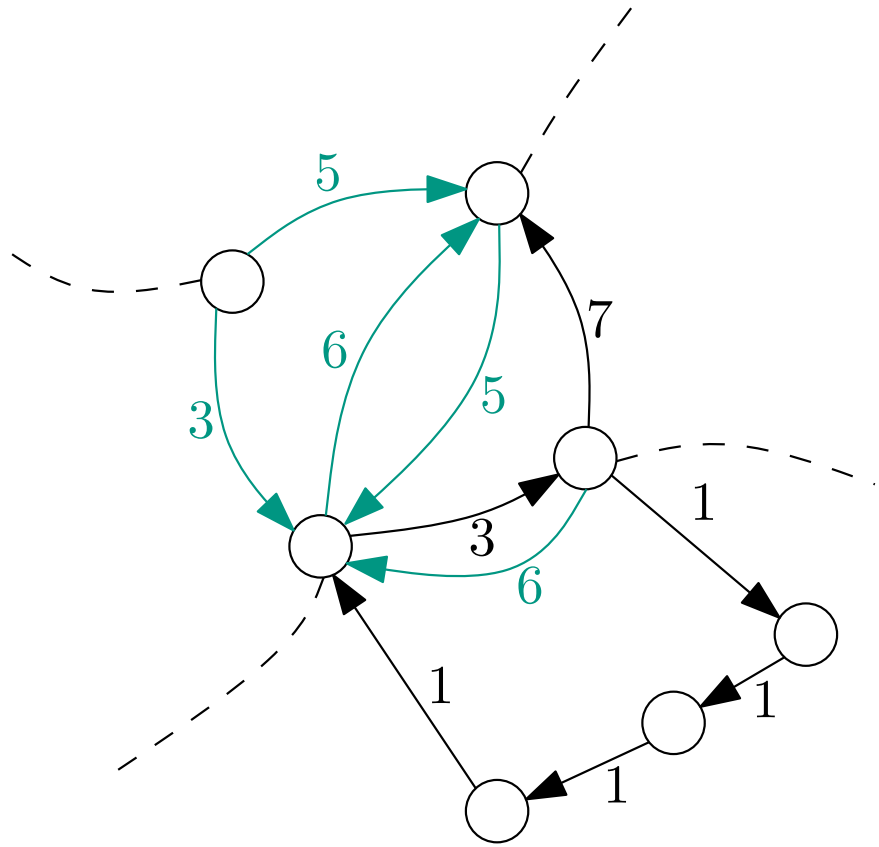


Bei Mehrfachkanten: Längere Kanten verwerfen

Knotenkontraktion von x

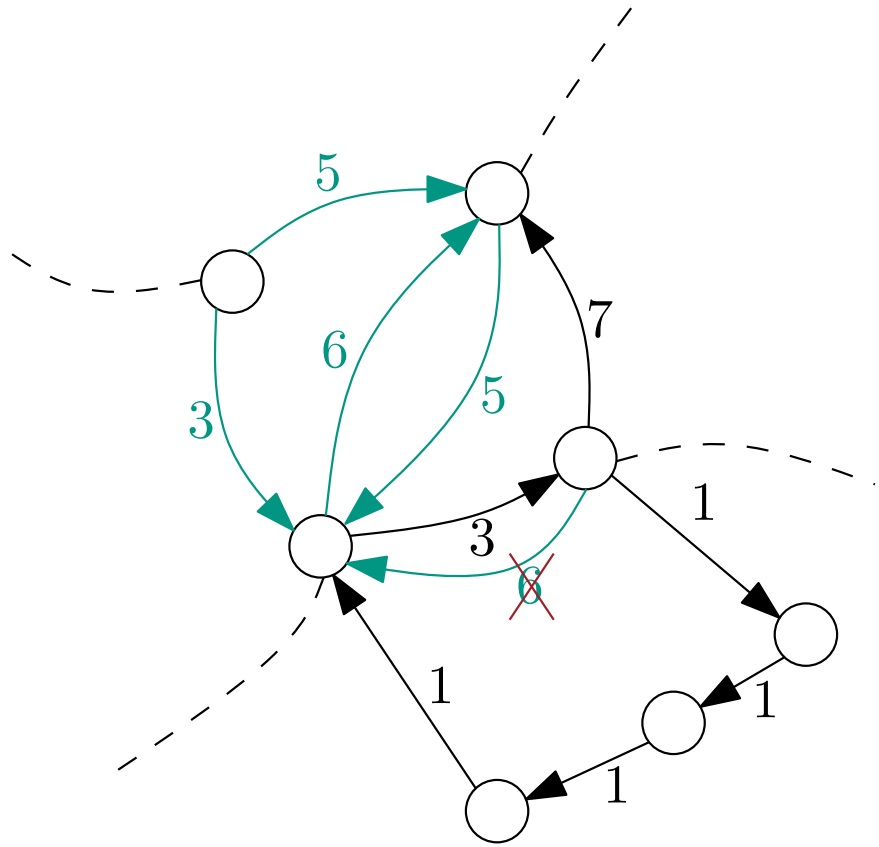


Knotenkontraktion von x



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 $y \to x \to z$ der einzige kürzeste $y - z$-Weg ist

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 $y \to x \to z$ 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

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 $y \to x \to z$ 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.

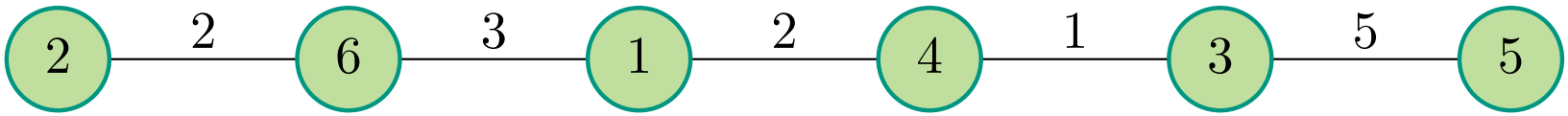
Contraction Hierarchy

Karlsruher Institut für Technologie

Grundidee

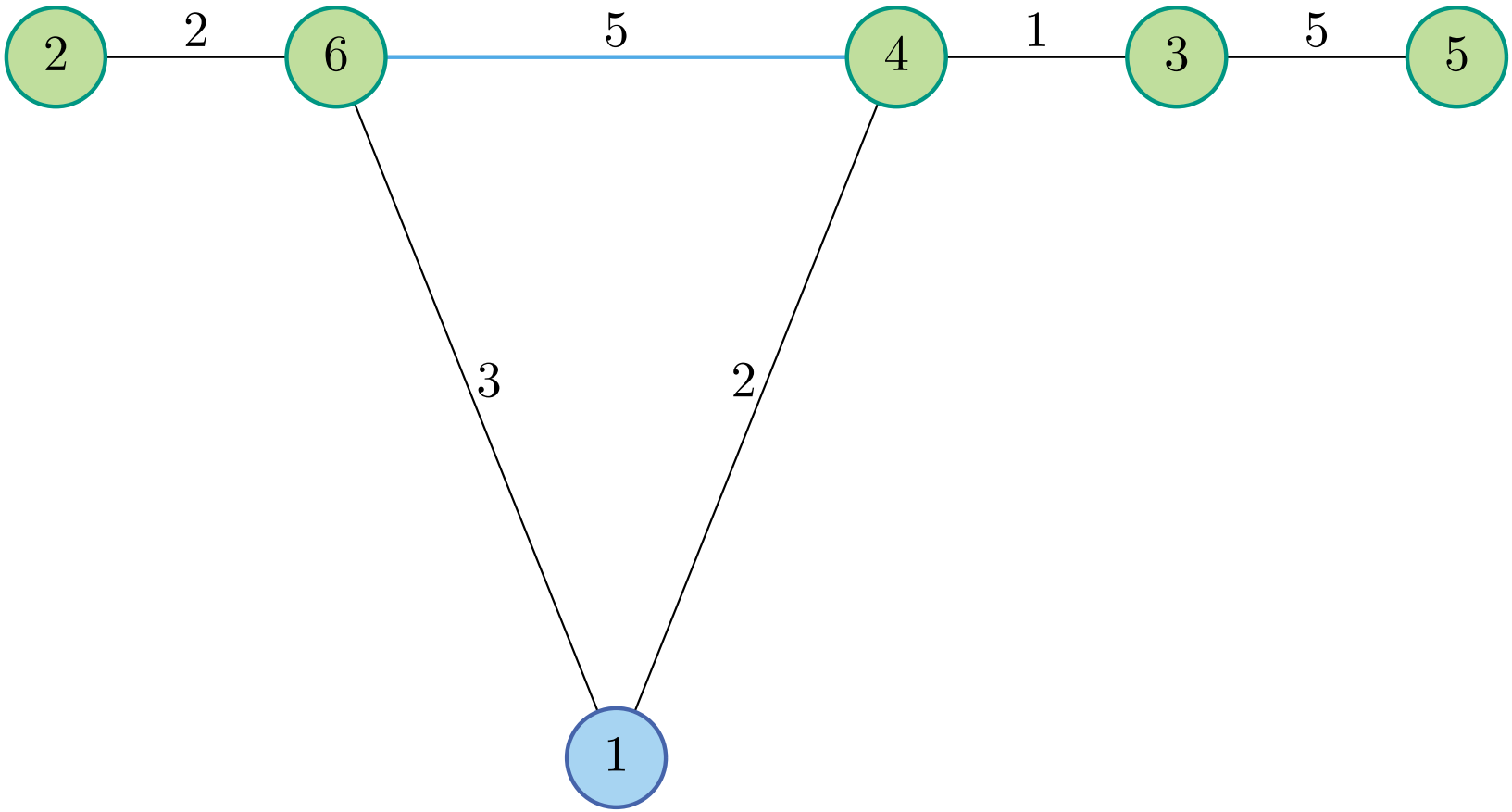
- Eingabegraph G
- Ordne Knoten von G nach "Wichtigkeit": $v_1 \dots v_n$
- Kontrahiere Knoten iterativ aus G heraus
  - zuerst den "unwichtigsten" Knoten v_1
  - den "wichtigsten" Knoten v_n als letztes
- Graph mit Shortcuts heißt augmentierter Graph

Contraction Hierarchy



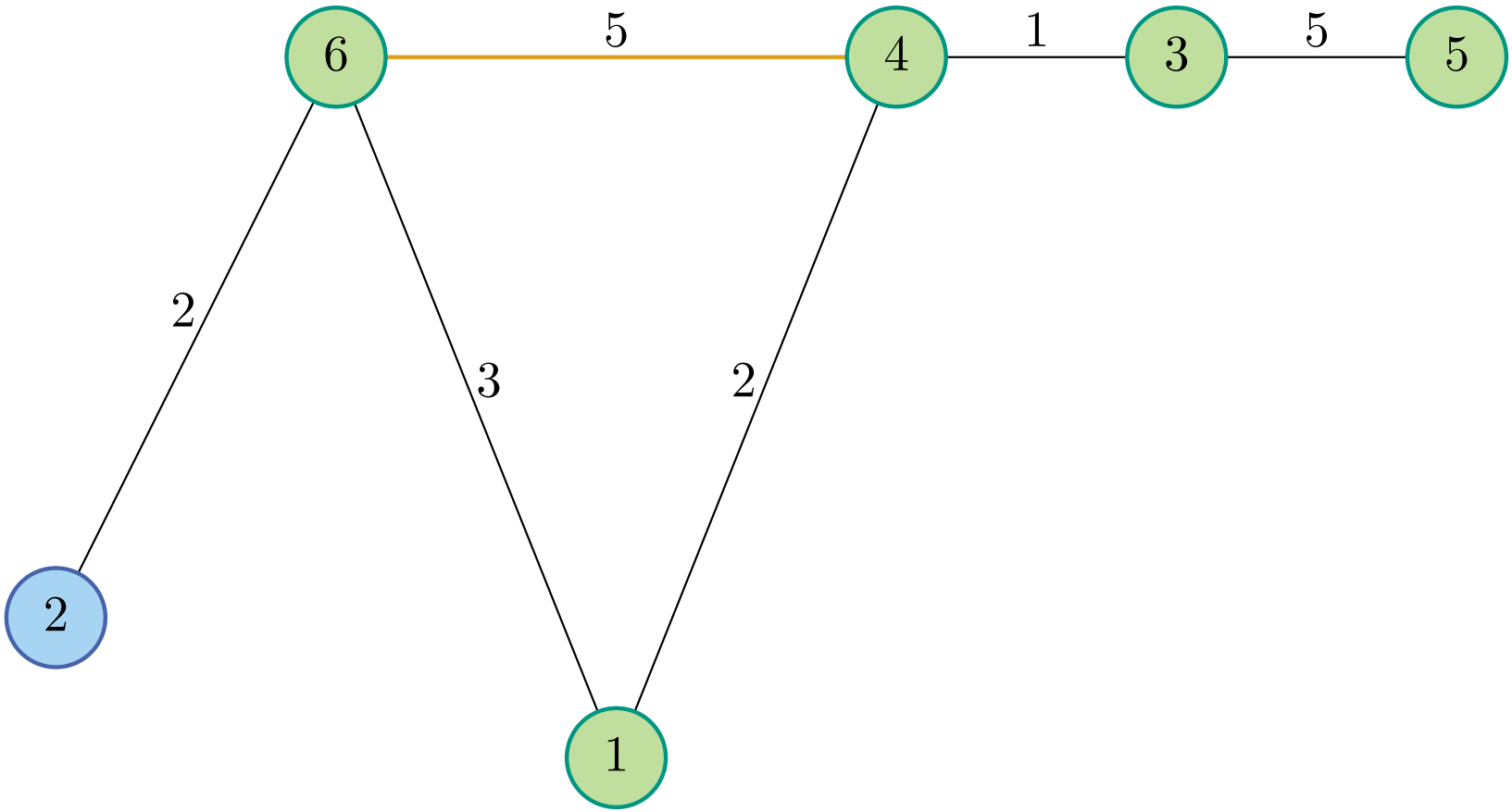
Knoten nummeriert nach “Wichtigkeit”

Contraction Hierarchy



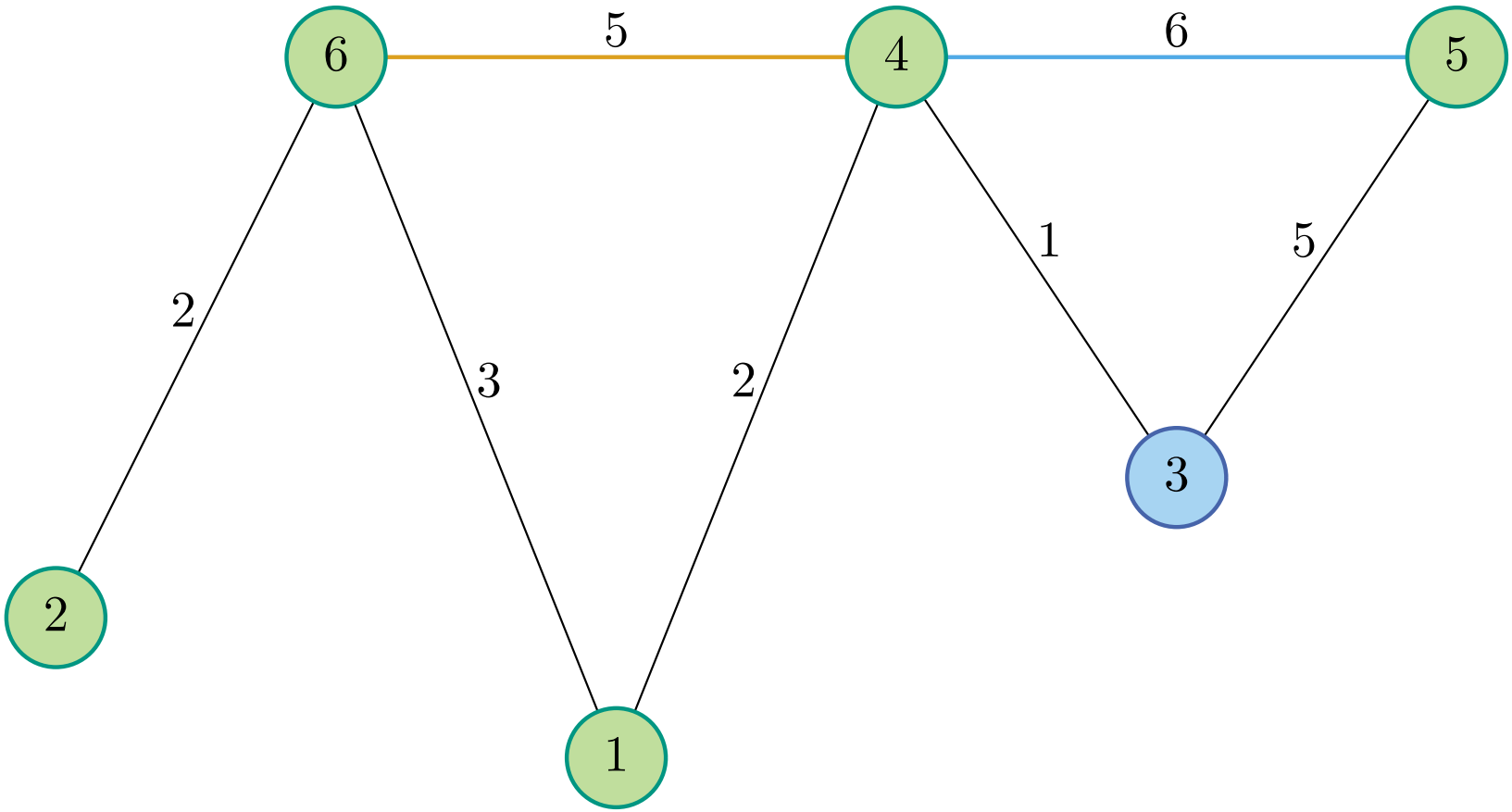
Knoten nummeriert nach “Wichtigkeit”

Contraction Hierarchy



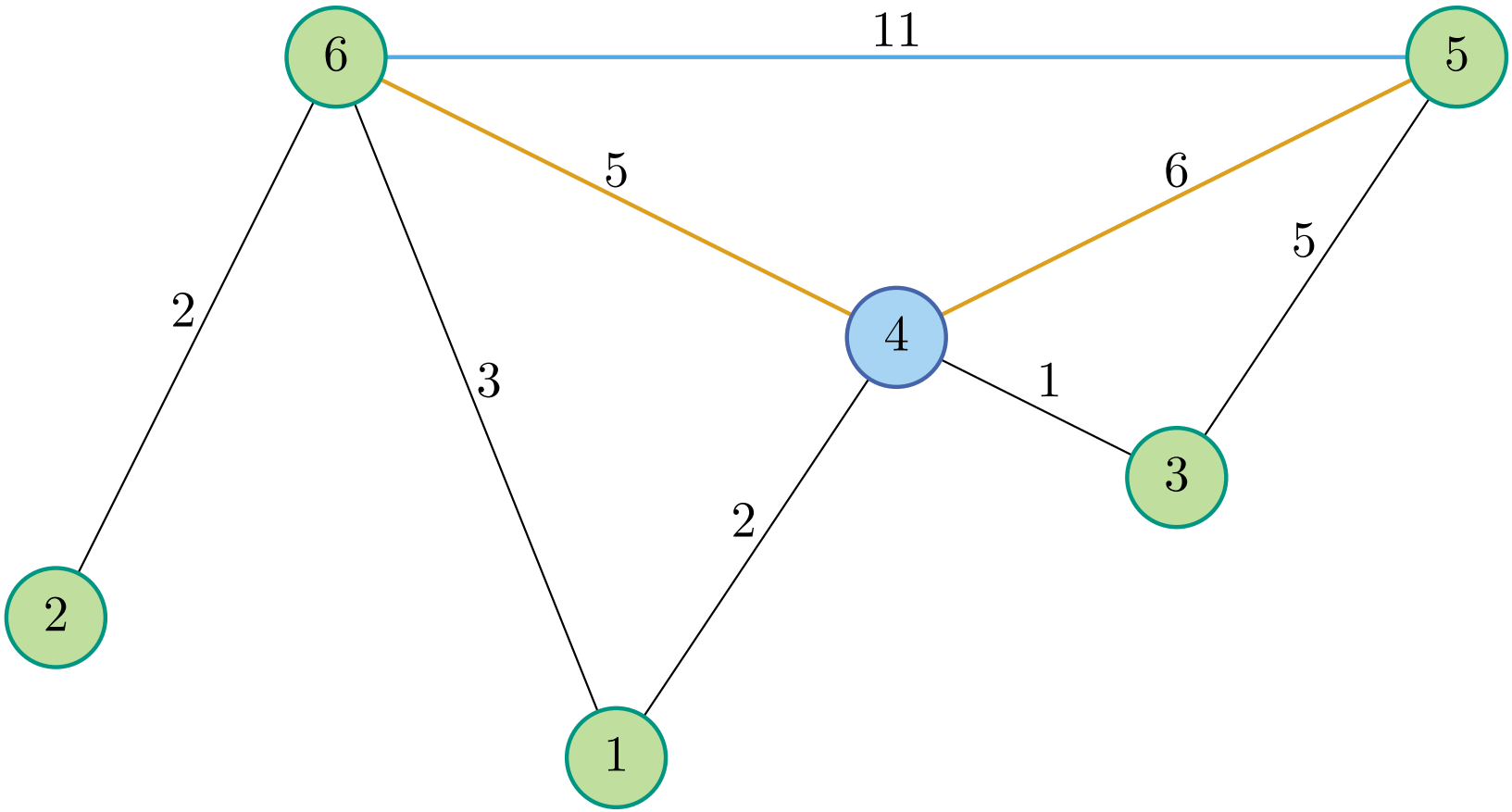
Knoten nummeriert nach “Wichtigkeit”

Contraction Hierarchy



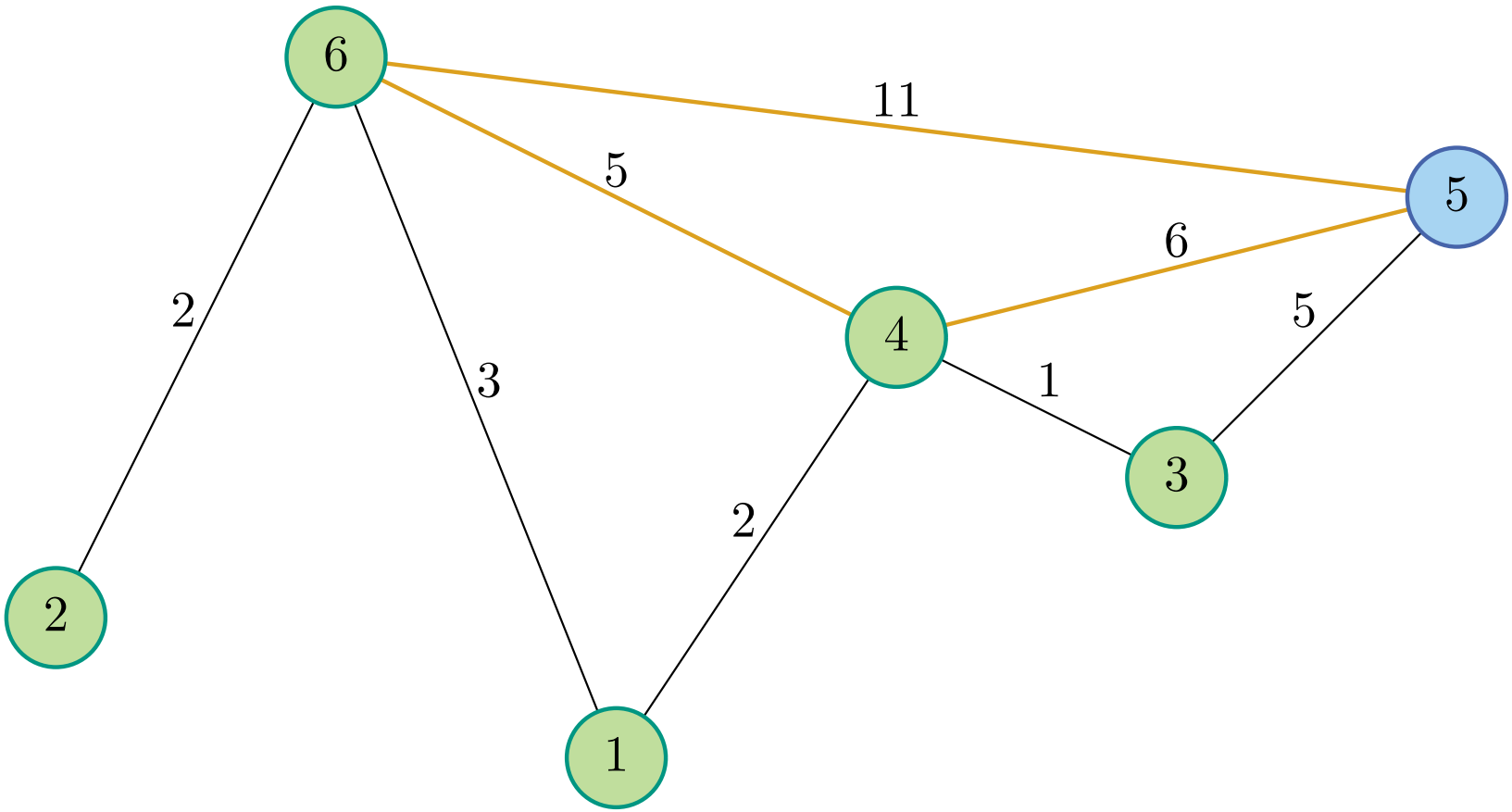
Knoten nummeriert nach “Wichtigkeit”

Contraction Hierarchy



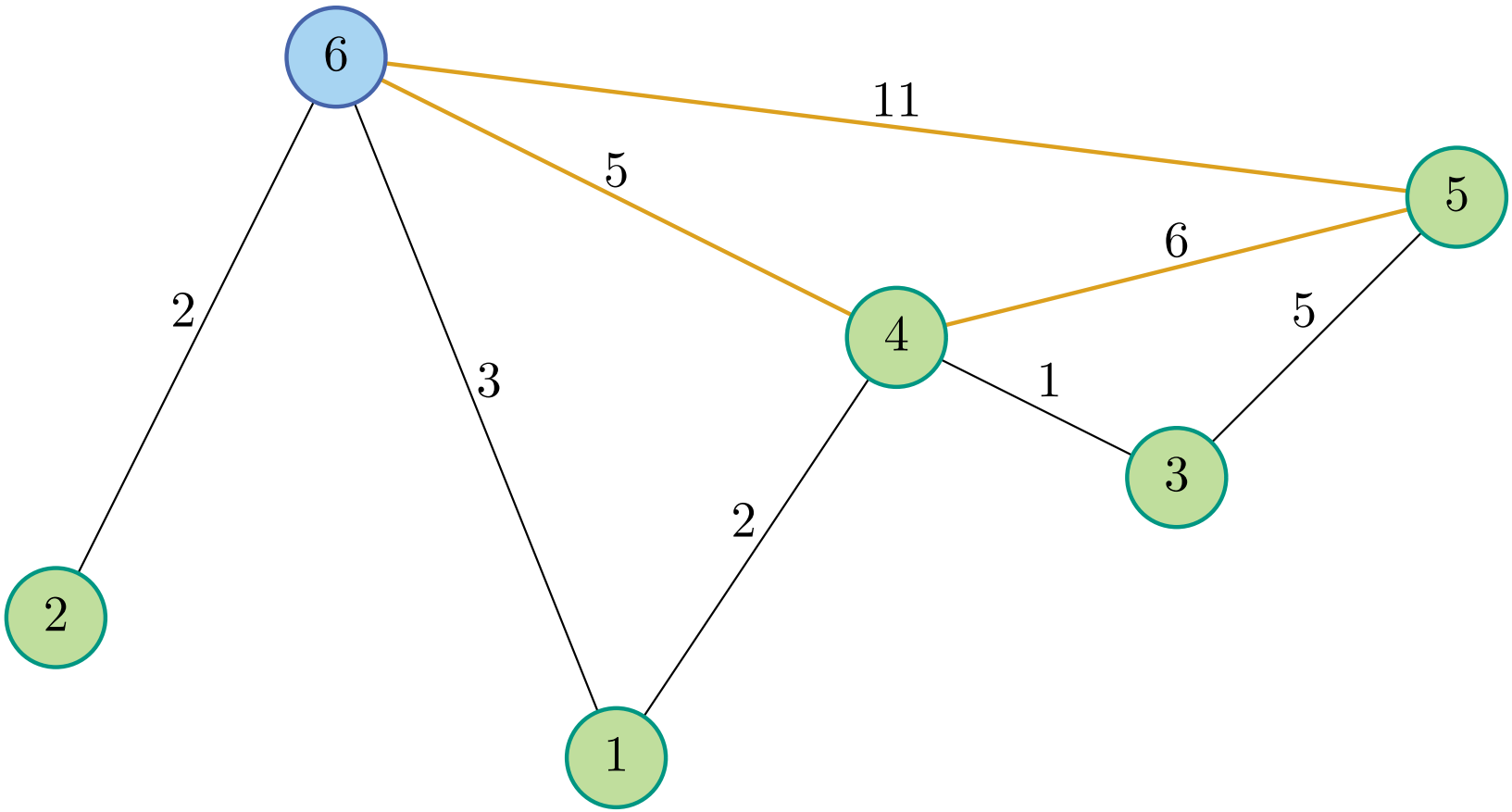
Knoten nummeriert nach “Wichtigkeit”

Contraction Hierarchy



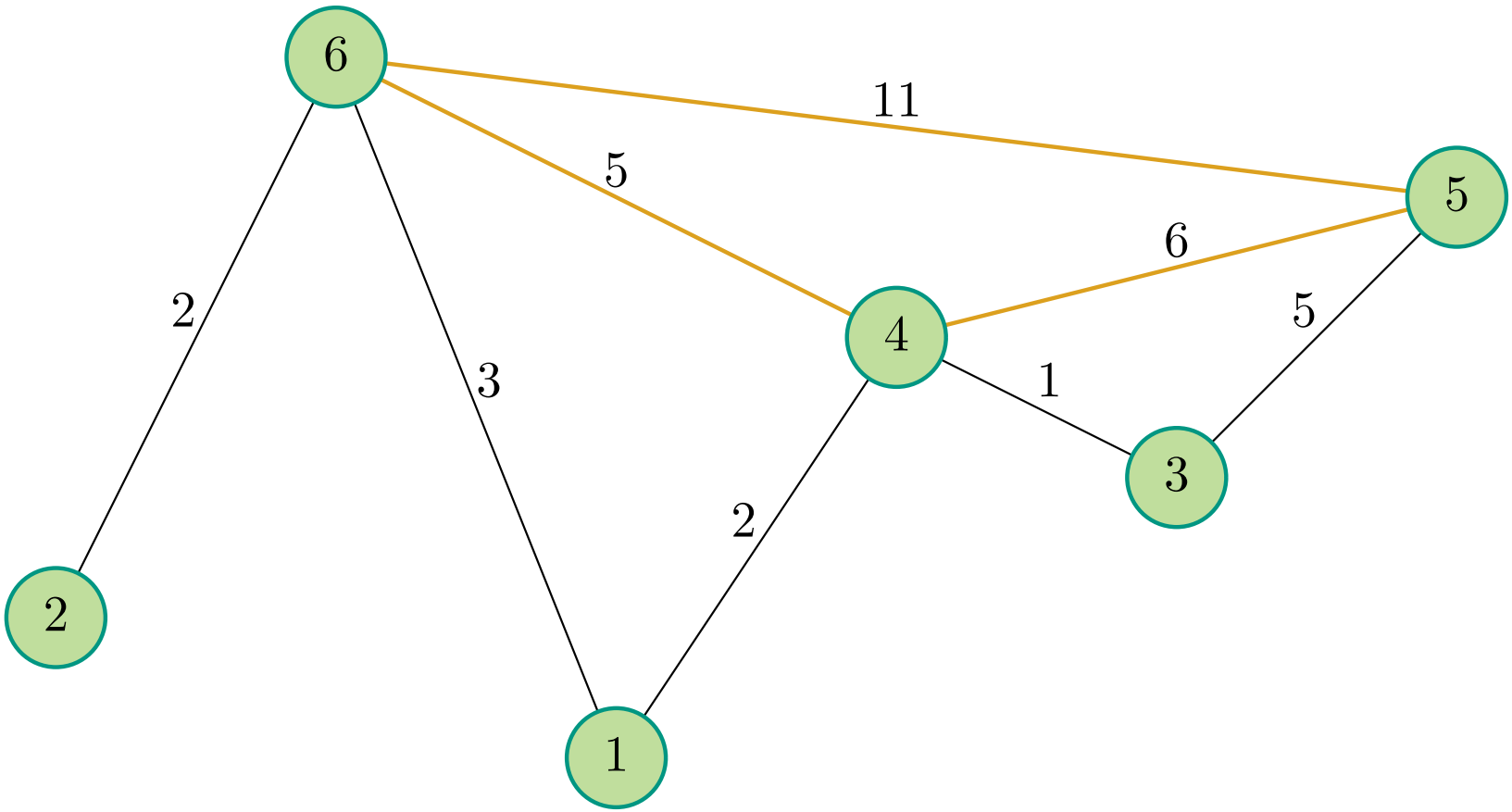
Knoten nummeriert nach “Wichtigkeit”

Contraction Hierarchy



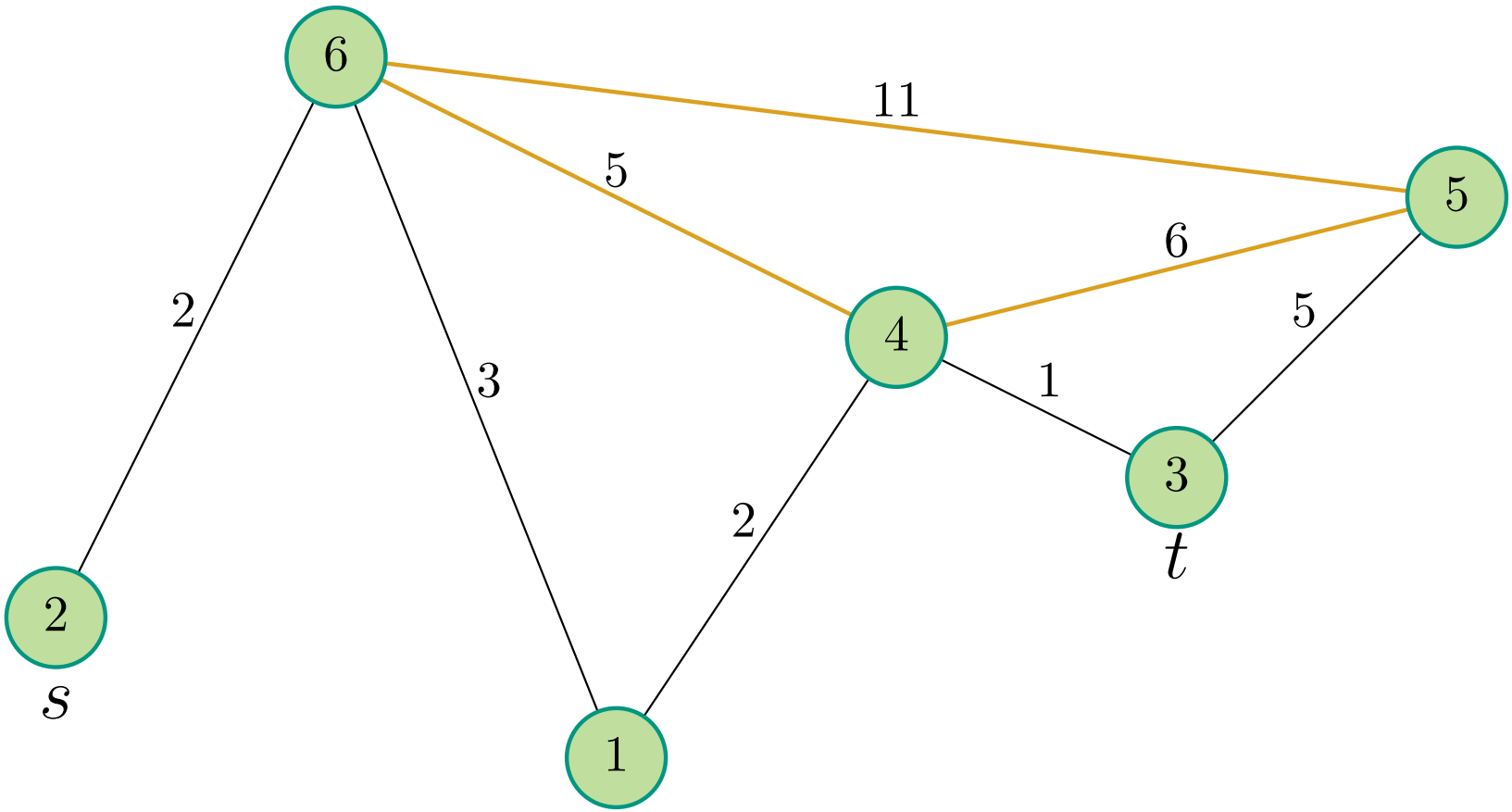
Knoten nummeriert nach “Wichtigkeit”

Contraction Hierarchy



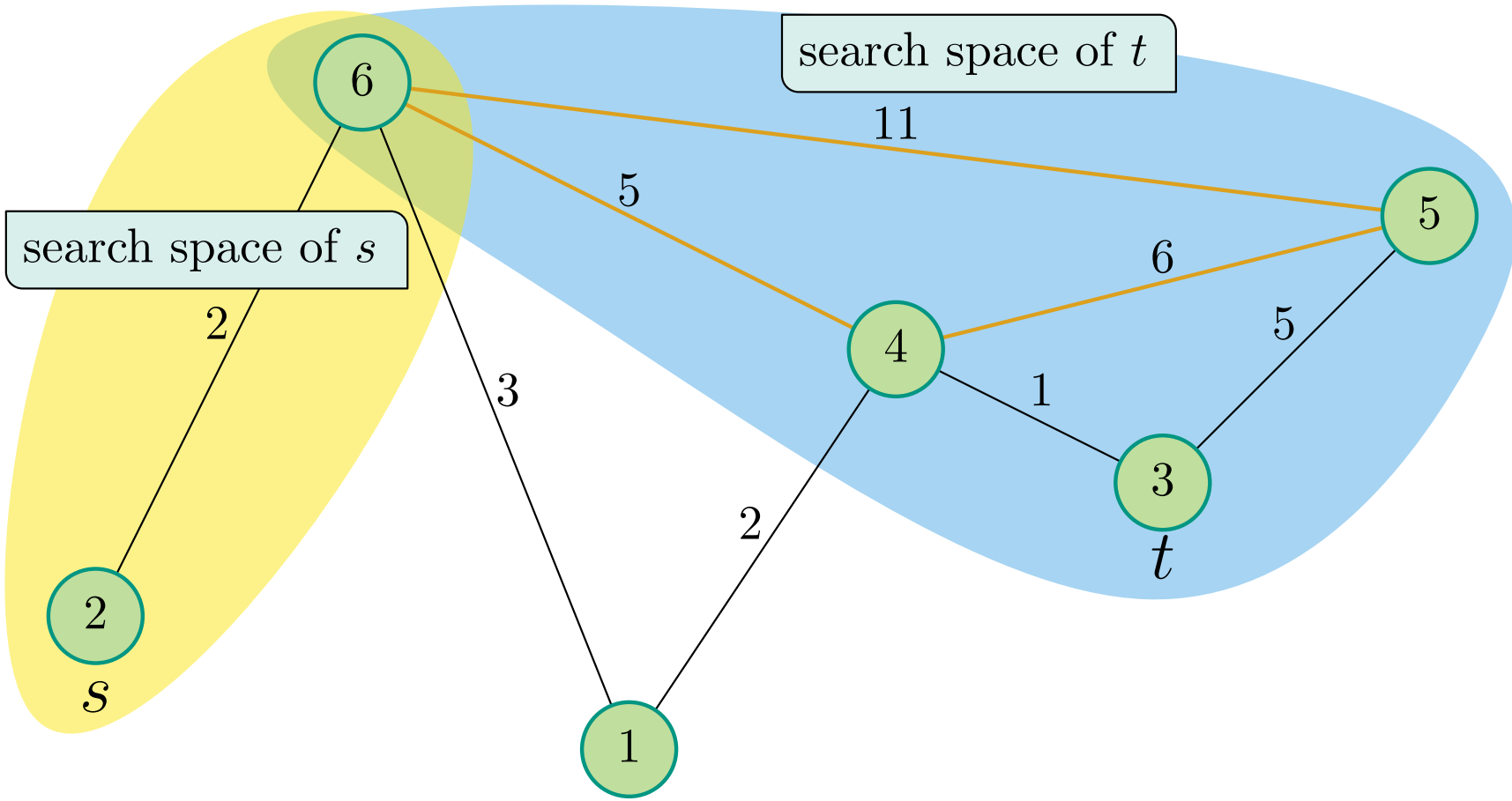
Knoten nummeriert nach “Wichtigkeit”

Contraction Hierarchy



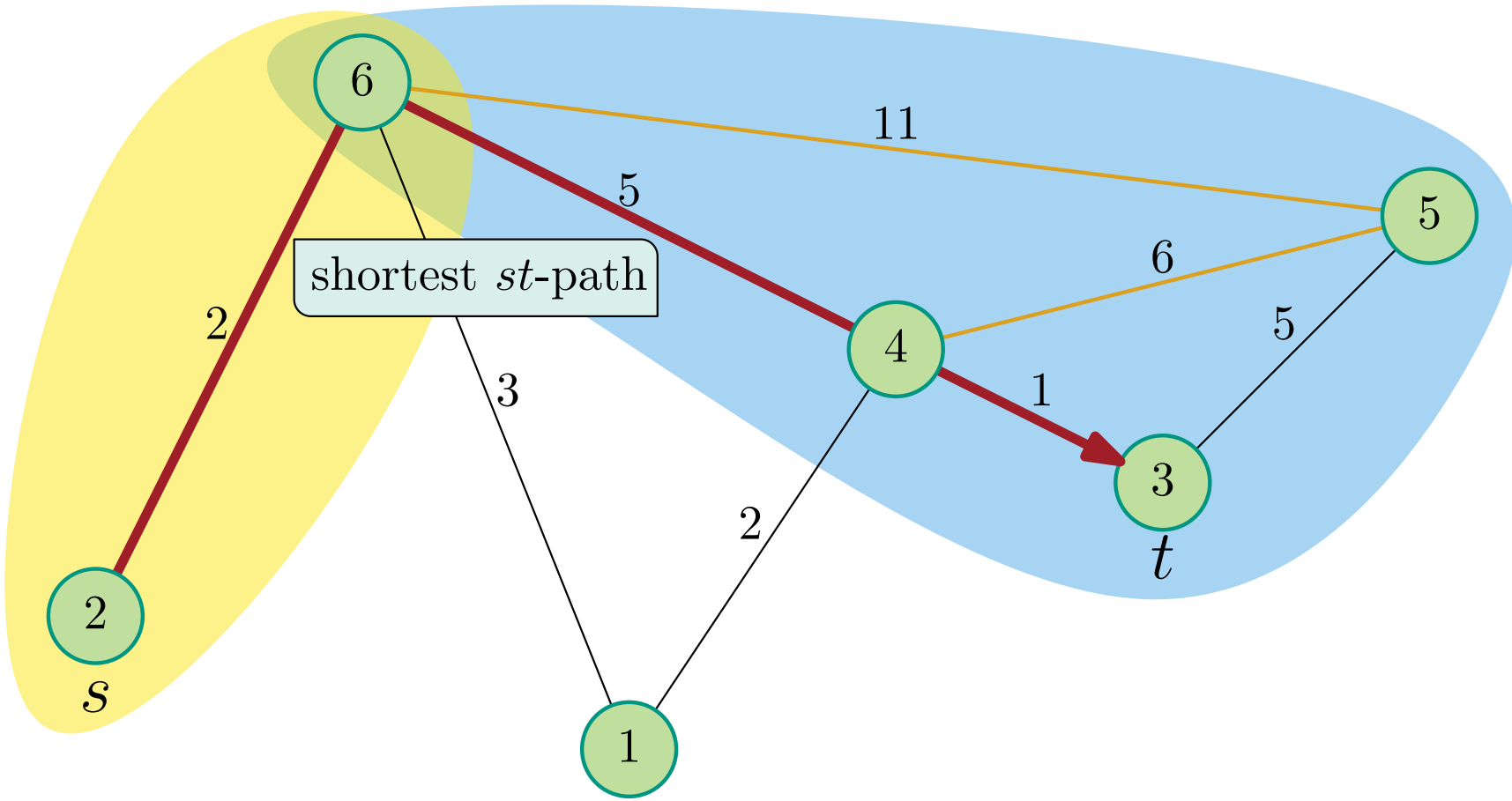
Knoten nummeriert nach “Wichtigkeit”

Contraction Hierarchy



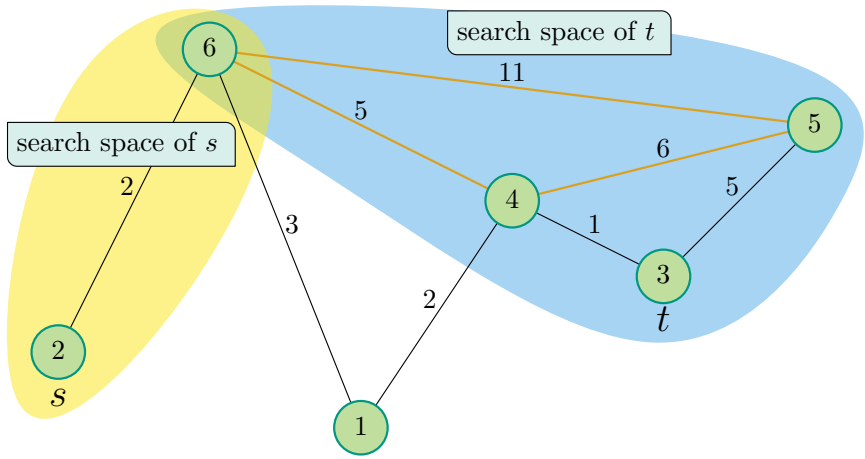
Knoten nummeriert nach “Wichtigkeit”

Contraction Hierarchy



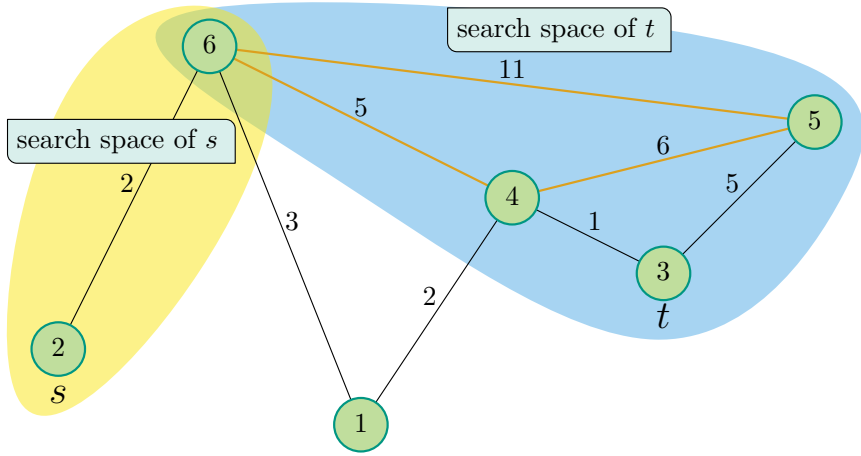
Für jeden ursprünglichen kürzesten Weg gibt es einen hoch-runter-Pfad

Anfrage



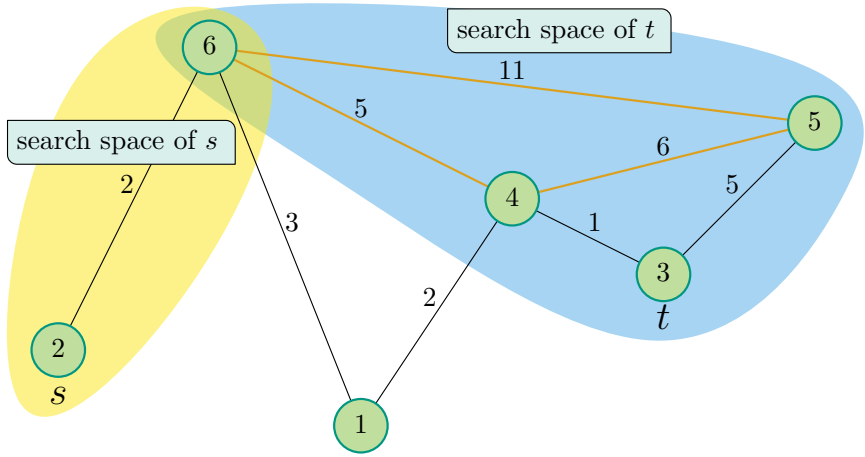
- Bidirektionale Variante von Dijkstras Algorithmus
- Verfolge nur Kanten zu wichtigeren Knoten (unwichtige sind durch Shortcuts abgedeckt)

Anfrage



- Bidirektionale Variante von Dijkstras Algorithmus
- Verfolge nur Kanten zu wichtigeren Knoten (unwichtige sind durch Shortcuts abgedeckt)
- Vorwärtssuche findet den “hoch”-Teil des Pfads
- Rückwärtssuche findet den “runter”-Teil des Pfads

Anfrage



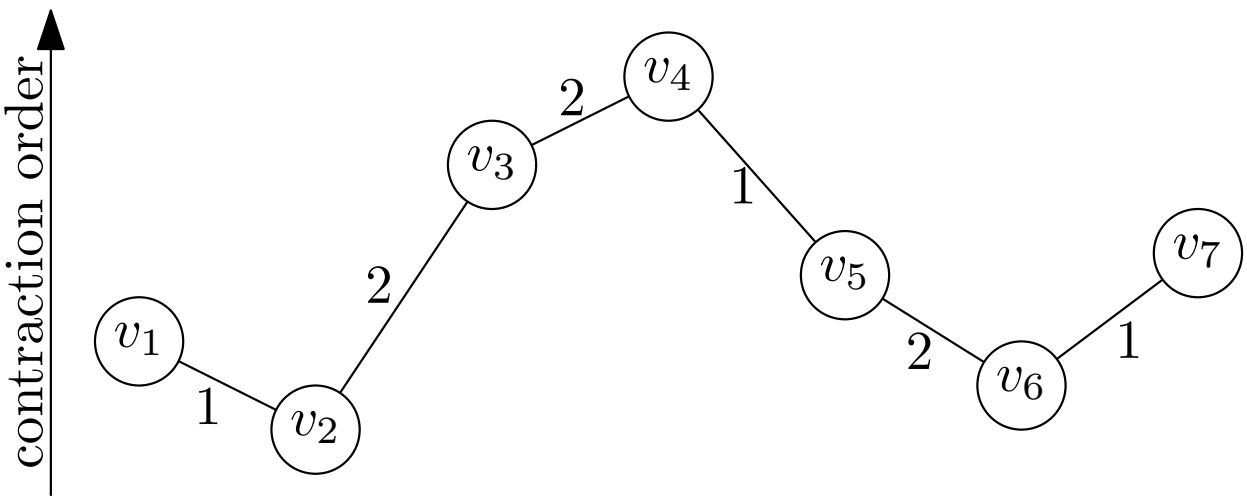
- Bidirektionale Variante von Dijkstras Algorithmus
- Verfolge nur Kanten zu wichtigeren Knoten (unwichtige sind durch Shortcuts abgedeckt)
- Vorwärtssuche findet den “hoch”-Teil des Pfads
- Rückwärtssuche findet den “runter”-Teil des Pfads
- Abbruch, wenn der min-key beider Queues größer ist als der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

Korrektheit

- Vorwärtssuche findet nur Aufwärtspfade.
- Rückwärtssuche findet nur Abwärtspfade.
- gefundene Pfade gehen erst hoch und dann runter
- **zu zeigen:** es gibt in G^+ immer einen kürzesten Pfad, der ein hoch-runter Pfad ist

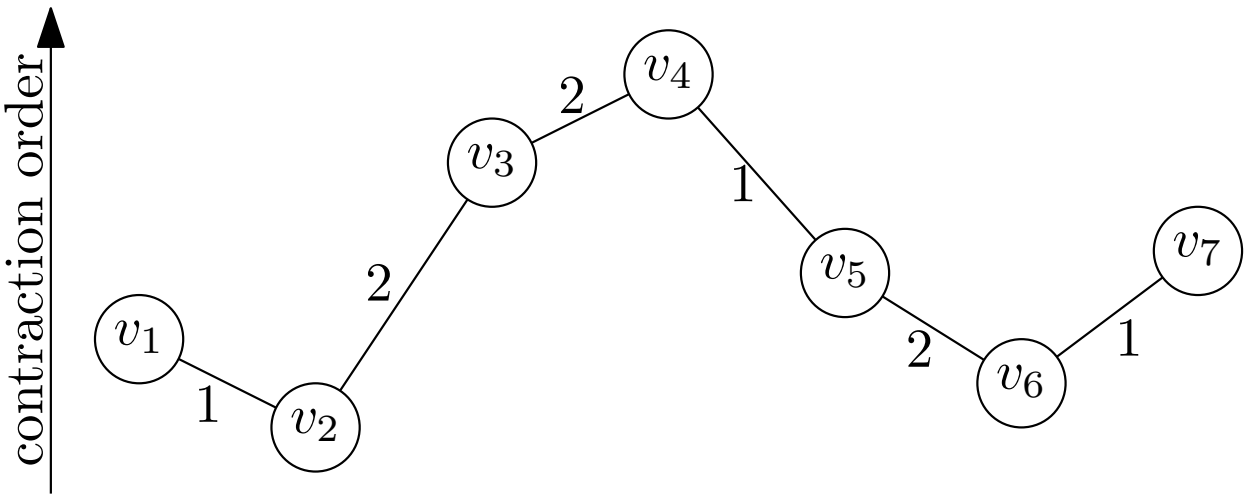
Korrektheit

- Vorwärtssuche findet nur Aufwärtspfade.
- Rückwärtssuche findet nur Abwärtspfade.
- gefundene Pfade gehen erst hoch und dann runter
- **zu zeigen:** es gibt in G^+ immer einen kürzesten Pfad, der ein hoch-runter Pfad ist
- Beweis durch Betrachten eines kürzesten Pfads



Korrektheit

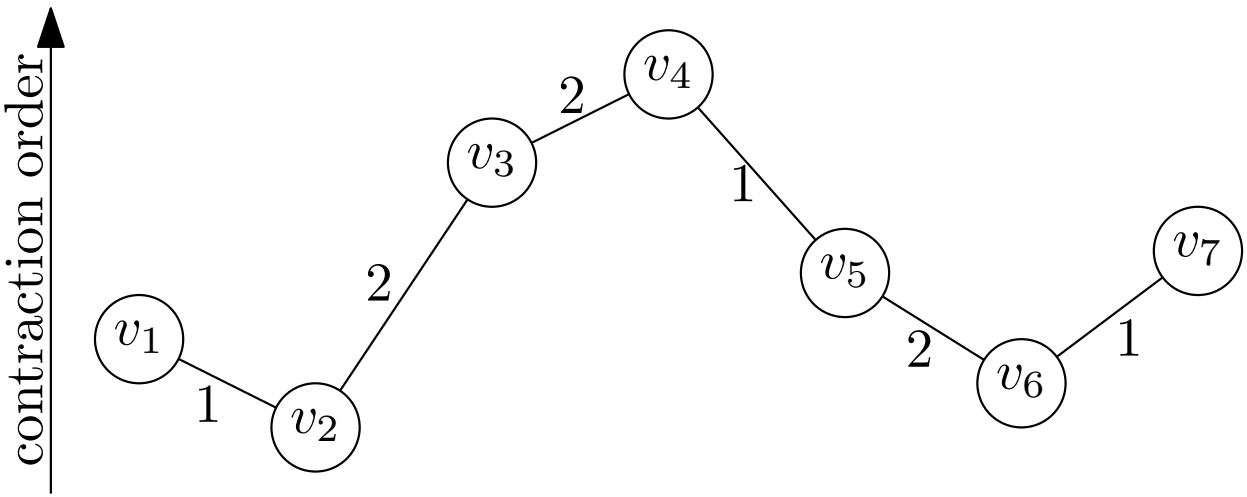
- **zu zeigen:** es gibt in G^+ immer einen kürzesten Pfad, der ein hoch-runter Pfad ist



- Es gibt einen kürzesten Weg P

Korrektheit

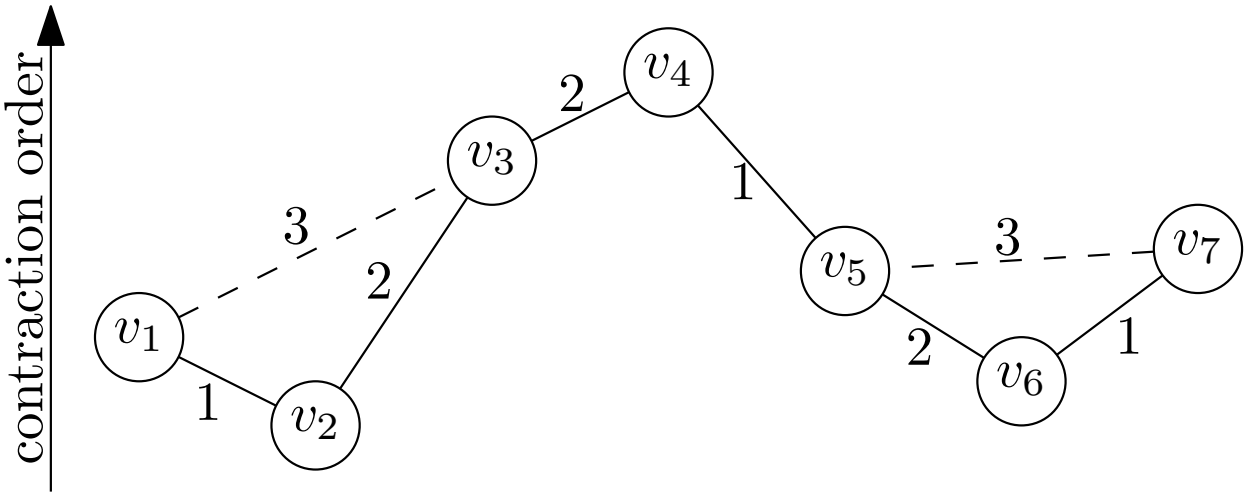
- **zu zeigen:** es gibt in G^+ immer einen kürzesten Pfad, der ein hoch-runter Pfad ist



- Es gibt einen kürzesten Weg P
- Wenn P kein hoch-runter Pfad ist, beinhaltet P es einen Knoten, dessen Nachbarn beide wichtiger sind

Korrektheit

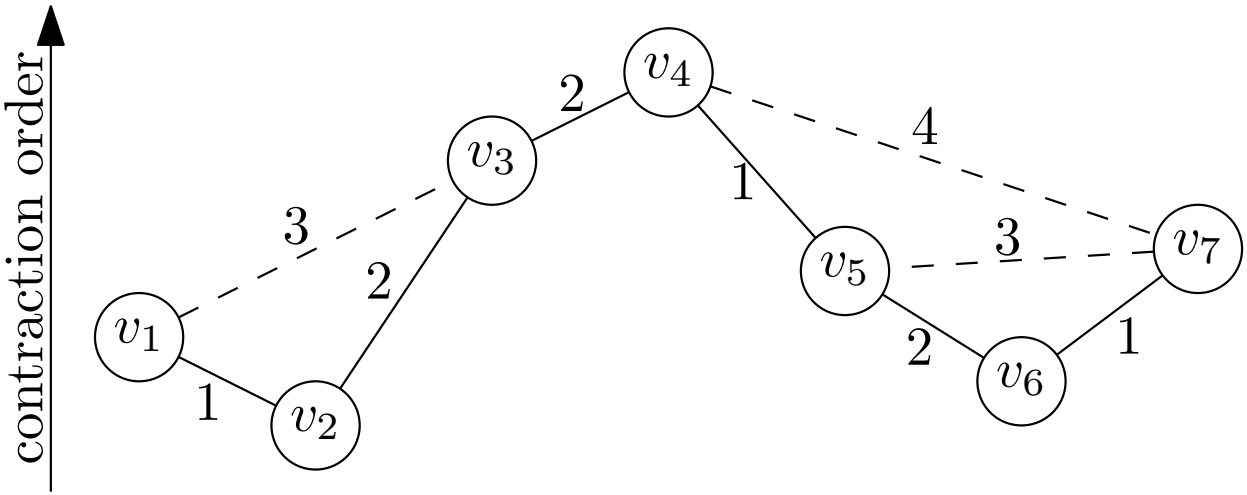
- **zu zeigen:** es gibt in G^+ immer einen kürzesten Pfad, der ein hoch-runter Pfad ist



- Es gibt einen kürzesten Weg P
- Wenn P kein hoch-runter Pfad ist, beinhaltet P es einen Knoten, dessen Nachbarn beide wichtiger sind
- Dann gibt es auch Shortcut oder Zeugen zw. den Nachbarn

Korrektheit

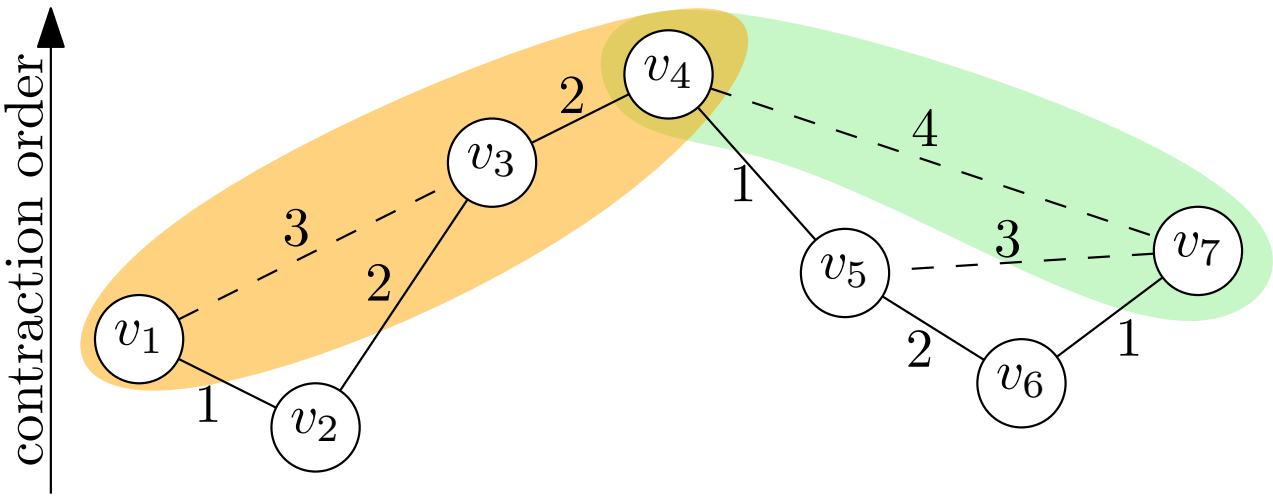
- **zu zeigen:** es gibt in G^+ immer einen kürzesten Pfad, der ein hoch-runter Pfad ist



- Es gibt einen kürzesten Weg P
- Wenn P kein hoch-runter Pfad ist, beinhaltet P es einen Knoten, dessen Nachbarn beide wichtiger sind
- Dann gibt es auch Shortcut oder Zeugen zw. den Nachbarn

Korrektheit

- **zu zeigen:** es gibt in G^+ immer einen kürzesten Pfad, der ein hoch-runter Pfad ist

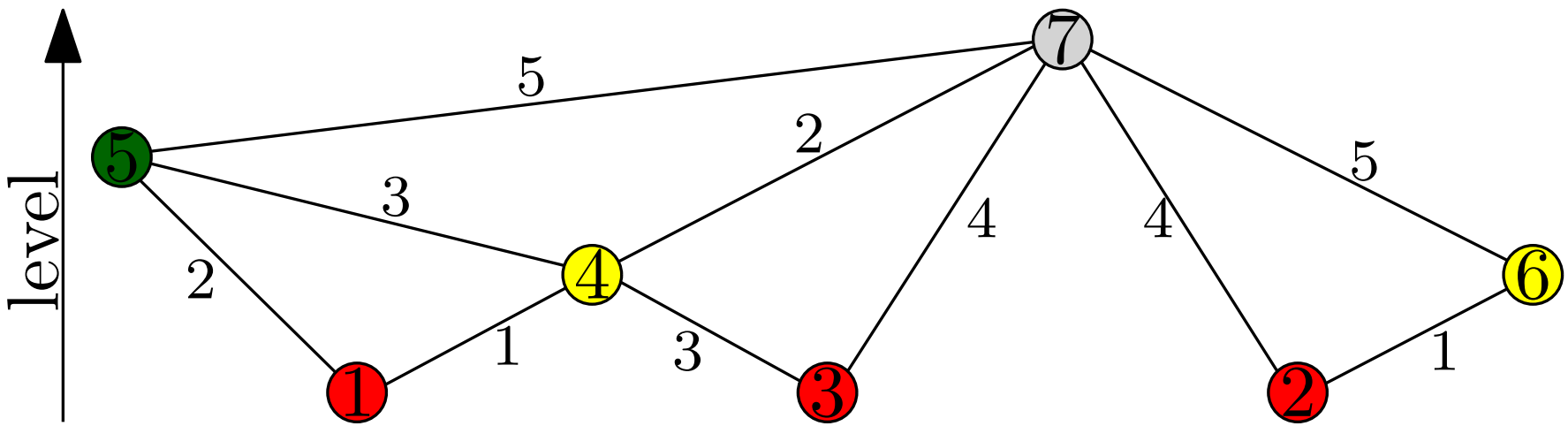


- Es gibt einen kürzesten Weg P
- Wenn P kein hoch-runter Pfad ist, beinhaltet P es einen Knoten, dessen Nachbarn beide wichtiger sind
- Dann gibt es auch Shortcut oder Zeugen zw. den Nachbarn

Für jeden kürzesten Pfad gibt es einen hoch runter Pfad gleicher Länge

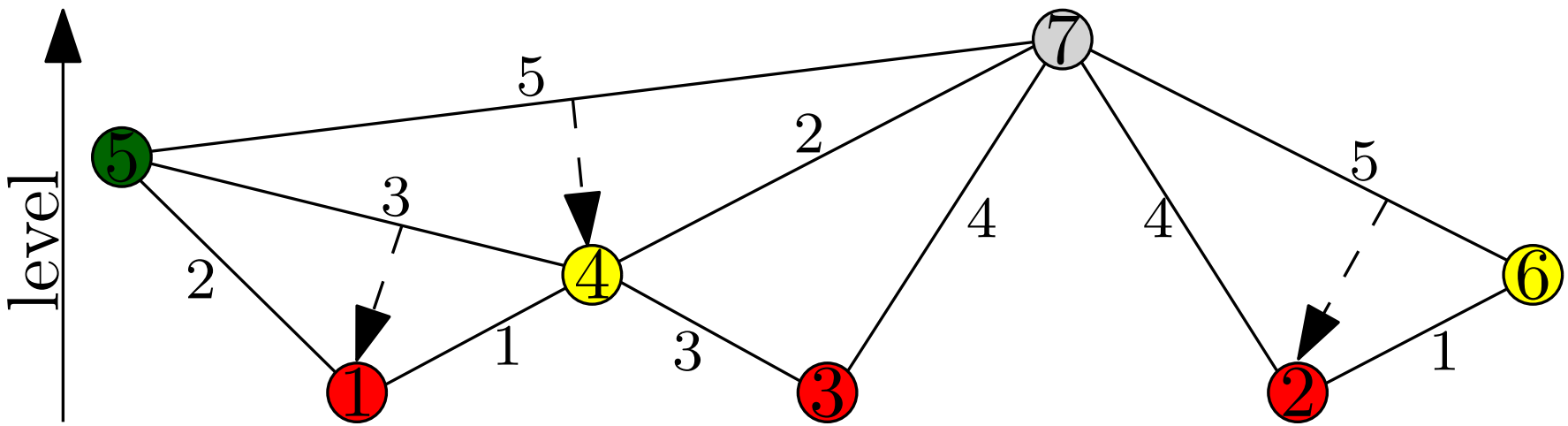
Pfadentpackung

- für jeden Shortcut (u, w) eines Pfades (u, v, w), speichere Mittelknoten v an der Kante
- expandiere Pfade mittels Rekursion



Pfadentpackung

- für jeden Shortcut (u, w) eines Pfades (u, v, w), speichere Mittelknoten v an der Kante
- expandiere Pfade mittels Rekursion



Nach “Wichtigkeit” Ordnen

Grund-Idee:

- Wir wollen wenig Shortcuts
- Ein Knoten ist “unwichtig”, wenn er wenig Shortcuts erzeugt

$\rightarrow$ simuliere Knotenkontraktion, um Knoten zu gewichten

Nach “Wichtigkeit” Ordnen

Grund-Idee:

- Wir wollen wenig Shortcuts
- Ein Knoten ist “unwichtig”, wenn er wenig Shortcuts erzeugt

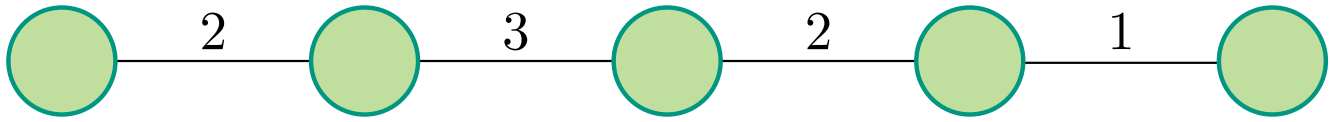
$\rightarrow$ simuliere Knotenkontraktion, um Knoten zu gewichten

Algorithmus:

- Baue eine nach “Wichtigkeit” sortierte Warteschlange mit allen Knoten
- Kontrahiere iterativ unwichtigsten Knoten
- Kontraktion eines Knotens kann “Wichtigkeit” der Nachbarn beeinflussen

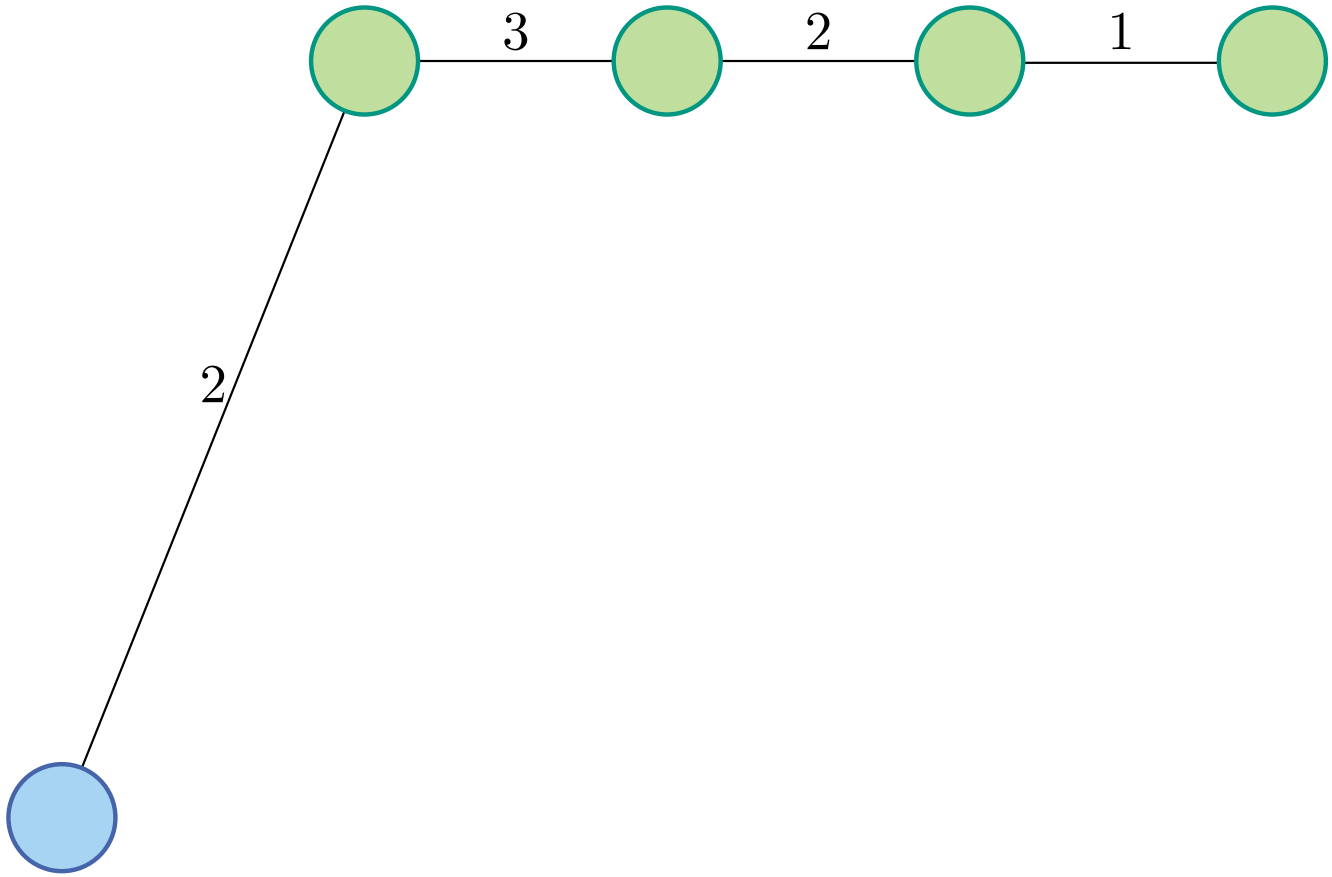
  $\rightarrow$ “Wichtigkeit” der Nachbarn neu berechnen

Problemfall: Pfad



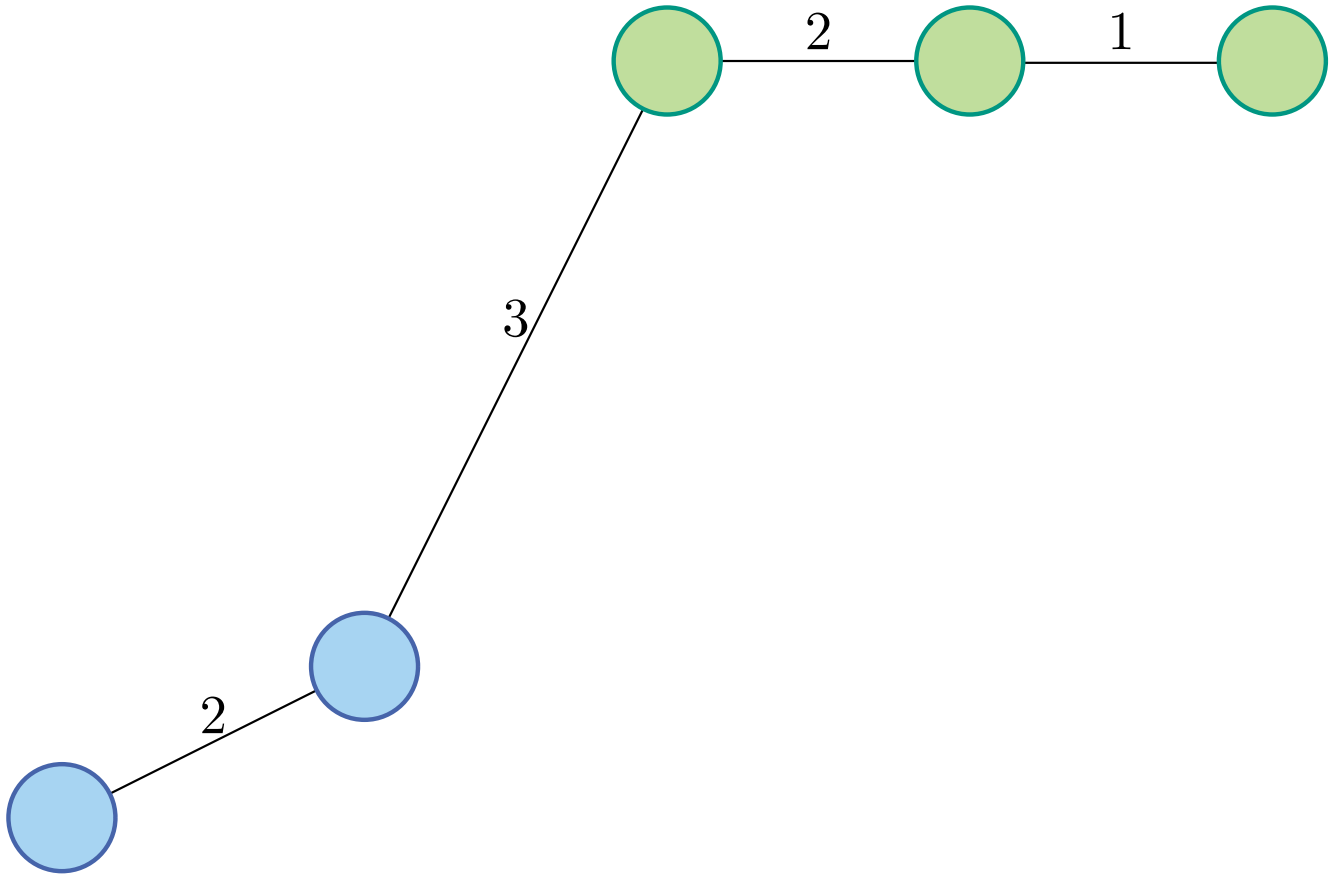
Linken Knoten kontrahieren erzeugt keine Shortcuts

Problemfall: Pfad



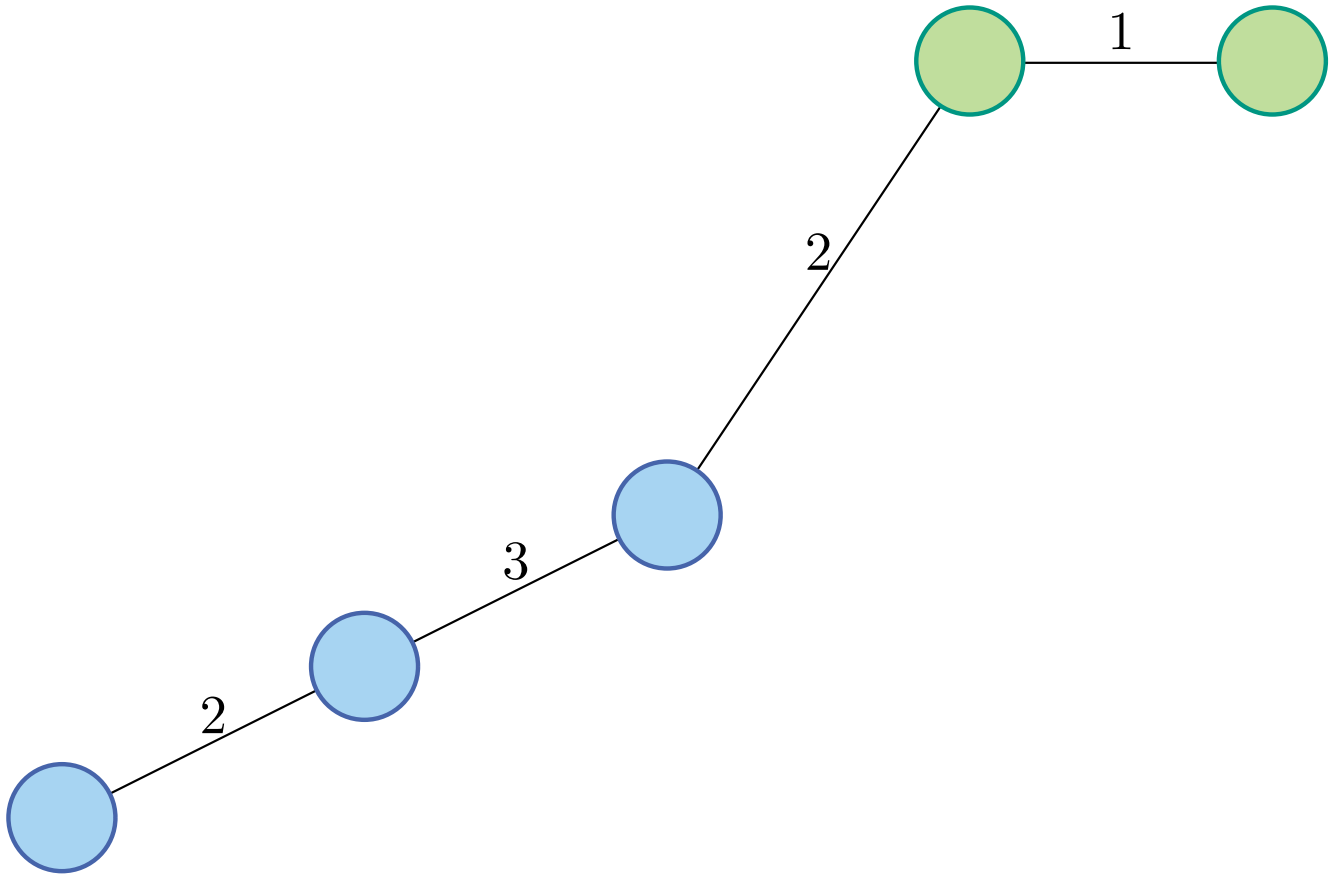
Linken Knoten kontrahieren erzeugt keine Shortcuts

Problemfall: Pfad



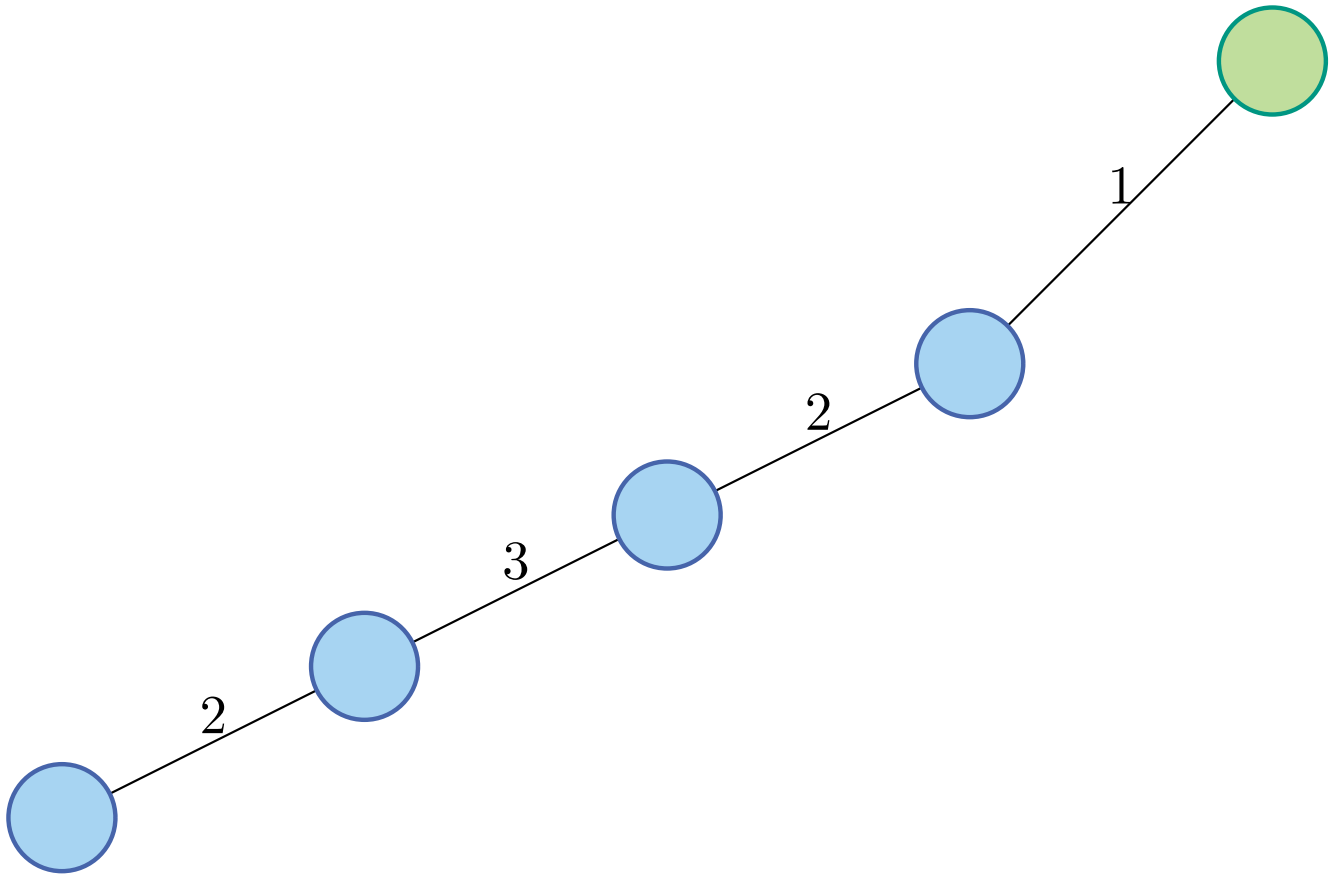
Linken Knoten kontrahieren erzeugt keine Shortcuts

Problemfall: Pfad



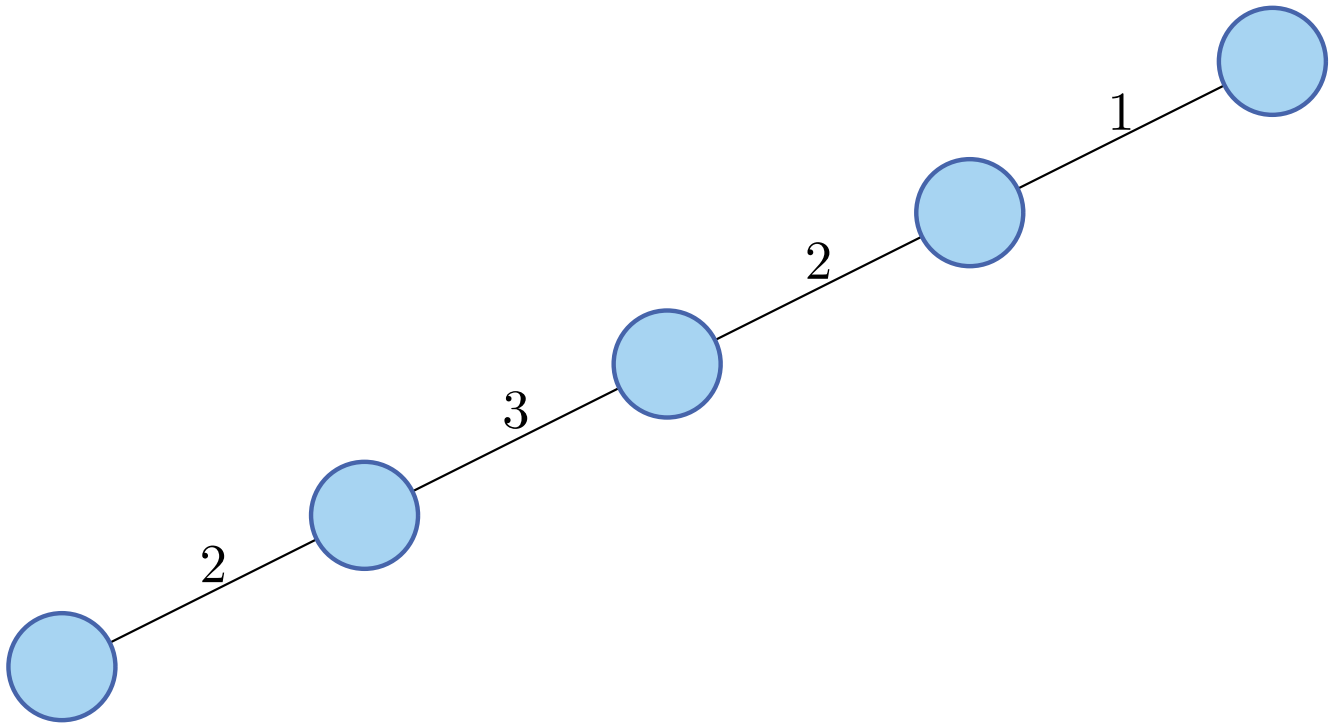
Linken Knoten kontrahieren erzeugt keine Shortcuts

Problemfall: Pfad



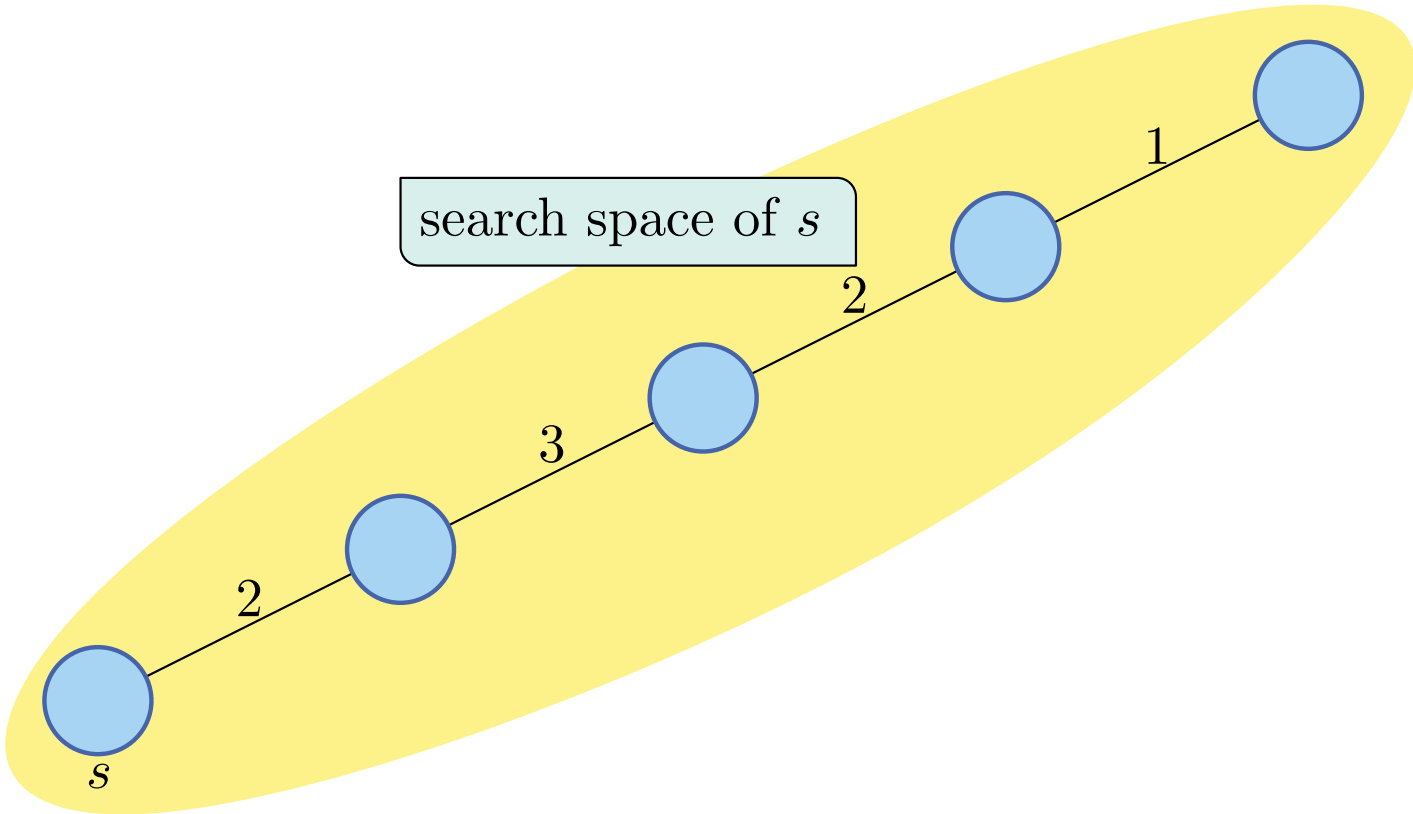
Linken Knoten kontrahieren erzeugt keine Shortcuts

Problemfall: Pfad



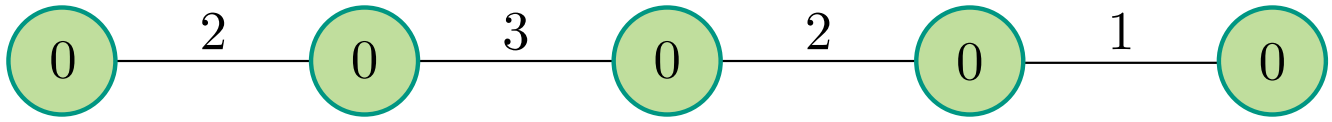
Linken Knoten kontrahieren erzeugt keine Shortcuts

Problemfall: Pfad



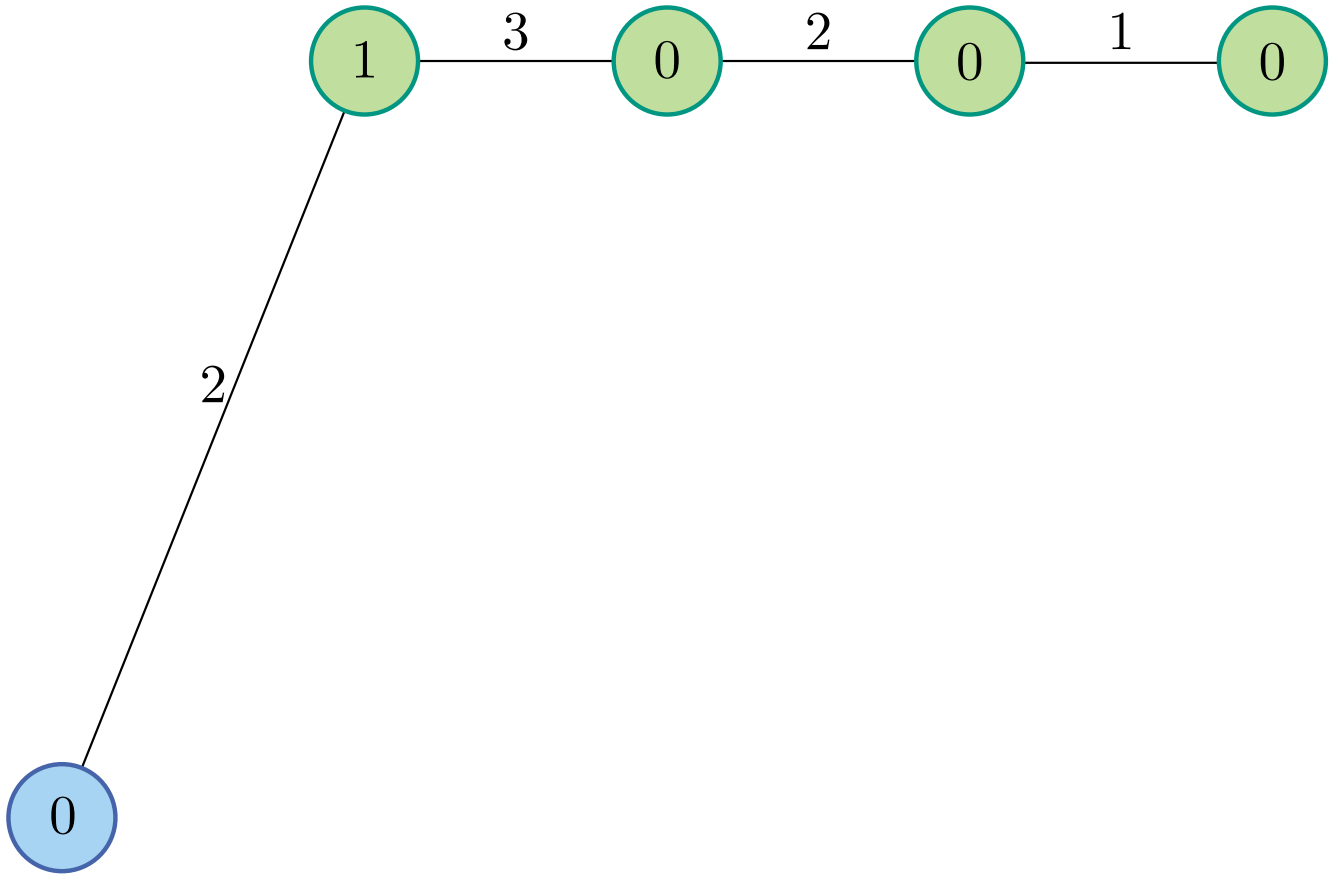
Suchraum von s ist der ganze Graph $\rightarrow$ keine Beschleunigung

Problemfall: Pfad



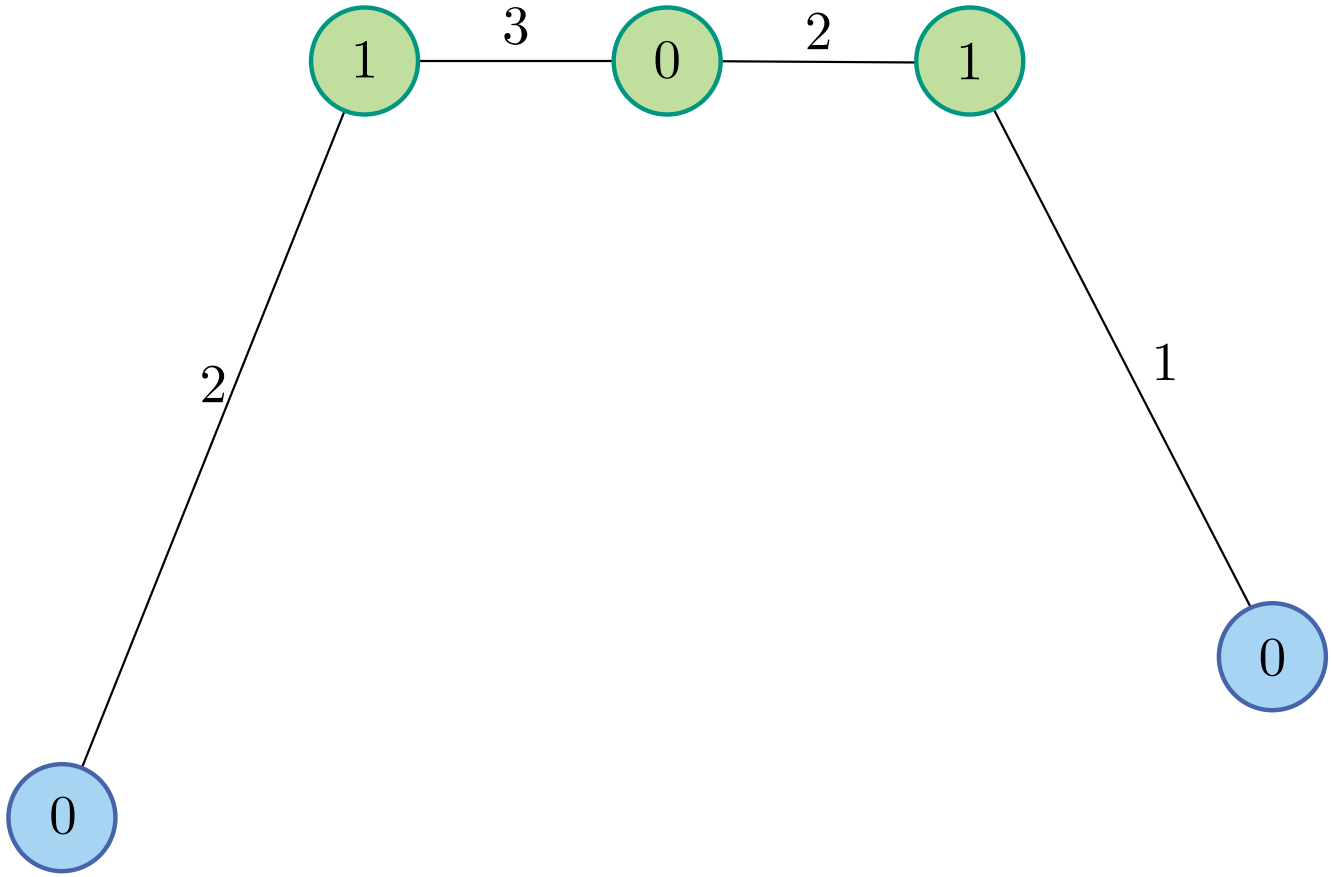
2-tes Kriterium: Das geschätzte Level $\ell(x)$ eines Knotens x kontrahiert $\rightarrow$ für alle Nachbarn y: $\ell(y) \leftarrow \max\{\ell(y), \ell(x) + 1\}$

Problemfall: Pfad



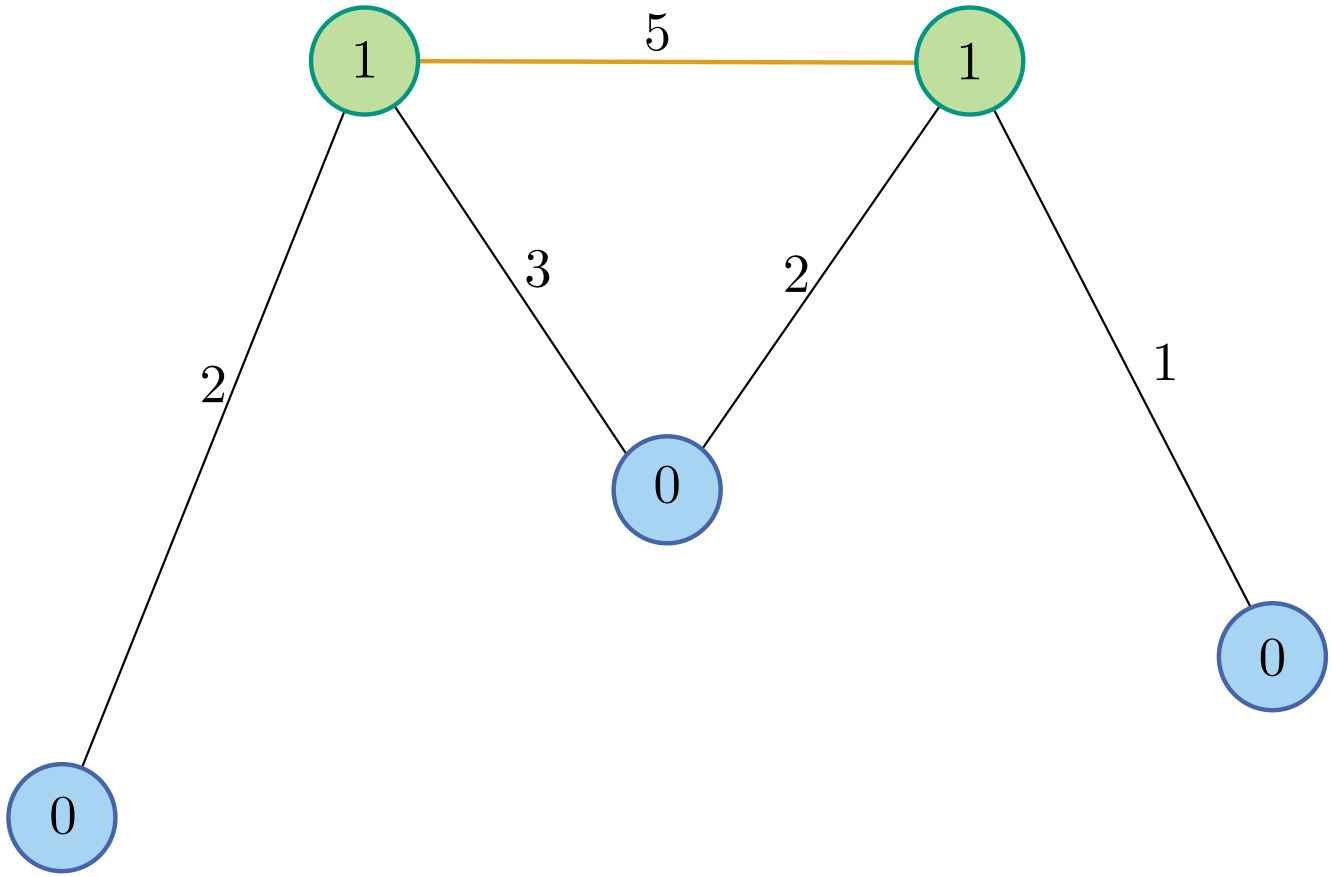
2-tes Kriterium: Das geschätzte Level $\ell(x)$ eines Knotens x kontrahiert $\rightarrow$ für alle Nachbarn y: $\ell(y) \leftarrow \max\{\ell(y), \ell(x) + 1\}$

Problemfall: Pfad



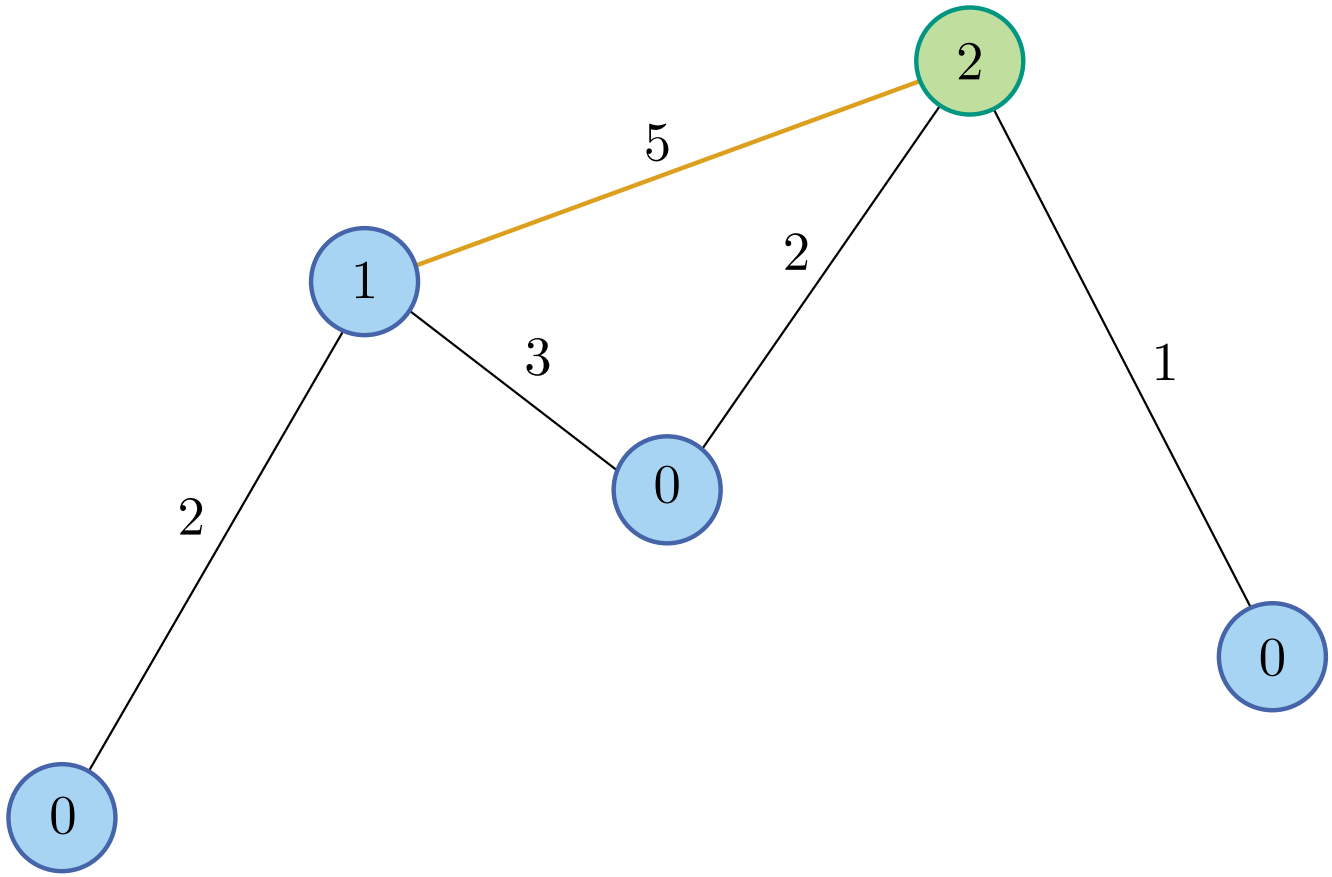
2-tes Kriterium: Das geschätzte Level $\ell(x)$ eines Knotens x kontrahiert $\rightarrow$ für alle Nachbarn y: $\ell(y) \leftarrow \max\{\ell(y), \ell(x)+1\}$

Problemfall: Pfad



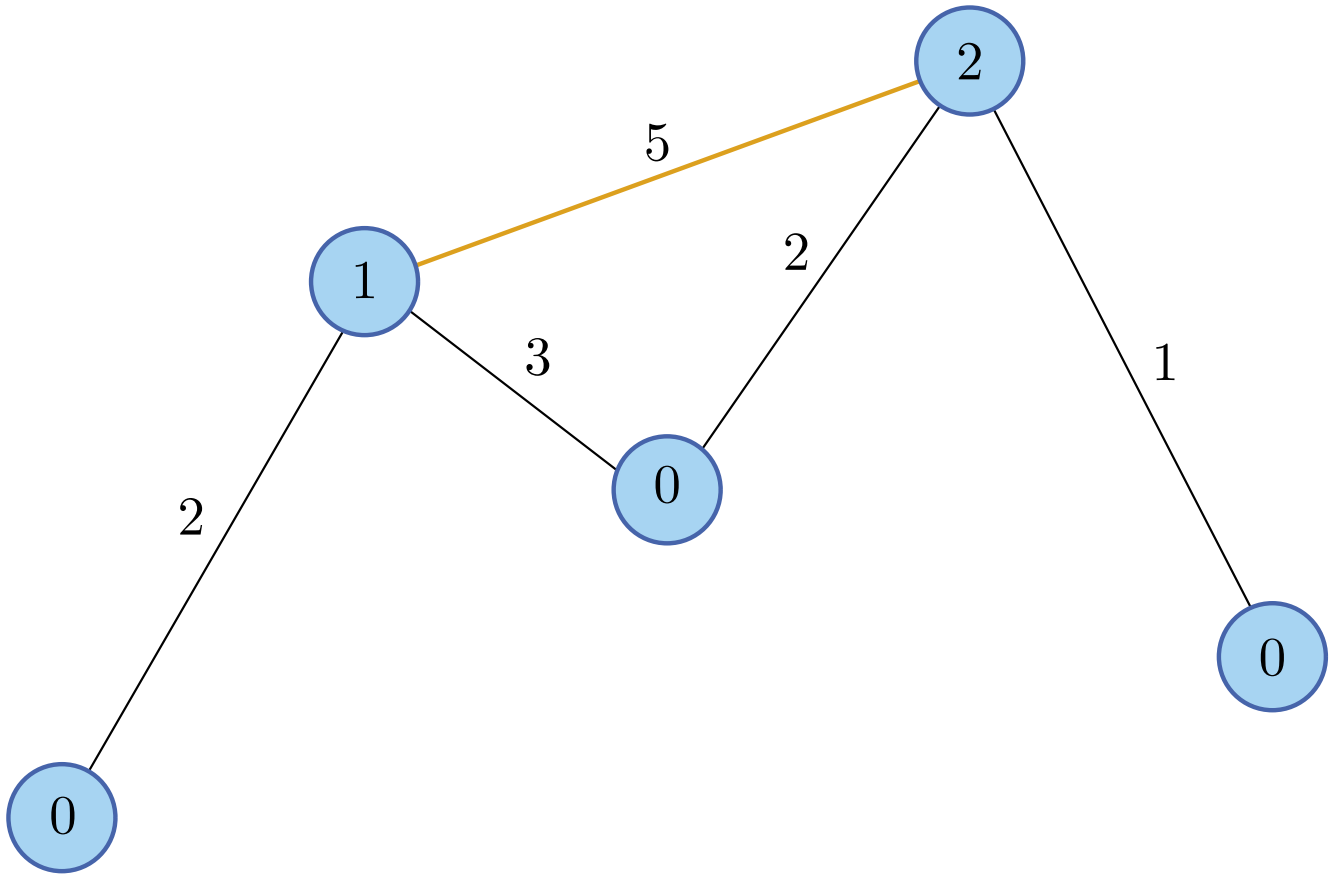
2-tes Kriterium: Das geschätzte Level $\ell(x)$ eines Knotens x kontrahiert $\rightarrow$ für alle Nachbarn y: $\ell(y) \leftarrow \max\{\ell(y), \ell(x)+1\}$

Problemfall: Pfad



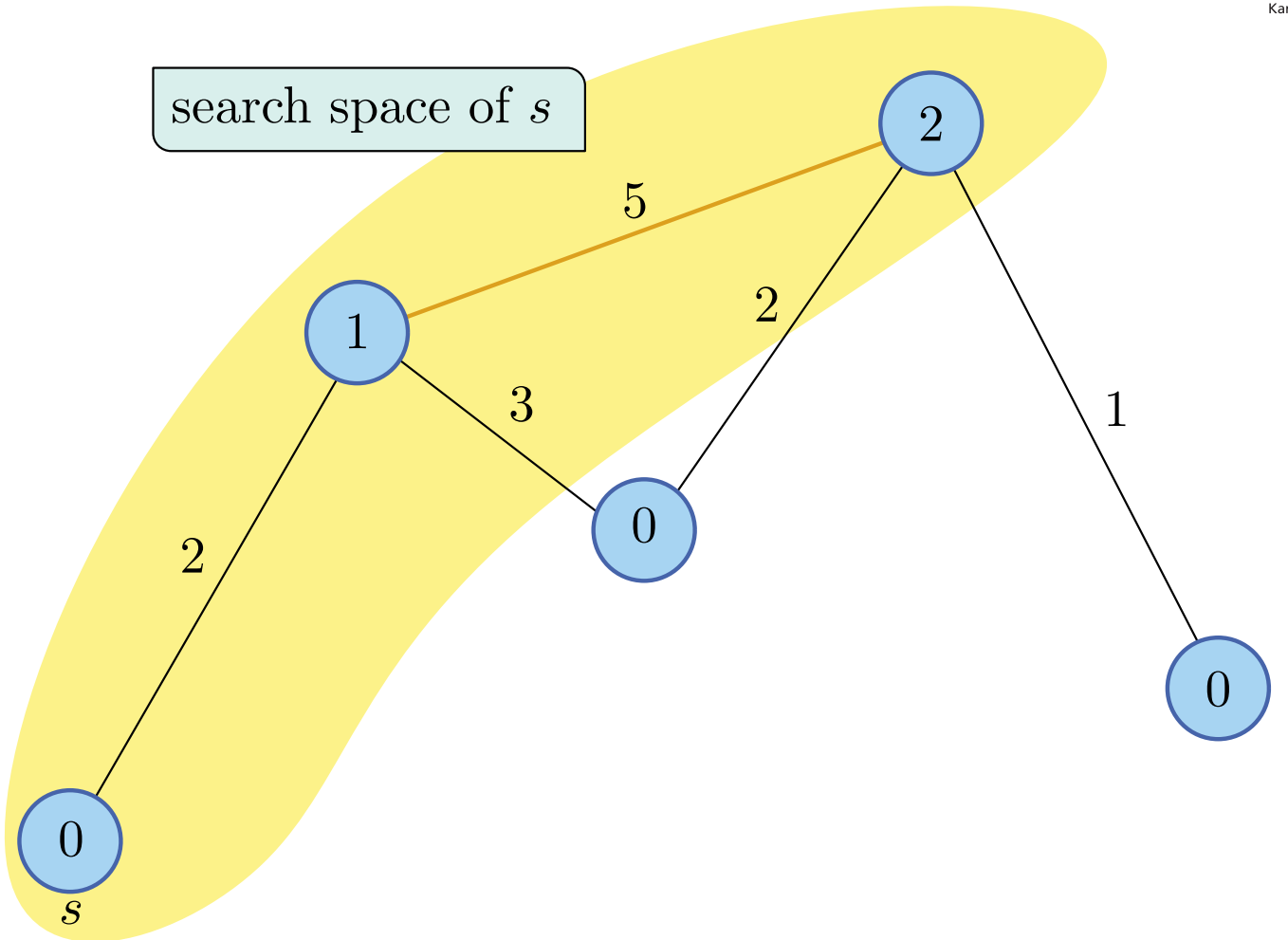
2-tes Kriterium: Das geschätzte Level $\ell(x)$ eines Knotens x kontrahiert $\rightarrow$ für alle Nachbarn y: $\ell(y) \leftarrow \max\{\ell(y), \ell(x) + 1\}$

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: $\ell(y) \leftarrow \max\{\ell(y), \ell(x)+1\}$

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

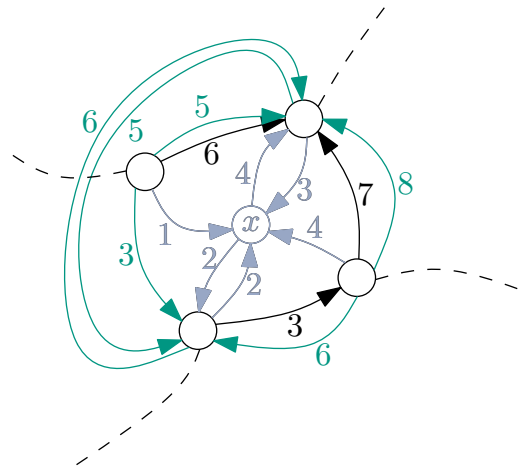
$$I(x) := \ell(x) + \frac{|A(x)|}{|D(x)|} + \frac{\sum_{e \in A(x)} h(e)}{\sum_{e \in D(x)} h(e)}$$

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

CH Grundstruktur Zusammenfassung

Phase 1 (Vorberechnung):

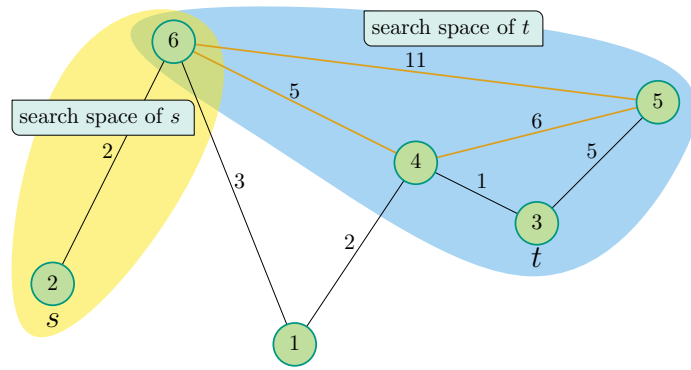
- Kontrahiere Knoten nach “Wichtigkeit”
- Zeugensuche reduziert eingefügte Shortcuts



Phase 2 (Query):

- Es gibt kürzesten Hoch-Runter-Pfad

$\rightarrow$ Biderektionale Suche nur zu wichtigeren Nachbarn

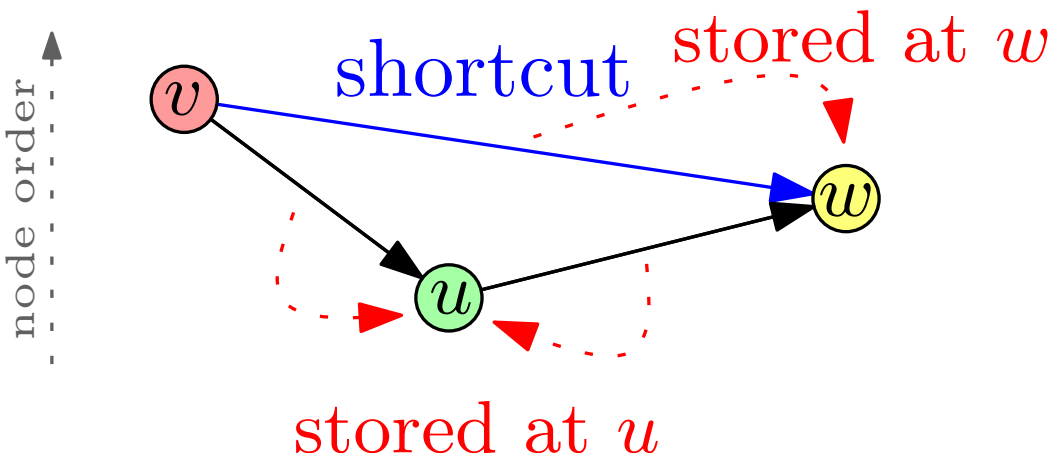


CH im Detail

Datenstruktur

Suchgraph:

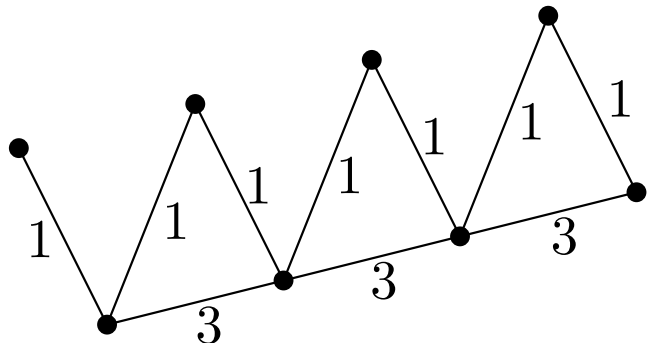
- normalerweise: speichere Kanten (v, w) in den Adjazenz-Arrays von v und w um bidirektionale Suche zu erlauben
- für die CH-Suche reicht es aus, die Kante nur an den Knoten mit geringerer "Wichtigkeit" (*rank* r), i.e. $\min\{r(v), r(w)\}$ zu speichern



Stall-On-Demand

Beobachtung:

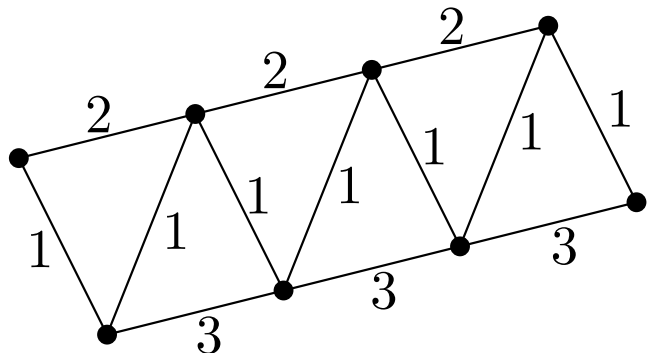
- Suchen können Knoten mit zu großer Distanz besuchen



Stall-On-Demand

Beobachtung:

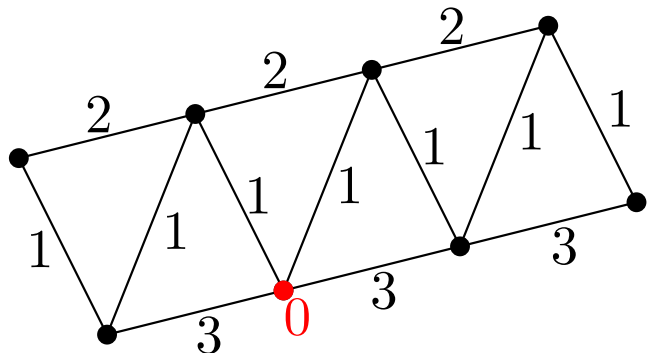
- Suchen können Knoten mit zu großer Distanz besuchen



Stall-On-Demand

Beobachtung:

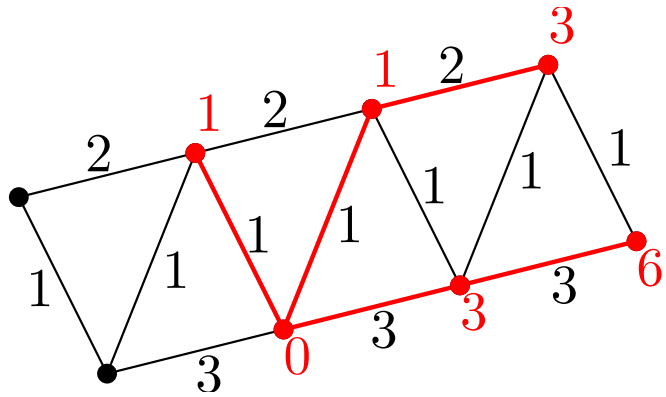
- Suchen können Knoten mit zu großer Distanz besuchen



Stall-On-Demand

Beobachtung:

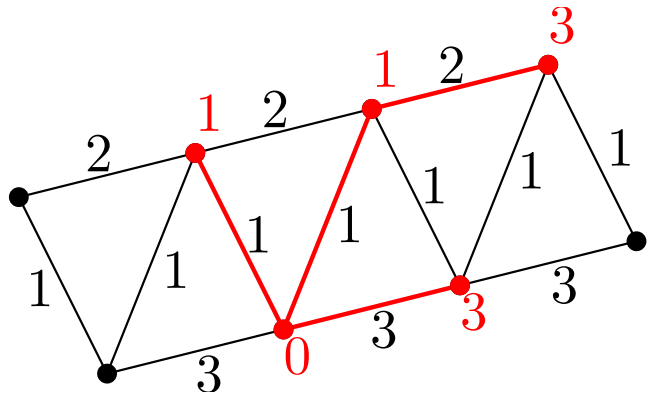
- Suchen können Knoten mit zu großer Distanz besuchen



Stall-On-Demand

Beobachtung:

- Suchen können Knoten mit zu großer Distanz besuchen

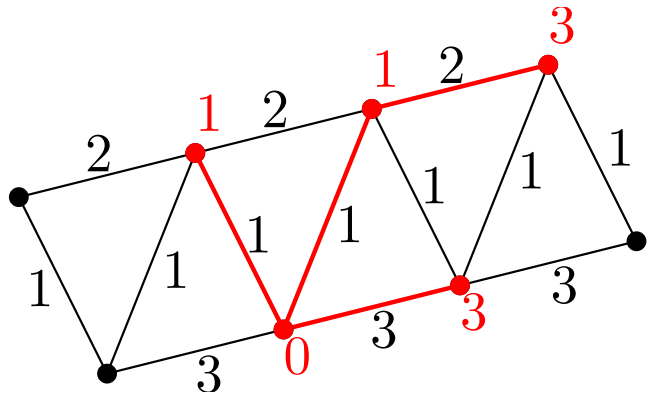


- Kann man zum prunen verwenden

Stall-On-Demand

Beobachtung:

- Suchen können Knoten mit zu großer Distanz besuchen

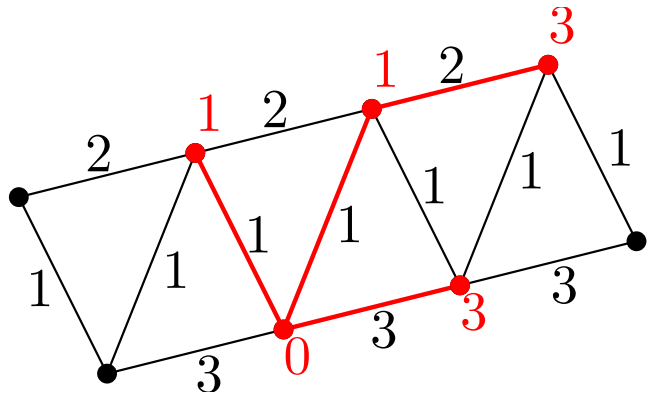


- Kann man zum prunen verwenden
- Jeder Teilpfad eines kürzesten hoch-runter-Pfads ist ein kürzester Pfad
- Bevor man einen Knoten settelt, sucht man nach kürzeren hoch-runter-Pfaden, gibt es diese kann man prunen

Stall-On-Demand

Beobachtung:

- Suchen können Knoten mit zu großer Distanz besuchen

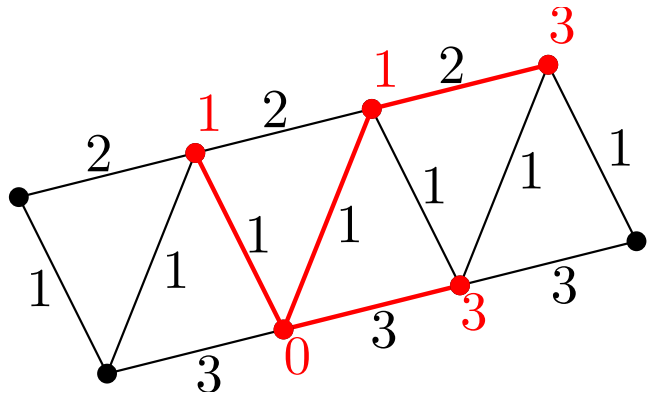


- Kann man zum prunen verwenden
- Jeder Teilpfad eines kürzesten hoch-runter-Pfads ist ein kürzester Pfad
- Bevor man einen Knoten settelt, sucht man nach kürzeren hoch-runter-Pfaden, gibt es diese kann man prunen
- Knoten v wird geprunt, wenn es Aufwärtsnachbar u von v gibt, so dass $d(u) + w(u, v) < d(v)$

Stall-On-Demand

Beobachtung:

- Suchen können Knoten mit zu großer Distanz besuchen



- Kann man zum prunen verwenden
- Jeder Teilpfad eines kürzesten hoch-runter-Pfads ist ein kürzester Pfad
- Bevor man einen Knoten settelt, sucht man nach kürzeren hoch-runter-Pfaden, gibt es diese kann man prunen
- Knoten v wird geprunt, wenn es Aufwärtsnachbar u von v gibt, so dass $d(u) + w(u, v) < d(v)$
- (Dies ist eine vereinfachte Version des ursprünglichen "Stall-On-Demand".)

Knotenordnung

Wie Knoten ordnen?

Knotenordnung

Wie Knoten ordnen?

- Bottom-Up, suche nach unwichtigen Knoten [GSSV12], haben wir schon gemacht
- Top-Down, suche nach wichtigen Knoten [ADGW12]
- Sampling Path-Greedy, Variante von Top-Down [DGPW14]

Top-Down

Path-Greedy

- Idee: Baue Ordnung von wichtig nach unwichtig
- Knoten wichtig, wenn er auf vielen kürzesten Wegen liegt

Top-Down

Path-Greedy

- Idee: Baue Ordnung von wichtig nach unwichtig
- Knoten wichtig, wenn er auf vielen kürzesten Wegen liegt

- initial ist die Ordnung leer
- Pfad ist überdeckt, wenn er einen Knoten in der Ordnung enthält
- sei U die Menge der nicht überdeckten kürzesten Pfade
- initial ist U die Menge aller kürzesten Pfade

Top-Down

Path-Greedy

- Idee: Baue Ordnung von wichtig nach unwichtig
- Knoten wichtig, wenn er auf vielen kürzesten Wegen liegt

- initial ist die Ordnung leer
- Pfad ist überdeckt, wenn er einen Knoten in der Ordnung enthält
- sei U die Menge der nicht überdeckten kürzesten Pfade
- initial ist U die Menge aller kürzesten Pfade

- Algo:
  - Solange die Ordnung nicht voll:
  - Packe Knoten v oben in die Ordnung, so dass v möglichst viele Pfade aus U überdeckt
  - Entferne diese Pfade aus U

Top-Down

Path-Greedy: Diskussion

- n mal all-pairs shortest paths
  $\rightarrow n^3 \log n$ mit Dijkstra auf dünnen Graphen
- linear Speicherverbrauch
- Kann auf $n^2 \log n$ gedrückt werden mit $O(n^2)$ Speicher
  $O(n^2)$ kommt daher, dass n Kürzeste-Wege-Bäume verwaltet werden
- gute Qualität der Ordnung, aber langsam

Path-Greedy: Datenstruktur

Ziel: $\mathcal{O}(n^2 \log n)$ Laufzeit für dünne Graphen
Allgemein:

- Deterministisches Tie-Breaking: eindeutige kürzeste Wege
- Speichere kürzeste-Wege-Bäume T_w mit Wurzel w
- Speichere nicht überdeckten kürzesten Pfade über Knoten w als $c(w)$

Path-Greedy: Datenstruktur

Ziel: $\mathcal{O}(n^2 \log n)$ Laufzeit für dünne Graphen
Allgemein:

- Deterministisches Tie-Breaking: eindeutige kürzeste Wege
- Speichere kürzeste-Wege-Bäume T_w mit Wurzel w
- Speichere nicht überdeckten kürzesten Pfade über Knoten w als $c(w)$

Berechne (bei Löschen von v):

- Amortisiert: jeder Knoten in jedem Baum T_u wird nur einmal gelöscht
  $\rightarrow$ Updates dort sind OK, zähle zusätzlich gelöschte Subbaumgröße δ_u

Path-Greedy: Datenstruktur

Ziel: $\mathcal{O}(n^2 \log n)$ Laufzeit für dünne Graphen
Allgemein:

- Deterministisches Tie-Breaking: eindeutige kürzeste Wege
- Speichere kürzeste-Wege-Bäume T_w mit Wurzel w
- Speichere nicht überdeckten kürzesten Pfade über Knoten w als $c(w)$

Berechne (bei Löschen von v):

- Amortisiert: jeder Knoten in jedem Baum T_u wird nur einmal gelöscht
  $\rightarrow$ Updates dort sind OK, zähle zusätzlich gelöschte Subbaumgröße δ_u
- Vorgängerknoten w von v in T_u muss auch aktualisiert werden
  $\rightarrow$ Aggregiere Updates über alle Bäume (in $\mathcal{O}(n \log n)$)

Path-Greedy: Datenstruktur

Bekannt:

- Eindeutige kürzeste Wege von allen Knoten w in T_w
- Nicht überdeckten kürzesten Pfade pro Knoten w als $c(w)$
- Gelöschte Subbaumgröße δ_w in T_w

Aggregiere Vorgänger Updates über alle Bäume (in $\mathcal{O}(n \log n)$):

Path-Greedy: Datenstruktur

Bekannt:

- Eindeutige kürzeste Wege von allen Knoten w in T_w
- Nicht überdeckten kürzesten Pfade pro Knoten w als $c(w)$
- Gelöschte Subbaumgröße δ_w in T_w

Aggregiere Vorgänger Updates über alle Bäume (in $\mathcal{O}(n \log n)$):

- Vorgänger Knoten liegen auf (eindeutigen) kürzesten Pfaden nach v
- Betrachte kürzeste-Wege-Baum I_v nach v

Path-Greedy: Datenstruktur

Bekannt:

- Eindeutige kürzeste Wege von allen Knoten w in T_w
- Nicht überdeckten kürzesten Pfade pro Knoten w als $c(w)$
- Gelöschte Subbaumgröße δ_w in T_w

Aggregiere Vorgänger Updates über alle Bäume (in $\mathcal{O}(n \log n)$):

- Vorgänger Knoten liegen auf (eindeutigen) kürzesten Pfaden nach v
- Betrachte kürzeste-Wege-Baum I_v nach v

  $\rightarrow$ Kürzeste Pfade (über v) eines Kindes u beinhalten den Vater w
- Verschiedene T_x beinhalten verschiedene Pfade (T_x kürzeste Pfade von x)

Path-Greedy: Datenstruktur

Bekannt:

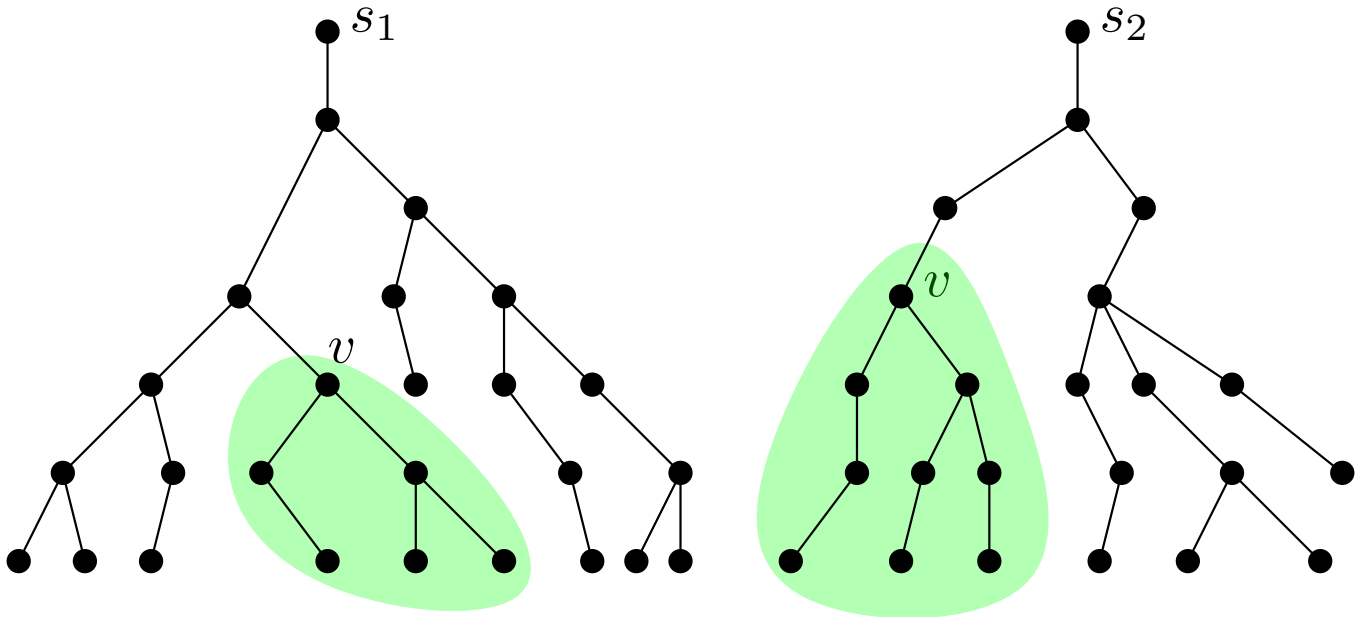
- Eindeutige kürzeste Wege von allen Knoten w in T_w
- Nicht überdeckten kürzesten Pfade pro Knoten w als $c(w)$
- Gelöschte Subbaumgröße δ_w in T_w

Aggregiere Vorgänger Updates über alle Bäume (in $\mathcal{O}(n \log n)$):

- Vorgänger Knoten liegen auf (eindeutigen) kürzesten Pfaden nach v
- Betrachte kürzeste-Wege-Baum I_v nach v

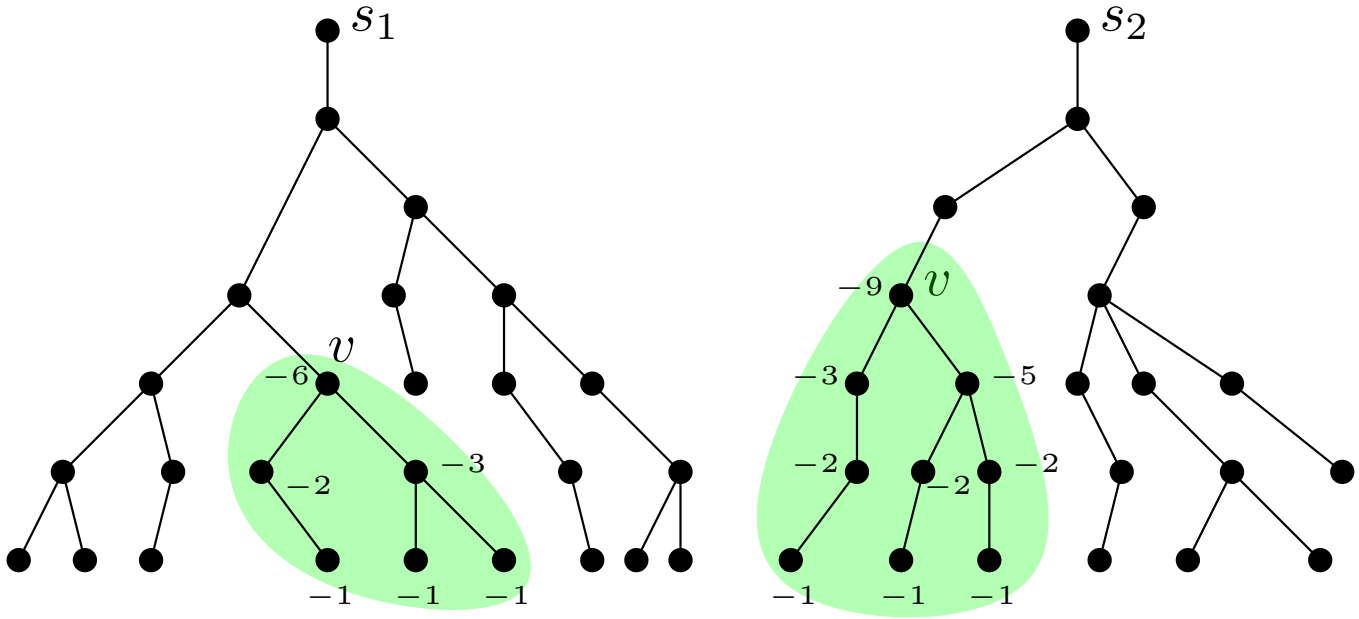
  $\rightarrow$ Kürzeste Pfade (über v) eines Kindes u beinhalten den Vater w
- Verschiedene T_x beinhalten verschiedene Pfade (T_x kürzeste Pfade von x)
- Reduziere $c(w)$ (Bottom-Up) um $r_w := \delta_w + \sum_{u \in \text{Children}(w)} r_u$

Path-Greedy: Beispiel



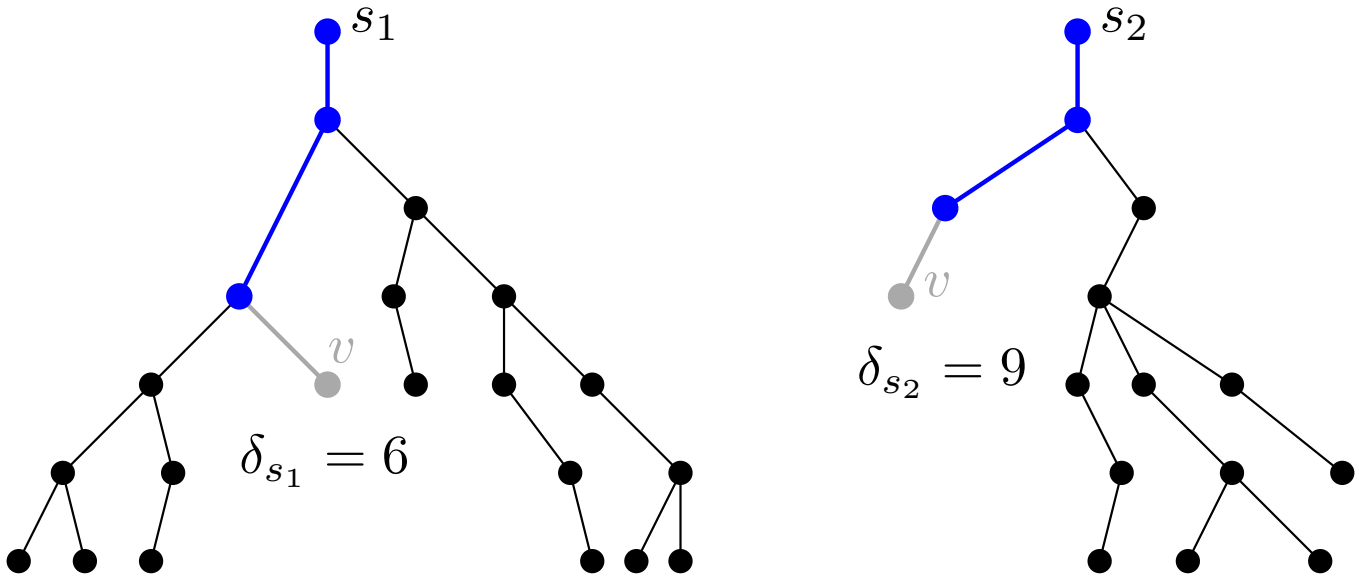
Finde Subbäume unter v

Path-Greedy: Beispiel



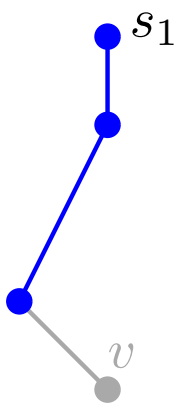
Aktualisiere $c(x)$ für alle Nachfahren von v (Bottom-Up)

Path-Greedy: Beispiel

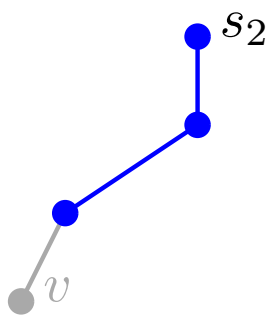


Lösche Subbäume unter v

Path-Greedy: Beispiel



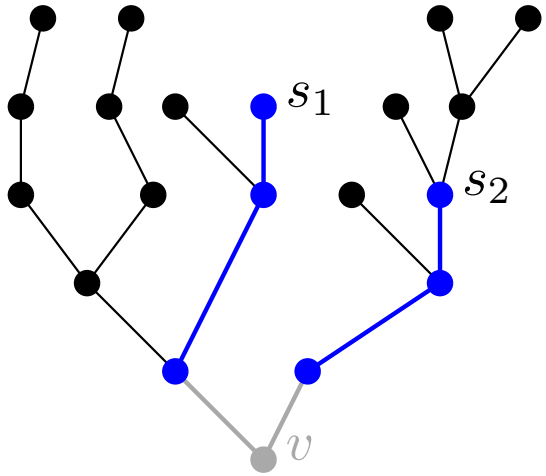
$\delta_{s_1} = 6$



$\delta_{s_2} = 9$

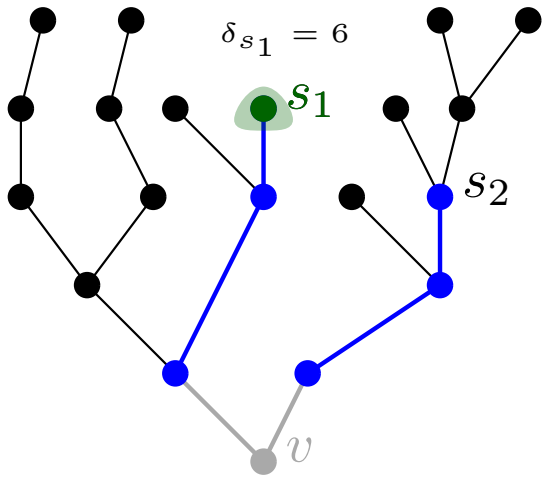
Vorfahren von v müssen noch $c(w)$ aktualisieren
Das sind Knoten auf s_i-v Pfaden

Path-Greedy: Beispiel



Betrachte I_v: alle zu aktualisierenden Pfade sind hier enthalten

Path-Greedy: Beispiel

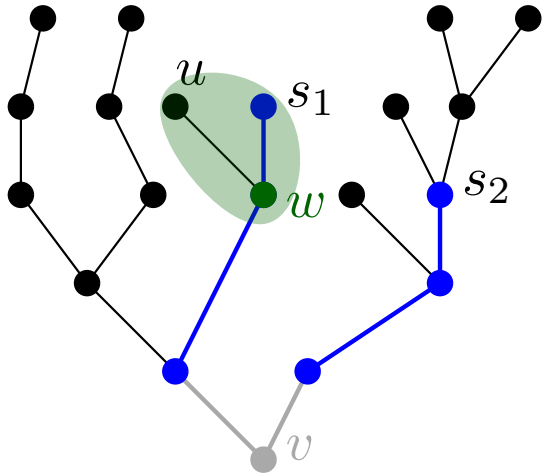


$$r_{s_1} = \delta_{s_1}$$

Reduziere $c(w)$ um $r_w := \delta_w + \sum_{u \in \text{Children}(w)} r_u \quad (= \delta_r + \sum_{u \in \text{Descendants}(w)} \delta_w)$

Aktualisiere Bottom-Up

Path-Greedy: Beispiel

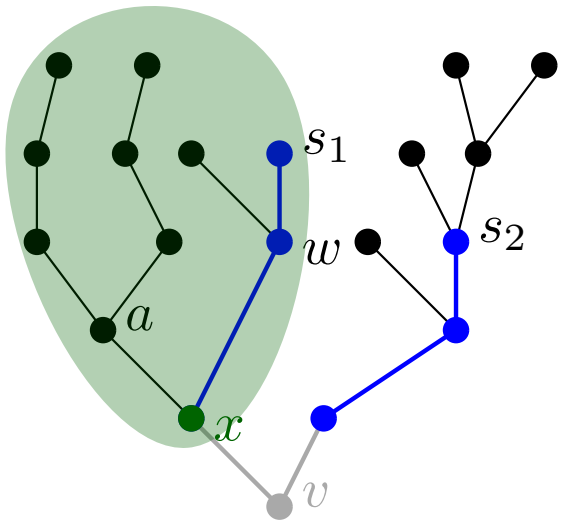


$$r_w = \delta_w + r_u + r_{s_1}$$

Reduziere $c(w)$ um $r_w := \delta_w + \sum_{u \in \text{Children}(w)} r_u \quad (= \delta_r + \sum_{u \in \text{Descendants}(w)} \delta_w)$

Aktualisiere Bottom-Up

Path-Greedy: Beispiel

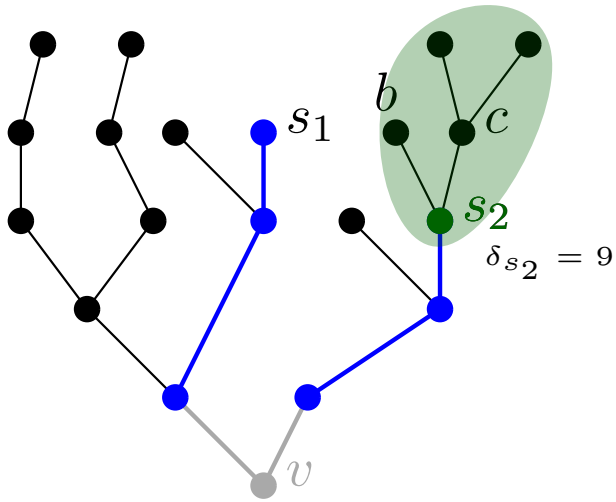


$$r_x = \delta_x + r_a + r_w$$

Reduziere $c(w)$ um $r_w := \delta_w + \sum_{u \in \text{Children}(w)} r_u \quad (= \delta_r + \sum_{u \in \text{Descendants}(w)} \delta_w)$

Aktualisiere Bottom-Up

Path-Greedy: Beispiel

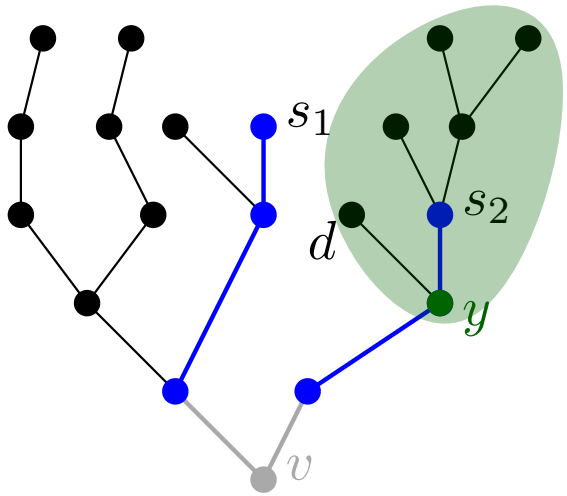


$$r_{s_2} = \delta_{s_2} + r_b + r_c$$

Reduziere $c(w)$ um $r_w := \delta_w + \sum_{u \in \text{Children}(w)} r_u \quad (= \delta_r + \sum_{u \in \text{Descendants}(w)} \delta_w)$

Aktualisiere Bottom-Up

Path-Greedy: Beispiel

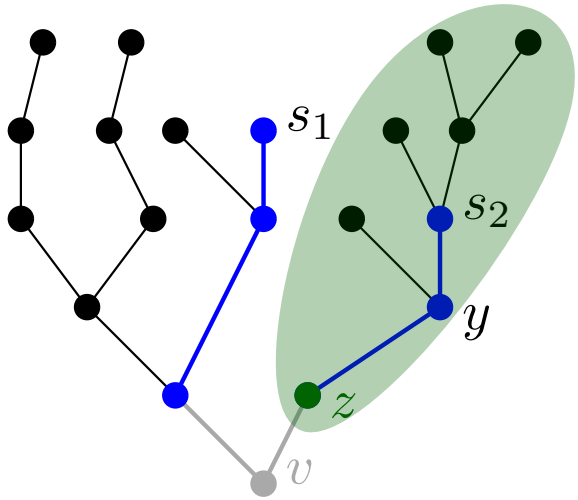


$$r_y = \delta_y + r_d + r_{s_2}$$

Reduziere $c(w)$ um $r_w := \delta_w + \sum_{u \in \text{Children}(w)} r_u \quad (= \delta_r + \sum_{u \in \text{Descendants}(w)} \delta_w)$

Aktualisiere Bottom-Up

Path-Greedy: Beispiel



$$r_z = \delta_z + r_y$$

Reduziere $c(w)$ um $r_w := \delta_w + \sum_{u \in \text{Children}(w)} r_u \quad (= \delta_r + \sum_{u \in \text{Descendants}(w)} \delta_w)$

Aktualisiere Bottom-Up

Sampling Path-Greedy [DGPW14]

- Problem von Top-Down: $O(n^2)$ Speicherverbrauch, da n Kürzeste-Wege-Bäume verwaltet werden
- Ansatz von Sampling Path-Greedy: speichere nur ein "paar" kürzeste Wege Bäume
- d.h., der Algorithmus sampelt Bäume

  Nicht gesampelte Knoten w tragen nicht zu $c(v)$ bei und haben $\delta_w = 0$

Sampling Path-Greedy [DGPW14]

Karlsruher Institut für Technologie

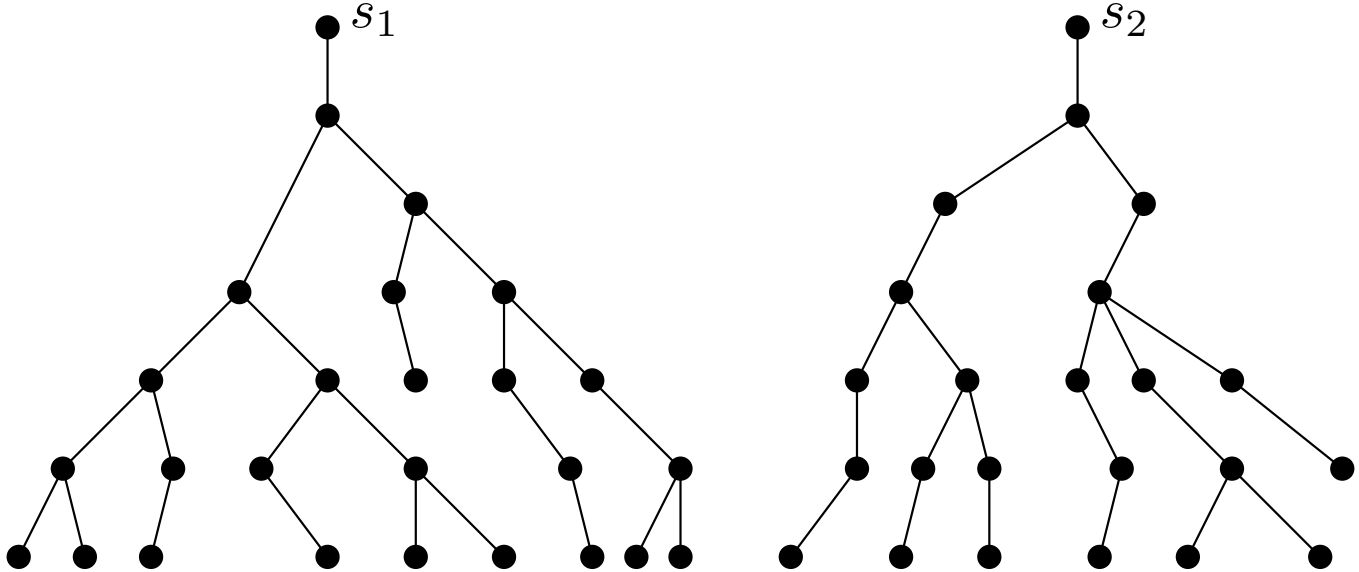
Idee

- Verwalte nur wenige zufällige Kürzeste-Wege-Bäume
  - Quellknoten: s_1, s_2 ...
- Wähle v, der auf den meisten Pfaden in diesen Bäumen liegt

Sampling Path-Greedy [DGPW14]

Idee

- Verwalte nur wenige zufällige Kürzeste-Wege-Bäume
  - Quellknoten: s_1, s_2 ...
- Wähle v, der auf den meisten Pfaden in diesen Bäumen liegt

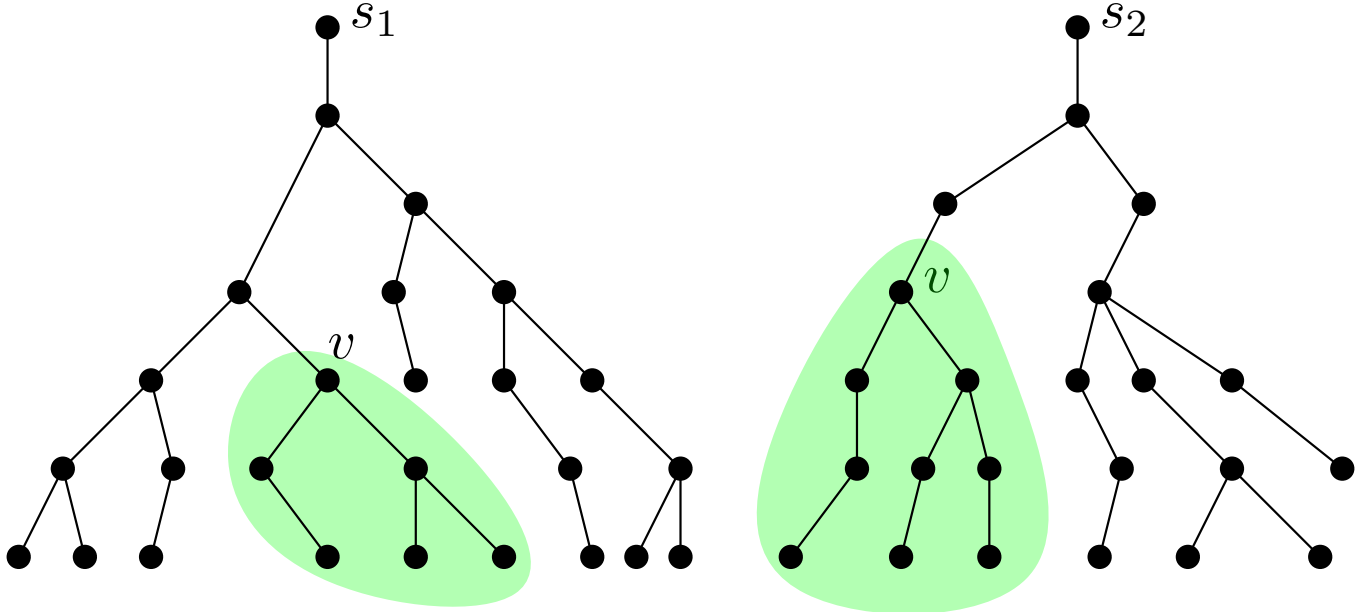


Die kürzeste Wege Bäume von s_1 und s_2

Sampling Path-Greedy [DGPW14]

Idee

- Verwalte nur wenige zufällige Kürzeste-Wege-Bäume
  - Quellknoten: s_1, s_2 ...
- Wähle v, der auf den meisten Pfaden in diesen Bäumen liegt



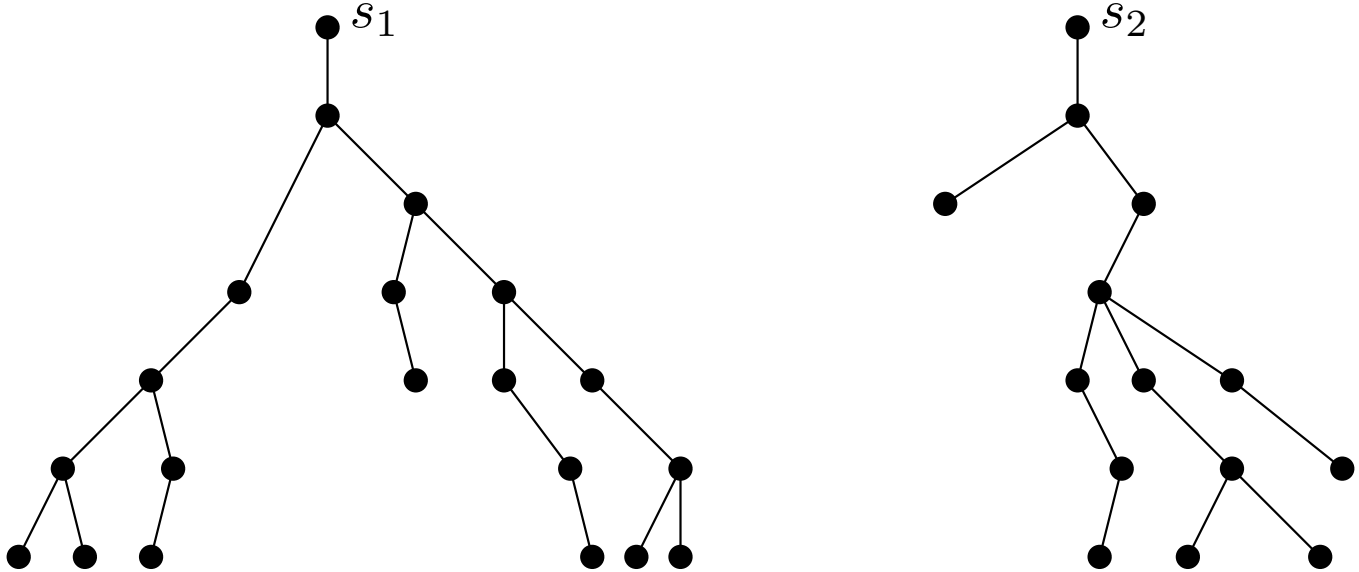
Die kürzeste Wege Bäume von s_1 und s_2

- v liegt auf so vielen Pfaden, die bei s_1 beginnen, wie sein s_1-Teilbaum groß ist (hier 6)

Sampling Path-Greedy [DGPW14]

Idee

- Verwalte nur wenige zufällige Kürzeste-Wege-Bäume
  - Quellknoten: s_1, s_2 ...
- Wähle v, der auf den meisten Pfaden in diesen Bäumen liegt



Die kürzeste Wege Bäume von s_1 und s_2

- v liegt auf so vielen Pfaden, die bei s_1 beginnen, wie sein s_1-Teilbaum groß ist (hier 6)
- Entferne Teilbäume in allen verwalteten Bäumen

Sampling Path-Greedy [DGPW14]

Experimente zeigen:

- Nur wenige Bäume notwendig um die wichtigsten Knoten zu finden
- Für das Mittelfeld und die unwichtigen Knoten werden mehr Bäume gebraucht

Beobachtung:

- Durch Löschen der Teilbäume wird Speicher frei

Idee:

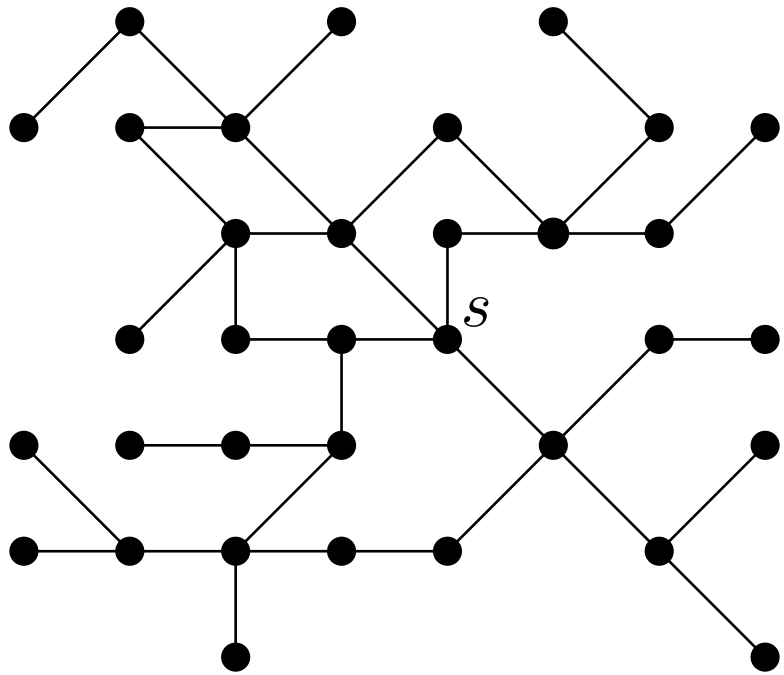
- Fülle frei gewordenen Speicher mit neuen Bäumen

Neue Bäume Aufbauen [DGPW14]

- Durch entfernen von Teilbäumen schrumpfen die Bäume
  $\rightarrow$ Baue neue Bäume auf
- Wir müssen uns nicht den ganzen Graph anschauen
  Gleiches Prunen funktioniert auch für l_v beim updaten von $c(w)$

Neue Bäume Aufbauen [DGPW14]

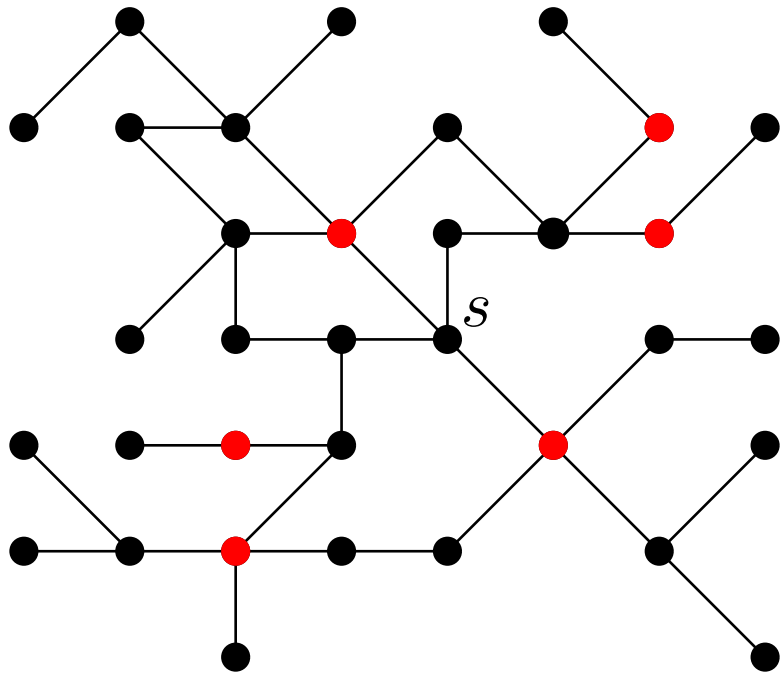
- Durch entfernen von Teilbäumen schrumpfen die Bäume
  $\rightarrow$ Baue neue Bäume auf
- Wir müssen uns nicht den ganzen Graph anschauen
  Gleiches Prunen funktioniert auch für l_v beim updaten von $c(w)$



s ist neue Baumwurzel

Neue Bäume Aufbauen [DGPW14]

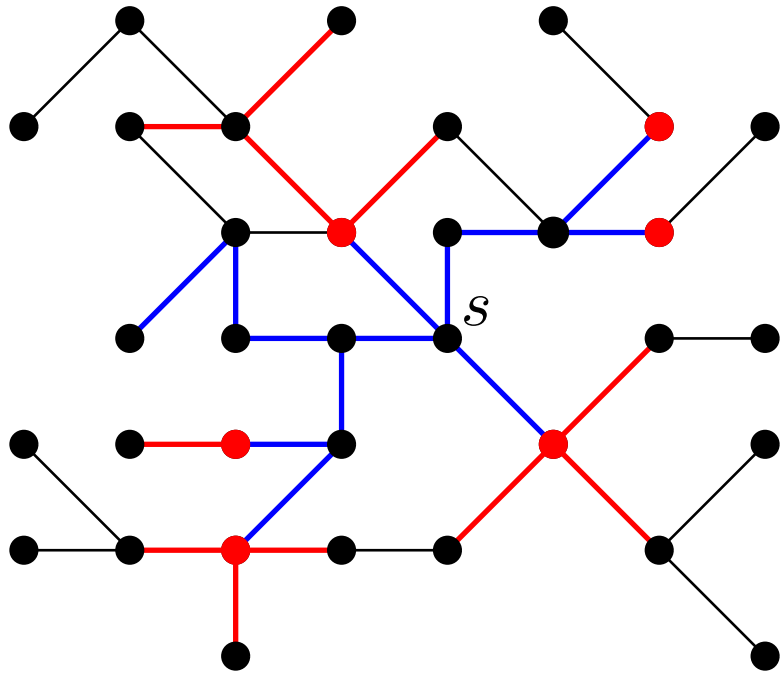
- Durch entfernen von Teilbäumen schrumpfen die Bäume
  $\rightarrow$ Baue neue Bäume auf
- Wir müssen uns nicht den ganzen Graph anschauen
  Gleiches Prunen funktioniert auch für l_v beim updaten von $c(w)$



rote Knoten sind bereits in der Ordnung

Neue Bäume Aufbauen [DGPW14]

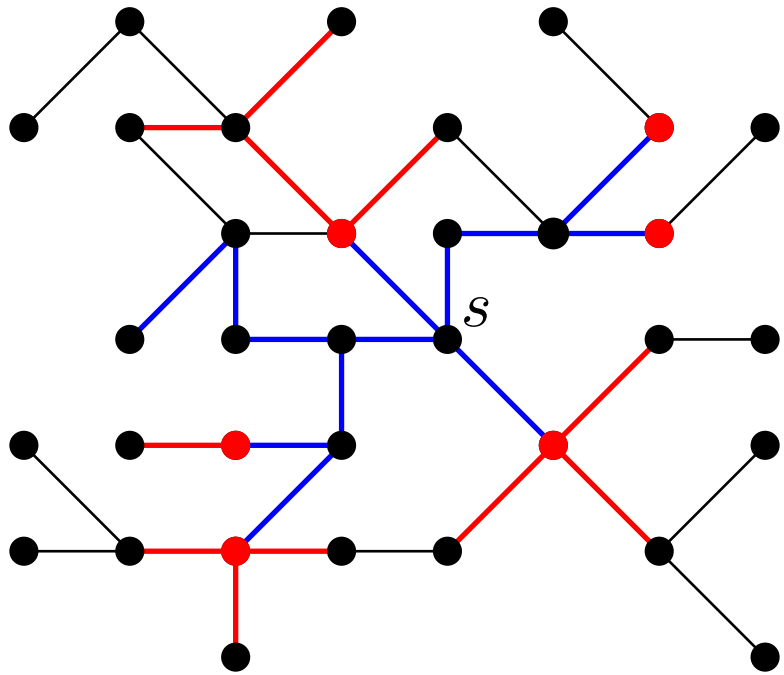
- Durch entfernen von Teilbäumen schrumpfen die Bäume
  $\rightarrow$ Baue neue Bäume auf
- Wir müssen uns nicht den ganzen Graph anschauen
  Gleiches Prunen funktioniert auch für l_v beim updaten von $c(w)$



lasse Dijkstra laufen

Neue Bäume Aufbauen [DGPW14]

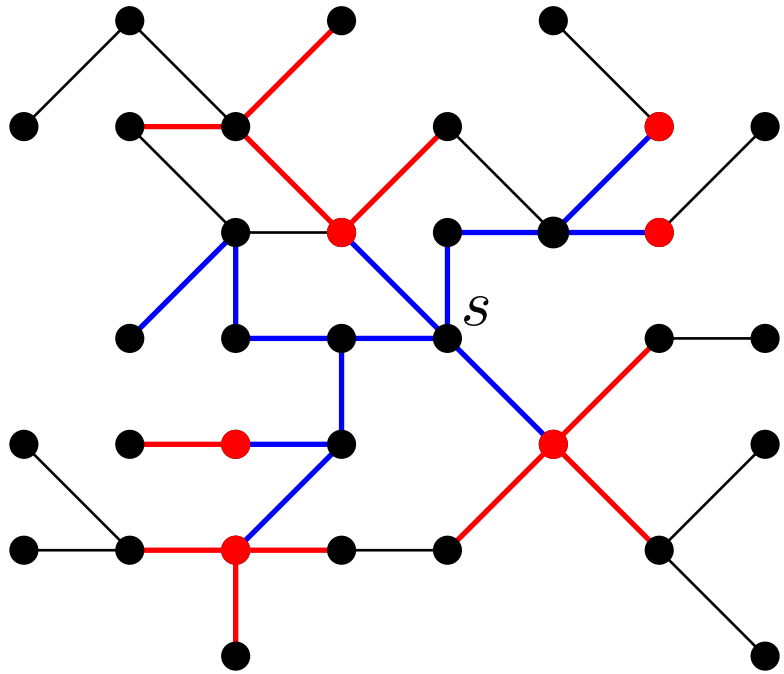
- Durch entfernen von Teilbäumen schrumpfen die Bäume
  $\rightarrow$ Baue neue Bäume auf
- Wir müssen uns nicht den ganzen Graph anschauen
  Gleiches Prunen funktioniert auch für l_v beim updaten von $c(w)$



speichere welche Knoten über rote Knoten erreicht wurden

Neue Bäume Aufbauen [DGPW14]

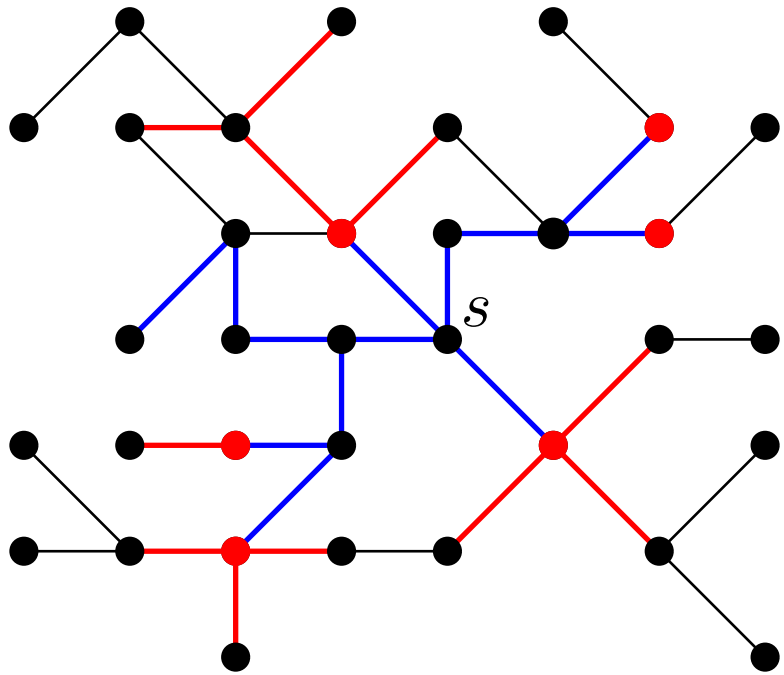
- Durch entfernen von Teilbäumen schrumpfen die Bäume
  $\rightarrow$ Baue neue Bäume auf
- Wir müssen uns nicht den ganzen Graph anschauen
  Gleiches Prunen funktioniert auch für l_v beim updaten von $c(w)$



breche ab, wenn die Queue nur noch Knoten enthält die über rote Knoten erreicht wurden

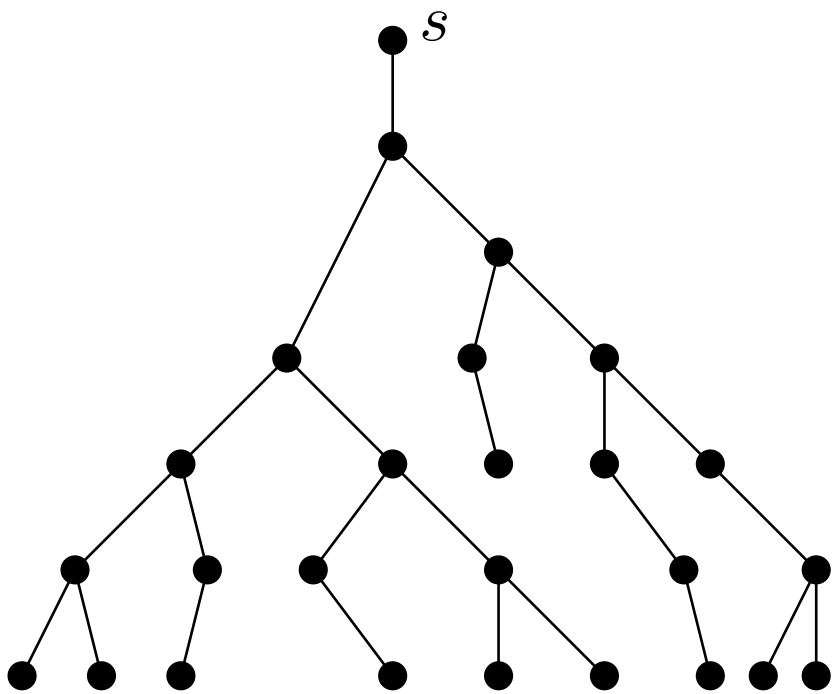
Neue Bäume Aufbauen [DGPW14]

- Durch entfernen von Teilbäumen schrumpfen die Bäume
  $\rightarrow$ Baue neue Bäume auf
- Wir müssen uns nicht den ganzen Graph anschauen
  Gleiches Prunen funktioniert auch für l_v beim updaten von $c(w)$



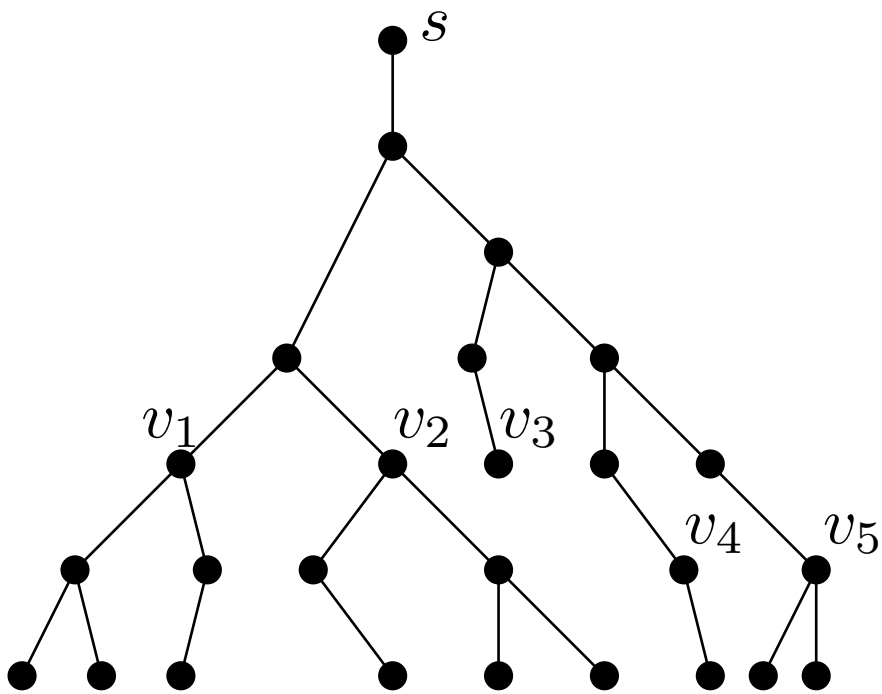
der blaue Teilgraph ist der neue Baum

Alternative Sichtweise [DGPW14]



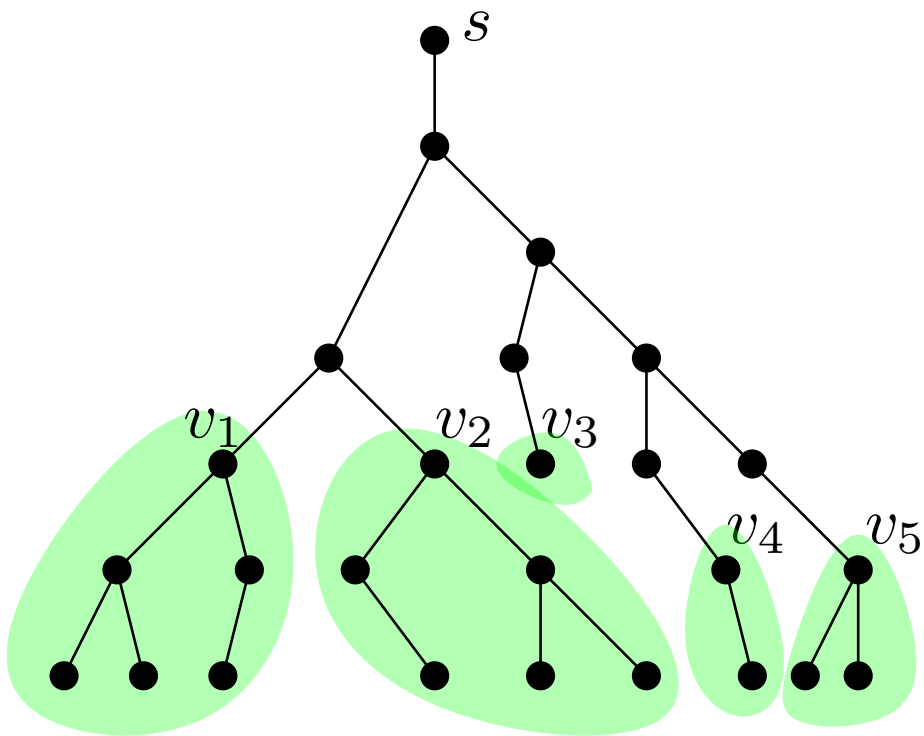
s ist neue Baumwurzel
Oben abgebildet ist der vollständige Kürzeste-Wege Baum

Alternative Sichtweise [DGPW14]



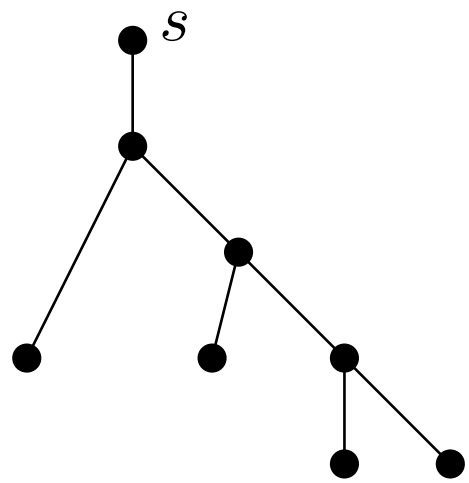
Die Knoten v_i wurden bereits ausgewählt

Alternative Sichtweise [DGPW14]



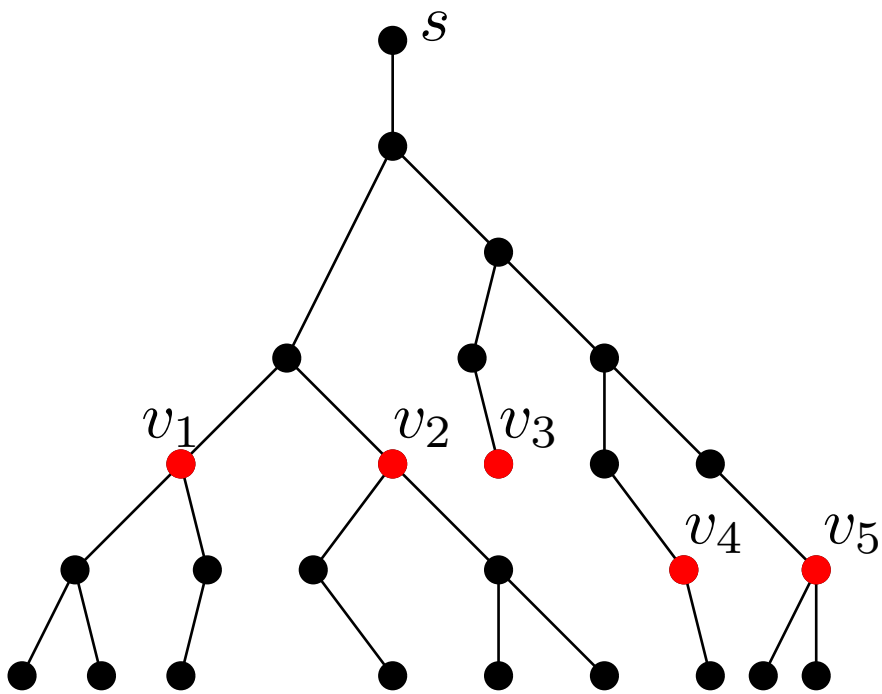
Ihre Teilbäume werden nicht gebraucht
$\rightarrow$ Wir wollen diese Teile gar nicht erst aufbauen

Alternative Sichtweise [DGPW14]



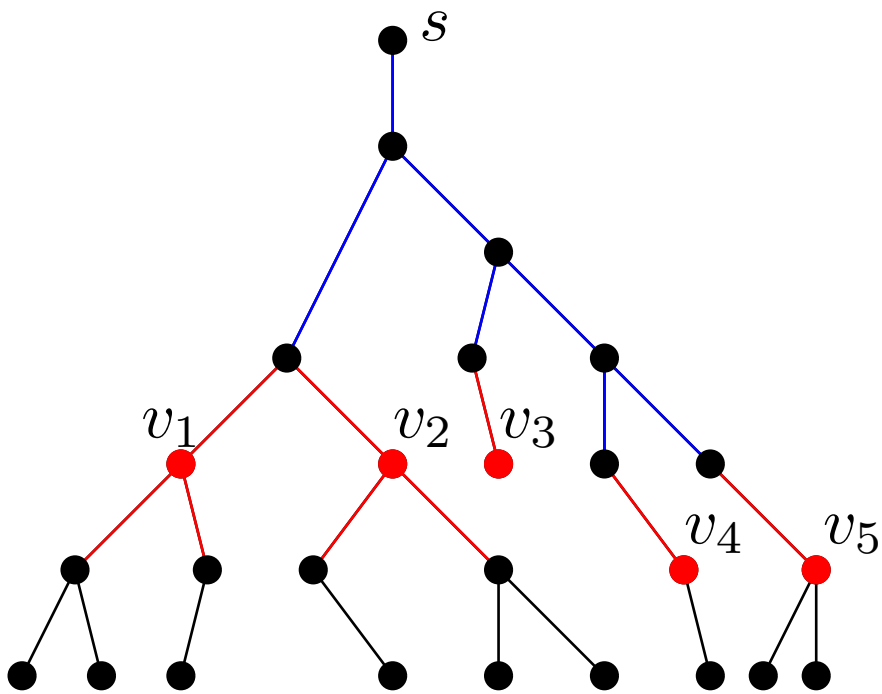
Dies ist der Baum den wir haben wollen

Alternative Sichtweise [DGPW14]



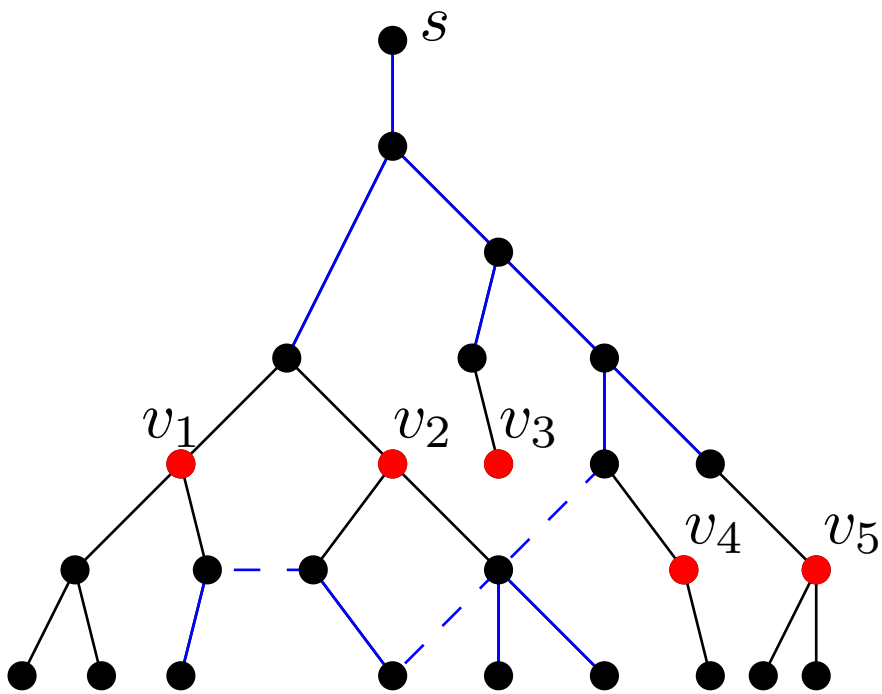
Die Knoten v_i sind bereits in der Ordnung und rot markiert

Alternative Sichtweise [DGPW14]



Um den Baum aufzubauen lassen wir Dijkstras Algorithmus von s aus laufen und brechen ab sobald alle Äste über rote Knoten gehen

Alternative Sichtweise [DGPW14]



Dies ist nicht das selbe wie an allen $v_1 \ldots$ zu prunen!
Wenn wir prunen findet die Suche neue Wege die gar nicht entlang des Kürzeste-Wege Baum von s entlang gehen.

Sampling Path-Greedy [DGPW14]

Algo:

1. Baue Kürzeste-Wege-Bäume auf mit zufälliger Wurzel, bis $k \cdot n$ Knoten in allen Bäumen sind
2. Wähle wichtigsten verbleibenden Knoten v aus
3. Lösche Teilbäume unter v
4. Gehe zu 1

- k ist ein Parameter der die Qualität steuert

Sampling Path-Greedy [DGPW14]

Vorteile von Sampling Path-Greedy:

- Funktioniert auf den meisten Graphen
- Gut auch bei hohen Knotengeraden, wo Bottom-Up sich schwer tut

Auf Straße

- Ordnungsqualität vergleichbar mit Bottom-Up
- Aber langsamer als Bottom-Up

$\rightarrow$ Nehmt Bottom-Up

Ergebnisse

	preprocessing		query	
method	time [h:m]	space [GB]	scans	time [μs]
MLD-3	$<$ 0:01	0.4	6074	912
MLD-4	$<$ 0:01	0.4	3897	707
CH	0:02	0.4	284	96.3
CH-15	0:04	0.4	231	85.0
CH-17	0:24	0.4	217	79.7

CH-x mit 2^x top-down Ordnung,
zur Erinnerung: Graph hat $> 2^{24}$ Knoten

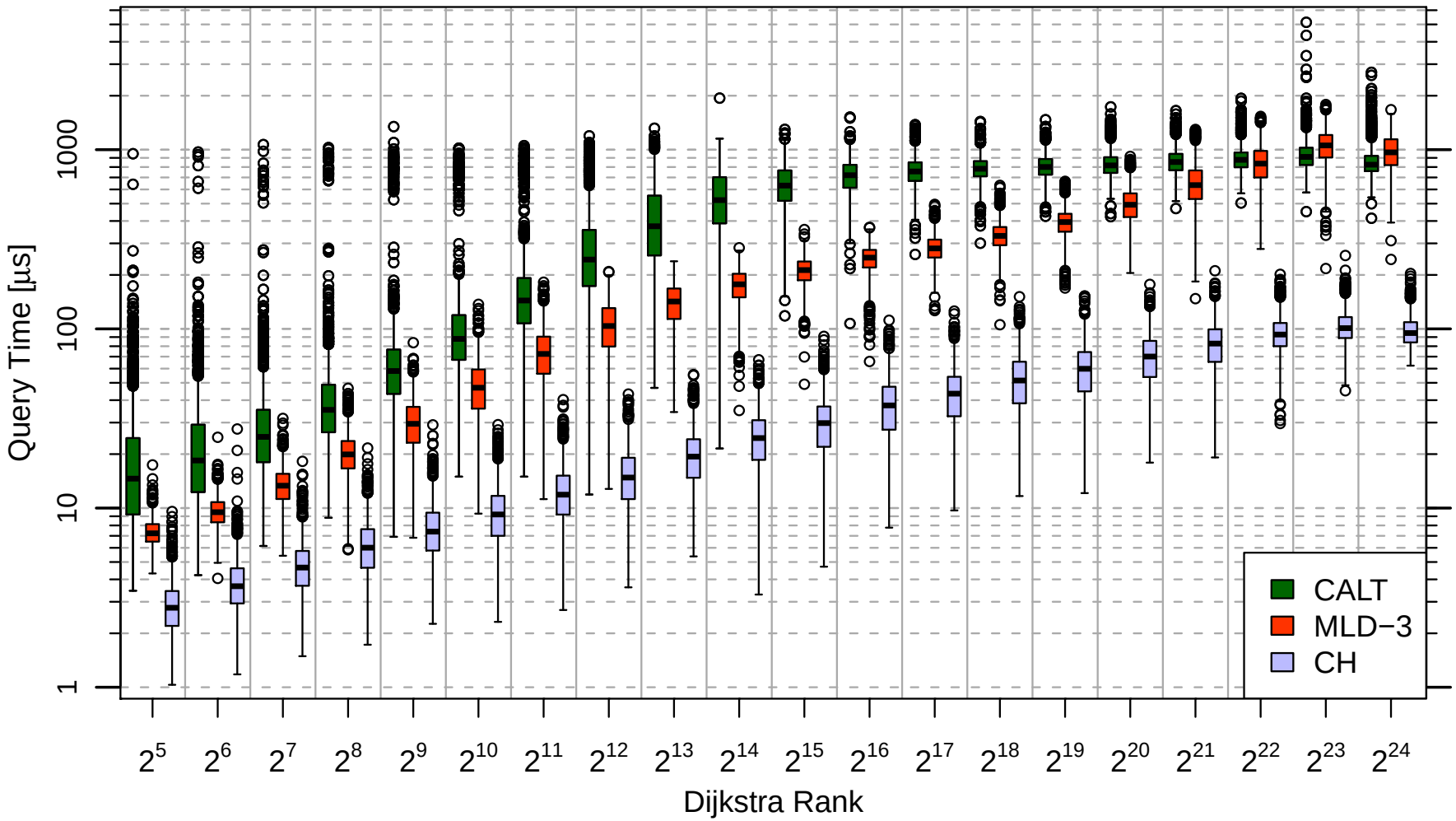
Ergebnisse

	preprocessing		query	
method	time [h:m]	space [GB]	scans	time [μs]
MLD-3	$<$ 0:01	0.4	6074	912
MLD-4	$<$ 0:01	0.4	3897	707
CH	0:02	0.4	284	96.3
CH-15	0:04	0.4	231	85.0
CH-17	0:24	0.4	217	79.7

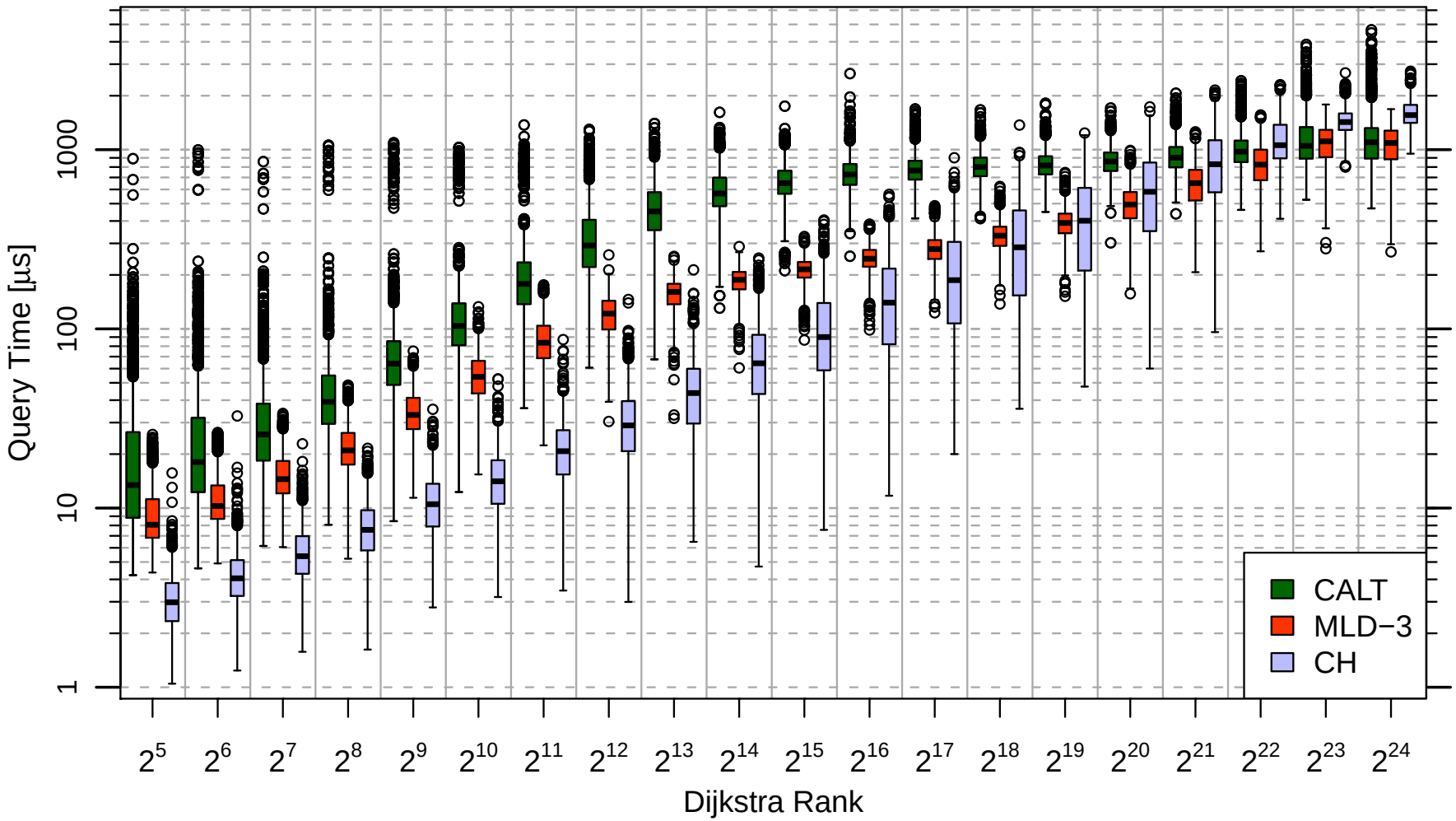
CH-x mit 2^x top-down Ordnung,
zur Erinnerung: Graph hat $> 2^{24}$ Knoten

- CH etwas langsamere Vorberechnung
- Faktor 10 schneller als MLD
- bottom-up Knotenordnung gut genug

Local Queries: Reisezeiten



Local Queries: Reisedistanzen



Literatur I

Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck.
Hierarchical hub labelings for shortest paths.
In *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*, volume 7501 of *Lecture Notes in Computer Science*, pages 24–35. Springer, 2012.

Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck.
Robust distance queries on massive networks.
In *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA'14)*, volume 8737 of *Lecture Notes in Computer Science*, pages 321–333. Springer, September 2014.

Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter.
Exact routing in large road networks using contraction hierarchies.
Transportation Science, 46(3):388–404, August 2012.