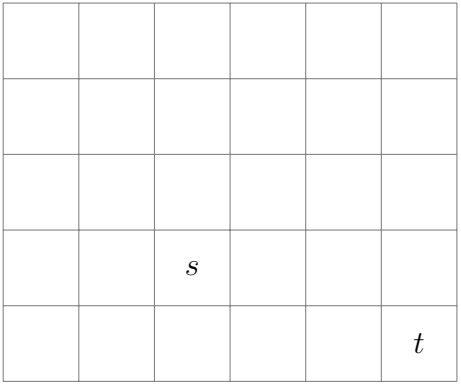
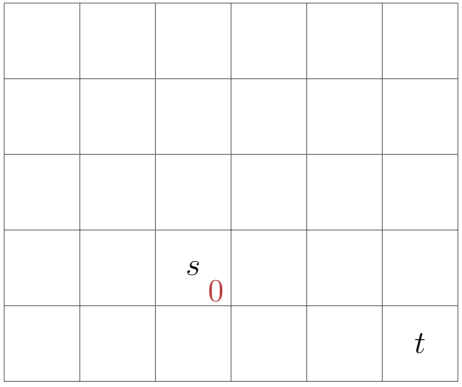
26.5 A*-Algorithmus

Motivation A*



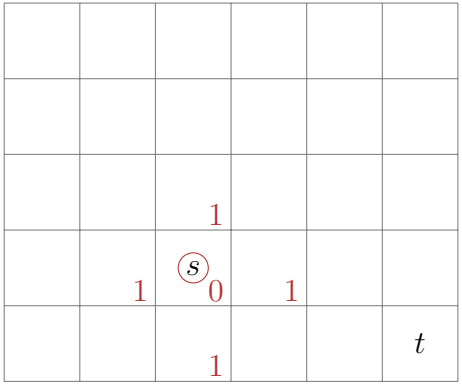
- Algorithmus von Dijkstra sucht die kürzesten Wege zu allen Knoten, in alle Richtungen

Motivation A*



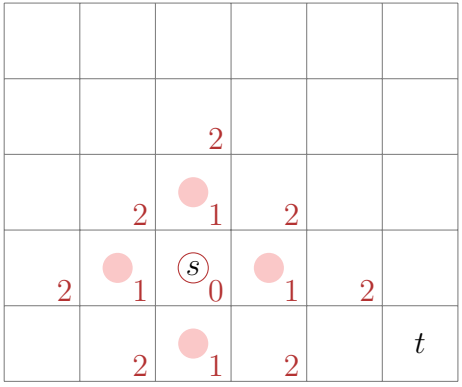
- Algorithmus von Dijkstra sucht die kürzesten Wege zu allen Knoten, in alle Richtungen

Motivation A*



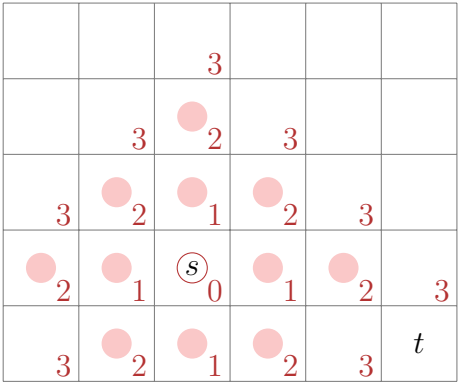
- Algorithmus von Dijkstra sucht die kürzesten Wege zu allen Knoten, in alle Richtungen

Motivation A*



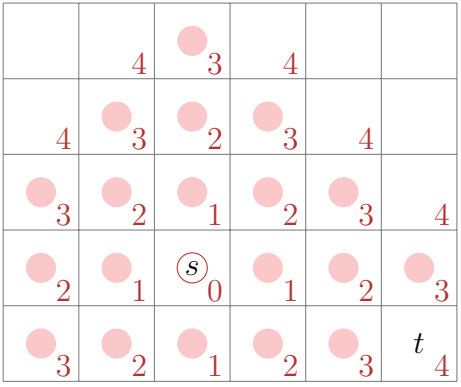
- Algorithmus von Dijkstra sucht die kürzesten Wege zu allen Knoten, in alle Richtungen

Motivation A*



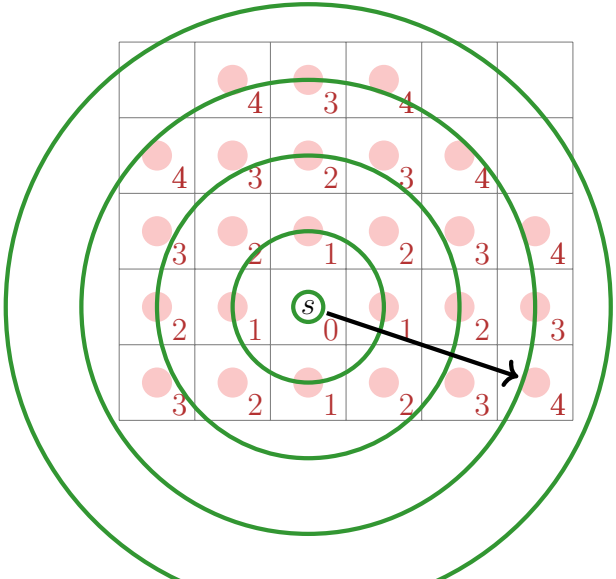
- Algorithmus von Dijkstra sucht die kürzesten Wege zu allen Knoten, in alle Richtungen

Motivation A*



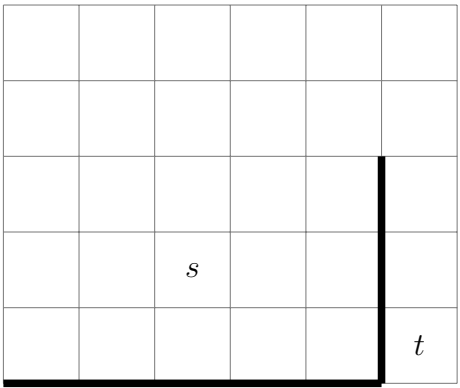
- Algorithmus von Dijkstra sucht die kürzesten Wege zu allen Knoten, in alle Richtungen

Motivation A*



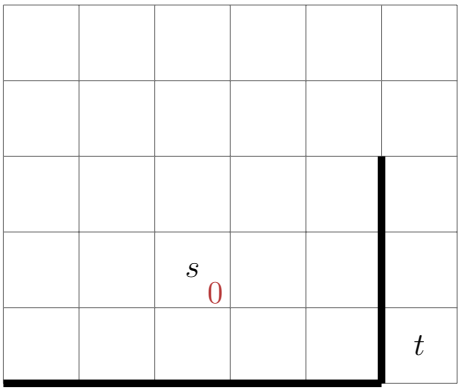
- Algorithmus von Dijkstra sucht die kürzesten Wege zu allen Knoten, in alle Richtungen

Motivation A*



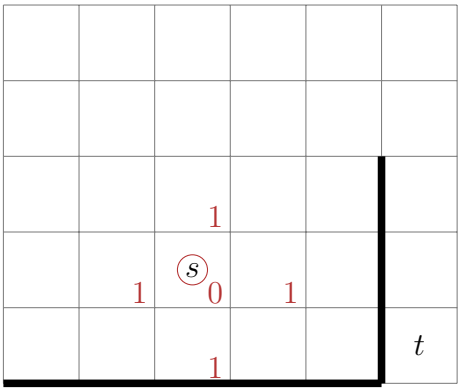
- Algorithmus von Dijkstra sucht die kürzesten Wege zu allen Knoten, in alle Richtungen
- was richtig ist, da die Stuktur des Graphen dem Algorithmus nicht bekannt ist

Motivation A*



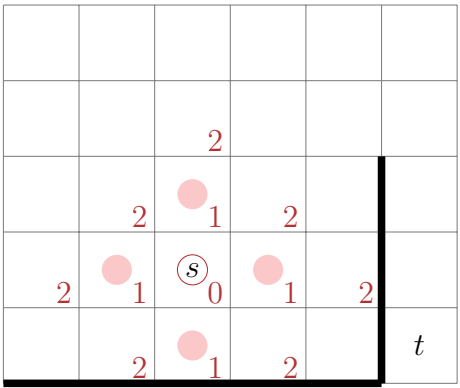
- Algorithmus von Dijkstra sucht die kürzesten Wege zu allen Knoten, in alle Richtungen
- was richtig ist, da die Stuktur des Graphen dem Algorithmus nicht bekannt ist

Motivation A*



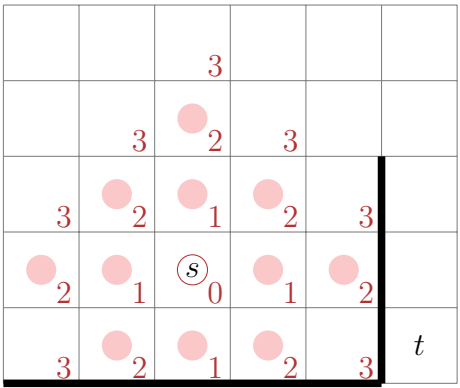
- Algorithmus von Dijkstra sucht die kürzesten Wege zu allen Knoten, in alle Richtungen
- was richtig ist, da die Stuktur des Graphen dem Algorithmus nicht bekannt ist

Motivation A*



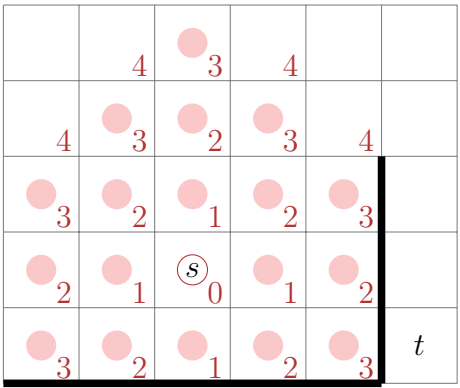
- Algorithmus von Dijkstra sucht die kürzesten Wege zu allen Knoten, in alle Richtungen
- was richtig ist, da die Stuktur des Graphen dem Algorithmus nicht bekannt ist

Motivation A*



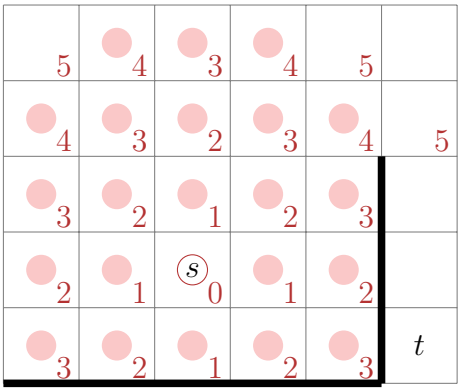
- Algorithmus von Dijkstra sucht die kürzesten Wege zu allen Knoten, in alle Richtungen
- was richtig ist, da die Stuktur des Graphen dem Algorithmus nicht bekannt ist

Motivation A*



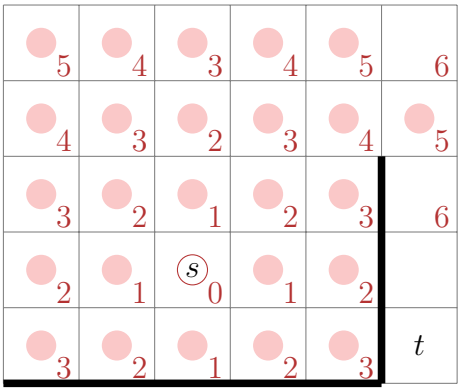
- Algorithmus von Dijkstra sucht die kürzesten Wege zu allen Knoten, in alle Richtungen
- was richtig ist, da die Stuktur des Graphen dem Algorithmus nicht bekannt ist

Motivation A*



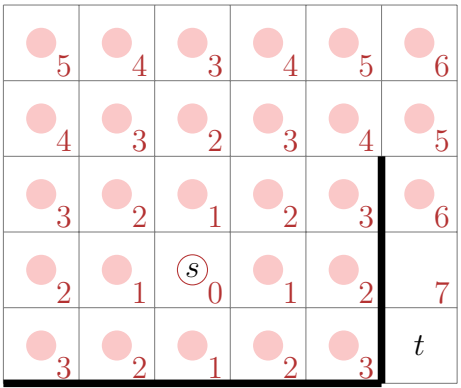
- Algorithmus von Dijkstra sucht die kürzesten Wege zu allen Knoten, in alle Richtungen
- was richtig ist, da die Stuktur des Graphen dem Algorithmus nicht bekannt ist

Motivation A*



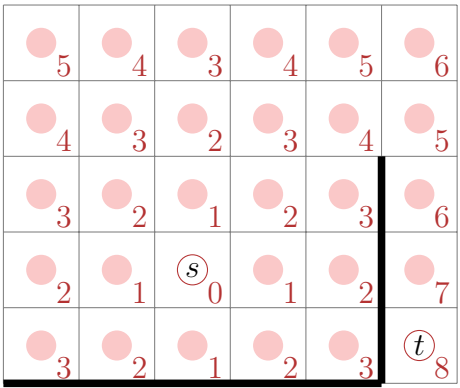
- Algorithmus von Dijkstra sucht die kürzesten Wege zu allen Knoten, in alle Richtungen
- was richtig ist, da die Stuktur des Graphen dem Algorithmus nicht bekannt ist

Motivation A*



- Algorithmus von Dijkstra sucht die kürzesten Wege zu allen Knoten, in alle Richtungen
- was richtig ist, da die Stuktur des Graphen dem Algorithmus nicht bekannt ist

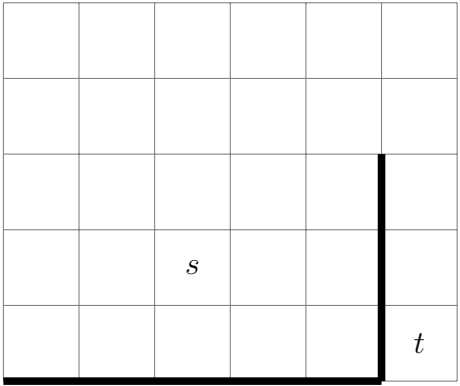
Motivation A*



- Algorithmus von Dijkstra sucht die kürzesten Wege zu allen Knoten, in alle Richtungen
- was richtig ist, da die Stuktur des Graphen dem Algorithmus nicht bekannt ist

A* in Action

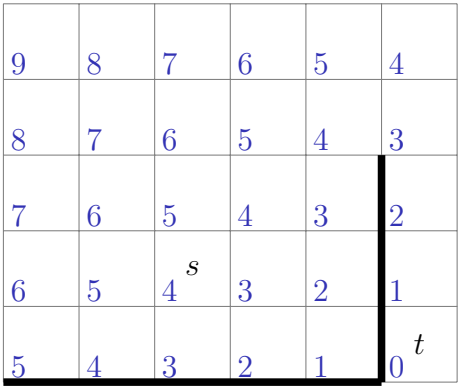
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ ($\hat{h} = \delta_x + \delta_y$ Manhattan-Distanz)



- Idee: führe den Algorithmus in eine bevorzugte Richtung mit Hilfe einer Abstandsheuristik $\hat{h}$
- Der Wert dieser Heuristik muss den wahren Abstand zu t unterschätzen und wird zum gefundenen Abstand d_s zu s addiert.

A* in Action

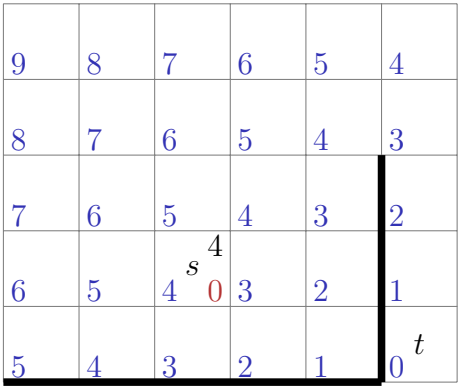
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ ($\hat{h} = \delta_x + \delta_y$ Manhattan-Distanz)



- Idee: führe den Algorithmus in eine bevorzugte Richtung mit Hilfe einer Abstandsheuristik $\hat{h}$
- Der Wert dieser Heuristik muss den wahren Abstand zu t unterschätzen und wird zum gefundenen Abstand d_s zu s addiert.

A* in Action

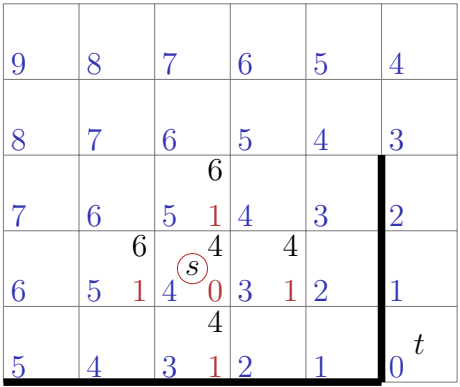
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ ($\hat{h} = \delta_x + \delta_y$ Manhattan-Distanz)



- Idee: führe den Algorithmus in eine bevorzugte Richtung mit Hilfe einer Abstandsheuristik $\hat{h}$
- Der Wert dieser Heuristik muss den wahren Abstand zu t unterschätzen und wird zum gefundenen Abstand d_s zu s addiert.

A* in Action

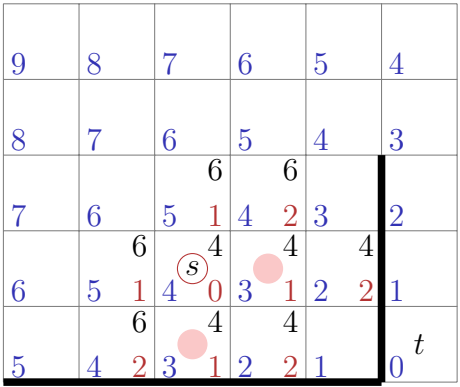
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ ($\hat{h} = \delta_x + \delta_y$ Manhattan-Distanz)



- Idee: führe den Algorithmus in eine bevorzugte Richtung mit Hilfe einer Abstandsheuristik $\hat{h}$
- Der Wert dieser Heuristik muss den wahren Abstand zu t unterschätzen und wird zum gefundenen Abstand d_s zu s addiert.

A* in Action

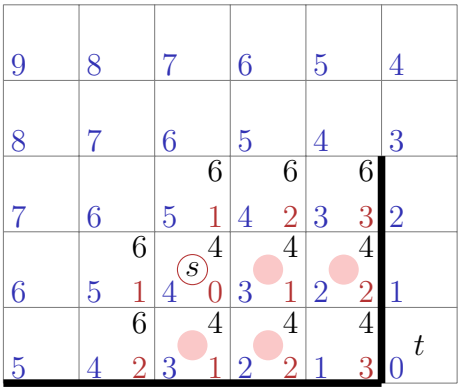
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ ($\hat{h} = \delta_x + \delta_y$ Manhattan-Distanz)



- Idee: führe den Algorithmus in eine bevorzugte Richtung mit Hilfe einer Abstandsheuristik $\hat{h}$
- Der Wert dieser Heuristik muss den wahren Abstand zu t unterschätzen und wird zum gefundenen Abstand d_s zu s addiert.

A* in Action

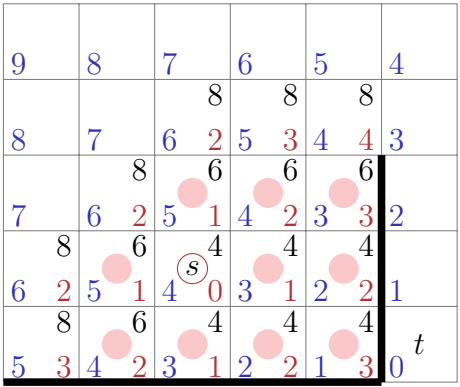
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ ($\hat{h} = \delta_x + \delta_y$ Manhattan-Distanz)



- Idee: führe den Algorithmus in eine bevorzugte Richtung mit Hilfe einer Abstandsheuristik $\hat{h}$
- Der Wert dieser Heuristik muss den wahren Abstand zu t unterschätzen und wird zum gefundenen Abstand d_s zu s addiert.

A* in Action

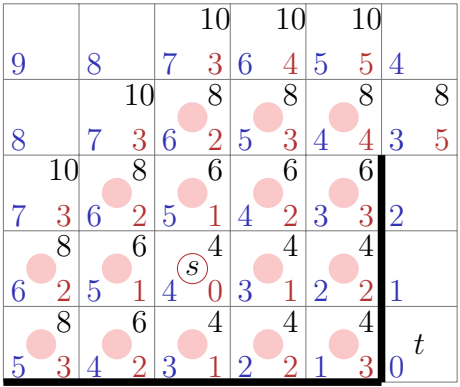
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ ($\hat{h} = \delta_x + \delta_y$ Manhattan-Distanz)



- Idee: führe den Algorithmus in eine bevorzugte Richtung mit Hilfe einer Abstandsheuristik $\hat{h}$
- Der Wert dieser Heuristik muss den wahren Abstand zu t unterschätzen und wird zum gefundenen Abstand d_s zu s addiert.

A* in Action

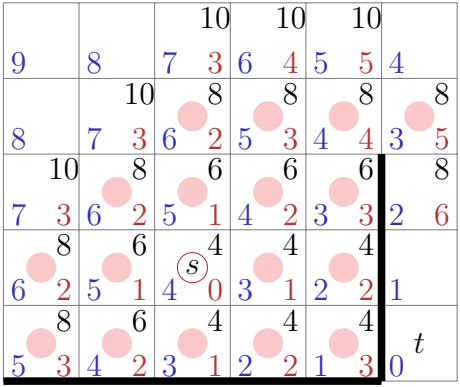
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ ($\hat{h} = \delta_x + \delta_y$ Manhattan-Distanz)



- Idee: führe den Algorithmus in eine bevorzugte Richtung mit Hilfe einer Abstandsheuristik $\hat{h}$
- Der Wert dieser Heuristik muss den wahren Abstand zu t unterschätzen und wird zum gefundenen Abstand d_s zu s addiert.

A* in Action

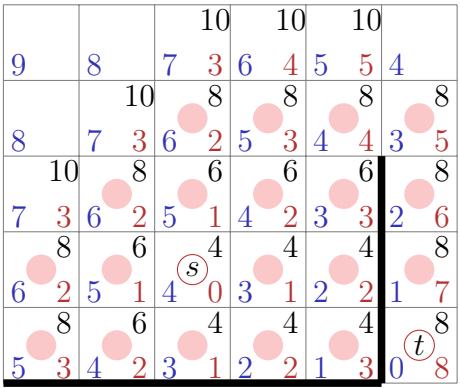
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ ($\hat{h} = \delta_x + \delta_y$ Manhattan-Distanz)



- Idee: führe den Algorithmus in eine bevorzugte Richtung mit Hilfe einer Abstandsheuristik $\hat{h}$
- Der Wert dieser Heuristik muss den wahren Abstand zu t unterschätzen und wird zum gefundenen Abstand d_s zu s addiert.

A* in Action

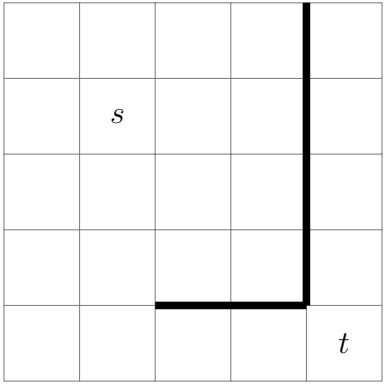
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ ($\hat{h} = \delta_x + \delta_y$ Manhattan-Distanz)



- Idee: führe den Algorithmus in eine bevorzugte Richtung mit Hilfe einer Abstandsheuristik $\hat{h}$
- Der Wert dieser Heuristik muss den wahren Abstand zu t unterschätzen und wird zum gefundenen Abstand d_s zu s addiert.

Weg zurückverfolgen

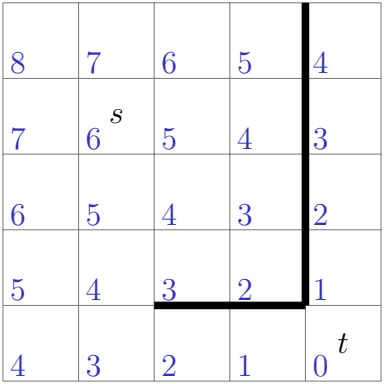
$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x + \delta_y)$



- Der Algorithmus funktioniert genauso wie der Dijkstra-Algorithmus.
- Anstelle des Wertes von d_s tritt für das Suchen des nächsten Elements in R der Wert von $\hat{f} = \hat{h} + d_s$

Weg zurückverfolgen

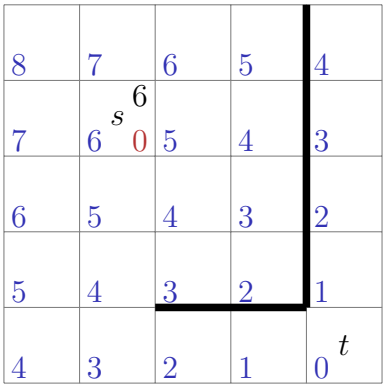
$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x + \delta_y)$



- Der Algorithmus funktioniert genauso wie der Dijkstra-Algorithmus.
- Anstelle des Wertes von d_s tritt für das Suchen des nächsten Elements in R der Wert von $\hat{f} = \hat{h} + d_s$

Weg zurückverfolgen

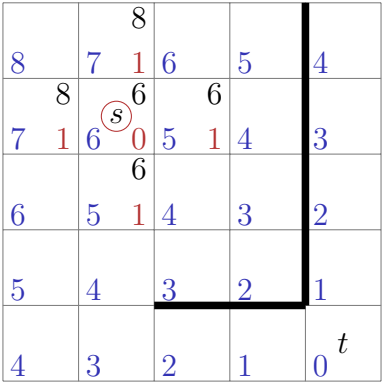
$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x + \delta_y)$



- Der Algorithmus funktioniert genauso wie der Dijkstra-Algorithmus.
- Anstelle des Wertes von d_s tritt für das Suchen des nächsten Elements in R der Wert von $\hat{f} = \hat{h} + d_s$

Weg zurückverfolgen

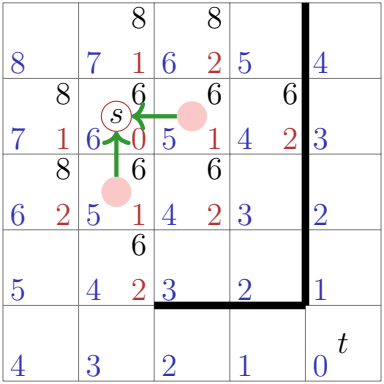
$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x + \delta_y)$



- Der Algorithmus funktioniert genauso wie der Dijkstra-Algorithmus.
- Anstelle des Wertes von d_s tritt für das Suchen des nächsten Elements in R der Wert von $\hat{f} = \hat{h} + d_s$

Weg zurückverfolgen

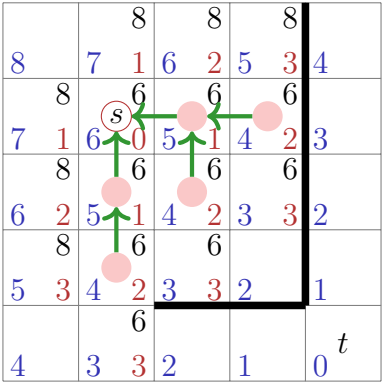
$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x + \delta_y)$



- Der Algorithmus funktioniert genauso wie der Dijkstra-Algorithmus.
- Anstelle des Wertes von d_s tritt für das Suchen des nächsten Elements in R der Wert von $\hat{f} = \hat{h} + d_s$

Weg zurückverfolgen

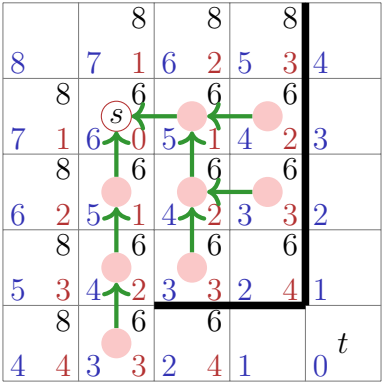
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x + \delta_y)$$



- Der Algorithmus funktioniert genauso wie der Dijkstra-Algorithmus.
- Anstelle des Wertes von d_s tritt für das Suchen des nächsten Elements in R der Wert von $\hat{f} = \hat{h} + d_s$

Weg zurückverfolgen

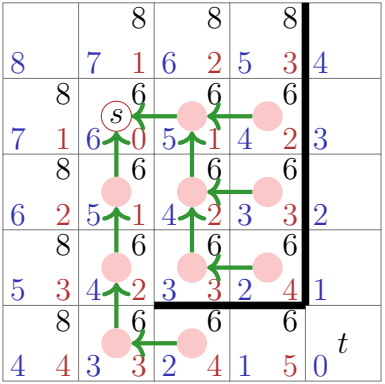
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x + \delta_y)$$



- Der Algorithmus funktioniert genauso wie der Dijkstra-Algorithmus.
- Anstelle des Wertes von d_s tritt für das Suchen des nächsten Elements in R der Wert von $\hat{f} = \hat{h} + d_s$

Weg zurückverfolgen

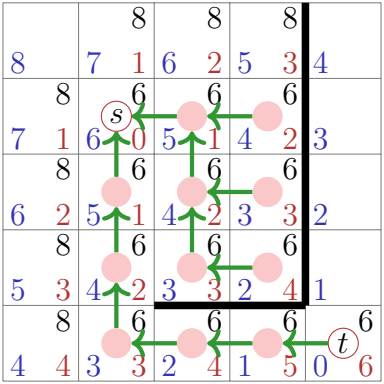
$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x + \delta_y)$



- Der Algorithmus funktioniert genauso wie der Dijkstra-Algorithmus.
- Anstelle des Wertes von d_s tritt für das Suchen des nächsten Elements in R der Wert von $\hat{f} = \hat{h} + d_s$

Weg zurückverfolgen

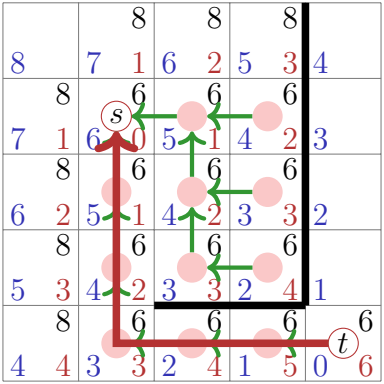
$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x + \delta_y)$



- Der Algorithmus funktioniert genauso wie der Dijkstra-Algorithmus.
- Anstelle des Wertes von d_s tritt für das Suchen des nächsten Elements in R der Wert von $\hat{f} = \hat{h} + d_s$

Weg zurückverfolgen

$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x + \delta_y)$$



- Der Algorithmus funktioniert genauso wie der Dijkstra-Algorithmus.
- Anstelle des Wertes von d_s tritt für das Suchen des nächsten Elements in R der Wert von $\hat{f} = \hat{h} + d_s$

A*-Algorithmus

Voraussetzungen

- Positiv gewichteter, endlicher Graph $G = (V, E, c)$
- $s \in V$, $t \in V$
- Abstandsschätzung $\widehat{h}_t(v) \leq h_t(v) := \delta(v, t) \; \forall \; v \in V$.
- Gesucht: kürzester Pfad $p : s \rightsquigarrow t$

A*-Algorithmus($G, s, t, \hat{h}$)

Input: Positiv gewichteter Graph $G = (V, E, c)$, Startpunkt $s \in V$, Endpunkt $t \in V$, Schätzung $\widehat{h}(v) \leq \delta(v, t)$

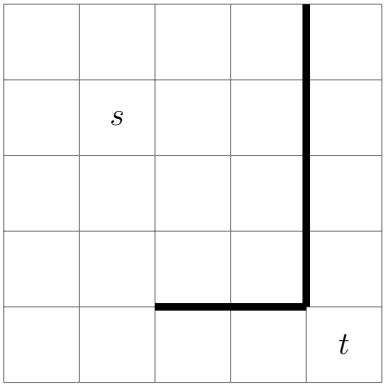
Output: Existenz und Wert eines kürzesten Pfades von s nach t

foreach $u \in V$ **do**
  $d[u] \leftarrow \infty$; $\widehat{f}[u] \leftarrow \infty$; $\pi[u] \leftarrow$ null

$d[s] \leftarrow 0$; $\widehat{f}[s] \leftarrow \widehat{h}(s)$; $N \leftarrow \{s\}$; $K \leftarrow \{\}$
while $N \neq \emptyset$ **do**
  $u \leftarrow \mathsf{ExtractMin}_{\widehat{f}}(N)$; $K \leftarrow K \cup \{u\}$
  if $u = t$ **then return** success
  foreach $v \in N^+(u)$ with $d[v] > d[u] + c(u, v)$ **do**
    $d[v] \leftarrow d[u] + c(u, v)$; $\widehat{f}[v] \leftarrow d[v] + \widehat{h}(v)$; $\pi[v] \leftarrow u$
    $N \leftarrow N \cup \{v\}$; $K \leftarrow K - \{v\}$
return failure

Was, wenn $\hat{h}$ nicht unterschätzt

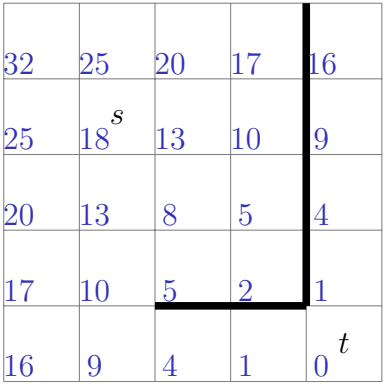
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- Algorithmus kann mit dem falschen Resultat terminieren, wenn $\hat{h}$ die Distanz zu t nicht unterschätzt.
- obwohl die Heuristik ansonsten vernünftig aussieht (sie ist z.B. monoton)

Was, wenn $\hat{h}$ nicht unterschätzt

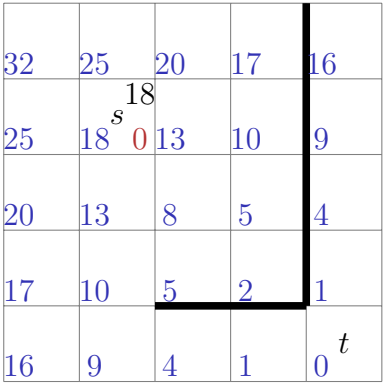
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- Algorithmus kann mit dem falschen Resultat terminieren, wenn $\hat{h}$ die Distanz zu t nicht unterschätzt.
- obwohl die Heuristik ansonsten vernünftig aussieht (sie ist z.B. monoton)

Was, wenn $\hat{h}$ nicht unterschätzt

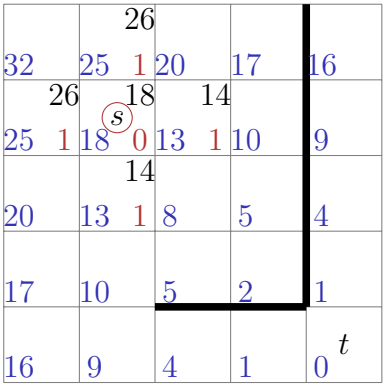
$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x^2 + \delta_y^2)$



- Algorithmus kann mit dem falschen Resultat terminieren, wenn $\hat{h}$ die Distanz zu t nicht unterschätzt.
- obwohl die Heuristik ansonsten vernünftig aussieht (sie ist z.B. monoton)

Was, wenn $\hat{h}$ nicht unterschätzt

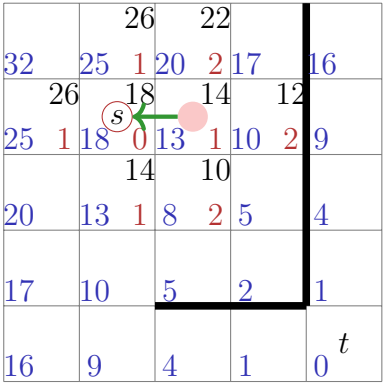
$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x^2 + \delta_y^2)$



- Algorithmus kann mit dem falschen Resultat terminieren, wenn $\hat{h}$ die Distanz zu t nicht unterschätzt.
- obwohl die Heuristik ansonsten vernünftig aussieht (sie ist z.B. monoton)

Was, wenn $\hat{h}$ nicht unterschätzt

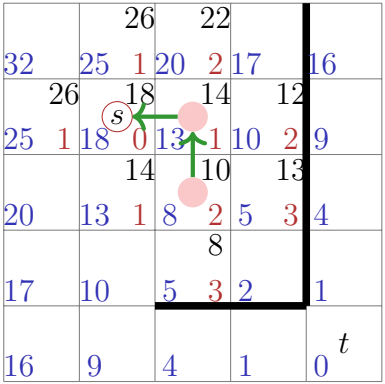
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- Algorithmus kann mit dem falschen Resultat terminieren, wenn $\hat{h}$ die Distanz zu t nicht unterschätzt.
- obwohl die Heuristik ansonsten vernünftig aussieht (sie ist z.B. monoton)

Was, wenn $\hat{h}$ nicht unterschätzt

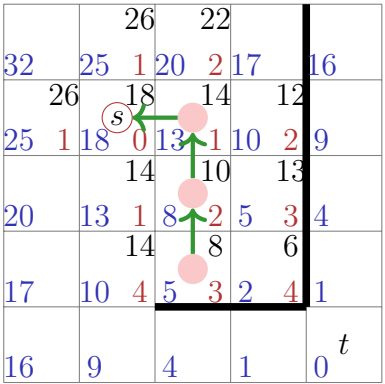
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- Algorithmus kann mit dem falschen Resultat terminieren, wenn $\hat{h}$ die Distanz zu t nicht unterschätzt.
- obwohl die Heuristik ansonsten vernünftig aussieht (sie ist z.B. monoton)

Was, wenn $\hat{h}$ nicht unterschätzt

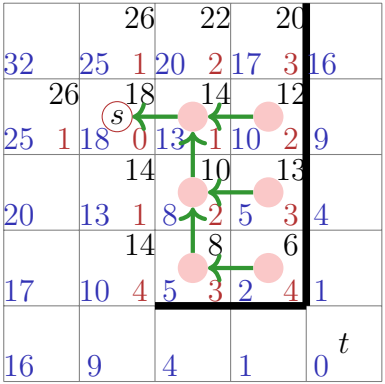
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- Algorithmus kann mit dem falschen Resultat terminieren, wenn $\hat{h}$ die Distanz zu t nicht unterschätzt.
- obwohl die Heuristik ansonsten vernünftig aussieht (sie ist z.B. monoton)

Was, wenn $\hat{h}$ nicht unterschätzt

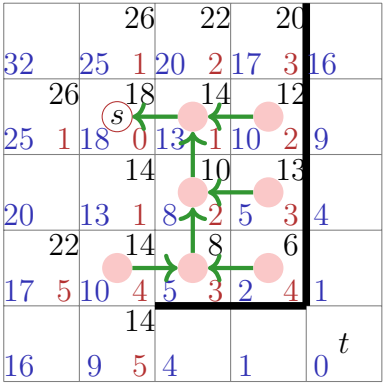
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- Algorithmus kann mit dem falschen Resultat terminieren, wenn $\hat{h}$ die Distanz zu t nicht unterschätzt.
- obwohl die Heuristik ansonsten vernünftig aussieht (sie ist z.B. monoton)

Was, wenn $\hat{h}$ nicht unterschätzt

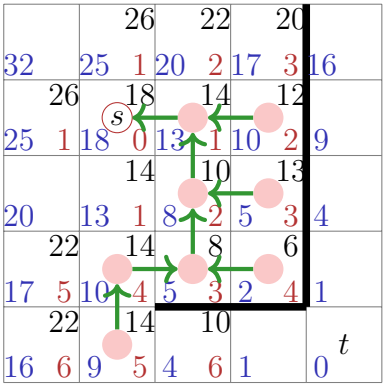
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- Algorithmus kann mit dem falschen Resultat terminieren, wenn $\hat{h}$ die Distanz zu t nicht unterschätzt.
- obwohl die Heuristik ansonsten vernünftig aussieht (sie ist z.B. monoton)

Was, wenn $\hat{h}$ nicht unterschätzt

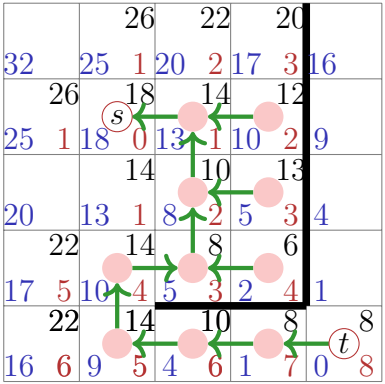
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- Algorithmus kann mit dem falschen Resultat terminieren, wenn $\hat{h}$ die Distanz zu t nicht unterschätzt.
- obwohl die Heuristik ansonsten vernünftig aussieht (sie ist z.B. monoton)

Was, wenn $\hat{h}$ nicht unterschätzt

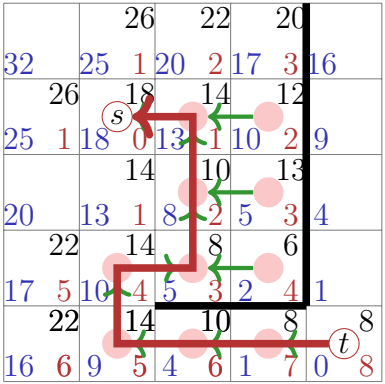
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- Algorithmus kann mit dem falschen Resultat terminieren, wenn $\hat{h}$ die Distanz zu t nicht unterschätzt.
- obwohl die Heuristik ansonsten vernünftig aussieht (sie ist z.B. monoton)

Was, wenn $\hat{h}$ nicht unterschätzt

$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- Algorithmus kann mit dem falschen Resultat terminieren, wenn $\hat{h}$ die Distanz zu t nicht unterschätzt.
- obwohl die Heuristik ansonsten vernünftig aussieht (sie ist z.B. monoton)

Erneutes Besuchen von Knoten

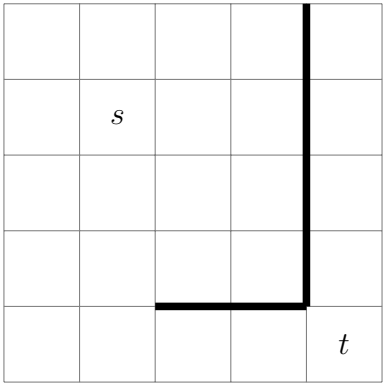
- Der A*-Algorithmus kann Knoten mehrfach aus der Menge R entnehmen und sie später wieder einfügen.
- Das kann zu suboptimalem Verhalten im Sinne der Laufzeit des Algorithmus führen.
- Wenn $\widehat{h}$ zusätzlich zur Zulässigkeit ($\widehat{h}(v) \leq h(v)$ für alle $v \in V$) auch noch monoton ist, d.h. wenn für alle $(u, u') \in E$:

$$\widehat{h}(u') \leq \widehat{h}(u) + c(u', u)$$

  dann ist der A* Algorithmus äquivalent zum Dijkstra-Algorithmus mit Kantengewichten $\tilde{c}(u, v) = c(u, v) + \widehat{h}(u) - \widehat{h}(v)$ und kein Knoten wird aus R entnommen und wieder eingefügt.
- Es ist allerdings nicht immer möglich, eine monotone Heuristik zu finden.

Ein verrücktes $\hat{h}$

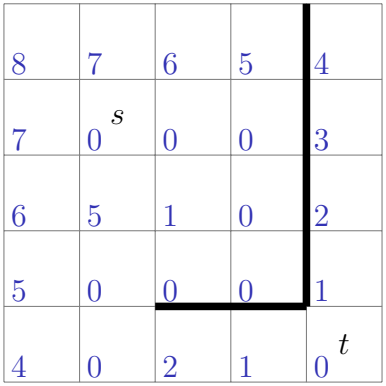
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithmus terminert mit dem korrekten Resultat, auch wenn die Abstandsheuristik nicht monoton ist
- Dann kann es kann vorkommen, dass Knoten mehrfach aus R entnommen und wieder eingefügt werden.

Ein verrücktes $\hat{h}$

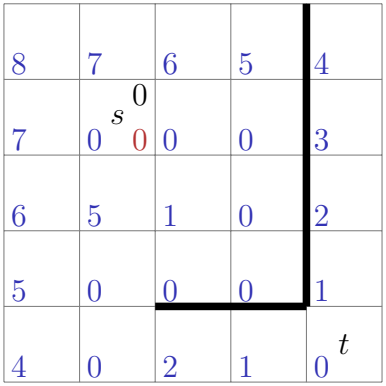
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithmus terminert mit dem korrekten Resultat, auch wenn die Abstandsheuristik nicht monoton ist
- Dann kann es kann vorkommen, dass Knoten mehrfach aus R entnommen und wieder eingefügt werden.

Ein verrücktes $\hat{h}$

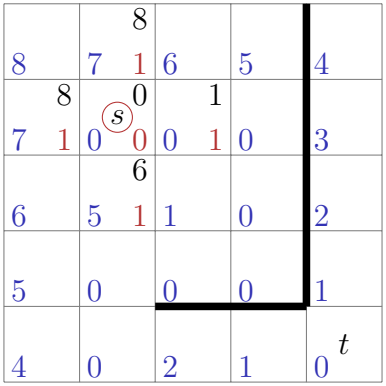
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithmus terminert mit dem korrekten Resultat, auch wenn die Abstandsheuristik nicht monoton ist
- Dann kann es kann vorkommen, dass Knoten mehrfach aus R entnommen und wieder eingefügt werden.

Ein verrücktes $\hat{h}$

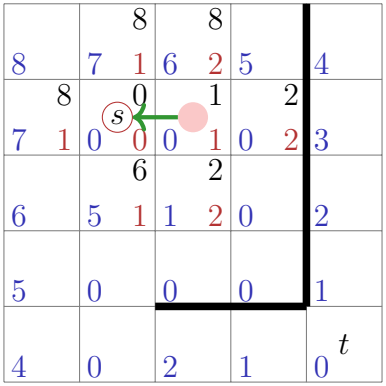
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithmus terminert mit dem korrekten Resultat, auch wenn die Abstandsheuristik nicht monoton ist
- Dann kann es kann vorkommen, dass Knoten mehrfach aus R entnommen und wieder eingefügt werden.

Ein verrücktes $\hat{h}$

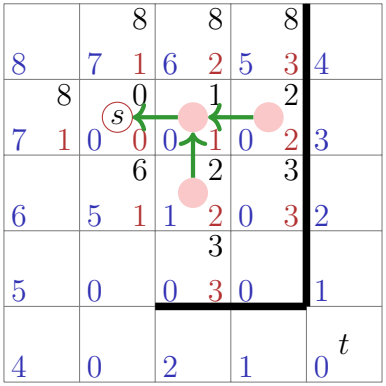
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithmus terminert mit dem korrekten Resultat, auch wenn die Abstandsheuristik nicht monoton ist
- Dann kann es kann vorkommen, dass Knoten mehrfach aus R entnommen und wieder eingefügt werden.

Ein verrücktes $\hat{h}$

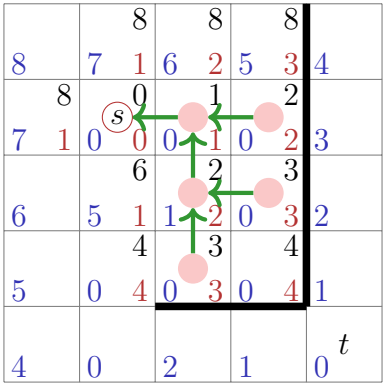
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithmus terminert mit dem korrekten Resultat, auch wenn die Abstandsheuristik nicht monoton ist
- Dann kann es kann vorkommen, dass Knoten mehrfach aus R entnommen und wieder eingefügt werden.

Ein verrücktes $\hat{h}$

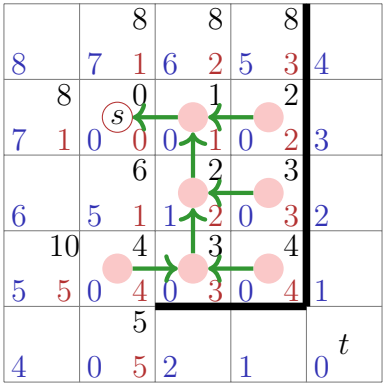
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithmus terminert mit dem korrekten Resultat, auch wenn die Abstandsheuristik nicht monoton ist
- Dann kann es kann vorkommen, dass Knoten mehrfach aus R entnommen und wieder eingefügt werden.

Ein verrücktes $\hat{h}$

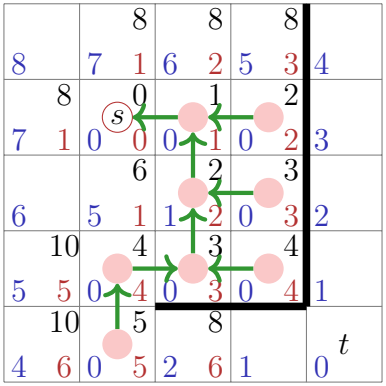
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithmus terminert mit dem korrekten Resultat, auch wenn die Abstandsheuristik nicht monoton ist
- Dann kann es kann vorkommen, dass Knoten mehrfach aus R entnommen und wieder eingefügt werden.

Ein verrücktes $\hat{h}$

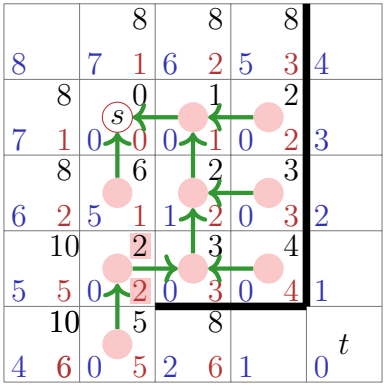
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithmus terminert mit dem korrekten Resultat, auch wenn die Abstandsheuristik nicht monoton ist
- Dann kann es kann vorkommen, dass Knoten mehrfach aus R entnommen und wieder eingefügt werden.

Ein verrücktes $\hat{h}$

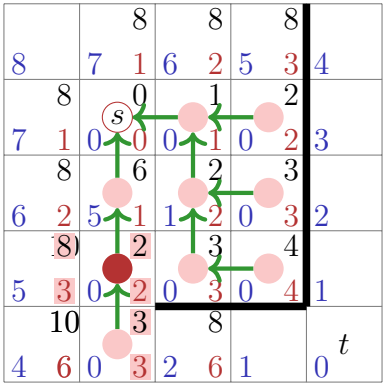
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithmus terminert mit dem korrekten Resultat, auch wenn die Abstandsheuristik nicht monoton ist
- Dann kann es kann vorkommen, dass Knoten mehrfach aus R entnommen und wieder eingefügt werden.

Ein verrücktes $\hat{h}$

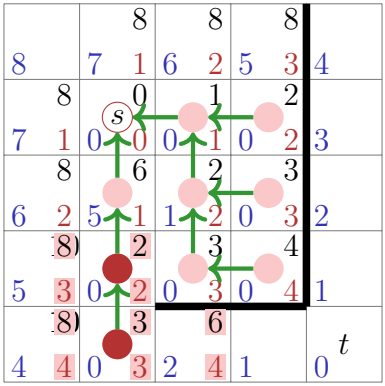
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithmus terminert mit dem korrekten Resultat, auch wenn die Abstandsheuristik nicht monoton ist
- Dann kann es kann vorkommen, dass Knoten mehrfach aus R entnommen und wieder eingefügt werden.

Ein verrücktes $\hat{h}$

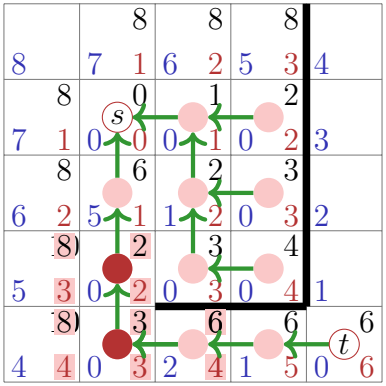
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithmus terminert mit dem korrekten Resultat, auch wenn die Abstandsheuristik nicht monoton ist
- Dann kann es kann vorkommen, dass Knoten mehrfach aus R entnommen und wieder eingefügt werden.

Ein verrücktes $\hat{h}$

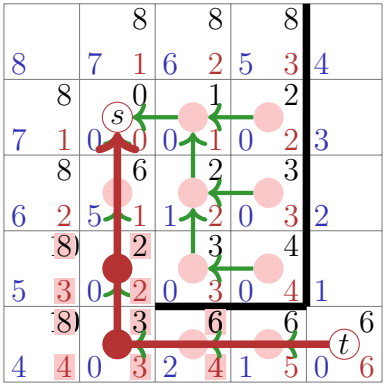
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithmus terminert mit dem korrekten Resultat, auch wenn die Abstandsheuristik nicht monoton ist
- Dann kann es kann vorkommen, dass Knoten mehrfach aus R entnommen und wieder eingefügt werden.

Ein verrücktes $\hat{h}$

$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithmus terminert mit dem korrekten Resultat, auch wenn die Abstandsheuristik nicht monoton ist
- Dann kann es kann vorkommen, dass Knoten mehrfach aus R entnommen und wieder eingefügt werden.

Zusammenfassung

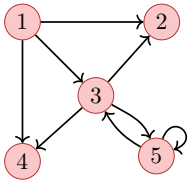
- Der A*-Algorithmus ist eine Erweiterung des Dijkstra-Algorithmus um eine Abstandsheuristik $\hat{h}$.
- A* = Dijkstra wenn $\hat{h} \equiv 0$.
- Wenn $\hat{h}$ den Abstand unterschätzt, funktioniert der Algorithmus korrekt.
- Wenn $\hat{h}$ ausserdem monoton ist, dann ist der Algorithmus auch effizient.
- In der Praxis (z.B. Routing) ist die Wahl von $\hat{h}$ oft intuitiv klar und führt zu deutlich verbessertem Verhalten im Vergleich zu Dijkstra.
- Beweis der Korrektheit im Handout

27. Transitive Hülle, alle kürzesten Wege

Reflexive transitive Hülle [Ottman/Widmayer, Kap. 9.2 Cormen et al, Kap. 25.2] Algorithmus von Floyd-Warshall [Ottman/Widmayer, Kap. 9.5.3 Cormen et al, Kap. 25.2]

Adjazenzmatrizen multipliziert

$$B := A_G^2 = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}^2 = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 2 \end{pmatrix}$$



Interpretation

Theorem 27

Sei $G = (V, E)$ ein Graph und $k \in \mathbb{N}$. Dann gibt das Element $a_{i,j}^{(k)}$ der Matrix $(a_{i,j}^{(k)})_{1 \leq i,j \leq n} = (A_G)^k$ die Anzahl der Wege mit Länge k von v_i nach v_j an.

Graphen und Relationen

Graph $G = (V, E)$
Adjazenzen $A_G \mathrel{\widehat{=}}$ Relation $E \subseteq V \times V$ auf V

Graphen und Relationen

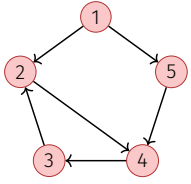
Graph $G = (V, E)$
Adjazenzen $A_G \mathrel{\widehat{=}}$ Relation $E \subseteq V \times V$ auf V

- **reflexiv** $\Leftrightarrow a_{i,i} = 1$ für alle $i = 1, \ldots, n$. (Schleifen)
- **symmetrisch** $\Leftrightarrow a_{i,j} = a_{j,i}$ für alle $i, j = 1, \ldots, n$ (ungerichtet)
- **transitiv** $\Leftrightarrow \ (u, v) \in E, (v, w) \in E \Rightarrow (u, w) \in E$. (Erreichbarkeit)

Reflexive Transitive Hülle

Reflexive transitive Hülle von $G \Leftrightarrow$ **Erreichbarkeitsrelation** E^*:
$(v, w) \in E^*$ gdw. $\exists$ Weg von Knoten v zu w.

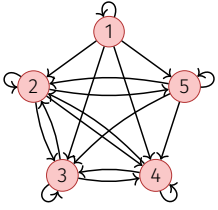
$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$



$G = (V, E)$

$\Rightarrow$

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$



$G^* = (V, E^*)$

Algorithmus $A \cdot A$

Input: (Adjazenz-)Matrix $A = (a_{ij})_{i,j=1\ldots n}$
Output: Matrixprodukt $B = (b_{ij})_{i,j=1\ldots n} = A \cdot A$

$B \leftarrow 0$
for $r \leftarrow 1$ **to** n **do**
  for $c \leftarrow 1$ **to** n **do**
    for $k \leftarrow 1$ **to** n **do**
      $b_{rc} \leftarrow b_{rc} + a_{rk} \cdot a_{kc}$ // Anzahl Pfade
return B

Berechnet Anzahl Pfade der Länge 2

Algorithmus $A \otimes A$

Input: Adjazenzmatrix $A = (a_{ij})_{i,j=1\ldots n}$
Output: Modifiziertes Matrixprodukt $B = (b_{ij})_{i,j=1\ldots n} = A \otimes A$

$B \leftarrow A$ // Pfade erhalten
for $r \leftarrow 1$ **to** n **do**
  for $c \leftarrow 1$ **to** n **do**
    for $k \leftarrow 1$ **to** n **do**
      $b_{rc} \leftarrow \max\{b_{rc}, a_{rk} \cdot a_{kc}\}$ // Pfad: ja/nein
return B

Berechnet Existenz von Pfaden der Längen 1 und 2

Berechnung Reflexive Transitive Hülle

Ziel: Berechnung von $B = (b_{ij})_{1 \leq i,j \leq n}$ mit $b_{ij} = 1 \Leftrightarrow (v_i, v_j) \in E^*$

Berechnung Reflexive Transitive Hülle

Ziel: Berechnung von $B = (b_{ij})_{1 \leq i,j \leq n}$ mit $b_{ij} = 1 \Leftrightarrow (v_i, v_j) \in E^*$ Erste Idee:

- Starte mit $B \leftarrow A$ und setze $b_{ii} = 1$ für alle i (Reflexivität).
- Berechne

$$B_n = \bigotimes_{i=1}^{n} B$$

wie?

Berechnung Reflexive Transitive Hülle

Ziel: Berechnung von $B = (b_{ij})_{1 \leq i,j \leq n}$ mit $b_{ij} = 1 \Leftrightarrow (v_i, v_j) \in E^*$ Erste Idee:

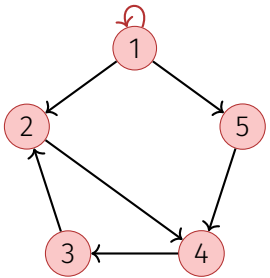
- Starte mit $B \leftarrow A$ und setze $b_{ii} = 1$ für alle i (Reflexivität).
- Berechne

$$B_n = \bigotimes_{i=1}^{n} B$$

mit Potenzen von 2 $B_2 := B \otimes B$, $B_4 := B_2 \otimes B_2$, $B_8 = B_4 \otimes B_4$ ...
$\Rightarrow$ Laufzeit $n^3 \lceil \log_2 n \rceil$

Verbesserung: Algorithmus von Warshall (1962)

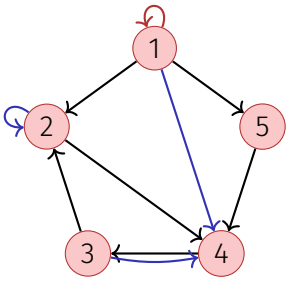
Induktiver Ansatz: Alle Wege bekannt über Knoten aus $\{v_i : i < k\}$.
Hinzunahme des Knotens v_k.



$$\begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Verbesserung: Algorithmus von Warshall (1962)

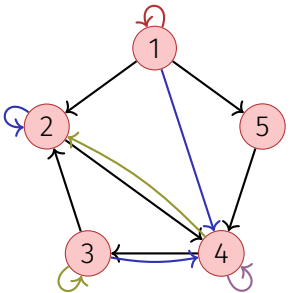
Induktiver Ansatz: Alle Wege bekannt über Knoten aus $\{v_i : i < k\}$.
Hinzunahme des Knotens v_k.



$$\begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Verbesserung: Algorithmus von Warshall (1962)

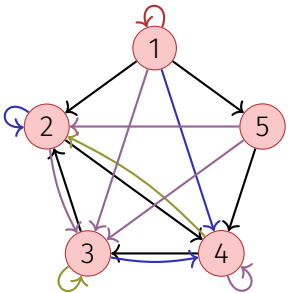
Induktiver Ansatz: Alle Wege bekannt über Knoten aus $\{v_i : i < k\}$.
Hinzunahme des Knotens v_k.



$$\begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Verbesserung: Algorithmus von Warshall (1962)

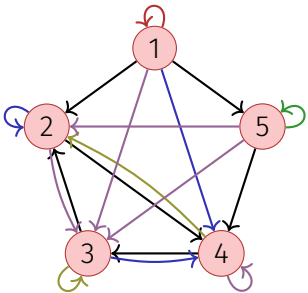
Induktiver Ansatz: Alle Wege bekannt über Knoten aus $\{v_i : i < k\}$.
Hinzunahme des Knotens v_k.



$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Verbesserung: Algorithmus von Warshall (1962)

Induktiver Ansatz: Alle Wege bekannt über Knoten aus $\{v_i : i < k\}$.
Hinzunahme des Knotens v_k.



$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Algorithmus TransitiveClosure(A_G)

Input: Adjazenzmatrix $A_G = (a_{ij})_{i,j=1\ldots n}$
Output: Reflexive Transitive Hülle $B = (b_{ij})_{i,j=1\ldots n}$ von G

$B \leftarrow A_G$
for $k \leftarrow 1$ **to** n **do**
    $b_{kk} \leftarrow 1$ // Reflexivität
    for $r \leftarrow 1$ **to** n **do**
        for $c \leftarrow 1$ **to** n **do**
            $b_{rc} \leftarrow \max\{b_{rc}, b_{rk} \cdot b_{kc}\}$ // Alle Wege über v_k
return B

Laufzeit des Algorithmus $\Theta(n^3)$.

Korrektheit des Algorithmus (Induktion)

Invariante (k): alle Wege über Knoten mit maximalem Index $< k$ berücksichtigt

- **Anfang ($k = 1$)**: Alle direkten Wege (alle Kanten) in A_G berücksichtigt.
- **Hypothese**: Invariante (k) erfüllt.
- **Schritt** ($k \to k + 1$): Für jeden Weg von v_i nach v_j über Knoten mit maximalen Index k: nach Hypothese $b_{ik} = 1$ und $b_{kj} = 1$. Somit im k-ten Schleifendurchlauf: $b_{ij} \leftarrow 1$.

Alle kürzesten Pfade

Ziel: Berechne das Gewicht eines kürzesten Pfades für jedes Knotenpaar.

- $|V|\times$ Anwendung von Dijkstras ShortestPath: $\mathcal{O}(|V| \cdot (|E| + |V|) \cdot \log |V|)$
  (Mit Fibonacci-Heap:: $\mathcal{O}(|V|^2 \log |V| + |V| \cdot |E|)$)
- $|V|\times$ Anwendung von Bellman-Ford: $\mathcal{O}(|E| \cdot |V|^2)$
- Es geht besser!

Induktion über Knotennummer.

Betrachte die Gewichte aller kürzesten Wege S^k mit Zwischenknoten in[44] $V^k := \{v_1, \ldots, v_k\}$, wenn Gewichte zu allen kürzesten Wegen S^{k-1} mit Zwischenknoten in V^{k-1} gegeben sind.

- v_k kein Zwischenknoten eines kürzesten Pfades von $v_i \rightsquigarrow v_j$ in V^k: Gewicht eines kürzesten Pfades $v_i \rightsquigarrow v_j$ in S^{k-1} dann auch das Gewicht eines kürzesten Pfades in S^k.
- v_k Zwischenknoten eines kürzesten Pfades $v_i \rightsquigarrow v_j$ in V^k: Teilpfade $v_i \rightsquigarrow v_k$ und $v_k \rightsquigarrow v_j$ enthalten nur Zwischenknoten aus S^{k-1}.

[44] wie beim Algorithmus für die reflexive transitive Hülle von Warshall

Induktion über Knotennummer.

$d^k(u, v)$ = Minimales Gewicht eines Pfades $u \rightsquigarrow v$ mit Zwischenknoten aus V^k

Induktion

$$d^k(u, v) = \min\{d^{k-1}(u, v), d^{k-1}(u, k) + d^{k-1}(k, v)\} (k \geq 1)$$
$$d^0(u, v) = c(u, v)$$

Algorithmus Floyd-Warshall(G)

Input: Graph $G = (V, E, c)$ ohne Tyklen mit negativem Gewicht.
Output: Minimale Gewichte aller Pfade d

$d^0 \leftarrow c$
for $k \leftarrow 1$ **to** $|V|$ **do**
  for $i \leftarrow 1$ **to** $|V|$ **do**
    for $j \leftarrow 1$ **to** $|V|$ **do**
      $d^k(v_i, v_j) = \min\{d^{k-1}(v_i, v_j), d^{k-1}(v_i, v_k) + d^{k-1}(v_k, v_j)\}$

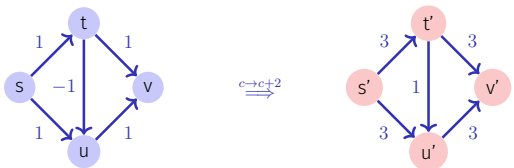
Laufzeit: $\Theta(|V|^3)$

Bemerkung: Der Algorithmus kann auf einer einzigen Matrix d (in place) ausgeführt werden.

Umgewichtung

Idee: Anwendung von Dijkstras Algorithmus auf Graphen mit negativen Gewichten durch Umgewichtung

Das folgende geht **nicht**. Die Graphen sind nicht äquivalent im Sinne der kürzesten Pfade.



Umgewichtung

Andere Idee: “Potentialfunktion” (Höhe) auf den Knoten

- $G = (V, E, c)$ ein gewichteter Graph.
- Funktion $h : V \to \mathbb{R}$
- Neue Gewichte

$$\tilde{c}(u, v) = c(u, v) + h(u) - h(v),\ (u, v \in V)$$

Umgewichtung

Beobachtung: Ein Pfad p ist genau dann kürzester Pfad in $G = (V, E, c)$, wenn er in $\tilde{G} = (V, E, \tilde{c})$ kürzester Pfad ist.

$$\begin{aligned}\tilde{c}(p) &= \sum_{i=1}^{k} \tilde{c}(v_{i-1}, v_i) = \sum_{i=1}^{k} c(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i) \\ &= h(v_0) - h(v_k) + \sum_{i=1}^{k} c(v_{i-1}, v_i) = c(p) + h(v_0) - h(v_k)\end{aligned}$$

Also $\tilde{c}(p)$ minimal unter allen $v_0 \rightsquigarrow v_k \iff c(p)$ minimal unter allen $v_0 \rightsquigarrow v_k$.

Zyklengewichte sind invariant: $\tilde{c}(v_0, \ldots, v_k = v_0) = c(v_0, \ldots, v_k = v_0)$

Johnsons Algorithmus

Hinzunahme eines neuen Knotens $s \notin V$:

$$
\begin{aligned}
G' &= (V', E', c') \\
V' &= V \cup \{s\} \\
E' &= E \cup \{(s, v) : v \in V\} \\
c'(u, v) &= c(u, v), \ u \neq s \\
c'(s, v) &= 0 (v \in V)
\end{aligned}
$$

Johnsons Algorithmus

Falls keine negativen Zyklen: wähle für Höhenfunktion Gewicht der kürzesten Pfade von s,

$$h(v) = d(s, v).$$

Für minimales Gewicht d eines Pfades gilt generell folgende Dreiecksungleichung:

$$d(s, v) \leq d(s, u) + c(u, v).$$

Einsetzen ergibt $h(v) \leq h(u) + c(u, v)$. Damit

$$\tilde{c}(u, v) = c(u, v) + h(u) - h(v) \geq 0.$$

Algorithmus Johnson(G)

Input: Gewichteter Graph $G = (V, E, c)$
Output: Minimale Gewichte aller Pfade D.

Neuer Knoten s. Berechne $G' = (V', E', c')$
if BellmanFord(G', s) = false **then** return "graph has negative cycles"
foreach $v \in V'$ **do**
    $h(v) \leftarrow d(s, v)$ // d aus BellmanFord Algorithmus
foreach $(u, v) \in E'$ **do**
    $\tilde{c}(u, v) \leftarrow c(u, v) + h(u) - h(v)$
foreach $u \in V$ **do**
    $\tilde{d}(u, \cdot) \leftarrow$ Dijkstra$(\tilde{G}', u)$
    foreach $v \in V$ **do**
        $D(u, v) \leftarrow \tilde{d}(u, v) + h(v) - h(u)$

Analyse

Laufzeiten

- Berechnung von G': $\mathcal{O}(|V|)$
- Bellman Ford G': $\mathcal{O}(|V| \cdot |E|)$
- $|V|\times$ Dijkstra $\mathcal{O}(|V| \cdot |E| \cdot \log |V|)$
  (Mit Fibonacci-Heap:: $\mathcal{O}(|V|^2 \log |V| + |V| \cdot |E|)$)

Insgesamt $\mathcal{O}(|V| \cdot |E| \cdot \log |V|)$
($\mathcal{O}(|V|^2 \log |V| + |V| \cdot |E|)$)